36

ADVANCES IN PARALLEL COMPUTING

Parallel Computing: Technology Trends

Edited by
Ian Foster
Gerhard R. Joubert
Luděk Kučera
Wolfgang E. Nagel
Frans Peters

IOS Press

Parallel Computing: Technology Trends

The year 2019 marked four decades of cluster computing, a history that began in 1979 when the first cluster systems using Components Off The Shelf (COTS) became operational. This achievement resulted in a rapidly growing interest in affordable parallel computing for solving compute intensive and large scale problems. It also directly lead to the founding of the Parco conference series.

Starting in 1983, the International Conference on Parallel Computing, ParCo, has long been a leading venue for discussions of important developments, applications, and future trends in cluster computing, parallel computing, and high-performance computing. ParCo2019, held in Prague, Czech Republic, from 10 – 13 September 2019, was no exception. Its papers, invited talks, and specialized mini-symposia addressed cutting-edge topics in computer architectures, programming methods for specialized devices such as field programmable gate arrays (FPGAs) and graphical processing units (GPUs), innovative applications of parallel computers, approaches to reproducibility in parallel computations, and other relevant areas.

This book presents the proceedings of ParCo2019, with the goal of making the many fascinating topics discussed at the meeting accessible to a broader audience. The proceedings contains 57 contributions in total, all of which have been peer-reviewed after their presentation. These papers give a wide ranging overview of the current status of research, developments, and applications in parallel computing.

ISBN 978-1-64368-070-5 (print)
ISBN 978-1-64368-071-2 (online)
ISSN 0927-5452 (print)
ISSN 1879-808X (online)

PARALLEL COMPUTING: TECHNOLOGY TRENDS

Advances in Parallel Computing

Parallel processing is ubiquitous today, with applications ranging from mobile devices such as laptops, smart phones and in-car systems to creating Internet of Things (IoT) frameworks and High Performance and Large Scale Parallel Systems. The increasing expansion of the application domain of parallel computing, as well as the development and introduction of new technologies and methodologies are covered in the *Advances in Parallel Computing* book series. The series publishes research and development results on all aspects of parallel computing. Topics include one or more of the following:

- Parallel Computing systems for High Performance Computing (HPC) and High Throughput Computing (HTC), including Vector and Graphic (GPU) processors, clusters, heterogeneous systems, Grids, Clouds, Service Oriented Architectures (SOA), Internet of Things (IoT), etc.
- High Performance Networking (HPN)
- Performance Measurement
- Energy Saving (Green Computing) technologies
- System Software and Middleware for parallel systems
- Parallel Software Engineering
- Parallel Software Development Methodologies, Methods and Tools
- Parallel Algorithm design
- Application Software for all application fields, including scientific and engineering applications, data science, social and medical applications, etc.
- Neuromorphic computing
- Brain Inspired Computing (BIC)
- AI and (Deep) Learning, including Artificial Neural Networks (ANN)
- Quantum Computing

Series Editor:
Professor Dr. Gerhard R. Joubert

Volume 36

Recently published in this series

Vol. 35. F. Xhafa and A.K. Sangaiah (Eds.), Advances in Edge Computing: Massive Parallel Processing and Applications
Vol. 34. L. Grandinetti, G.R. Joubert, K. Michielsen, S.L. Mirtaheri, M. Taufer and R. Yokota (Eds.), Future Trends of HPC in a Disruptive Scenario
Vol. 33. L. Grandinetti, S.L. Mirtaheri, R. Shahbazian, T. Sterling and V. Voevodin (Eds.), Big Data and HPC: Ecosystem and Convergence

Volumes 1–14 published by Elsevier Science.

ISSN 0927-5452 (print)
ISSN 1879-808X (online)

Parallel Computing: Technology Trends

Edited by

Ian Foster

Argonne National Laboratory and University of Chicago, Chicago, USA

Gerhard R. Joubert

Technical University Clausthal, Clausthal-Zellerfeld, Germany

Luděk Kučera

Charles University, Prague, Czech Republic

Wolfgang E. Nagel

Technical University Dresden, Dresden, Germany

and

Frans Peters

formerly Philips Research, Eindhoven, Netherlands

IOS Press

Amsterdam • Berlin • Washington, DC

ISBN 978-1-64368-070-5 (print)
ISBN 978-1-64368-071-2 (online)
Library of Congress Control Number: 2020934256
doi: 10.3233/APC36

Publisher
IOS Press BV
Nieuwe Hemweg 6B
1013 BG Amsterdam
Netherlands
fax: +31 20 687 0019
e-mail: order@iospress.nl

For book sales in the USA and Canada:
IOS Press, Inc.
6751 Tepper Drive
Clifton, VA 20124
USA
Tel.: +1 703 830 6300
Fax: +1 703 830 2300
sales@iospress.com

Preface

The *ParCo2019* conference was hosted by Charles University, Prague, Czech Republic. This event marked thirty-six years of *ParCo* conferences. During the past decades the conference proved to be not only a mirror of the advances made in the development and use of parallel computing, but also a stimulator for advancing research and development of many new technologies.

During the opening session of the conference it was noted that 2019 also marked four decades of cluster computing. In 1979 a cluster system with a simple MIMD (Multiple Instruction Multiple Data) architecture using COTS (Components Off The Shelf) be-came operational. A short historical overview of the development and results obtained with the cluster system is given in the note titled: *Four Decades of Cluster Computing* included in these proceedings. The research done with this system brought about in-ternational cooperation involving a number of researchers. These international connections resulted in the start of the international ParCo conferences in 1983, which in turn initiated the journal Parallel Computing (Elsevier) and later the book series Advances in Parallel Computing (Elsevier and IOS Press).

The scientific program of the *ParCo2019* conference consisted of invited talks, con-tributed papers and papers presented as part of symposia. The invited speakers were:

- Torsten Hoeffler: Performance Portability with Data-Centric Parallel Programming
- Thomas Lippert: Scalability, Cost-Effectiveness and Composability of Large-Scale Supercomputers through Modularity
- Ian Foster: Coding the Continuum (slides at www.parco.org/slides/Foster.pdf)
- Jaroslav Cervinka: High Performance Computing at Skoda Auto
- Mikhail Dyakonov: Will we ever have a quantum computer?
- Erik D'Hollander: Empowering Parallel Computing with Field Programmable Gate Arrays
- Jean-Pierre Panziera: Addressing the Exascale Challenges (slides at www.parco.org/slides/Panziera.pdf)
- Jean-Marc Denis: European Processor Initiative: The European Vision for Exascale Ages and Beyond (slides at www.parco.org/slides/Denis.pdf)

Two Invited Speakers chose to submit a paper for inclusion in the proceedings. In addition slides presented by three speakers are available on the ParCo Website.

Contributed papers presented at the conference were selected based on paper proposals. Submitted papers were reviewed in a first round prior to the conference and again in a second round at the presentation of the papers. The results and recommendations from the two review rounds were communicated to authors. Authors of accepted papers were given the option to submit a revised version of their paper. The proceedings thus only include papers that were accepted after their presentation at the conference. The papers ultimately selected for publication give a wide ranging overview of the current status of Parallel Computing research, developments and applications.

Four symposia were organised and presented as part of the conference. Organisers of Symposia were responsible for the reviewing of the respective papers presented and submitted for publication.

The Editors are indebted to all persons who assisted in making the conference a suc-cess. These include the staff at the registration desk and the technical staff who ensured the functioning of all equipment. A particular word of thanks is due to Siavash Ghiasvand from the Dresden University of Technology for his support in compiling the final version of the manuscript of these proceedings.

A final note needs mentioning: This is the first volume in the *Advances in Parallel Computing* book series that is published as an Open Access (OA) book, making the contents of the book freely accessible to everyone. The publication of the proceedings as an OA book does not change the indexing of the published material in any way.

Ian Foster
Gerhard Joubert
Luděk Kučera
Wolfgang Nagel
Frans Peters

Conference Organisation

Conference Committee
Gerhard Joubert, Germany (Conference Chair)
Ian Foster, USA
Luděk Kučera, Czech Republic
Thomas Lippert, Germany
Wolfgang Nagel, Germany
Frans Peters, Netherlands

Program Committee
Ian Foster, USA
Wolfgang Nagel, Germany

Symposium Committee
Gerhard Joubert, Germany
Thomas Lippert, Germany
Luděk Kučera, Czech Republic

Organising & Exhibition Committee
Luděk Kučera, Czech Republic

Finance Committee
Frans Peters, Netherlands (Finance Chair)

ParCo2019 Sponsors

EPI (European Processor Initiative)
hoComputer & Intel
Jülich Supercomputing Centre, Jülich Forschungszentrum, Germany
The University of Chicago, USA
Charles University, Czech Republic
Technical University Clausthal, Germany

Program Committee

Ian Foster, USA (Program Committee Chair)
Wolfgang Nagel, Germany (Program Committee Chair)

David Abramson, Australia
Marco Aldinucci, Italy
Christian Bischof, Germany
Jens Breitbart, Germany
Kris Bubendorfer, New Zealand
Andrea Clematis, Italy
Umit Catalyurek, USA
Sudheer Chunduri, USA
Massimo Coppola, Italy
Luisa D'Amore, Italy
Pasqua D'Ambra, Italy
Erik D'Hollander, Belgium
Bjorn De Sutter, Belgium
Ewa Deelman, USA
Frédéric Desprez, France
Didier El Baz, France
Ian Foster, USA
Geoffrey Fox, USA
Franz Franchetti, USA
Basilio B. Fraguela, Spain
Karl Fürlinger, Germany
Edgar Gabriel, USA
Efstratios Gallopoulos, Greece
José Daniel Garcia Sanchez, Spain
Michael Gerndt, Germany
William Gropp, USA
Georg Hager, Germany
Kevin Hammond, UK
Lei Huang, USA
Emmanuel Jeannot, France
Odej Kao, Germany
Wolfgang Karl, Germany
Carl Kesselman, USA
Christoph Kessler, Sweden
Harald Köstler, Germany
Dieter Kranzlmüller, Germany
Herbert Kuchen, Germany
Alexey Lastovetsky, Ireland
Jin-Fu Li, Taiwan
Jay Lofstead, USA
Ignatio Martin Llorente, Spain
Allen D. Malony, USA
Simon McIntosh-Smith, UK
Bernd Mohr, Germany
Wolfgang E. Nagel, Germany
Kengo Nakajima, Japan
Bogdan Nicolae, USA
Dimitrious Nikolopoulos, UK
Manish Parashar, USA
Christian Pérez, France
Sege Petiton, France
Oscar Plata, Spain
Sabri Pllana, Sweden
Enrique S. Quintana-Ortí, Spain
Carl Raicu, USA
J. (Ram) Ramanujam, USA
Matei Ripeanu, Canada
Dirk Roose, Belgium
Peter Sanders, Germany
Henk Sips, Netherlands
Domenico Talia, Italy
Michela Taufer, USA
Valerie Taylor, USA
Doug Thain, USA
George Thiruvathukal, USA
Massimo Torquati, Italy
Denis Trystram, France
Sudharshan Vashkudai, USA
Jose Luis Vazquez-Poletti, Spain
Jon Weissman, USA

Mini-Symposia

Tools and Infrastructure for Reproducibility in Data-intensive Applications

Organisers

Sandro Fiore, USA
Ian Foster, USA
Carl Kesselman, USA

ParaFPGA 2019: Parallel Computing with FPGAs

Organisers

Erik D'Hollander, Belgium
Abdellah Touhafi, Belgium

Program Committee:

Frank Hannig, Germany
Yun Liang, China
Tsutomu Maruyama, Japan
Dionisios Pnevmatikatos, Greece
Viktor Prasanna, USA
Dirk Stroobandt, Belgium
Wim Vanderbauwhede, UK
Sotirios G. Ziavras, USA

Energy-efficient Computing on Parallel Architectures (ECO-PAR)

Organisers

Enrico Calore, Italy
Nikela Papadopoulou, Greece
Sebastiano Fabio Schifano, Italy
Vladimir Stegailov, Russia

ELPA – A Parallel Dense Eigensolver for Symmetric Matrices with Applications in Computational Chemistry

Organisers

Thomas Huckle, Germany
Bruno Lang, Germany

This page intentionally left blank

Contents

Parallel Architecture

Parallel Methods

Parallel Performance

Parallel Programming

This page intentionally left blank

Opening

This page intentionally left blank

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200017

Four Decades of Cluster Computing

Gerhard JOUBERT[a,1], Anthony MAEDER[b]
[a] *Clausthal University of Technology, Germany*
[b] *Flinders University, Adelaide, Australia*

Abstract.

During the latter half of the 1970s high performance computers (HPC) were constructed using specially designed and manufactured hardware. The preferred architectures were vector or array processors, as these allowed for high speed processing of a large class of scientific/engineering applications. Due to the high cost of the development and construction of such HPC systems, the number of available installations was limited. Researchers often had to apply for compute time on such systems and wait for weeks before being allowed access. Cheaper and more accessible HPC systems were thus in great need. The concept to construct high performance parallel computers with distributed Multiple Instruction Multiple Data (MIMD) architectures using standard off-the-shelf hardware promised the construction of affordable supercomputers. Considerable scepticism existed at the time about the feasibility that MIMD systems could offer significant increases in processing speeds. The reasons for this were due to Amdahl's Law, coupled with the overheads resulting from slow communication between nodes and the complex scheduling and synchronisation of parallel tasks. In order to investigate the potential of MIMD systems constructed with existing off-the-shelf hardware a first simple two processor system was constructed that finally became operational in 1979. In this paper aspects of this system and some of the results achieved are reviewed.

Keywords. MIMD parallel computer, cluster computer, parallel algorithms, speed-up, gain factor.

1. Introduction

During the 1960s and 1970s the solution of increasingly complex scientific problems resulted in a demand for more powerful computers. The available sequential processors proved unable to meet these demands. The attempts implemented in the late 1960s to optimise the execution of sequential program code by analysing program execution patterns resulted in optimised execution strategies [1, 2]. These attempts to increase the processing speeds of sequential SISD (Single Instruction Single Data) computers were limited and did not offer the compute power needed for the processing of compute intensive problems. A typical problem at the time was to be able to compute a 24 hour weather forecast in less than 24 hours.

A next step was to speed up the execution of compute intensive sections of a program through specially designed hardware. An often occurring operation in scientific computations is the processing of vectors and matrices. Such operations can be executed in parallel by SIMD (Single Instruction Multiple Data) processors. It was thus a natural approach in the 1970's to develop vector and array processors as the supercom-

[1] Lange-Feld-Str. 45, Hanover, Germany. E-mail: gerhard.joubert@tu-clausthal.de

puters of the day. Examples are the ICL DAP (Distributed Array Processor), ILLIAC, CRAY, etc.

The problem was that the development of such specially designed and built machines was expensive. The use of such supercomputers by researchers as well as software developers was limited due to the high cost of purchasing and running these systems. In addition the programming of applications software often had to resort to machine level instructions in order to utilise the particular hardware characteristics of the available machine.

The development of integrated circuits during the early 1970's, which enabled the large scale production of processors at ever lower cost, opened up the possibility to use such components to construct MIMD parallel computers at low cost. The concept proposed in a non-published talk in 1976 [3] was that the future of high performance computing at acceptable costs was possible by using standard COTS (Components Off The Shelf) to construct low-cost parallel computers. The architecture of such systems could be adapted by using standard as well as special compute nodes, different storage architectures and various interconnection networks.

The concept of developing such systems was, however, deemed unattractive during the late 1970's mainly due to two aspects. The first was Amdahl's Law [4] that only a relatively small percentage of programs could be parallelised, and the second was that the synchronisation and communication requirements would create an overhead, which made parallel systems highly inefficient. A further aspect that hampered the acceptance of MIMD systems, was Grosch's Law [5], which stated that computer performance increases as the square of the cost, i.e. if a computer costs twice as much one could expect it to be four times more powerful. This does not apply to MIMD systems as the addition of nodes results in a linear increase in compute power. Moore's Law [6] maintained in 1965 that the number of components per integrated circuit doubled every year, which was revised in 1975 to double every two years. This resulted in an estimated doubling of computer chip performance due to design improvements about every 18 months. It was an open question in how far these developments could offset the inherent disadvantages of MIMD systems.

In 1977 Prof. Tsutomu Hoshino and Prof. Kawai started a project in Japan to construct a parallel computer using standard components. Their aim was to develop a parallel system architecture that could be used to solve particular problems. The system was later called the PAX computer [7]. This approach was different from that described in the following sections, where the general applicability of MIMD systems to solve compute intensive problems was the main objective.

2. A Simple MIMD Parallel Computer

In 1976/77 a project was started at the University of Natal, South Africa to investigate the possibilities of achieving higher compute performances by connecting standard available mini-computers [8]. The final development stage was reached in 1979 when the system was upgraded to have both nodes with identical hardware. The parallel system was later named the CSUN (Computer System of the University of Natal) [8].

The project involved three aspects, viz. hardware and architecture, network and software.

2.1 Hardware and Architecture

The available hardware consisted of two standard HP1000 mini-computers. The processors were identical, but the memory sizes differed initially. The architecture decided on was a master-slave configuration with distributed memories. No commonly accessible memory was available. The HP1000 offered a micro programming capability, which allowed for special functions to be executed at high speed.

Fig. 1: The cluster system, admired by Chris Handley[2]

2.2 Network

The connection of the two nodes had to offer high communication speeds. This was realised by using a high-speed connection available for HP1000 mini computers for logging high volumes of data collected by scientific instruments. The cable was adapted by HP to supply a computer interface at both ends allowing the interconnection of the two nodes via interface cards installed in each machine. These interfaces were user configurable by means of adjustable switch settings for timing or logistic characteristics, allowing a computer-to-computer mode. The maximum transmission speed was one million 16 bit words per second.

2.3 Software

The Real Time Operating System (RTOS), HP-RTE, available for the HP1000 offered the basic platform for running and managing the nodes. The system had to be enhanced

[2] Later: University of Otago, New Zealand

by additional software modules to achieve the control of the overall parallel computer system. A monitor was developed to create an interface for users to input and run programs. Programs and data were provided on punched cards or tape.

A critical component was the communication between the two nodes. For this drivers were developed that also allowed for the synchronisation of tasks. With the master-slave organisation of the system the slave always had to be under control of the master. In an interrupt-driven environment this is easily accomplished. The communication available between the two nodes did not allow to transmit specific interrupt signals between the two machines. Thus data controlled transmission, i.e. sending all messages with header information, was used. Both sender and receiver had to wait for acknowledgement from the counterpart before message transmission could begin. This caused an additional overhead for the synchronisation of tasks.

The master node was responsible for all controlling activities. It prepared tasks for execution by the slave, downloaded these together with the data needed to the slave, which then started executing the tasks. The master in the meantime prepared its own tasks and executed these in parallel, exchanging intermediate results with the slave. The master also executed any serial tasks as required. The later upgrade of the system to have two equally equipped nodes simplified task scheduling.

Such a setup is of course very sensitive to the volume and frequency of data transmission. This must thus be considered by programmers when selecting an algorithm for solving a particular problem.

No programming tools for developing parallel software were available at the time. The standard programming language for scientific applications was FORTRAN. A precompiler was developed that processed instructions from programmers to automatically create parallel tasks that were inserted in the FORTRAN program code. The compiler subsequently created tasks that could be executed in parallel, which information was used to schedule the parallel execution of tasks.

3. Applications

The aim with the project was to show that at least some algorithms could be executed in less time by a cluster constructed with standard components. The two-node cluster was a starting point that could be easily expanded by adding more, not necessarily identical, nodes.

The physical limitations of the available nodes as well as the architecture of the cluster limited the classes of problems that could possibly be efficiently executed. Thus, a comparatively low volume of interprocessor data transfers as well as few synchronisation points relative to the amount of computational work, was an advantage.

Problems implemented on the cluster were, for example:

- Partial Differential Equations: One-dimensional heat equation solved by explicit and implicit difference methods [9]
- Solution of tridiagonal linear systems [10]
- Numerical integration [11].

4. Gain Factor

Several methods for assessing parallel computer performance are available, such as speedup, cost, etc. These metrics proved insufficient, especially in view of Amdahl's Law [4], for a comparison of the overall time used to solve a problem on a sequential processor and the MIMD system described above.

The measurement needed was a comparison of overall sequential compute time, T_s, and overall parallel compute time, T_p. A further aspect was that the optimal sequential and parallel algorithms may differ substantially. Thus, in the comparisons, the optimal algorithm for each processing mode—sequential or parallel—was used.

A large number of aspects influence the value of T_p, such as organisation and speed of processors (these need not be identical, thus potentially resulting in a heterogeneous system), interprocessor communication speed, communications software design, construction of algorithms, etc. In practice time measurements can be made to obtain values for T_s and T_p for particular algorithms. This gives a Gain Factor:

$$G = (T_s - T_p)/T_s$$

If $0 < G \leq 1$ parallel processing offers an advantage over sequential processing. The upper limit, $G = 1$, is obtained when T_p, the overall time used to solve a problem with the parallel machine, is zero. When $G \leq 0$ parallel computation offers no advantage. Note that G applies equally well to the performance measurement of heterogeneous systems, and includes communication and administration overheads and covers the limitations expressed in Amdahl's Law.

Results obtained for a number of test cases using the two node cluster, are [12]:

- Solution of tridiagonal linear systems, 120x120: G = 0.42
- One-dimensional diffusion equation, 30.000 time steps: G = 0.481
- Numerical integration, 30.000 steps: G = 0.497.

With a two node cluster the value of $G \leq 0.5$.

These results showed that, at least in some cases, parallel processing using an MIMD system with distributed memories may offer significant advantages.

5. Conclusions

The results obtained with the simple two-node MIMD parallel system showed that clusters constructed with standard components can be used to boost the execution of parallel algorithms for solving certain classes of problems.

The results obtained with the system prompted further research on the effects of more nodes, different connection networks and suitable algorithms.

This work resulted in the start of the international **Parallel Computing (ParCo) conference series** with the first conference held in 1983 in West-Berlin. The aims with these events was to stimulate research and development of all types of parallel systems, as it was clear from the outset that not one architecture is suitable for solving all problems.

It took more than a decade for the idea of using standard components to construct HPC systems to be adopted by industry on a comprehensive scale. It was also only gradually realised that the flexibility of cluster systems allowed for the processing of a

wide range of compute intensive and/or large scale problems. The resulting advent of cheaper parallel systems built with commodity hardware lead to many specially designed HPC systems becoming less competitive due to their high price tags and limited application spectrum. The resulting major crisis in the supercomputing industry during the late 1980's and early 1990's lead to the demise of many companies supplying specially designed hardware aimed at particular problem classes..

Exascale computing is presently the next step in HPC and this will require extreme parallelism, employing many thousands or millions of nodes, to achieve its goals.

With the end of Moore's Law approaching, new technologies may emerge, to achieve the future development of HPC beyond exascale.

References

[1] Anderson, D. W., Sparacio, F. J., Tomasulo, R. M.: The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling (1967), See: http://home.eng.iastate.edu/~zzhang/courses/cpre585-f04/reading/ibm67-anderson-360.pdf

[2] Schneck, Paul B.: The IBM 360-91, In: Supercomputer Architecture, The Kluwer International Series in Engineering and Computer Science (Parallel Processing and Fifth generation Computing), Springer, Bo ston, MA, Vol. 31, 53-98 (1987)

[3] Joubert, G.: Invited Talk, Helmut Schmidt University, Hamburg, January 1976

[4] Amdahl, Gene M.: Validity of the Single Processor Approach to Achieving Large-Scale Computing Cap abilities. AFIPS Conference Proceedings (30): 483–485. doi:10.1145/1465482.1465560, (1967)

[5] Grosch, H.R.J.: High Speed Arithmetic: The Digital Computer as a Research Tool, Journal of the Optical Society of America. **43** (4): 306–310 (1953). doi:10.1364/JOSA.43.000306

[6] Moore, Gordon: Cramming More Components onto Integrated Circuits, Electronics Magazine. **38** (8): 114–117 (1965)

[7] Hoshino, Tsutomu: PAX Computer, Reading, Massachusetts, etc.: Addison Wesley Publishing Company (1989)

[8] Proposed by U. Schendel, Free University of Berlin (1979)

[9] Joubert, G. R., Maeder, A. J.: An MIMD Parallel Computer System, Computer Physics Communications, Amsterdam: North Holland Publishing Company, Vol. 26, 253-257 (1982)

[10] Joubert, Gerhard, Maeder, Anthony: Solution of Differential Equations with a Simple Parallel Com puter, International Series on Numerical Mathematics (ISNM), Birkhäuser: Basel, Vol. 68,137-144 (1982)

[11] Joubert, G. R., Cloete, E.: The Solution of Tridiagonal Linear Systems with an MIMD Parallel Computer, ZAMM Zeitschrift für Angewandte Mathematik und mechanik, Vol. 65, 4, 383-385 (1985)

[12] Joubert, G. R., Maeder, A. J., Cloete, E.: Performance measurements of Parallel Numerical Algorithms on a Simple MIMD Computer, Proceedings of the the Seventh South African Symposium on Numerical Mathematics, Computer Science Department, University of Natal, Durban, ISBN 0 86980 264 X, 25-36 (1981)

Invited Talks

This page intentionally left blank

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200019

Will We Ever Have a Quantum Computer?

M.I. Dyakonov[1]

Laboratoire Charles Coulomb, Université Montpellier, France

Abstract. In the hypothetical quantum computing one replaces the classical two-state **bit** by a quantum element (**qubit**) with two **basic** states, ↑ and ↓. Its arbitrary state is described by the wave function $\psi = a\uparrow + b\downarrow$, where a and b are complex amplitudes, satisfying the normalization condition. Unlike the classical bit, that can be only in **one** of the two states, ↑ or ↓, the qubit can be in a continuum of states defined by the quantum amplitudes a and b. **The qubit is a continuous object**. At a given moment, the state of a quantum computer with N qubits is characterized by 2^N quantum amplitudes, which are continuous variables restricted by the normalization condition only. Thus, the hypothetical quantum computer is an **analog machine** characterized by a super-astronomical number of continuous variables (even for N~100÷1000). Their values cannot be arbitrary, they must be under our control. Thus the answer to the question in title is: When physicists and engineers will learn to keep under control this number of continuous parameters, which means **- never**.

Keywords. Quantum computing, qubits

1. Introduction

The idea of quantum computing was first put forward in a rather vague form by the Russian mathematician Yuri Manin in 1980. In 1981 it was independently proposed (also in a vague form) by Richard Feynman. Realizing that (because of the exponential increase of the number of quantum states) computer simulations of quantum systems become impossible when the system is large enough, he advanced the idea that to make them efficient the computer itself should operate in the quantum mode: "Nature isn't classical and if you want to make a simulation of Nature, you'd better make it quantum mechanical, and by golly it's a wonderful problem, because it doesn't look so easy". David Deutsch in 1985, formally described the universal quantum computer, as a quantum analog of the Universal Turing machine.

The subject did not attract much attention until Peter Shor in 1994 proposed an algorithm allowing to factor very large numbers on an *ideal* quantum computer much faster compared to the conventional (classical) computer. This outstanding theoretical result has triggered an explosion of general interest in quantum computing and many thousands of research papers, mostly theoretical, have been and still continue to be published at an increasing rate.

[1] Laboratoire Charles Coulomb, Université Montpellier, cc 070, 34095 Montpellier, France **michel.dyakonov@gmail.com**

2. Progress

During the last 20 years one can hardly find an issue of any science digest magazine, or even of a serious physical journal, that does *not* address quantum computing. Quantum Information Centers are opening all over the globe, funds are generously distributed, and breathtaking perspectives are presented to the layman by enthusiastic scientists and journalists. Many researchers feel obliged to justify whatever research they are doing by claiming that it has some relevance to quantum computing.

Computer scientists are proving and publishing new theorems related to quantum computers at a rate of ~ *ten articles per day*. A huge number of proposals have been published for various physical objects that could serve as quantum bits, or *qubits.* As of October 7, 2019, Google gives 6 970 000 results for "quantum computing", and 201 000 results for "quantum computing with", and these numbers increase every day. The impression has been created that quantum computing is going to be the next technological revolution of the 21st century. When will we have useful quantum computers? The most optimistic experts say: "In 10 years", others predict 20 to 30 years (note that those expectations have remained unchanged during the last 20 years), and the most cautious ones say: "Not in my lifetime". The present author belongs to the meager minority answering "Not in any foreseeable future", and this point of view is being explained below.

At a given moment the state of the *classical* computer is described by a sequence (↑↓↑↑↓↑↓↓…), where ↑ and ↓ represent *bits* of information – realized as the *on* and *off* states of individual transistors. With N transistors, there are 2^N different possible states of the computer. The computation process consists in a sequence of switching some transistors between their ↑ and ↓ states according to a prescribed program.

In *quantum* computing one replaces the classical two-state element by a quantum element with two *basic states*, the quantum bit, or *qubit.* The simplest object of this kind is the electron internal angular momentum, *spin*, with the peculiar quantum property of having only *two* possible projections on *any* axis: +1/2 or -1/2 (in units of the Planck constant). For some chosen axis, we again denote the two basic quantum states of the spin as ↑ and ↓.

However, an *arbitrary* spin state is described by the *wave function* $\psi = a\uparrow + b\downarrow$, where a and b are complex numbers, satisfying the condition $|a|^2 + |b|^2 = 1$, so that $|a|^2$ and $|b|^2$ are the *probabilities* for the spin to be in the basic states ↑ and ↓ respectively.

In contrast to the classical *bit* that can be only in *one of the two* states, ↑ and ↓, the *qubit* can be in a *continuum* of states defined by the quantum amplitudes a and b. This property is often described by the rather mystical and frightening statement that the qubit can exist *simultaneously* in *both* of its ↑ and ↓ states. (This is like saying that a vector in the xy plane directed at 45° to the x-axis *simultaneously* points *both* in the x- and y-directions - a statement that is true in some sense, but does not have much useful content.)

Note that since a and b are complex numbers satisfying the normalization condition, and since the overall phase of the wave function is irrelevant, there remain two free parameters defining the state of a single qubit (exactly like for a classical vector whose

orientation in space is defined by two polar angles). This analogy does not apply any more when the number of qubits is 2 or more.

With two qubits, there are $2^2 = 4$ basic states: (↑↑), (↑↓), (↓↑), and (↓↓). Accordingly, they are described by the *wave function* $\psi = a(\uparrow\uparrow) + b(\uparrow\downarrow) + c(\downarrow\uparrow) + d(\downarrow\downarrow)$ with 4 complex amplitudes *a, b, c*, and *d*. In the general case of N qubits, the state of the system is described by 2^N complex amplitudes restricted by the normalization condition only.

While the state of the classical computer with *N bits* at any given moment coincides with *one* of its 2^N possible discreet states, the state of a quantum computer with *N* qubits is described by the *values of* 2^N *continuous variables,* the quantum amplitudes.

This is the origin of the supposed power of the quantum computer, but it is also the reason for it's great fragility and vulnerability. The information processing is supposed to be done by applying unitary transformations (*quantum gates*), that change these amplitudes *a, b, c*... in a precise and controlled manner. The number of qubits needed to have a useful machine (i.e. one that can compete with your laptop in solving certain problems, like e.g. factoring very large numbers by Shor's algorithm) is estimated to be $10^3 - 10^5$. Thus the number of continuous variables describing the state of such a quantum computer at any given moment is at least 2^{1000} ($\sim 10^{300}$) which is much, much greater than the number of particles in the whole Universe (this is only $\sim 10^{80}$)!

At this point a normal engineer, or an experimenter, looses interest. Indeed, possible errors in a classical computer consist in the fact that one or more transistors are switched off instead of being switched on, or vice versa. This certainly is an unwanted occurrence, but can be dealt with by relatively simple methods employing *redundance*.

In contrast, accomplishing the Sisyphean task of keeping under control 10^{300} *continuous variables* is absolutely unimaginable. However, the QC theorists have succeeded in transmitting to the media and to the general public the belief that the feasibility of large-scale quantum computing has been *proved* via the famous *threshold theorem*: once the error per qubit per gate is below a certain value, indefinitely long quantum computation becomes feasible, at a cost of substantially increasing the number of qubits needed (the *logical* qubit is encoded by several *physical* qubits). Very luckily, the number of qubits increases only *polynomially* with the size of computation, so that the total number of qubits needed must increase from $N = 10^3$ to $N = 10^6 - 10^9$ only (with a corresponding increase of the *unimaginable* number of 2^N continuous parameters defining the state of the whole machine!!!).

3. Experimental studies

Experimental studies related to the idea of quantum computing make only a small part of the huge QC literature. They represent the *nec plus ultra* of the modern experimental technique, they are extremely difficult and inspire respect and admiration. The goal of such proof-of-principle experiments is to show the possibility to realize the basic quantum operations, as well as to demonstrate some elements of quantum algorithms.

The number of qubits used is below 10, usually from 3 to Apparently, going from 5 qubits to 50 (the goal set by the ARDA Experts Panel road map for the year 2012!) presents hardly surmountable experimental difficulties and the reasons for this should be understood. Most probably, they are related to the simple fact that $2^5 = 32$, while $2^{50} = 1125899906842624$.

By contrast, the *theory* of quantum computing, which largely dominates in the literature, does not appear to meet any substantial difficulties in dealing with millions of qubits. Various noise models are being considered, and it has been proved (under certain assumptions) that errors generated by "local" noise can be corrected by carefully designed and very ingenious methods, involving, among other tricks*, massive parallelism*: many thousands of gates should be applied simultaneously to different pairs of qubits and many thousands of measurements should be done simultaneously too.

An important issue is related to the *energies* of the ↑ and ↓ states. While the notion of *energy* is of primordial importance in all domains of physics, both classical and quantum, it is not in the vocabulary of QC theorists. (Surprisingly, they also have no use for other indispensable attributes of Quantum Mechanics, like Hamiltonian and Schroedinger equation).

They implicitly assume that the energies of all 2^N states of an ensemble of qubits *are exactly equal*. Otherwise, the existence of an energy difference ΔE leads to oscillations of the quantum amplitudes with a frequency $\Omega = \Delta E/ \hbar$, where $\hbar$ is the Planck constant, and this is a basic fact of Quantum Mechanics. (For example, one of the popular candidates for a qubit, the electron spin, will make a precession around the direction of the Earth's magnetic field with a frequency ~ 1 MHz). Should the Earth's magnetic field be screened, and if yes, with what precision?

Whatever is the nature of qubits, some energy differences will necessarily exist because of stray fields, various interactions, etc. resulting in a chaotic dynamics of the whole system, which will completely disorganize the performance of the quantum machine. I am not aware of any studies of this very general problem.

Let us recall that our laptops have originated from the construction of the elementary **electronic calculator** which replaced the abacus in the 60is. Step by step improvements and developments of this simple device resulted in the supercomputers that we have today.

With quantum computing, this natural process has been reversed: the field started with fantastic promises of breaking security codes and changing our world forever. However, after more than 20 years of unprecedented hype there still is nothing real to show. Forget "quantum supremacy" and factoring atrociously large numbers. Just show us some working quantum device, however simple, e.g. a quantum school calculator which could perform operations like 3+5, or 3x5, and maybe even factor 15 by using Shor's algorithm! I would not mind if this quantum calculator had the size of a 3 story building and immersed in liquid Helium...

However, 25 years after Shor's seminal theoretical work, which triggered the whole field of quantum computing, and many, many billions of dollars spent, these

elementary tasks are still far beyond our capabilities. This fact does not inspire any confidence.

4. Conclusions.

The hypothetical quantum computer is a system with an unimaginable number of continuous degrees of freedom - the values of the 2^N quantum amplitudes with $N \sim 10^3$–10^5 . These values cannot be arbitrary, they should be under our control with a high precision (which has yet to be defined).

Riding a bike, after some training, we learn to successfully control 3 degrees of freedom: the velocity, the direction, and the angle that our body makes with respect to the pavement. A circus artist manages to ride a one-wheel bike with 4 degrees of freedom. Now, imagine a bike having 1000 (or 2^{1000} !) joints that allow free rotations of their parts with respect to each other. Will anybody be capable of riding this machine?

Thus, the answer to the question in title is: As soon as the physicists and the engineers will learn to control this number of degrees of freedom, which means - *NEVER*.

References

Some previous papers of the author on the same subject:

Quantum computing: a view from the enemy camp, Future Trends in Microelectronics. The nano Millenium, S. Luryi, J. Xu, and A. Zaslavsky (eds), Wiley (2002), pp. 307-318; arXiv:cond-mat/0110326

State of the art and prospects for quantum computing. Future Trends in Microelectronics. Frontiers and Innovations, S. Luryi, J. Xu, and A. Zaslavsky (eds), Wiley (2013), pp. 266-285; arXiv:1212.3562

Prospects for quantum computing: extremely doubtful, J. of Modern Physics, Conf. Series, **33**, 1460357 (2014); arXiv:1401.3629

The case against quantum computing, IEEE Spectrum, (March issue, 2019)
https://spectrum.ieee.org/computing/hardware/the-case-against-quantum-computing

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200020

Empowering Parallel Computing with Field Programmable Gate Arrays

Erik H. D'HOLLANDER [a,1]
[a] *Electronics and Information Systems Department, Ghent University, Belgium*

Abstract. After more than 30 years, reconfigurable computing has grown from a concept to a mature field of science and technology. The cornerstone of this evolution is the field programmable gate array, a building block enabling the configuration of a custom hardware architecture. The departure from static von Neumann-like architectures opens the way to eliminate the instruction overhead and to optimize the execution speed and power consumption. FPGAs now live in a growing ecosystem of development tools, enabling software programmers to map algorithms directly onto hardware. Applications abound in many directions, including data centers, IoT, AI, image processing and space exploration. The increasing success of FPGAs is largely due to an improved toolchain with solid high-level synthesis support as well as a better integration with processor and memory systems. On the other hand, long compile times and complex design exploration remain areas for improvement. In this paper we address the evolution of FPGAs towards advanced multi-functional accelerators, discuss different programming models and their HLS language implementations, as well as high-performance tuning of FPGAs integrated into a heterogeneous platform. We pinpoint fallacies and pitfalls, and identify opportunities for language enhancements and architectural refinements.

Keywords. FPGAs, high-level synthesis, high-performance computing, design space exploration

1. Introduction

Parallel Computing is predominantly focusing on high performance computing. It typically covers three major domains: architectures, algorithms and applications. These three components are considered a joint force to accelerate computations. Modern applications have an ever growing processing need. This is the case for many areas such as artificial intelligence, image processing, Internet of Things, and big data, not to mention cyber physical systems, which is the topic of one of the Horizon 2020 calls by the European Commission [14]. Despite an exponential rise of computing power for CPUs in general, we see that since 2003 the performance rate is dropping [19], first of all due to the Dennard scaling [9], which is actually a problem of warming up caused by a continuously rising clock frequency. A second aspect, not related to technology but to the type of application, is Amdahl's law [1], which is analyzed using parallel program models in section 5. The third one is the end of Moore's law [26], because transistors cannot prac-

[1] E-mail: Erik.DHollander@UGent.be.

tically be made smaller than a few silicon atoms. This means that we have to go new ways to accelerate the computations. One of the accelerators notoriously present in our today's desktops and supercomputers is the graphics processing unit. A GPU is an excellent device for parallel number crunching, originally rooted in image processing, but which since many years has outgrown this application domain by an order of magnitude. While GPUs are voracious number crunchers, graphics processing units are not a one size fits all architecture for every problem on the earth. A GPU is best suited for massively parallel computations. It has a fixed architecture consisting of many parallel processing engines, and a fast thread manager which is able to switch very quickly between threads waiting for data from the memory.

A major alternative emerging as a rising star is the field programmable gate array. In contrast to the GPU, the field programmable gate array has not a fixed but a flexible, reconfigurable architecture. An FPGA behaves metaphorically speaking like a chameleon because it is able to adapt itself to the type of the algorithm. This compute engine allows to build an ad hoc architecture which is a one-to-one mapping of the algorithm onto the hardware. There is no program, no instructions, just an interconnection of computing elements and control logic performing the task implied by the algorithm. This brings us to a very regular layout of the field programmable gate array: it consists of logic elements, computing elements such as digital signal processors, memory elements or RAM blocks, input-output elements at the border, clock signals to synchronize all the operations and finally a network of programmable interconnections used to configure the control and data paths of the design (Fig. 1).

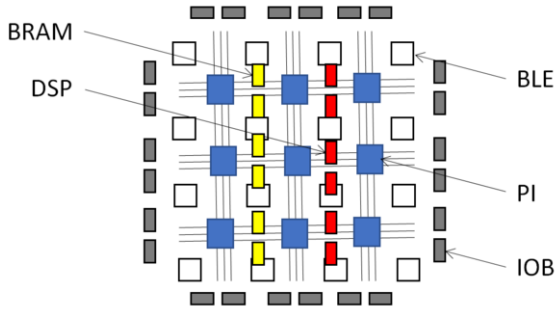


Figure 1. FPGA architecture with basic logic elements, programmable interconnects, I/O pins, block RAM and digital signal processors.

The so-called programming of an FPGA involves mapping a logic design, including DSPs and their interconnections, onto the physical hardware of the FPGA. This is done in two steps. First, a high-level synthesis or HLS compiler converts the program into a task graph, assigns the nodes of the graph to computing resources of the FPGA, maps the edges to control and data paths between the computing elements and generates a low-level hardware description in VHDL or Verilog. In the second step, the hardware description is mapped, placed and routed onto the physical available resources of the FPGA. The result is stored in a binary file, called the bitstream, which is uploaded to the configuration memory of the FPGA. After configuration, i.e. setting the interconnections, the programmed FPGA can operate e.g. as a data flow engine.

The rest of this paper is organized as follows. The origins of FPGAs are presented in section 2. The back-end of an FPGA compiler is described in section 3 and section 4 introduces the front end of high-level synthesis compilers, including performance factors and design exploration. The differences between FPGA and GPU accelerators are addressed in section 5, in particular focusing on the typical programming styles and application areas of both accelerators when using the same programming language OpenCL. Section 6 explores an integrated FPGA-CPU environment, including shared virtual memory, coherent caching and high bandwidth, as presented in the HARP-v2 platform. The power of this platform is illustrated by a guided image filter application. In section 7 some common misconceptions are addressed and concluding remarks are given in section 8.

2. History

Field programmable gate arrays have come a long way. They found their origin in the 60s when researchers were looking for new computer architectures as an alternative for the stored program concept of von Neumann. Not that the stored program concept was a bad idea, on the contrary: it is an ingenious innovation to put instructions and data in the same memory and to reconfigure the processor during instruction fetch and decode cycles. Why then look for new architectures? Well, by configuring the control and data paths at each instruction, the von Neumann computer trades in performance for flexibility.

The first person to challenge the von Neumann concept was Gerald Estrin from UCLA [12]. His idea was to extend the traditional computer, which he called a fixed architecture, with a variable part, used to implement complex logic functions. He built a six by six motherboard to hold configurable computing and memory elements. The compute elements implemented functions much more complex than the simple instructions in the fixed architecture. The backside of the motherboard consists of a wiring panel to interconnect the different components. Programming the "fix+variable" computer was quite a challenge, e.g. the computation of the eigenvalues of a symmetric matrix is described in a publication of 20 pages and resulted in a speedup of 4 with respect to an existing IBM computer [13]. Although these results are modest, they show that implementing an algorithm directly into hardware is beneficial. The question remains how this can be done efficiently.

In 1975 Jack Dennis of MIT pioneered a new architecture which he called the dataflow processor [10]. The architecture consists of a large set of computing elements which can be arbitrarily interconnected and which fire when data is present on the inputs. Programming the dataflow processor consists of interconnecting the computing elements according to the dataflow graph of the algorithm and percolating the data from the top. There are two performance gains. First, since the program is not stored in a memory, the instruction fetch and instruction decode phases are eliminated, leading to a performance gain of 20 to 40%. Second, the parallelism is maximally exploited, because it is only limited by the dependencies in the algorithm and by the available computing elements. A drawback of the first dataflow architecture is that the computing resources may be underutilized when occupied by a single data stream. Therefore Arvind, a colleague of Jack Dennis, proposed the tagged dataflow architecture [2] in which several separate data streams are identified by tokens with different colors and tokenized data compete for the compute resources in the system.

Why did these new architectures not materialize? Obviously, the technology was lagging behind, compared to the large-scale integration of today and also silicon compilers were still in their infancy. These problems have been largely overcome by the confluence and synergistic effects of hardware and software developments together with a major progress in the detection and management of parallelism in programs. Parallel computing techniques and versatile micro-electronics created a fertile ground for a new kind of architecture, the field programmable gate array. Almost simultaneously two companies were established. In 1974 Altera (now Intel) was founded by graduates of the University of Minnesota and in 1975 Xilinx was created with people from university of Illinois. Together these companies own more than 80% of the FPGA market. Nowadays, the field of FPGAs has grown into a mature hardware technology enhanced by two software developments. First, the integrated development tools now incorporate cycle accurate logic simulation, power and performance estimations and resource usage reports. Second, the support of high-level synthesis languages allows algorithms and programs to be written in C or OpenCL, and converted into low-level VHDL for implementation on the FPGA, thereby gaining several orders of magnitude in design exploration and development time. Nevertheless, programming FPGAs with high-level synthesis tools still requires a minimal notion of the underlying hardware.

3. Compiler back end

The smallest compute element of an FPGA, the lookup table or LUT, calculates an arbitrary logic function of a number of input variables. Several logic functions with the same inputs are calculated in parallel with the same LUT. Since the propagation delay of a LUT is much smaller than the clock cycle time of the FPGA, many LUTs can be cascaded into a long chain within the time frame of one clock cycle and thus create complex functions, such as an integer addition or multiplication. When the combined propagation delay becomes larger than the clock cycle time, the result is put into a flip-flop. The combination of a lookup table and the flip-flop is called a basic logic element. This element can be configured either as a logic function or as a state element, depending on the use of the flip-flop.

A high-level synthesis compiler converts a high-level program into LUTs, DSPs, gates, and their interconnections. Consider a sequence of functions operating on a data stream, $y = f_1(f_2(\ldots f_n(x)))$. Each of the functions is synthesized such that the combinational execution time is less than one clock cycle time and the result is stored in a flip-flop where it is ready for the next function. This creates a pipeline with a latency equal to the number of functions and a throughput of one result per cycle, as soon as the pipeline is filled with data. How is a pipeline created by the compiler? Let us look at a simple example, $y = ax + b$, represented by the task graph in figure 2. Assuming that the multiplication and the addition require one clock cycle each, then the compiler inserts registers or flip-flops to hold the intermediate results. The computation takes three clock cycles, one to read the input elements, one for the multiplication and one for the addition. Let us now extend this operation for arrays of n elements, $y[i] = a[i]x[i] + b[i]; \quad i = 0 \ldots n$ as described in a loop statement. In this case, the compiler creates extra control logic to fetch the input variables, launch the computation and store the output.

The output of an HLS compiler is a description of the data- and control paths, memory, registers, flip-flops, and interconnection logic, written in a low-level hardware lan-

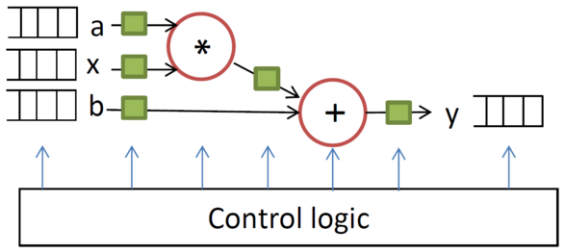


Figure 2. Task graph, registers (squares), I/O data streams and control logic for $y = ax + b$.

guage such as VHDL or Verilog. Next, the placement and routing of the hardware description are done by mapping, placing and routing the design onto the FPGA, taking into account the available resources. The resulting configuration of the hardware design is written into a bitstream file used to set the interconnections in the FPGA. The bitstream is moved to a persistent memory (e.g. flash memory) and from there it is stored in the configuration memory at the boot time of the FPGA. It is important to note that high-level synthesis compilers such as Vivado HLS of Xilinx or OpenCL of Intel run typically in about five minutes whereas the hardware place and route tools run in the order of five hours.

4. Compiler front end

Before long there were early birds attempting to create high-level synthesis compilers, mostly based on the knowledge and ideas of parallel computing developed in the 80s. The efficiency of these compilers was limited, partly due to the lack of detailed inside knowledge and proprietary information from the FPGA vendors [27]. Everything has turned around since Xilinx and Altera, now Intel, have developed high-level synthesis tools for their own devices, leading to Vivado HLS [28] by Xilinx or OpenCL [17] by Altera and Intel. Besides these imperative C-like languages, Maxeler developed a dataflow-oriented language Maxeler Java [3].

4.1. Design space exploration

The approach to obtain efficient hardware with high-level synthesis is the same for OpenCL and Vivado HLS: maximize the compute power using DSPs and maximize the reuse of local data, since on-chip BRAM memory is limited. This calls for algorithms with a high arithmetic density, i.e. a large number of operations per byte I/O [8]. The first step is to adapt the algorithm to the computational model of an FPGA and carry out an initial performance evaluation. Next, pragmas are inserted to improve the performance taking into account the available resource budget of DSPs, BRAMS, and LUTs. In general, it is not required to know low-level programming or detailed characteristics of the FPGA because most information is available in the compiler reports. Further refining can be done by simulation, emulation, and profiling. The interactive phase of generating an efficient design is called the design space exploration.

4.2. Performance factors

Key performance factors during the design space exploration are:

- *clock frequency.* A small clock cycle time limits the amount of work that can be done in one cycle and therefore more flip-flops and cycles will be needed to implement the design. This has an impact on speed and resource consumption. An HLS compiler will optimize the clock frequency, based on the value requested by the user and the capabilities of the hardware.
- *pipelining.* If the compiler is able to create pipelines with n stages, the execution speed increases to n times the speed of the non-pipelined version.
- *parallelism.* When an algorithm has m independent data streams, the compiler can organize m parallel pipelines and therefore multiply the speedup by the number of pipelines. Examples are executing pipelined dot product calculations of a matrix multiplication in parallel or using parallel streams in systolic computations [16]. The challenge here is to feed all pipelines simultaneously.

The bottom line is that the performance of an FPGA depends on two basic principles: cram as much as possible operations into a pipeline and create as many pipelines as possible which can be fed simultaneously.

4.3. The initiation interval

For maximum performance, pipelines need new data in each and every clock cycle. Pipeline bubbles and gaps are caused by memory or data dependencies. Memory dependencies occur when the number and width of data ports to the cache or DDR memory are insufficient to provide continuous parallel data streams to all pipelines. Data dependencies occur when computing elements have to wait more than one cycle for the results of other computing elements in order to carry out the computation. Both memory and data dependencies involve an action from the programmer. Off-chip memory dependencies due to bandwidth limitations are removed by using on-chip cache cores or reduced using fast interfaces, such as PCI-Express. Memory dependencies caused by contention for the limited number of on-chip BRAM memory ports are avoided by partitioning data over many parallel accessible memory banks. Data dependencies occur between loop iterations when the result of one iteration is used in a subsequent iteration. The impact of loop carried dependencies is expressed by the initiation interval, II. The initiation interval of a loop is the number of cycles one has to wait before a new iteration can start. Ideally, the initiation interval $II = 1$. However, due to data or memory dependencies, the initiation interval may become larger than 1 for example, $II = 2$. In that case, the performance drops by 50%. The initiation interval is therefore one of the most important loop characteristics shown in the compiler reports.

There are a number of hints to improve loop operations: loops are unrolled to increase the number of parallel iterations and generate more opportunities for pipelining. Complete unrolling may lead to an overuse of resources, requiring partial loop unrolling. The bandwidth usage is improved by coalescing load operations, leading to single wide data fetches instead of doing sequential loads. Memory locations unknown at compile time, caused by pointer arithmetic or complex index calculations, are an area of concern. In nested loops it is better to represent matrices as multidimensional arrays instead of using a computed index into a single array.

To illustrate the impact of design space exploration, consider the program in listing 1. A constant is added to a matrix in a double nested loop, l1 and l2. The pragmas of the Vivado HLS compiler used are unroll, pipeline and array partitioning. The pragmas are applied selectively according to the scenarios in table 1. Unrolling the outer loop gives no speedup. Unrolling the inner loop creates 512 parallel iterations, generating a modest 6-fold speedup. The reason is that FPGA memory banks have 2 ports, therefore we can only read two values per cycle, i.e. start 2 iterations in parallel in each cycle. The inner loop takes 512/2 = 256 cycles, whereas the same loop without pragmas takes 512*3 = 1536 cycles (3 cycles for respectively read, add and write). The solution is to increase the number of memory banks by partitioning the arrays `din` and `dout`. This scenario leads to a better speedup of 512, but still leaves an initiation interval of $II = 3$ in the outer loop, because the parallel iterations of the inner loop don't overlap and therefore an inner loop iteration takes 3 cycles. Pipelining the outer loop implies unrolling the inner loop according to the compiler documentation. Without partitioning, we again obtain a speedup of 6, due to the memory bottleneck. Pipelining the inner loop, plus array partitioning, creates overlapping iterations, an initiation interval $II = 1$ and a speedup of 1529. The same happens if we pipeline the outer loop, which implies unrolling and pipelining the inner loop completely.

Listing 1 Design space exploration, optimal case, II=1.

```
#define N 512
void nestedloop(int din[N][N], int dout[N][N]) {
#pragma HLS ARRAY_PARTITION variable=din  factor=256 dim=2
#pragma HLS ARRAY_PARTITION variable=dout factor=256 dim=2
l1:  for (int i = 0; i < N; i++) {
#pragma HLS PIPELINE
l2:      for (int j = 0; j < N; j++) {
             dout[i][j] = din[i][j] + 40;
         }
     }
  return;
}
```

Scenario	Outer (l1)	Inner (l2)	Array partitioning	II	Speedup
Unroll	x				1
Unroll		x		256	6
Unroll		x	x	3	512
Pipeline	x			256	6
Pipeline		x	x	1	1,529
Pipeline	x		x	1	1,529

Table 1. Design space exploration using unrolling, pipelining and array partitioning.

This simple design space exploration shows a performance improvement of more than 1500 with a number of well-chosen pragmas.

5. FPGA vs GPU programming: pipelining vs parallelism

Since more than two decades the performance gain of processors is hampered by clock rate and transistor scaling constraints. In accelerators such as GPUs, low clock rates have been replaced by explicit parallelism, yielding chips with hundreds or even thousands cores per chip die [21]. The exponential growth of parallel cores requires a proportional scaling of the problem parallelism. Recently, Hennessy and Patterson have illustrated that the law of Amdahl comes into play for the diminishing processing speed between the years 2011 and 2015 [19]. In this period GPUs were growing at a fast pace, and it reminds us that the basic power of GPUs is not always applicable in every algorithm. Amdahl's law specifies that we cannot diminish the execution time below the critical execution path of serial calculations, even if everything else is parallelized. This has two consequences for HPC programming: 1) the amount of serial computations in a program needs to be minimized and 2) highly parallel computers or GPUs require applications with ample parallelism. The first consequence is visible in figure 3, displaying the maximum speedup with p processors as a function of the critical serial part. Even with only 2% serial operations, the speedup is limited by 50, irrespective of the number of processors used, see figure 3. For the impact of the second consequence, we look at the distribution of the parallelism in a program.

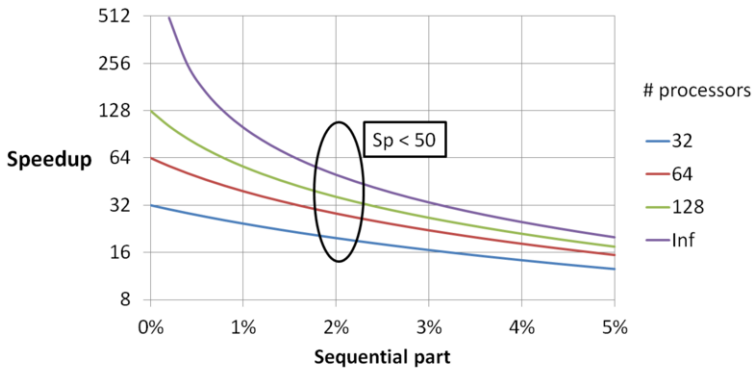


Figure 3. Application dependent speedup limit, Amdahl's law.

5.1. Distribution of parallelism in ordinary programs

Ideally, all p processors of a multiprocessor work all the time, so the parallelism is p. Ruby Lee studied three other distributions [23]: the equal time, equal work and inverse work hypothesis. Equal time means that the program executes an equal amount of time with respectively 1, 2 up to p processors operating in parallel. Equal work means that the amount of work executed with i processors is proportional to $i, i = 1 \ldots p$. The inverse work hypothesis means that the amount of work done by i processors in parallel is inversely proportional to i. These hypotheses give rise to three analytical formulas for the speedup, respectively $S_{et} = O(p)$ for the equal time distribution, $S_{ew} = O(p/\ln(p))$ for the equal work distribution and $S_{iw} = O(\ln(p))$ for the inverse work distribution, with

et = equal time, *ew* = equal work and *iw* = inverse work hypothesis. As an example, a workload of 48 units executed on 4 processors has a minimal execution time of 12 units. Equal time, equal work and inverse work distributions of the parallelism increase the execution time to respectively 19.2, 25.0 and 32.8 units. Ruby Lee also analyzed the parallelism of a large number of programs using the Paraphrase compiler at Illinois [22,23]. The results are shown in table 2.

Available processors	Speedup S_p	Hypothesis
1–10	$O(p)$	Equal time
11–1000	$O(p/\ln(p))$	Equal work
1001–10000	$O(\ln(p))\ldots O(p/\ln(p))$	Transient
>10000	$O(\ln(p))$	Inverse work

Table 2. Speedup bounds for applications with diminshing parallelism.

In this study, the exploitable parallelism diminishes with a growing number of available processors. Although this is an old empirical observation, it may reflect that Amdahl's law is actually a law of diminishing parallelism. Since GPUs are largely dependent on massive parallelism this may also explain the decreasing rise of the computing speed in the years 2011-2015 due to Amdahl's law.

5.2. The case of OpenCL

Partly due to the success of OpenCL for programming GPUs, FPGA vendors Altera/Intel and also Xilinx have selected this language for high-level synthesis [7]. However, GPUs and FPGAs are fundamentally different and this is reflected in the way an OpenCL compiler for FPGAs is designed and used.

GPUs have ample parallel cores called streaming multi-processors. These are most useful for SIMD calculations, which are launched in OpenCL by the NDRange kernel. An NDRange kernel executes the same iteration on parallel processors for each index in the *n*-dimensional iteration space of a nested loop. In a GPU architecture, the index points of a parallel loop are organized into groups of 32 threads, each executing the same instruction of the iteration for different index values [24]. Such a group of 32 threads is called a warp. All warps compete for execution and the warp scheduler assigns a SIMD instruction to a warp for which all data are available. This means that a GPU operates strictly in a single instruction multiple data or SIMD mode. When the data for a warp are not available, the warp scheduler assigns another warp for which data are ready. The fast thread switching allows to hide the memory latency of the waiting threads.

GPU	FPGA
Fast warp (thread) scheduler	Fixed configuration, no thread switching
Independent iterations	Loop carried dependencies OK
Massively parallel (SIMD)	Pipelined execution (MISD)
Large memory	Small memory footprint
Send $\rightarrow$ Calculate $\rightarrow$ Receive	Streaming data, comp./comm. overlap

Table 3. Operational differences between GPU and FPGA.

When implementing this mode of operation on an FPGA, we are faced with a number of discrepancies and operational differences, see table 3. First, an FPGA has no warp scheduler, since the design of a program is fixed in the configuration memory of the FPGA. This precludes thread switching to hide the memory latency, one of the major performance factors in GPUs. As a consequence, all data for the warp need to be available at all times, creating an extra hurdle since FPGAs usually have a small memory footprint and no cache on chip. Second, all threads in a warp operate independently and therefore require parallel loops without loop carried dependencies between iterations. This precludes the generation of long pipelines, which are favorable for execution on FPGAs. Third, since a GPU has its own hierarchy of memories and caches, the data is moved to the GPU, processed, and the results are sent back to the CPU. In contrast, FPGAs are best fit for long streams of data that are processed using overlapping computation and buffered communication. It becomes clear that FPGAs and GPUs are not so much competitors, but have a complementary role when it comes to different application domains.

5.3. Vector types in OpenCL

As an alternative to the resource-hungry NDRange loop control, OpenCL for FPGAs supports the more efficient vector data types. Vector types allow to operate in a SIMD fashion on vector data. This creates parallel pipelines, thereby multiplying the pipeline performance by the number of parallel data streams operating in lockstep. An example is the use of a vector type in the matrix multiplication $C = A \times B$ by specifying the rows of matrix B as `float8` vectors. In this way, 8 elements of the product matrix are calculated simultaneously within the pipelined inner loop (see listing 2). This results in 8 new values per cycle.

Listing 2: Vectored pipelines. Matrix B has data type float8. Loop j creates 8 pipelines each operating in SIMD fashion on 8 vector elements in loop k.

```
#define T float8
__kernel void __attribute__((task))
matmul (__global float * restrict A,
        __global T * restrict B, __global T * restrict C)
{
  for (int i = 0; i<N; i++)
    for (int j = 0; j<N; j += 8)
    { // 8 parallel pipelines
        T vsum = 0;
#pragma unroll 8
      // generate pipelined k-loop
        for (int k = 0; k<n; k++)
            vsum += A[i*N+k] * B[k*N/8 +j/8];
        C[i*N/8+j/8] = vsum;
    }
}
```

6. Advancing FPGA accelerator integration

FPGAs have a limited amount of on-chip storage, much less than GPUs. Therefore, the interaction between an FPGA accelerator and the CPU is more crucial than for GPUs. FPGA connects with a data stream either using a PCI-express bus, by accessing device data directly such as with video input or via an on-chip interconnect such as the AXI bus in the Zynq. A tighter integration is beneficial, but presents a number of other problems such as using a common cache, translating the virtual addresses of the CPU into addressable locations in the FPGA and sharing virtual memory between CPU and FPGA.

6.1. The HARP platform

In order to explore several design improvements, Intel has created a research platform consisting of a fast Xeon Broadwell processor and an Arria 10 FPGA, together with extra hardware for caching, and a high-speed CPU-FPGA interconnection [29]. In addition, the OpenCL language is supported for this platform [4]. The Heterogeneous Architecture Research Platform (HARP) introduces three innovations shown in figure 4. First, the local DDR RAM at the FPGA side is replaced by a transparently shared DDR memory at the CPU side. This includes IP cores translating physical FPGA addresses into virtual CPU addresses, a Quick Path Interconnect and two PCI-Express channels, providing a combined maximum bandwidth of 28 GB/s, managed by the channel steering logic. Second, the FPGA contains a fixed hardware cache of 64 KB as well as an application-specific cache, generated by the OpenCL compiler. Third, the OpenCL implementation includes coherent shared virtual memory (SVM) support between the FPGA and the CPU.

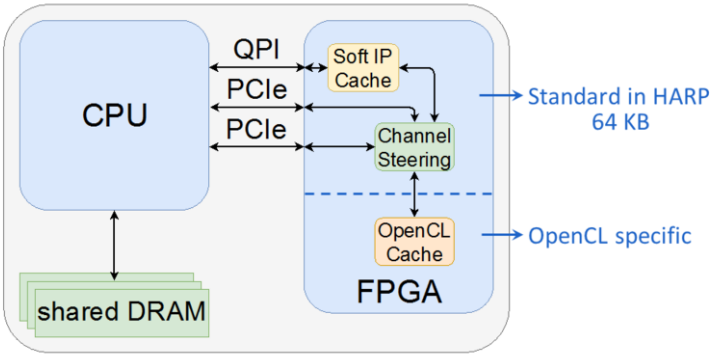


Figure 4. HARP architecture with coherent shared memory, fast interconnection and 2 local caches.

There are a number of questions that can be addressed: how efficient is the cache, how large is the bandwidth between the FPGA and the CPU and what is the benefit of the shared memory between the CPU and the FPGA. Regarding the bandwidth, we found that the real achieved bandwidth is between 15 and 16 GB/s, depending on the buffer size [15]. Regarding the cache, we have to make the distinction between the fixed 64 KB Soft IP cache of the FPGA Interface Unit (FIU) and a special OpenCL cache which is implemented by the compiler based on the data structures and usage in the OpenCL

program. In order to study the cache performance, we exploited the temporal locality by repeatedly reading a buffer with increasing length from the CPU. The specialized OpenCL cache was able to double the bandwidth with respect to the fixed cache in the Arria 10, see figure 5.

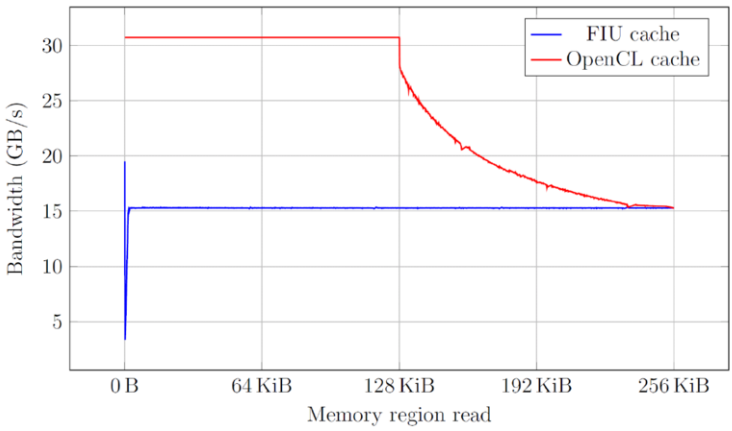


Figure 5. Cache efficiency: the OpenCL application-generated cache doubles the available bandwidth (red). The FIU cache was measured by disabling the OpenCL cache using the volatile keyword (blue).

Finally, in a traditional FPGA-CPU configuration, the CPU memory cannot be accessed by the FPGA, therefore OpenCL buffers are required to explicitly send and receive data, which is time-consuming. Furthermore, cache coherency is lost. In the HARP platform, a shared virtual memory (SVM) space has been implemented to transparently access data in the memory of the processor. As a result, data is sent to, or read from the FPGA on demand. This improves the communication efficiency and avoids extra buffer space in the FPGA.

6.2. Case study: a guided image filter

The impact of the innovations were tested using the "guided image filter" (GIF) image processing algorithm [18]. The guided image filter takes two images: a raw noisy input image I and an image with extra information G, for example LIDAR data, which is used to improve the noisy image. GIF is a convolution algorithm in which an output pixel O_i is obtained by averaging the input pixels I_j multiplied with weighted guidance pixels $W(G_j)$ in a window of radius r ($r = 1,3,5...$) centered around pixel i. The general form is

$$O_i = \sum_j W(G_j) I_j \tag{1}$$

The algorithm uses a sliding window buffer to receive streaming input data and reuses most of the data in consecutive pixel calculations, see figure 6.

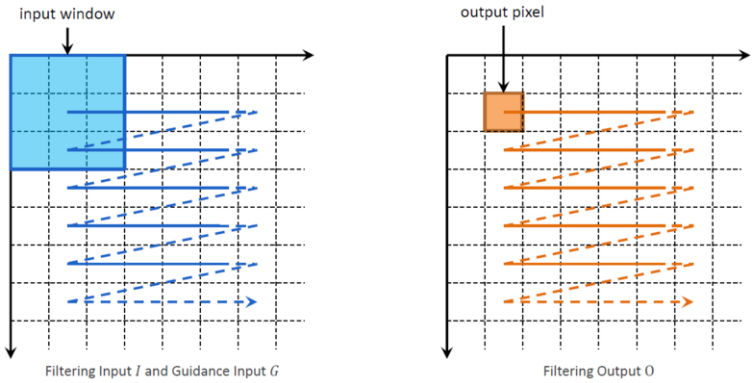


Figure 6. Sliding window with radius $r = 1$ in the guided filter algorithm. Windows I and G are used to calculate pixel O.

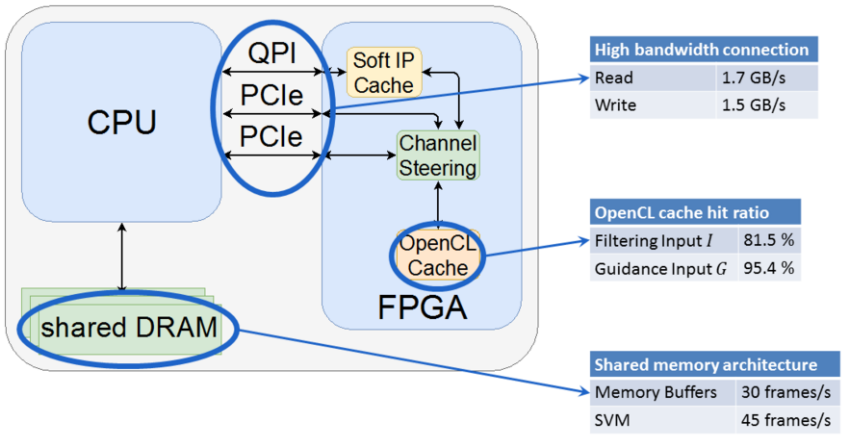


Figure 7. Measurements of the guided filter on the HARPv2 platform: bandwidth, OpenCL generated cache hit rate, impact of using shared virtual memory vs. copying data into local memory buffers.

6.3. Results

The key performance indicators (KPIs) of this example are the bandwidth, the cache efficiency and the impact of the shared memory. The KPIs obtained in [15] are summarized in figure 7. The algorithm has been tested on full HD color images (1920x1080x3 pixels) with radius $r = 3$. The pipelined and vectorized design is compute-bound and has a data throughput of 1.7 GB/s for reads and 1.5 GB/s for writes. The OpenCL cache hit rate is 81.5% and 95.4% for the input and guidance images respectively, showing the favorable impact of an algorithm-specific designed cache. The use of shared virtual memory instead of loading data into local buffers increases the frame rate from 30 to 45 frames per second. These results are obtained using floating-point operations and correspond to the maximum design fitting on the FPGA. As mentioned in [15], a larger radius generates more computations per frame. Using fixed-point instead of floating-point calculations, a radius $r = 6$ can be implemented, yielding 74 frames per second.

This example shows that OpenCL and the HARP platform go together well. The HARP platform offers a high bandwidth and a shared memory architecture. OpenCL, on the other hand, adds a design-specific generated cache and the use of shared virtual memory to the application developer.

7. Fallacies and pitfalls

The world of FPGAs is exciting, but also challenging, there are a lot of high expectations, but also misconceptions. We would like to address some of these particular items or issues here.

One kernel fits all sizes

It is reasonable to expect that one compute kernel can be used for any size of data. This is mostly not true because the hardware generated depends on the kernel arguments. E.g. the number of rows and columns of a matrix multiplication are used to reserve local buffers, since the FPGA has no dynamic memory. Smaller matrices are fine, but one loses the unused resources. Larger matrices require other techniques such as block matrix multiplication to get the job done [11]. An FPGA kernel is thus less flexible than CPU or GPU procedures.

Switching kernels in an FPGA or a GPU is equally fast

This is true only if there are enough resources to store multiple kernels simultaneously in the logic fabric. Otherwise, switching between multiple kernels implies a full or partial reconfiguration for each kernel, and this takes in the order of milliseconds or even seconds [30]. In a GPU, a kernel switch is as fast as a procedure call. However, running multiple kernels simultaneously by sharing processors is more complicated and involves merging several kernels [31].

OpenCL is a standardized language for FPGAs

While OpenCL is a standard, its usage, attributes, pragmas and programming model are not standardized across the FPGA vendors. This applies also to the compiler reports which are used to optimize the design. E.g. the initiation interval is tabulated for each loop in the Xilinx reports whereas it is specified for basic blocks in the Intel reports. The Xilinx software gives the total number of cycles as well as the expected frequency which allows to calculate the execution time of the kernel and even the number of GFlops straight from the compiler report. Intel shows the number of cycles for each node in the task graph and each basic block. It is not obvious to calculate the total number of cycles from the task graph of the program. See also the tutorial [20].

FPGA programming eats time

As is mentioned in section 3, compiling an HLS program typically requires in the order of several minutes, whereas implementing the hardware design into a bitstream can take many hours. Since HLS resource usage and cycle time reporting is quite accurate and not time-hungry, HLS design space exploration shows a much more favorable development cycle time than what is generally assumed.

An OpenCL program for a GPU is easily portable to an FPGA

The GPU architecture differs fundamentally from the FPGA, e.g. FPGAs have a small memory and no hardware thread switching. Hardware thread reordering, proposed in [25], doubles the resource usage due to arbitration logic and extra memory management. The GPU parallel programming style using the NDrange kernel consumes a lot of resources when used in the FPGA. OpenCL kernels for FPGAs are mostly single work items, which are then pipelined. Finally, FPGAs require a thorough design space exploration to optimize resource usage or to obtain a significant speedup. For these reasons, it is often better to redesign the algorithm from the ground up to orient the program to the characteristic features and optimizations of an FPGA [5].

8. Conclusion

Field programmable gate arrays have become a significant player in parallel computing systems. While the support of a common OpenCL language suggests a transparent use of GPUs and FPGAs accelerators, the application domain, algorithm selection and programming style is substantially different. The initial high expectations have made way for a more realistic view and a focus on better tools and design concepts, with good results. A number of key improvements to be expected are a tight integration of FPGAs in SoCs, enhanced compiler reporting, standardized pragmas as well as increased use of HLS languages for reconfigurable computing in software engineering curricula [6].

References

[1] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the Spring Joint Computer conference - AFIPS '67 (Spring)*, page 483, Atlantic City, New Jersey, April 1967. ACM Press.

[2] Arvind, Kathail, Vinod, and Pingali, Keshav. A Dataflow Architecture with Tagged Tokens. Technical report MIT-LCS-TM-174, Massachusetts Institute of Technology, September 1980.

[3] Tobias Becker, Oskar Mencer, and Georgi Gaydadjiev. Spatial Programming with OpenSPL. In Dirk Koch, Frank Hannig, and Daniel Ziener, editors, *FPGAs for Software Programmers*, pages 81–95. Springer International Publishing, Cham, 2016.

[4] Anthony M. Cabrera and Roger D. Chamberlain. Exploring Portability and Performance of OpenCL FPGA Kernels on Intel HARPv2. In *Proceedings of the International Workshop on OpenCL - IWOCL'19*, pages 1–10, Boston, MA, USA, 2019. ACM Press.

[5] Nicola Cadenelli, Zoran Jaksi, Jord Polo, and David Carrera. Considerations in using OpenCL on GPUs and FPGAs for throughput-oriented genomics workloads. *Future Generation Computer Systems*, 94:148–159, May 2019.

[6] Luca P. Carloni, Emilio G. Cota, Giuseppe Di Guglielmo, Davide Giri, Jihye Kwon, Paolo Mantovani, Luca Piccolboni, and Michele Petracca. Teaching Heterogeneous Computing with System-Level Design Methods. In *Proceedings of the Workshop on Computer Architecture Education - WCAE'19*, pages 1–8, Phoenix, AZ, USA, 2019. ACM Press.

[7] Tomasz S. Czajkowski, David Neto, Michael Kinsner, Utku Aydonat, Jason Wong, Dmitry Denisenko, Peter Yiannacouras, John Freeman, Deshanand P. Singh, and Stephen Dean Brown. OpenCL for FPGAs: Prototyping a Compiler. In *Proceedings of the 2012 International Conference on Engineering of Reconfigurable Systems & Algorithms*, pages 3–12. CSREA Press, July 2012.

[8] Bruno da Silva, An Braeken, Erik H. D'Hollander, and Abdellah Touhafi. Performance Modeling for FPGAs: Extending the Roofline Model with High-Level Synthesis Tools. *International Journal of Reconfigurable Computing*, 2013:1–10, 2013.

[9] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, October 1974.

[10] Jack B. Dennis and David P. Misunas. A preliminary architecture for a basic data-flow processor. In *ACM SIGARCH Computer Architecture News*, volume 3, pages 126–132. ACM, 1975.

[11] Erik H. D'Hollander. High-Level Synthesis Optimization for Blocked Floating-Point Matrix Multiplication. *ACM SIGARCH Computer Architecture News*, 44(4):74–79, January 2017.

[12] Gerald Estrin, Bertram Bussell, Rein Turn, and James Bibb. Parallel processing in a restructurable computer system. *IEEE Transactions on Electronic Computers*, 12(6):747–755, 1963.

[13] Gerald Estrin and C. R. Viswanathan. Organization of a "Fixed-Plus-Variable" Structure Computer for Computation of Eigenvalues and Eigenvectors of Real Symmetric Matrices. *Journal of the ACM (JACM)*, 9(1):41–60, 1962.

[14] European Commission. Computing technologies and engineering methods for cyber-physical systems of systems. RIA Research and Innovation action ICT-01-2019, Horizon 2020, October 2018.

[15] Thomas Faict, Erik H. D'Hollander, and Bart Goossens. Mapping a Guided Image Filter on the HARP Reconfigurable Architecture Using OpenCL. *Algorithms*, 12(8):149, July 2019.

[16] Xinyu Guo, Hong Wang, and Vijay Devabhaktuni. A Systolic Array-Based FPGA Parallel Architecture for the BLAST Algorithm. *ISRN Bioinformatics*, 2012:1–11, 2012.

[17] Hasitha Muthumala Waidyasooriya, Masanori Hariyama, and Kunio Uchiyama. *Design of FPGA-Based Computing Systems with OpenCL*. Springer International Publishing AG, 2018.

[18] Kaiming He, Jian Sun, and Xiaoou Tang. Guided Image Filtering. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(6):1397–1409, June 2013.

[19] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Communications of the ACM*, 62(2):48–60, January 2019.

[20] Kenter, Tobias. OpenCL design flows for Intel and Xilinx FPGAs - using common design patterns and dealing with vendor-specific differences. In *Sixth International Workshop on FPGAs for Software Programmers*, Barcelona, September 2019.

[21] Peter Kogge and John Shalf. Exascale Computing Trends: Adjusting to the" New Normal" for Computer Architecture. *Computing in Science & Engineering*, 15(6):16–26, 2013.

[22] Ruby Bei Loh Lee. Performance bounds in parallel processor organizations. In *High Speed Computer and Algorithm Organization*, pages 453–455. Academic Press, Inc, 1977.

[23] Ruby Bei-Loh Lee. *Performance characterisation of parallel processor organisations*. PhD thesis, Stanford University, Stanford, 1980.

[24] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2):39–55, March 2008.

[25] Amir Momeni, Hamed Tabkhi, Gunar Schirner, and David Kaeli. Hardware thread reordering to boost OpenCL throughput on FPGAs. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 257–264, Scottsdale, AZ, USA, October 2016. IEEE.

[26] G.E. Moore. Cramming More Components Onto Integrated Circuits. *Proceedings of the IEEE*, 86(1):82–85, January 1998.

[27] A. Podobas, H. R. Zohouri, N. Maruyama, and S. Matsuoka. Evaluating high-level design strategies on FPGAs for high-performance computing. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, 2017.

[28] Moritz Schmid, Christian Schmitt, Frank Hannig, Gorker Alp Malazgirt, Nehir Sonmez, Arda Yurdakul, and Adrian Cristal. Big Data and HPC Acceleration with Vivado HLS. In Dirk Koch, Frank Hannig, and Daniel Ziener, editors, *FPGAs for Software Programmers*, pages 115–136. Springer International Publishing, Cham, 2016.

[29] Greg Stitt, Abhay Gupta, Madison N. Emas, David Wilson, and Austin Baylis. Scalable Window Generation for the Intel Broadwell+Arria 10 and High-Bandwidth FPGA Systems. In *FPGA '18*, pages 173–182, Monterey, CA, 2018. ACM Press.

[30] Kizheppatt Vipin and Suhaib A Fahmy. FPGA Dynamic and Partial Reconfiguration: A Survey of Architectures, Methods, and Applications. *ACM Computing Surveys*, 1(1):39, January 2017.

[31] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo. Simultaneous Multikernel GPU: Multi-tasking throughput processors via fine-grained sharing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 358–369, March 2016.

This page intentionally left blank

Main Track

Deep Learning Applications

This page intentionally left blank

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200022

First Experiences on Applying Deep Learning Techniques to Prostate Cancer Detection

Eduardo José GÓMEZ-HERNÁNDEZ [a,1] and José Manuel GARCÍA [a]
[a] *Computer Engineering Department, University of Murcia, Murcia, Spain*

Abstract. Nowadays, machine learning techniques based on deep neural networks are everywhere, from image classification and recognition or language translation to autonomous driving or stock market prediction. One of the most prominent fields of application is medicine, where AI techniques promote and promise the personalized medicine. In this work, we entered in this field to study the prostate cancer prediction from images digitalized from hematoxylin and eosin stained biopsies. We chose this illness since prostate cancer is a very common type of cancer and the second cause of death in men. We did this work in collaboration with Hospital Reina Sofia of Murcia. As newcomers, we faced a lot of problems to start with, and questioned ourselves about many issues. This paper shows our experiences in developing and training two convolutional neural networks from scratch, exposing the importance of both the preprocessing steps (cropping raw images to tiles, labeling, and filtering), and the postprocessing steps (i.e., to obtain results understandable for doctors). Therefore, the paper describes lessons learned in building CNN models for prostate cancer detection from biopsy slides.

Keywords. Deep learning, Prostate cancer detection, Neural networks

1. Introduction

Deep learning has become more and more ubiquitous in everybody's day life. Specifically, deep neural networks have been incorporated into numerous fields, such as image classification, language processing, economics, video games, and medicine. In some tasks, this new technology is being able to outperform human performance.

Progress in hardware technologies and cost reduction have caused new approaches in deep neural networks, outperforming older machine learning techniques. Nowadays it is possible to afford more and more complex problems, then it is important to have some practice and experience on it.

One of the most prominent fields of application of deep learning techniques is in medicine. Initially, applications of deep learning in medicine were limited to radiology images [1], but later (since end of 2016) it began to apply for other kinds of images [2–5].

Medical image analysis has started to implement deep learning for screening and localization of malignant zones. Additionally, other medical areas are working with these

[1]Corresponding Author: Computer Engineering Department, University of Murcia, 30.100 - Murcia (Spain); E-mail: eduardojose.gomez@um.es.

kind of techniques as well, like the analysis of the genetic information inside DNA and RNA series [6]. The common objective is not replace physicians with deep learning techniques, but supporting them to make better diagnoses.

Although our research group is focused on High-Performance Computing (HPC) and its applications, some years ago we got attracted by using our knowledge on HPC techniques to improve the precision and execution time of deep learning workloads from a real medical case. Then, we joined Prof. Enrique Poblet-Martínez and his research team from the Hospital Reina Sofía of Murcia to work in the field of "Detection of Prostate Cancer by biopsy slides". The final objective of our new research line is to create a model able to recognize tumorous zones in biopsy slides as a preliminary screening, hence allowing doctors to focus on the tumorous cases.

This paper presents our first experiences in this field, showing the way we took to learn about Deep Neural Networks (DNNs) tackling a real problem from the medicine field. We started by choosing the MXNet framework as our platform where our codes have been run. The first lesson we learned was the major role that the preprocessing steps play to observe a good behaviour of the CNN network . Next, we realized about modifying the hyper-parameters of the network to tune its behaviour and further improve its "accuary". Finally, we discovered the importance of the postprocessing steps to present the neural network's output in a format understandable by pathologists. In this first attempt, we achieved an AUC satistic metric of 82% in discriminating healthy from cancerous images using Inception V3.

The rest of the paper is organized as follows: Section II introduces the main concepts managed throughout this paper, related to deep learning frameworks, accelerators, and prostate cancer. Section III reports the machine and configuration used. The methodology is described in Section IV. Section V contains the obtained experimental results. Finally, Section VI exposes our conclusions and give some hint for future work.

2. Background

2.1. Machine Learning & Deep Learning

Theoretical and mathematical models of the artificial intelligence techniques were developed in the twentieth century. One of these models are ANNs (Artificial Neural Networks), a type of brain-inspired learning algorithm, built from small units called neurons. The most classical one, MLP (MultiLayer Perceptron) network, With enough layers, and enough perceptrons per layer, is able to represent any mathematical function [7]. However, when the amount of data grows, networks built exclusively from perceptrons can be very inefficient. Therefore, new types of neural networks should be made, being CNNs (Convolutional Neural Networks) the most known ones. The most distinguished layers in these networks are convolution and pooling, taking input data structured as channels of two dimensions.

Once the network is defined, with more or fewer layers, there are two different phases: inference, and training. In the inference phase, a set of inputs are presented to the network, and a set of outputs are given by the network, like any mathematical function. But training phase is more complicated, using an algorithm, it starts to teach the network to do something useful.

Before starting training, a initialization step is needed to set the different parameters of the network. This step might appear trivial or optional, but a bad starting point may make the network never be able to learn. Also, it is possible to bring this data from another neural network model, called Transfer Learning [8].

SGD (Stochastic Gradient Descendent) is the most known training algorithm for neural networks, but it is not the only one, there others like Adam [9] and DCASGD [10] among others. SGD is a variant of GD (Gradient Descendent) but used with batches. A batch is a group of input data of fixed size.

Then, the iteration process is as follow: First the Feed-Forward step, where data is presented to the network in batches, storing the result for later use. Then, the Back-Propagation step that compares all the results from the previous step with ground truth, and propagates backward on the network to calculate the gradient estimate. Finally, with the gradient estimation, all weights and biases are updated in the Update step.

There are some metrics to observe the precision of the neural network, being the most common accuracy, mse, macc, and cross-entropy. In classification, the most used is accuracy, giving the percentage of correctly predicted cases over the total. In classification, AUC statistic metric has started to be used in neural networks for medicine. AUC is the Area Under the Curve, to be specific, under the ROC curve. A ROC curve is a Receiver Operating Characteristic Curve [11], and it is commonly used to know how good is a binary classifier.

2.2. Frameworks & HPC

Machine learning techniques could be difficult to code and debug, therefore many frameworks have been developed to ease its use. Most of them are open source with software for most of the types of neural networks. The most known ones are Caffe, Caffe2, Tensorflow, Theano, PyTorch, Mxnet, and CNTK among others [12]. And there are frameworks like Keras, providing a more high-level experience, running on top of some of the aforementioned frameworks.

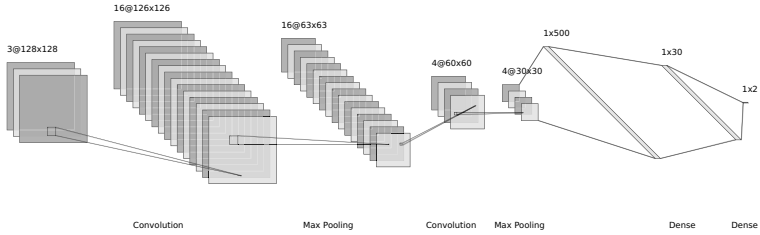
Specifically, the training phase is very time-consuming, since it is evaluating an optimization problem with hundreds, thousands, or even millions of parameters. Therefore, the reduction of the training phase execution time is a desirable feature for all frameworks. Thanks to this shorter training time, scientists using theses frameworks can explore a wide solution space, and even develop more complex networks.

The rise of High-Performance Computing (HPC) applied not only to grand challenge problems but also to common problems has revolutionized the machine learning field. All major vendors offer products which can be used for deep learning, as GPUs from Nvidia and AMD or scalable Xeons from Intel. Also, proprietary designs have emerged using ASICs, FPGAs or systolic arrays. Maybe the most well known is the development of the TPUs from Google [13].

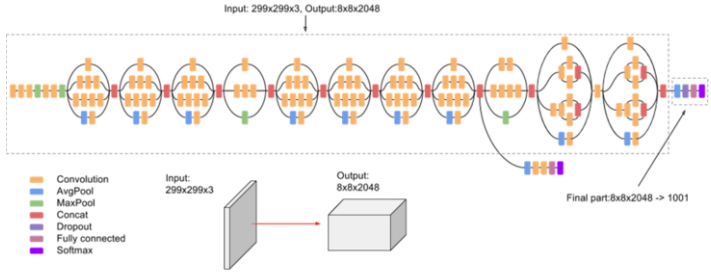
2.3. Prostate Cancer

Prostate cancer is the most common cancer and the second leading cause of death in men [14]. Nowadays, pathologists have a large number of slides to diagnose, making diagnosis very long. Reduce this diagnosis time would help to focus on the needed cases.

Prostate biopsies are hematoxylin and eosin stained (H&E) and normally stored inside a crystal. These biopsies need to be transformed to digital images to be used by



(a)



(b)

Figure 1. a: **Custom Neural Network** inspired on LeNet. **b**: **Inception V3** https://cloud.google.com/tpu/docs/inception-v3-advanced.

deep learning techniques. To do that, there are systems able to scan biopsies into a high-resolution image, called WSI (whole slide images). These WSI allow the application of image analysis techniques to prostate biopsies.

Using these WSIs, there are some approaches developing deep learning models to detect prostate cancer in biopsy slides [15–17]. Some of them try to find the tumorous zones and classify them using the Gleason's pattern.

3. Our Experience

3.1. Settings

In this study, we have used the MXNet 1.3.0 framework running with Cuda 9.2, and cuDNN 7.4.1. Statistical data was obtained with scikit-learn 0.20.2. Our compute machine is running CentOS Linux 7.5.1804 with Linux 3.10.0-862.14.4, powered by two Intel(R) Xeon(R) CPU E5-2603 v3 @ 1.60GHz with 64 GiB RAM memory, and a Geforce GTX 1080 8GB GDDR5X. For storage, we have a 500GB Samsung SSD 850. Finally, the scanner used to digitalize biopsies was iScan Coreo Ventana, able to produce BIF, TIFF and JPEG2000 image formats, from 1x to 40x magnification.

Two neural networks architectures have been used. The first one is a basic CNN (Figure 1) based on LeNet [18] and previously used in a medical environment [2, 19]. The second one is Inception v3, a very common network used for image classification (Figure 1). To clarify these figures, next we detail the different layers from the LeNet-based network. At the beginning is the input image (3 channels of 128x128 pixels); then

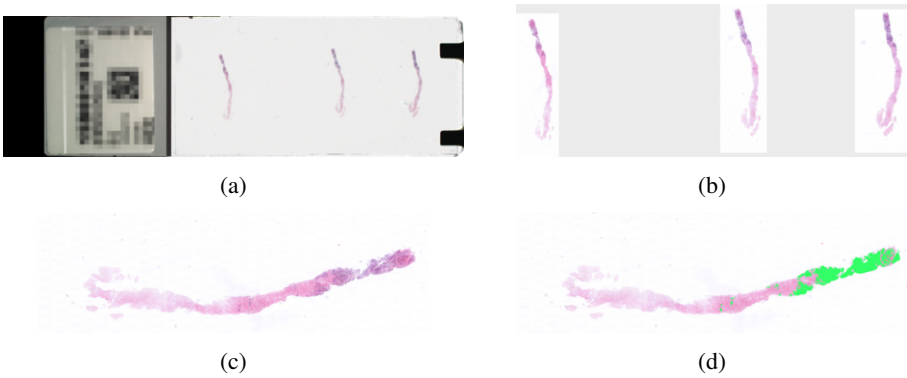


Figure 2. An example of a biopsy. a: a crystal with 3 slabs of a biopsy. **b**: scanned biopsy with all slabs. **c**: a slab extracted from the scanned image. **d**: a labeled slab of a biopsy, green zones denote tumors.

a convolution of 16 filters with a kernel size of 3x3; its size is reduced by a 2x2 max pooling layer; then, another convolution is applied, but with a kernel size of 5x5 and 4 filters, followed again with another 2x2 max pooling; in the end, we have an MLP with sizes 500, 30, and 2, each one with sigmoid activation; and a SoftMax layer at the end.

3.2. Preprocessing Database

As mentioned before, WSI images have a high resolution scanned image. Our selected framework, MXNet, cannot use this multiple format. Then, we chose the TIFF format to convert it later to JPG. These TIFFs have multiple layers, the first one is an image of the biopsy (Figure 2), the next one is a thumbnail, and the consecutive ones are 20x, 10x, 5x, 2.5x, 1.25x, 0.625x, 0.3125x, 0.15625x, 0.078125x magnifications.

The start point was to set a magnification value for the images. Pathologists usually select 20x magnification to find tumors, therefore we extract this specific image from the multilayer TIFF image (Figure 2), resulting in a size of $200 \sim 500$ megapixels. Also, as all slabs from the biopsy are very similar, doctors decided which one will be used .

Next, we ask pathologists for labeling tumorous zones in the biopsy slide, distinguishing affected biopsies from healthy ones, and locating the affected areas (like Figure 2).

However, even using only one slab from the biopsy, the image is too big to be used as such. To cope with this problem, we followed the approach of cropping the image in many rectangular tiles, treating the image as many tiles on a wall. We did this inspired on previous works in the medicine field [3]. Making this, we could process each tile independently from others, solving the size problem. As a downside, we missed some relevant information such as the relative position of the file in the image and the surrounding area.

The slide's background had much noise. Our first approach was to try to relax the white's definition. Then, we considered that every pixel with each RGB component above value 215 is white, instead of 255. After that, it was necessary to define how many white pixels should have the tile to be marked as white. We studied our database to find a good percentage (Figure 3). We selected 3 values (95%, 50%, and 20%), and test if there was any difference. As it can be seen in the Figure 3, 95% was the best value.

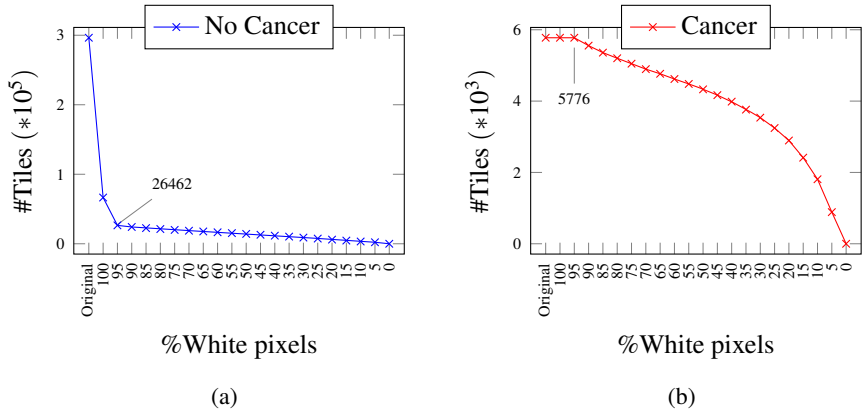


Figure 3. Number of tiles left. **a**: amount of no cancer tiles in our database after removing tiles with x% of white pixels. **b**: amount of cancer tiles in our database after removing tiles with x% of white pixels.

The next question we faced was the following: How should be the proportion of cancerous tiles against healthy tiles? To answer this, we prepared 3 datasets with different proportions: 30% cancer - 70% no cancer, 50% cancer - 50% no cancer, and 70% cancer - 30% no cancer. We found that, in our case, giving enough epochs, all datasets got approximately the same precision. Another problem when 50% - 50% is not used is the class imbalance which can force the network to favour classes most common in the database, sometimes to the limit of outputting always that class.

As images are between 0 and 255 in all of their components, we finished the preprocessing step adding a normalization stage, with the objective of mapping all values between 0 and 1. In this case, this stage divides all the components by 255.

When working with small datasets, a common practice is data augmentation. There are a lot of data augmentation techniques, from rotating and flipping to brightness and contrast changes. We started using random rotations and flips, achieving a noticeable improvement in the accuracy of the network.

3.3. Neural Network

As mentioned, on other studies, there are plenty of different neural networks, such as ResNet [20], Inception, Alexnet, VCG, LeNet, etc. In this case, we decided to test two networks (Figure 1). The first one is a small convolution neural network inspired in LeNet. And the second one is Inception V3, very used in medical image analysis. Initially, we thought that the smaller network would be faster and more accurate because it could specialize more than larger one.

Transfer learning is very popular today, especially when the amount of available data is very low. This technique allows to use a good starting point and requiring less epoch to learn the problem. However, in this work, we wanted to start from the beginning. As we do not used it, all weights were randomly initialized.

3.4. Postprocessing

From the last layer of the network (SoftMax), we got two probabilities, the first one was the probability of being a healthy tile, and the second was the probability of being a

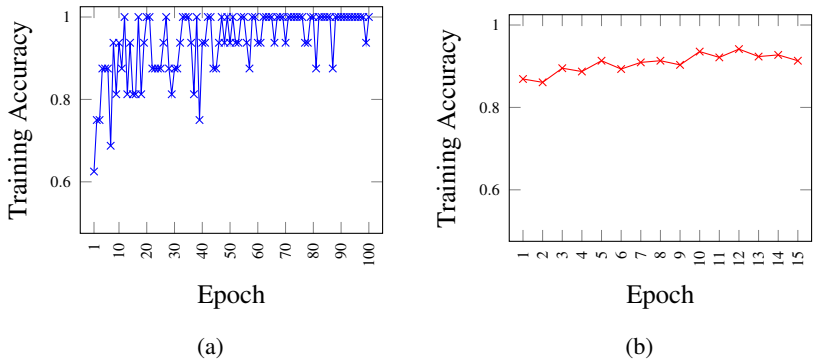


Figure 4. Training accuracy per epoch. **a:** LeNet based network with 100 epochs. **b:** Inception V3 network with 15 epochs.

tumorous tile. Then, we took a threshold to determine when an image is cancerous and when it was not, in this case we used 50% as the threshold. Later, we started to use alpha blending to get a heatmap.

Then, the output gives the probability for each tile of being healthy or not. However, this is not enough. In the clinical environment, doctors want to see the output in a more visual manner. We proposed two ways to approach this: masking and recoloring. Masking implies making an image to be superimposed to the original, allowing to see both images at the same time. Recoloring is similar to the mask but applied directly to the image. Both methods were very similar but imply different results. For this study, we chose recolor, because it allowed us to have only one image at the output and not carrying both.

4. Results & Lessons Learned

In the training phase, we run our two neural network models with random rotations and flips, a learning rate of 0.001, and 0.9 as momentum. All values initialized with Xavier average at 1. The LeNet based network was run for a total of 100 epochs with 8 images per batch. And Inception V3 for 15 epochs with 16 images per batch.

During this work, obtaining data was a very complex task, and we ended using 21 prostate biopsies (391,174 tiles). From that, 17 was for training (302,186 tiles), and after cleaning and data normalization, we ended with 11,552 tiles. These numbers could seem quite large, but they are from 17 biopsies. Therefore, in Figure 4, we can observe that LeNet network is overfitted by the small input data.

In the testing phase, we used 4 biopsies (88,988 tiles), and after cleaning and data normalization, we ended with 7,954 tiles. We achieved an AUC of 82% using Inception V3, and an AUC of 65% in our small network. We show an example tissue, the ground truth by pathologists, and the result obtained from the network reconstructed (Figure 5).

Regarding to Table 1 and Figure 5, our small network was able to address a little about the recognition of tumorous tiles in a prostate biopsy in a reasonable time. Also the inference of a new biopsy is really fast. On the other hand, Inception V3 takes the double of time making only 15% of epochs, but achieving better results. The main problem was the shortage of the data, because our data did not gather all kind of tumorous biopsies.

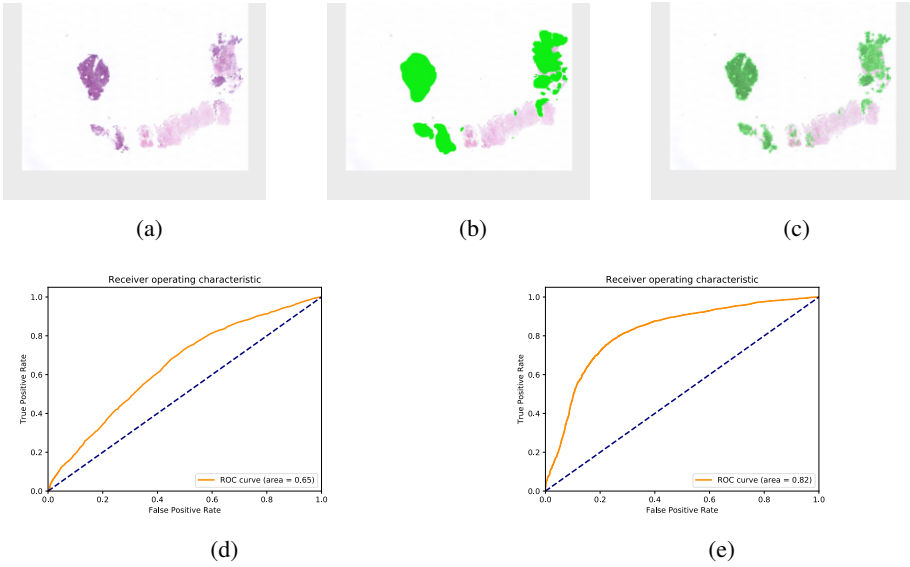


Figure 5. Obtained Results: **a**: Example Tissue, **b**: Ground Truth, **c**: Output Image. ROC Curves with AUC value (**d**: for own LeNet based network, **e**: Inception V3).

Task	Time
Initial Preprocessing	15 mins
Data Base	1 min
Data sieve	3 hours
Postprocessing	18 mins

(a)

Phases	Networks	
	LeNet	V3
Training	1 hour	2 hours
Inference	2 min	6 min

(b)

Table 1. Execution Times: a: Times for common processing tasks in both networks. **b:** Time for each network phase.

From all experiences we collected throughout this experiment, we would like to show the main lessons we have learned:

- **The importance of preprocessing:** We started using the raw image and quickly we found that image dimensions and image size were a problem. With the tiling we realized that the network was not able to learn anything, excluding distinguish background from the tissue. Only after making at least one preprocessing technique, we started to get some acceptable results.
- **Quantity and quality of the data:** Neural networks need a lot of data, not all fields of study have data available, and can take too much time to generate them. But letting this aside, the data have to be good data. We need a good sample from all possible values to get good results.
- **Neural network architecture:** There are a lot of pre-made neural networks ready to be trained. We started with a basic convolutional neural network, although soon we tested a more complex neural network. The complex configuration obtained better results, although this made the training phase slower and more memory bound limited.

- **Postprocessing may be critical in some cases:** In real-world applications, the output needs to be understandable. Moreover, in the medical field is mandatory to visualize the output so doctors can check the works done by the deep learning algorithms. Therefore, in this sector is important both to obtain a good accuracy and to properly show the output in a clear and user-friendly way.
- **HPC requirements:** The training phase is very expensive in computational power, due to tthe many calculations made to crunch the big data used. It is really easy to have memory boundary problems when working with neural networks. Machines with a reasonable amount of memory and high performance computing help to reduce this phase from months to only some hours.

5. Conclusions & Future Work

In this paper, we have exposed the common problems found when developing a neural network for the first time for a medical case. Starting from the raw data, it is a challenging problem the selection of which kind of data choose and how organize it. Also, there are many parameters to be tuned to start learning patterns from the input. Besides, output format can be relevant and may incur a complex step.

As developing a neural network is a very complex task, we have attempted with this work to show our errors and problems on it to help newcomers. We have concluded that preprocessing and the quantity and quality of the data are very important when looking for good accuracy. Also, our small neural network was outperformed by Inception V3, showing us that the prostate cancer detection could be very complex for our simple network.

Further research could be conducted in various directions. The problem exposed to the neural network may reach a better accuracy obtaining more data and more precise labeling. Also, a more refined neural network and preprocessing steps could help. Additionally, preprocessing and postprocessing techniques could be improved to take less time and get near instant results.

Acknowledgments

We would like to thank Prof. Enrique Poblet-Martinez, Dr. Eduardo Alcaraz-Mateos, and Dr. Francisco Garcia-Molina for obtaining and labeling data, and answering all our questions about this clinical case. Also, we thank Andrés García Meroño for several ideas in image processing. And the rest of laboratory workmates, Francisco Muñoz Martínez, and David Corbalán Navarro for their fruitful discussions. This work was partially funded by the AEI (State Research Agency, Spain) and the ERDF (European Regional Development Fund, EU), under the Contract RTI2018-098156-B-C53.

The data used is not publicly available, but all code and scripts used in this work are available at: https://gitlab.com/OdnetninI/prostate-cancer-detection-using-mxnet-framework.

References

[1] Maciej A. Mazurowski, Mateusz Buda, Ashirbani Saha, and Mustafa R. Bashir. Deep learning in radiology: an overview of the concepts and a survey of the state of the art. *CoRR*, abs/1802.08717, 2018.

[2] Hideharu Ohsugi, Hitoshi Tabuchi, Hiroki Enno, and Naofumi Ishitobi. Accuracy of deep learning, a machine-learning technology, using ultra–wide-field fundus ophthalmoscopy for detecting rhegmatogenous retinal detachment. *Scientific Reports*, 7:9425, 2017.

[3] Angel Cruz-Roa, Hannah Gilmore, Ajay Basavanhally, Michael Feldman, Shridar Ganesan, Natalie N.C. Shih, John Tomaszewski, Fabio A. González, and Anant Madabhushi. Accurate and reproducible invasive breast cancer detection in whole-slide images: A deep learning approach for quantifying tumor extent. *Scientific Reports*, 7(1):46450, 2017.

[4] Nicolas Coudray, Paolo Santiago Ocampo, Theodore Sakellaropoulos, Navneet Narula, Matija Snuderl, David Fenyö, Andre L. Moreira, Narges Razavian, and Aristotelis Tsirigos. Classification and mutation prediction from non–small cell lung cancer histopathology images using deep learning. *Nature Medicine*, 24:1559–1567, 2018.

[5] Andre Esteva, Brett Kuprel, Roberto A. Novoa, Justin Ko, Susan M. Swetter, Helen M. Blau, and Sebastian Thrun. Dermatologist-level classification of skin cancer with deep neural networks. *Nature*, 542(7639):115–118, January 2017.

[6] Christof Angermueller, Tanel Pärnamaa, Leopold Parts, and Oliver Stegle. Deep learning for computational biology. *Molecular Systems Biology*, 12(7):878, 2016.

[7] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251 – 257, 1991.

[8] Vivienne Sze, Yu Hsin Chen, Tien Ju Yang, and Joel S. Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017.

[9] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[10] Shuxin Zheng, Qi Meng, Taifeng Wang, Wei Chen, Nenghai Yu, Zhiming Ma, and Tie-Yan Liu. Asynchronous stochastic gradient descent with delay compensation for distributed deep learning. *CoRR*, abs/1609.08326, 2016.

[11] Kelly H Zou, A James O'Malley, and Laura Mauri. Receiver-operating characteristic analysis for evaluating diagnostic tests and predictive models. *Circulation*, 115(5):654–657, Feb 2007.

[12] W. G. Hatcher and W. Yu. A survey of deep learning: Platforms, applications and emerging research trends. *IEEE Access*, 6:24411–24432, 2018.

[13] Norman P. Jouppi, Cliff Young, and Nishant et al Patil. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, 2017. ACM.

[14] Simon Rodney, Taimur Tariq Shah, Hitendra RH Patel, and Manit Arya. Key papers in prostate cancer. *Expert Review of Anticancer Therapy*, 14(11):1379–1384, 2014.

[15] Geert Litjens, Clara I. Sánchez, Nadya Timofeeva, Meyke Hermsen, Iris Nagtegaal, Iringo Kovacs, Christina Hulsbergen - van de Kaa, Peter Bult, Bram van Ginneken, and Jeroen van der Laak. Deep learning as a tool for increased accuracy and efficiency of histopathological diagnosis. *Scientific Reports*, 6(1):26286, 2016.

[16] Kunal Nagpal, Davis Foote, and Yun Liu et al. Development and validation of a deep learning algorithm for improving gleason scoring of prostate cancer. *CoRR*, abs/1811.06497, 2018.

[17] Eirini Arvaniti, Kim S. Fricker, Michael Moret, Niels J. Rupp, Thomas Hermanns, Christian Fankhauser, Norbert Wey, Peter J. Wild, Jan H. Rueschoff, and Manfred Claassen. Automated gleason grading of prostate cancer tissue microarrays via deep learning. *bioRxiv*, 2018.

[18] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.

[19] S. Sarraf and G. Tofighi. Deep learning-based pipeline to recognize alzheimer's disease using fmri data. In *2016 Future Technologies Conference (FTC)*, pages 816–820, Dec 2016.

[20] Songtao Guo and Zhouwang Yang. Multi-channel-resnet: An integration framework towards skin lesion analysis. *Informatics in Medicine Unlocked*, 12:67–74, 2018.

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200023

Deep Generative Model Driven Protein Folding Simulations

Heng MA [a] Debsindhu BHOWMIK [a] Hyungro LEE [b] Matteo TURILLI [b]
Michael YOUNG [a] Shantenu JHA [b,c] Arvind RAMANATHAN [d]
[a] *CSED, Oak Ridge National Laboratory, Oak Ridge, TN 37830*
[b] *RADICAL, ECE, Rutgers University, Piscataway, NJ 08854, USA*
[c] *Brookhaven National Laboratory, Upton, New York, 11973*
[d] *Data Science and Learning, Argonne National Laboratory, Lemont, IL 60439*

Abstract. Significant progress in computer hardware and software have enabled molecular dynamics (MD) simulations to model complex biological phenomena such as protein folding. However, enabling MD simulations to access biologically relevant timescales (e.g., beyond milliseconds) still remains challenging. These limitations include (1) quantifying which set of states have already been (sufficiently) sampled in an ensemble of MD runs, and (2) identifying novel states from which simulations can be initiated to sample rare events (e.g., sampling folding events). With the recent success of deep learning and artificial intelligence techniques in analyzing large datasets, we posit that these techniques can also be used to adaptively guide MD simulations to model such complex biological phenomena. Leveraging our recently developed unsupervised deep learning technique to cluster protein folding trajectories into partially folded intermediates, we build an iterative workflow that enables our generative model to be coupled with all-atom MD simulations to fold small protein systems on emerging high performance computing platforms. We demonstrate our approach in folding Fs-peptide and the β-β-α (BBA) fold, FSD-EY. Our adaptive workflow enables us to achieve an overall root-mean squared deviation (RMSD) to the native state of 1.6 $\AA$ and 4.4 $\AA$ respectively for Fs-peptide and FSD-EY. We also highlight some emerging challenges in the context of designing scalable workflows when data intensive deep learning techniques are coupled to compute intensive MD simulations.

Keywords. Deep learning, Workflows, Molecular dynamics, Protein folding

1. Introduction

Multiscale molecular simulations are widely used to model complex biological phenomena, such as protein folding, protein-ligand (e.g., small molecule, ligand/ drug, protein) interactions, and self-assembly [1,2]. However, much of these phenomena occur at timescales that are fundamentally challenging for molecular simulations to access, even with advances in both hardware and software technologies [3]. Hence, there is a need to develop scalable, adaptive simulation strategies that can enable sampling of timescales relevant to these biological phenomena.

Many adaptive sampling techniques [4,5,6,7,8,9] have been proposed. All these techniques share some similar characteristics, including (a) the need for efficient and automated approaches to identify a small number of relevant conformational coordinates

(either through clustering and/or dimensionality reduction techniques) [10,11,12], and (b) the identification of the 'next' set of simulations to run such that more trajectories are successful in attaining a specific end goal (e.g., protein that is well folded, protein bound to its target ligand, etc.) [8,9].

These adaptive simulations present methodological and infrastructral challenges. Ref. [4] provides important validation of the power of adaptive methods over traditional "vanilla" molecular dynamics (MD) simulations or "ensemble" simulations. Ref. [13] highlights challenges of such workflows on high-performance computing platforms.

We recently developed a deep learning based approach that uses convolutions and a variational autoencoder (CVAE) to cluster simulations in an unsupervised manner [14]. We have shown that our CVAE can discover and identify intermediate states from protein folding pathways; further, the CVAE-learned latent dimensions cluster conformations into biophysically relevant features such as number of native contacts, or root mean squared deviation (RMSD) to the native state.

We posit that the CVAE learned latent features can be used to drive adaptive sampling within MD simulations, where the next set of simulations to run are decided based on a measure of 'novelty' of the simulation/ trajectory frame observed.

Integrating CVAE concurrently with large-scale ensemble simulations on high-peformance computing platforms entails the aforementioned complexity of adaptive workflows [13], while introducing additional infrastructural challenges. These arise from the concurrent and adaptive execution of heterogeneous simulations and learning workloads requiring sophisticated workload and performance balancing, inter alia.

In this paper, we implement a baseline version of our deep learning driven adaptive sampling workflow with multiple concurrent instances of MD simulations and CVAEs. Our contributions can be summarized as follows:

- We demonstrate that deep learning based approaches can be used to drive adaptive MD simulations at scale. We demonstrate our approach in folding small proteins, namely Fs-peptide and the β-β-α-fold (BBA) protein and show that it is possible to fold them using deep learning driven adaptive sampling strategy.
- We highlight parallel computing challenges arising from the unique characteristics of the worklfow, viz., training of deep learning algorithms can take almost as much time as running simulations, necessitating novel developments to deal with heterogeneous task placement, resource management and scheduling.

Taken together, our approach demonstrates the feasibility of coupling deep learning (DL) and artificial intelligence (AI) workflows with conventional all-atom MD simulations.

2. Related Work

Adaptive sampling techniques have been widely developed for MD simulations. A thorough review of this area of research is beyond the scope of this paper – however, we refer the interested reader to [15] for more details. From a computational point of view, adaptive sampling techniques require the use of specialized middle-ware that allows for scheduling, managing and orchestrating hundreds (if not thousands) of loosely coupled simulations that are guided by statistical approaches [16,17,18].

More recently, the development of machine learning techniques such as Markov State Models (MSM) and its integration with adaptive sampling techniques have proven quite useful for selecting the optimal number of stating points (for simulations) while simultaneously improving the convergence in these simulations [19]. When combined with sampling techniques such as umbrella sampling, MSM-based estimators can provide further insights into complex biological processes [20]. With advances in machine learning techniques, especially deep learning methods [21,14,22,8], we examined whether using such generative models could be advantageous in the context of accelerating protein folding simulations.

3. Methods

3.1. Workflow description

Two key components of the workflow include the MD simulation module and the deep-learning based CVAE module, which are described below.

Molecular dynamics (MD) simulations: The MD simulations are performed on GPUs with OpenMM 7.3.0 [23]. Both the Fs-peptide and BBA systems were modeled using the Amberff99SB-ildn force field [24] in implicit Onufriev-Bashford-Case GBSA solvent model [25]. The non-bonded interactions are cut off at 10.0 $\mathring{A}$ and no periodic boundary condition was applied. All the bonds to hydrogen are fixed to their equilibrium value and simulations were run using a 2 fs time step. Langevin integrator was used to maintain the system temperature at 300 K with a friction coefficient at 91 ps^{-1}. The initial configuration was optimized using L-BFGS local energy minimizer with tolerance of 10 kJ/-mol and maximum of 100 iterations. The initial velocity is assigned to each atom from a Boltzmann distribution at 300 K. We also added a new reporter to calculate the contact matrix of C_α atoms in the protein (using a distance cut-off of 8 $\mathring{A}$ in hdf5 format using the MDAnalysis module [26,27] that could be used as inputs to the deep learning module (described below). Each simulation run outputs a frame every 50 ps.

Convolutional Variational Autoencoder (CVAE): Autoencoder is a deep neural network architecture that can represent high dimensional data in a low dimensional latent space while retaining the key information [28]. With its unique hourglass shaped architecture, an autoencoder compresses input data into a latent space with reduced dimension and reconstructs it to the original data. We use the CVAE to cluster the conformations from our simulations in an unsupervised manner [14,29]. Currently in our workflow, we use the number of latent dimensions as a hyperparameter (varying between $\{3,\ldots,6\}$) and use the CVAE that most accurately reconstructs the input contact maps [14,29]. CVAE was implemented using Keras/TensorFlow and trained on a V100 GPU for 100 epochs.

Assembling our workflow: As illustrated in Figure 1, our prototype workflow couples the two components. In the first stage, the objective is to initially train the CVAE to determine the optimal number of latent dimensions required to faithfully reconstruct the simulation data. We commence our runs as an ensemble of equilibrium MD simulations. Ensemble MD simulations are known to enable better sampling of the conformational landscape, and also can be run in an embarrassingly parallel manner. The simulation data is converted into a contact map representation (to overcome issues with rotation/translation within the simulation box) and are streamed at regular intervals into the CVAE module. The output from the first stage is an optimally learned latent representation of the simu-

lation data, which organizes the landscape into clusters consisting of conformations with similar biophysical features (e.g., RMSD to the native state). Note that this is an emergent property of the clustering and the RMSD to the native state is not used as part of training data.

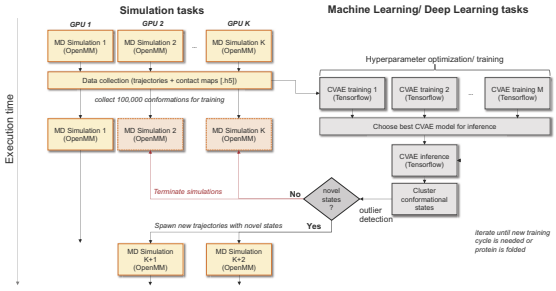


Figure 1.: Deep generative model driven protein folding simulation workflow.

In the second stage, our objective is to identify the most viable/ promising next set of starting states for propagating our MD simulations towards the folded state. We switch the use of CVAE to infer from newly generated contact maps (from simulations) and observe how they are clustered. Based on their similarity to the native state (measured by the RMSD), a subset of these conformations are selected for propagating additional MD runs. The workflow is continued until the protein is folded (i.e., conformations reach a user-defined RMSD value to the native state).

3.2. Implementation, Software and Compute Platform

We used the Celery software to implement the aforementioned workflow. Celery is an asynchronous task scheduler with a flexible distributed system to process messages and manage operations, which enables real-time task processing and scheduling. The tasks can be executed and controlled by the Celery worker server asynchronously or synchronously. Celery applications use callables to represent the modules that are part of the workflow. Once called, the task client adds to the task queue a message where its unique name is referred so that the worker can locate the right function to execute. The flexibility of Celery framework enables real-time interfacing to manage resource and excise control over the task scheduling and execution. With MD simulation and CVAE tasks modulized as Celery compatible callables, they are monitored and controlled through Celery Python interface. According to the strategy demonstrated in Figure 1, we simply build multi-task workflows, which supports a large volume of concurrent tasks with real-time interfacing and decision-making. The use of Celery framework allows us to establish a baseline for estimating the compute requirements of our workflow.

We tested our deep learning driven adaptive simulation framework on the NVIDIA DGX-2 system at Oak Ridge National Laboratory (ORNL). The DGX-2 system provides more than 2 petaflops of computational power from a single node that leverages its 16 interconnected NVIDIA Tesla V100-SXM3-32GB GPUs. This enables us to distribute the MD simulations and CVAE training onto 12 and 4 GPUs respectively. All the components in the workflow are encapsulated within a Python script that manages the various tasks through Celery. It first initializes the Celery worker along with the selected broker, RabbitMQ. All 16 GPUs are then employed for MD simulations to first generate 100,000 conformers as the initial training data for CVAE. With 5 minute interval be-

tween iterations, the trained CVAE periodically compress MD simulation conformers from MD trajectories into data points of latent space, which are subsequently evaluated with density-based spatial clustering of applications with noise (DBSCAN) for outlying conformations [30]. We used DBSCAN for its relative simplicity and also to establish a baseline implementation of our code. For Fs-peptide, outliers were collected with all four trained CVAE models and only CVAE with 6 dimensional latent space was applied for BBA outlier searching. In each iteration, the MD runs are examined for outliers. Simulations that pass an initial threshold of 20,000 frames (1 μs) for Fs-peptide and 10,000 (0.5 μs) for BBA, but do not produce any outliers for the last 5000 frames (250 ns of simulation time) are purged. With the available GPUs from such MD runs, new MD simulations are launched from the the outliers to ensure appropriate resource management and usage.

4. Results

In previous work [14], we have shown that the CVAE can learn a latent space from the Fs-peptide simulations such that the conformations from the simulations cluster into distinct clusters consisting of folded and unfolded states. When parameters such as the RMSD (to the native state) and the fraction of native contacts are used to annotate the latent dimensions [31], we showed that these latent representations correspond to reaction coordinates that describe how a protein may fold (beginning with the unfolded state ensemble). Thus, we posit that we can propagate the simulations along these low-dimensional representations and can drive simulations to sample folded states of the protein in a relatively short number of iterations.

Figure 2 summarizes the results of our folding simulations of Fs-peptide. The peptide consists of 21 residues – Ace-A_5(AAARA)$_3$A-NME – where Ace and NME represent the N- and C-terminal end caps of the peptide respectively, and A represents the amino acid Alanine, where as R represents the amino acid Arginine. It is often used as a prototypical system for protein folding and adopts a fully helical structure as part of its native state ensemble [32]. Previous simulations used implicit solvent simulations using the GBSA-OBC potentials and the AMBER-FF99SB-ILDN force-field with an aggregate simulation time of 14 μs at 300K [32]. We used the same settings for our MD simulations and initiated our workflow. Summary statistics of the simulations are provided in Table 1. A total of 90 iterations of the workflow was run to obtain a total sampling of 54.198 μs. Note that the sampling time of the MD simulations is an aggregate measure similar to the ones reported in previous studies.

We began by examining the RMSD with respect to the native state from all of our simulations. As shown in Figure 2A, 13 of the total of 31 simulations are unproductive – i.e., they do not sample the native state consisting of the fully formed α-helix. This is not entirely surprising given that the starting state consists of a nearly linear peptide with no residual secondary structures. Based on this observation, we posited that our CVAE model can be used to identify partially folded states from the simulations. We also examined the histogram of the RMSD values computed for each conformation with respect to the native state ensemble (Figure 2B). Based on the histograms, we can reasonably choose a threshold of 3.1$\mathring{A}$ or less to depict the folded state ensemble, followed by 4.6 $\mathring{A}$ for partially folded states, and 8.3 $\mathring{A}$ for the unfolded states. Any trajectory that shows RMSD values beyond 8.3 $\mathring{A}$ are only sampling the unfolded state of the protein.

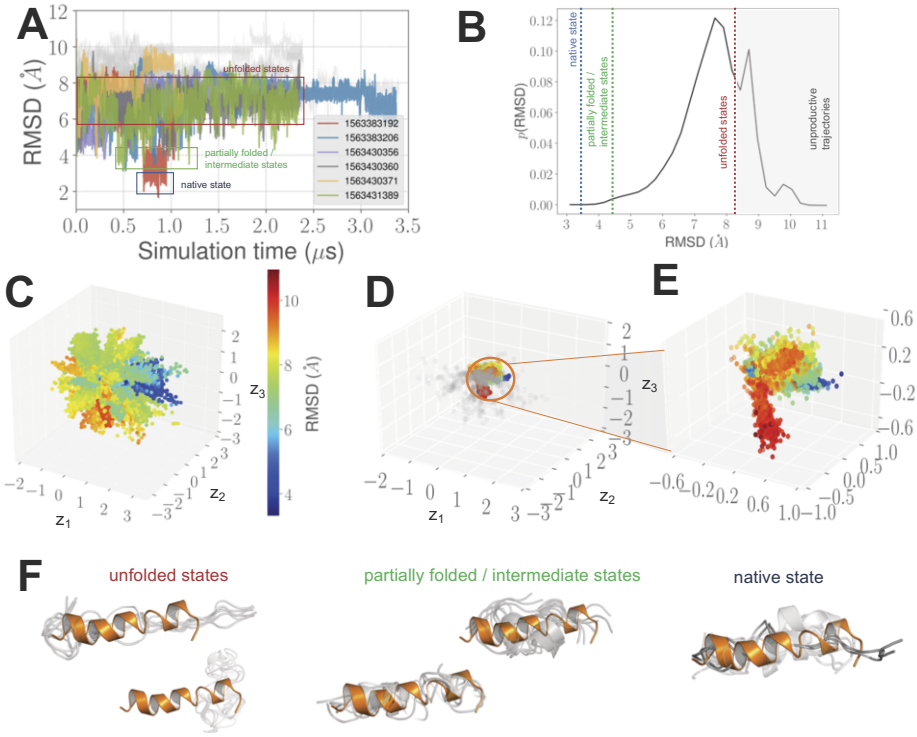


Figure 2. CVAE-driven folding simulations of Fs-peptide.(A) Root mean squared deviation (RMSD) with respect to the native/ folded state from the 31 trajectories generated using our adaptive workflow for the Fs-peptide system. Only productive simulations – i.e., simulations that achieve a RMSD cut-off of 4.5 $\AA$ or less are highlighted for clarity. The rest of the simulations are shown in light gray. (B) A histogram of the RMSD values in panel (A) depicting the RMSD cut-off for identifying folded, partially folded, and unfolded ensembles from the data. The corresponding regions are also marked in panel (A). (C) Using the RMSD to the native state as a measure of foldedness of the system, we project the simulation data onto a three dimensional latent representation learned by the CVAE. Note that the folded states (low RMSD values highlighted in deeper shades of blue) are separated from the folding intermediate (shades of green and yellow) and the unfolded states (darker shades of red).(D) A zoomed in projection of the last 0.5 μs of simulations generated along with the original projections (shown in pale gray, subsampled at every 100^{th} snapshot). (E) highlights the same but just showing the samples from the last 0.5 μs to highlight the differences between folded and unfolded states. (F) shows representative snapshots from our simulations with respect to the unfolded, partial folded, and native state ensembles. Note that the cartoon representation shown in orange represents the native state (minimum RMSD of 1.6 $\AA$ to reference structure) determined from our simulations.

The projections of all the 31 simulations onto the learned CVAE is depicted in Figure 2C. Collectively, z_1-z_3 provide a description of the Fs-peptide folding process. Notably, much of the folded conformational states (highlighted in blue, indicating low RMSD to the native state) are clustered together. Similarly, the unfolded conformations (conformations colored in darker shades of red with higher RMSD to the native state ensemble) are also clustered together. Taking this further, we examined if the similarity in the conformations hold even with a smaller partition of the data (see Figures 2D and E), namely the last 10% of the overall simulation data. This can be treated as a test set from which new simulations are initiated. Notably, from these simulations we observe the presence of roughly three arms in the projections (Figure 2E) consisting of: (1) partially folded

System	Total no. simulations	Total simulation time (μs)	(Shortest*, Longest) simulations (μs)	Iterations	Min. RMSD ($\AA$)
Fs-peptide	31	54.198	1.01, 3.4	90	1.6
BBA (FSD-EY)	45	18.562	0.517, 0.873	100	4.44

Table 1. Summary statistics of simulations. *Only considering the simulations that pass the initial threshold.

highlighted in shades of green/yellow, (2) unfolded state ensemble highlighted in shades of red, and (3) a much smaller ensemble of folded states (highlighted in blue).

For each of these states, we can also extract the structural characteristics with respect to the folded state (Figure 2F). Many of the unfolded states do not consist of any secondary structural features (top and bottom left panels). The partially folded states consist of partial turns/ helical structures. The final folded state (with RMSD of 1.6 $\AA$) consists of most (if not all) helical turns in the protein.

4.1. Folding simulations of FSD-EY

The BBA protein namely, FSD-EY is a designed protein that adopts a β-β-α-fold in its native state; however this protein tends to be dynamic in solution [33,34]. Similar to other zinc-finger proteins, the structure of the protein can potentially vary, and represents a challenging use-case for testing our workflow. As shown in Figure. 3, our simulations do start with a completly unfolded state of the protein (average RMSD to native state is about 12 $\AA$. Using an aggregated MD sampling time of 18 μs, we note that we reach a RMSD value of 4.44 $\AA$.

Although we do not sample the native state of the protein consisting of the β-β-α-fold, we are still able to sample regions of the landscape that consist of a defined hydrophobic core consisting of the highlighted residues in Figure 3D. Except for the dynamic C-terminal end, where the hydrophobic interactions between F21 and F25 are not entirely stable, the conformations that exhibit low RMSD values to the native state depict the presence of this hydrophobic core. We expect that extending these simulations further using the CVAE-driven protocol will enhance these interactions allowing it to fold completely.

5. Discussion

As artificial intelligence (AI) and deep learning (DL) techniques become more pervasive for analyzing scientific datasets, there is an emerging need for supporting AI/DL coupled workflows to traditional HPC applications such as MD simulations. Our approach provides a proof-of-concept for how we can guide MD simulations to sample folded state ensemble of small proteins using DL techniques. The approach that we chose was based on building a generative model for protein conformations and identifying new starting conformations for additional MD sampling. Although the generative model was only used to identify novel conformations for extending our MD simulations, it nevertheless allowed us to guide the MD simulations towards sampling folded conformations of the protein systems we considered.

Although DL approaches can take significantly longer time to train, we deliberately chose a prototypic DL approach, namely CVAE, to train on our MD simulation data (Ta-

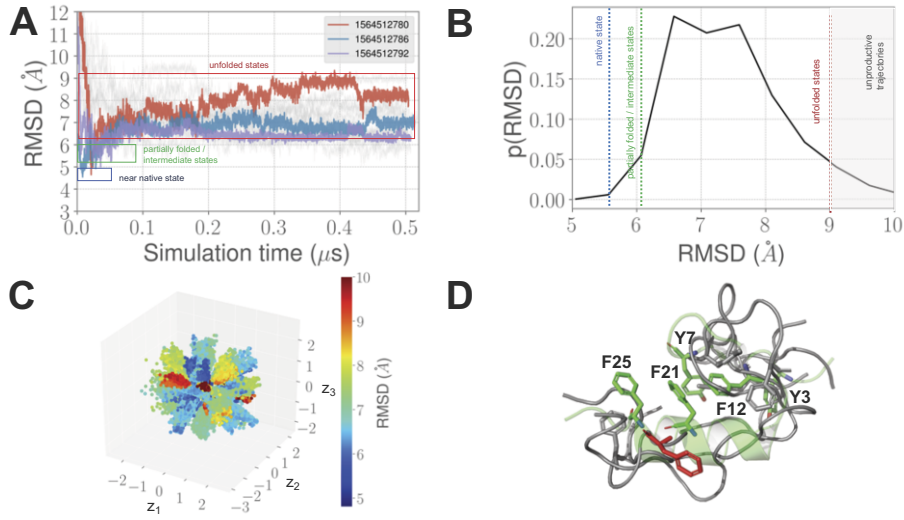


Figure 3. CVAE-driven folding simulations of BBA-fold, FSD-EY. (A) RMSD plots with respect to the native state of FSD-EY depicting the near-native state (blue), partially folded states (green) and unfolded (red) trajectories similar to Figure 2. (B) A histogram of the RSMD values to the native state. (C) The learned projections from the CVAE for the trajecotries; similar to the Fs-peptide system, we can observe the clustering of conformations based on their RMSD values to the native state. We have used a RMSD cut-off of 10$\AA$ to highlight states closer to the native state. (D) Although we could not fully fold the protein, we do observe the presence of a well-formed hydrophobic core except for one residue (F25) at the C-terminal end of the protein.

System	DL training (100 epochs; minutes)	Time per epoch (seconds)	Inference time (ms/frame)	MD simulations (ns per minute)
Fs-peptide	7	5	5.13	1.25
BBA	11	7	1.27	1.20

Table 2. Summary statistics of time taken by the individual components of our workflow: (1) train and infer from the CVAE for each system, and (2) running the MD simulation.

ble 2). As can be seen from the table, the computational cost of training and inference times for the CVAE model is on par with the cost for running our MD simulations. That is, within the time required to train our CVAE model, our MD simulations progress only by about a nanosecond. Thus, starting up of new MD simulations based on the guidance received from our CVAE model will not affect the workflow's overall performance. Further, our MD simulations were run using implicit solvent models, which also significantly reduces their computational times. Further, each contact map is no more than a couple of kilobytes of data and hence we did not require the utilization of intrinsic capabilities of the NVIDIA DGX-2, including the ability to potentially stream data across GPUs/ processors.

A primary motivation for this work was to use ML/DL based analysis to drive MD simulations, and to calibrate results against non ML/DL driven approaches. In Ref. [12] Fox et al introduced the concept of "Effective Performance" that is achieved by combining learning with simulation and without changing the traditional system characteristics. Our selection of physical systems, in particular the BBA peptide allows to provide a coarse-grained estimate for the effective performance of CVAE based adaptive sam-

pling. Using Ref. [4] as reference data, we find that the effective performance of CVAE based sampling is at least a factor of 20 greater than "vanilla" MSM based sampling approaches. Our estimate is based upon the convergence of simulated BBA structures to its reference structures to within 4.5 Å. Note that the ExTASY based simulations in Ref. [4] are at least two orders of magnitude more efficient that reference DE Shaw simulations. In future work, we will extend our effective performance estimate to Villin head piece (VHP) and refine our estimates for BBA.

We expect that the concomitant increase in data sizes for larger MD simulations would necessitate the use of streaming approaches. Similarly, as the time required to train our DL models with data-/model- parallel approaches increases, it will require the use of emerging memory hierarchy architectures to facilitate efficient data handling/ transfer across compute nodes that are dedicated for training and simulation. Further, data intensive techniques such as reinforcement learning and/or active learning could also have been used to guide our MD simulations.

The requirements of the ML/DL driven simulations outlined in this paper are representative of ML/DL driven adaptive workflows — where the status of the intermediate data analysis drive subsequent computations. Adaptive workflows pose significant challenges [13] compared to workflows whose execution trajectory is predetermined a priori. Further, the integration of diverse ML/DL approaches as the intermediate analysis driving subsequent computations adds additional complexity, which includes but is not limited to heterogeneous workload, load balancing and resource management. Scalable execution and modern HPC platforms implies the need for specialized middleware that support address these challenges. We are addressing these aspects as part of ongoing work and future development built upon RADICAL-Cybertools [13,35].

The source code, associated datasets including the generated simulations and deep learning models are available on https://github.com/acadev/Parco2019_baseline.

Acknowledgements

We thank Vivek Balasubramanian and Jumana Dakka for helpful discussions and early contributions. We also thank Chris Layton for helping set up our runs on the NVIDIA DGX-2 compute systems. We also acknowledge support by NSF DIBBS 1443054 and NSF RADICAL-Cybertools 1440677. This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of the manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan). This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05- 00OR22725.

References

[1] Ron O. Dror, Robert M. Dirks, J.P. Grossman, Huafeng Xu, and David E. Shaw. Biomolecular simulation: a computational microscope for molecular biology. *Annu Rev Biophys*, 41(1):429–452, 2012.

[2] Eric H. Lee, Jen Hsin, Marcos Sotomayor, Gemma Comellas, and Klaus Schulten. Discovery through the computational microscope. *Structure*, 17(10):1295–1306.

[3] Gregory R. Bowman, Kyle A. Beauchamp, George Boxer, and Vijay S. Pande. Progress and challenges in the automated construction of markov state models for full protein systems. *J Chem Phys*, 131(12):124101, 2009.

[4] Eugen Hruska, Vivekanandan Balasubramanian, John R Ossyra, Shantenu Jha, and Cecilia Clementi. Extensible and scalable adaptive sampling on supercomputers. *arXiv preprint arXiv:1907.06954*, 2019.

[5] Nina Singhal Hinrichs and Vijay S. Pande. Calculation of the distribution of eigenvalues and eigenvectors in markovian state models for molecular dynamics. *The Journal of Chemical Physics*, 126(24):244101, 2007.

[6] Jeffrey K. Weber and Vijay S. Pande. Characterization and rapid sampling of protein folding markov state model topologies. *J Chem Theory Computat*, 7(10):3405–3411, 2011. PMID: 22140370.

[7] S. Doerr, I. Ariz-Extreme, M. J. Harvey, and G. De Fabritiis. Dimensionality reduction methods for molecular simulations. *ArXiv e-prints*, October 2017.

[8] Shriyaa Mittal and Diwakar Shukla. Recruiting machine learning methods for molecular simulations of proteins. *Molecular Simulation*, 44(11):891–904, 2018.

[9] Zahra Shamsi, Kevin J. Cheng, and Diwakar Shukla. Reinforcement learning based adaptive sampling: Reaping rewards by exploring protein conformational landscapes. *The Journal of Physical Chemistry B*, 122(35):8386–8395, 09 2018.

[10] Michael R. Shirts and Vijay S. Pande. Mathematical analysis of coupled parallel simulations. *Phys Rev Lett*, 86(22):4983–4987, May 2001.

[11] A. J. Savol, V. M. Burger, P. K. Agarwal, A. Ramanathan, and C. S. Chennubhotla. QAARM: quasi-anharmonic autoregressive model reveals molecular recognition pathways in ubiquitin. *Bioinformatics*, 27(13):52–60, Jul 2011.

[12] Geoffrey Fox, James A Glazier, JCS Kadupitiya, Vikram Jadhao, Minje Kim, Judy Qiu, James P Sluka, Endre Somogyi, Madhav Marathe, Abhijin Adiga, and Shantenu Jha. Learning everywhere: Pervasive machine learning for effective high-performance computation. *High-Performance Big Data Computing, IPDPS Workshop, Rio de Janeiro*, 2019.

[13] Vivek Balasubramanian, Travis Jensen, Matteo Turilli, Peter M. Kasson, Michael R. Shirts, and Shantenu Jha. Implementing adaptive ensemble biomolecular applications at scale. *CoRR*, abs/1804.04736, 2018.

[14] Debsindhu Bhowmik, Shang Gao, Michael T. Young, and Arvind Ramanathan. Deep clustering of protein folding simulations. *BMC Bioinformatics*, 19(18):484, 2018.

[15] Peter M Kasson and Shantenu Jha. Adaptive ensemble simulations of biomolecules. *Current Opinion in Structural Biology*, 52:87 – 94, 2018. Cryo electron microscopy: the impact of the cryo-EM revolution in biology • Biophysical and computational methods - Part A.

[16] Sander Pronk, Per Larsson, Iman Pouya, Gregory R. Bowman, Imran S. Haque, Kyle Beauchamp, Berk Hess, Vijay S. Pande, Peter M. Kasson, and Erik Lindahl. Copernicus: A new paradigm for parallel adaptive molecular dynamics. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 60:1–60:10, New York, NY, USA, 2011. ACM.

[17] Vivek Balasubramanian, Travis Jensen, Matteo Turilli, Peter Kasson, Michael Shirts, and Shantenu Jha. Implementing adaptive ensemble biomolecular applications at scale. *arXiv preprint arXiv:1804.04736*, 15:800–809, 2018.

[18] Matthew C. Zwier, Joshua L. Adelman, Joseph W. Kaus, Adam J. Pratt, Kim F. Wong, Nicholas B. Rego, Ernesto Suárez, Steven Lettieri, David W. Wang, Michael Grabe, Daniel M. Zuckerman, and Lillian T. Chong. Westpa: An interoperable, highly scalable software package for weighted ensemble simulation and analysis. *Journal of Chemical Theory and Computation*, 11(2):800–809, 2015. PMID: 26392815.

[19] Nuria Plattner, Stefan Doerr, Gianni De Fabritiis, and Frank Noé. Complete protein–protein association kinetics in atomic detail revealed by molecular dynamics simulations and markov modelling. *Nature Chemistry*, 9(10):1005–1011, 2017.

[20] Sunhwan Jo, Donghyuk Suh, Ziwei He, Christophe Chipot, and Benoît Roux. Leveraging the information from markov state models to improve the convergence of umbrella sampling simulations. *The*

Journal of Physical Chemistry B, 120(33):8733–8742, 08 2016.

[21] Andreas Mardt, Luca Pasquali, Hao Wu, and Frank Noé. Vampnets for deep learning of molecular kinetics. *Nature Communications*, 9(1):5, 2018.

[22] Frank Noé, Alexandre Tkatchenko, Klaus-Robert Müller, and Cecilia Clementi. Machine learning for molecular simulation. *arXiv preprint arXiv:1911.02792*, 2019.

[23] Peter Eastman, Jason Swails, John D. Chodera, Robert T. McGibbon, Yutong Zhao, Kyle A. Beauchamp, Lee-Ping Wang, Andrew C. Simmonett, Matthew P. Harrigan, Chaya D. Stern, Rafal P. Wiewiora, Bernard R. Brooks, and Vijay S. Pande. Openmm 7: Rapid development of high performance algorithms for molecular dynamics. *PLOS Computational Biology*, 13(7):1–17, 07 2017.

[24] Kresten Lindorff-Larsen, Stefano Piana, Ron O. Dror, and David E. Shaw. How fast-folding proteins fold. *Science*, 334(6055):517–520, 2011.

[25] Alexey Onufriev, Donald Bashford, and David A. Case. Exploring protein native states and large-scale conformational changes with a modified generalized born model. *Proteins: Structure, Function, and Bioinformatics*, 55(2):383–394, 2004.

[26] Naveen Michaud-Agrawal, Elizabeth J. Denning, Thomas B. Woolf, and Oliver Beckstein. Mdanalysis: A toolkit for the analysis of molecular dynamics simulations. *J Comput Chem*, 32(10), 2011.

[27] R. J. Gowers, M. Linke, Jonathan Barnoud, Tyler J. E. Reddy, Manuel N. Melo, Sean L. Seyler, Jan Domanski, David L. Dotson, Sebastien Buchoux, Ian M. Kenney, and Oliver Beckstein. MDAnalysis: A python package for the rapid analysis of molecular dynamics simulations. In Sebastian Benthall and Scott Rostrup, editors, *Proceedings of the 15th Python in Science Conference*, pages 98 – 105, 2016.

[28] Carl Doersch. Tutorial on variational autoencoders. *arXiv preprint arXiv:1606.05908*, 2016.

[29] Raquel Romero, Arvind Ramanathan, Tony Yuen, Debsindhu Bhowmik, Mehr Mathew, Lubna Bashir Munshi, Seher Javaid, Madison Bloch, Daria Lizneva, Alina Rahimova, Ayesha Khan, Charit Taneja, Se-Min Kim, Li Sun, Maria I. New, Shozeb Haider, and Mone Zaidi. Mechanism of glucocerebrosidase activation and dysfunction in gaucher disease unraveled by molecular dynamics and deep learning. *Proceedings of the National Academy of Sciences*, 116(11):5086–5095, 2019.

[30] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. pages 226–231. AAAI Press, 1996.

[31] Jörg Gsponer and Amedeo Caflisch. Molecular dynamics simulations of protein folding from the transition state. *Proceedings of the National Academy of Sciences*, 99(10):6719–6724, 2002.

[32] Robert T. McGibbon. Fs MD Trajectories. 5 2014.

[33] Kresten Lindorff-Larsen, Paul Maragakis, Stefano Piana, Michael P. Eastwood, Ron O. Dror, and David E. Shaw. Systematic validation of protein force fields against experimental data. *PLOS ONE*, 7(2):e32131–, 02 2012.

[34] Catherine A Sarisky and Stephen L Mayo. The $\beta\beta\alpha$-fold: explorations in sequence space. *Journal of Molecular Biology*, 307(5):1411 – 1418, 2001.

[35] Matteo Turilli, Vivek Balasubramanian, Andre Merzky, Ioannis Paraskevakos, and Shantenu Jha. Middleware building blocks for workflow systems. *Computing in Science & Engineering (CiSE) special issue on Incorporating Scientific Workflows in Computing Research Processes*, https://arxiv.org/abs/1903.10057.

This page intentionally left blank

Economics

This page intentionally left blank

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200025

A Scalable Approach to Econometric Inference

Philip Nadler [a] Rossella Arcucci [a] and Yi-Ke Guo [a]
[a] *Data Science Institute, Imperial College London*

Abstract. We propose a novel approach combining vector autoregressive models and data assimilation to conduct econometric inference for high dimensional problems in cryptocurrency markets. We label this new model TVP-VAR-DA. As the resulting algorithm is computationally very expensive, it mandates the introduction of a problem decomposition and its implementation in a parallel computing environment. We study its scalability and prediction accuracy under various specifications.

Keywords. Econometrics, Inference, Cryptocurrencies, Data Assimilation, Parallel Computing, Problem Decomposition

1. Introduction

The ongoing digitisation of the economy has led to an abundance of data. Some institutions such as the Bank of England [8] argue that it is only a matter of time until the nature of economic data changes and will be as granular and continuous as is already the case for traffic and weather data. A very prominent example is the emergence of cryptocurrency markets, in which every single transaction is traceable on a publicly available ledger and is updated on a minute by minute bases [14,4], with the ledgers data size continuously growing. Hence, this new economic phenomenon provides an ideal example for the digitisation of economic data and also exposes the limitations of econometric inference at scale.

Due to these developments, economic modeling needs to take computational constraints more into account and incorporate sensible ways to speed up and enable larger scale analysis without the loss of accuracy. Whilst uncommon in economics, this resembles the approach in computational sciences in which researchers often face the problem of trade-offs between accuracy and computational efficiency. Our work therefore aims at bridging this gap by presenting a mathematical formulation of a generalisable economic model in which we explicitly account for scalability and parallelisation as well as evaluate and compare accuracy.

The model class we consider are vector autoregressive models with time varying parameters (TVP-VAR) [9]. TVP-VARs are time series models often used for economic policy analysis and forecasting, describing the interrelationship of economic variables dynamically over time and also model latent economic states whose structure economists exploit

This work is supported by Worley Parsons

for policy analysis [5]. Most conducted studies are small scale [15] because the inclusion of time varying latent states with multiple lagged coefficients leads to parameter matrices whose size increases with the square of the number of variables in the model. This makes CPU time requirements highly nonlinear with respect to the number of variables and thus calls for parallel computing methods [7].

Due to the fact that economists need to study and compare various model parameterisations such as lag length selection, it is important that such models can be computed in reasonable amounts of time without losing accuracy. We thus introduce a domain decomposition approach for significant performance gains. The parallelisation we introduce is on the mathematical formulation of the problem. We decompose the datasets in time windows with possible overlaps and we propose a mathematical model formulation based on domain decomposition. We present and study the performance of the parallel algorithm implemented on a distributed computing architecture. Also, the algorithm's scalability is studied taking into account the execution time.

To study the models performance, we use generated data as well as a dataset that consists of up to 121 identified exchanges on the Bitcoin blockchain. The data is available in multiple frequencies and spans multiple years. Each exchange is represented as an on-chain address cluster of hundreds of addresses. Thus for each exchange, multiple time series are of interest: the aggregate amount of Bitcoin they hold over time, the number of inflows and outflows, as well as transaction rate within a given timeframe. We further expand this with available off-chain data such as price and trading volume of exchanges. This allows for investigation of economic questions such as if a large inflow of Bitcoins has a negative effect of the price of a given exchange, i.e if the rules of supply and demand hold for individual exchanges.

2. The Economic Model

In order to conduct economic inference we combine a TVP-VAR with a Data Assimilation (DA) Framework. DA is an uncertainty quantification technique used to incorporate observational data into a prediction model ([2]) in order to improve numerical forecasted results. As previously discussed, TVP-VAR is a time series model for the analysis of economic systems using latent state variables. The model is outlined in Eq. 1 to 3. We propose a new model which combines TVP-VAR with DA, naming it TVP-VAR-DA. Due to the high dimensionality of the model and the number of state variables used to describe cryptocurrency markets, the TVP-VAR-DA is a large scale problem that should be solved in suitable acceptable time. It mandates the use of parallel computing environments. In this paper, we formally address the parallelism problem by defining the parallel TVP-VAR-DA model based on a problem decomposition approach. In fact, as claimed in [7], the partitioning problem (i.e, decomposability: to break the problem into small enough independent less complex subproblems) is a universal source of scalable parallelism.

To exemplify the model, the basic structure of the univariate TVP-VAR model is:

$$\tilde{y}_t = \tilde{x}_t \phi_t + \tilde{\varepsilon}_t \tilde{\sigma}_t \tag{1}$$

$$\phi_t = \phi_{t-1} + \tilde{v}_t \tag{2}$$

$$log(\tilde{\sigma}_t^2) = log(\tilde{\sigma}_{t-1}^2) + \tilde{\xi}_t \tag{3}$$

where scalar $\tilde{y}_t$ is the time t value of the dependent variable for $t = 1,..,T$, $\tilde{x}_t$ is a $1 \times q$ vector of predictors and lagged dependent variables and ϕ_t the coefficient vector of corresponding dimension. The errors follow the distributions: $\tilde{\varepsilon}_t \sim N(0,1)$, $\tilde{\nu}_t \sim N(0,\tilde{Q}_t)$ and $\tilde{\xi}_t \sim N(0,\tilde{R}_t)$. Eq. 1 describes the relationship of an economic variable with other series of interest. The evolution of this relationship over time is given by Eq. 2, whereas Eq. 3 models changes in the volatility of variables over time.

Generalisation In order to generalise the model in Eq. 1-3 to arbitrary lag length and number of variables under consideration, it is necessary to re-express the model in a more general matrix notation:

$$y_t = \Phi_1 y_{t-1} + ... + \Phi_l y_{t-l} + \mu_t + \varepsilon_t \sigma_t \tag{4}$$

where y_t is now a $q \times 1$ vector of variables we wish to study, μ_t a vector of means and Φ_l is a $q \times q$ coefficient matrix. σ_t is of the same dimension as y_t whereas ε_t is a diagonal matrix. To write this compactly, we define $X_t = [y'_{t-1},...,y'_{t-l},1]'$ and $\Phi = [\Phi_1,...,\Phi_l,\mu]'$. Define $K = (ql+1)$ as the product of variables q and lag length l including a constant. Thus X_t is of dimension $K \times 1$ and Φ of dimension $K \times q$.
Vectorisation of Φ is performed in order to include variable lag length which is necessary for policy analysis while preserving markovian properties of the resulting state-space model. Therefore define $x_t = I \otimes X_t$ using Kronecker product $\otimes$, where I is the identity matrix and define $\beta_t = vec(\Phi)$, where $vec()$ stacks the columns of a matrix. This allows us to rewrite the VAR in compact notation similar to the univariate case and express it in state space form:

$$y_t = x'_t \beta_t + \varepsilon_t \sigma_t \tag{5}$$

$$\beta_t = F\beta_{t-1} + v_t \tag{6}$$

$$log(\sigma_t^2) = log(\sigma_{t-1}^2) + \xi_t \tag{7}$$

where β_t is now of dimension $qK \times 1$ with corresponding transition matrix F and x'_t of dimension $q \times qK$, where $log(\sigma_t^2)$ in Eq. 7 scales accordingly. The terms ε_t, v_t and ξ_t are zero mean errors and will be denoted in the linearization and algorithm section for clarity. Distributional assumptions remain the same with $\varepsilon_t \sim N(0,I)$, $v_t \sim N(0,Q_t)$ and $\xi_t \sim N(0,R_t)$ with the covariance matrices scaling accordingly. The above equations form a state space model with a stochastic volatility term, that can be approximated via a variety of filtering techniques (see e.g. [10], [3], [11]). Our TVP-VAR-DA methodology is able to produce point forecasts in high-dimensional settings, while also generating the history of latent state variables for analysis, taking into account the interdependencies of many variables, incorporating more information and thus reducing omitted variable bias. The solution to this model which takes into account parallelisation due to domain decomposition is described in the next section.

3. The Parallel Model

The optimal values of parameters β_t and σ_t are obtained via the following DA methodology:

$$\beta_t, \sigma_t = \underset{\beta,\sigma}{\operatorname{argmin}}\ J(\beta, \sigma) \tag{8}$$

with

$$J(\beta, \sigma) = \left|\left|\beta - \beta_{t-1} + \nu_t\right|\right|_{Q_t^{-1}} + ||log\left(\frac{\sigma_{t-1}^2}{\sigma^2}\right) + \xi_t||_{R_t^{-1}} \tag{9}$$

In order to partition the problem, let $\Omega \subset \mathbb{R}^M$ denote the domain[2]. $DD(\Omega)$ then constitutes a partitioning with overlaps such that $DD(\Omega) = \{\Omega_i\}_{1,\ldots,p}$ with $\Omega_i = (x_{i,t}, y_{i,t})_{t=0,\ldots T_i}$ where $T_i < T$, $\Omega_i \subset \mathbb{R}^{r_i}$, $r_i \leq M$ and for $i = 1, \ldots, p$ is such that $\Omega = \cup_{i=1}^{p} \Omega_i$ with $\Omega_i \cap \Omega_j = \Omega_{ij} \neq \varnothing$. This allows us to restate the problem as:

$$J_i(\beta_i, \sigma_i) = ||\beta_i - \beta_{i,t-1} + \nu_{i,t}||_{Q_{i,t}^{-1}} + ||log\left(\frac{\sigma_{i,t-1}^2}{\sigma_i^2}\right) + \xi_{i,t}||_{R_{i,t}^{-1}} \tag{10}$$

where furthermore $y_{i,t}$, $x_{i,t}$ are the restrictions of the corresponding quantities in (5), (6), (7) and (8) on the subdomains which constitute the decomposition. In order to minimise the function in (10) on each subdomain, we pose $\nabla(J_i(\beta_i, \sigma_i)) = 0$ and we solve the normal equations which conduct to a modified version of the Kalman Filter [2,12]:

$$\beta_{i,t} = \beta_{i,t-1} + K_t(y_{i,t} - x_{i,t}\beta_{i,t-1}), \quad t = 1, \ldots, T_i \tag{11}$$

$$log(\sigma_{i,t}^2) = log(\sigma_{i,t-1}^2) + K_{i,t}^*(y_{i,t}^* - log(\sigma_{i,t-1}^2)), \quad t = 1, \ldots, T_i \tag{12}$$

where

$$K_{i,t} = Q_{i,t-1}x_{i,t}'(x_{i,t}Q_{i,t-1}x_{i,t}' + \sigma_{i,t})^{-1} \tag{13}$$

and

$$K_{i,t}^* = R_{i,t-1}(R_{i,t-1})^{-1}, \quad y_{i,t}^* = log((y_{i,t} - x_{i,t}\beta_{i,t-1})^2) - ln(\xi^2) \tag{14}$$

as described in detail in Algorithm 1 applied to each subdomain Ω_i where for ease of notation we drop subdomain subscript i. The Algorithm depicts the main steps and also includes a condition $\bar{\sigma}$, in which $log(\sigma_{i,t}^2)$ can be modeled as time varying or constant over time.

4. Dataset

We conduct our experiments on two datasets. The first one is generated artificially in order to check model specifications and to generate more well-behaved data. The second dataset consists of on-chain as well as off-chain data of the cryptocurrency market.

[2]The training data set is defined as $\mathscr{D} = \{(x_i, y_i), i = 1, \ldots, M\}$

Algorithm 1 The TVP-VAR-DA algorithm $A(\Omega_i, p)$ based on domain decomposition for each subdomain Ω_i, where $DD(\Omega) = \{\Omega_i\}_{1,\ldots,p}$

Input $\Omega_i = (x_{i,t}, y_{i,t})_{t=0,\ldots T}$
Initialise Priors Q_0, R_0, ξ, F, β_0, σ_0
specify $\bar{\sigma}$
for for t=1...T **do**
Prediction Step
    $\beta_{t-1} = F\beta_{t-1}$ ▷ Predicted Mean
    $Q_{t-1} = FQ_{t-1}F'$ ▷ Predicted Variance
    $\eta_{t-1} = y_t - x_t\beta_{t-1}$ ▷ Forecast Error
Stochastic Volatility
    if $\sigma_t \neq \bar{\sigma}$ **then** ▷ Define Constant or Stochastic Volatility
        Construct $y_t^* = log((y_t - x_t\beta_{t-1})^2) - ln(\xi^2)$
        $K_t^* = R_{t-1}(R_{t-1})^{-1}$ ▷ Gain Matrix
        $log(\sigma_t^2) = log(\sigma_{t-1}^2) + K_t^*(y_t^* - log(\sigma_{t-1}^2))$ ▷ Posterior Mean of $log(\sigma_t^2)$
        $R_t = (I - K_t^*)R_{t-1}$ ▷ Posterior Variance of $log(\sigma_t^2)$
    else
        $\sigma_t = \bar{\sigma}$
    end if
Updating Step
    $f_t = x_tQ_{t-1}x_t' + \sigma_t$ ▷ Forecast Variance
    $K_t = Q_{t-1}x_t'(x_tQ_{t-1}x_t' + \sigma_t)^{-1}$ ▷ Gain Matrix
    $\beta_t = \beta_{t-1} + K_t(y_t - x_t\beta_{t-1})$ ▷ Posterior Mean of β_t
    $Q_t = (I - K_tx_t)Q_{t-1}$ ▷ Posterior Variance of β_t
end for

Dataset 1 In order to verify the correct tracking of the state equation we create an artifical dataset that is generated according to the following underlying specifications:

$$y_t = X_t'\gamma_t + e_{1,t} \tag{15}$$

$$\gamma_t = \gamma_{t-1} + e_{2,t} + e_{3,t} \tag{16}$$

Where γ is the time varying state variable, and e_1, e_2, e_3 are $\sim i.i.d\, N(0, e_i)$ white noise processes. Data matrix X_t is generated in a similar fashion as standardised i.i.d process. γ_t and y_t are generated as outlined above.
The dimensions of the dataset are created to match the real dataset and are of problem size $M = 1100$, $M = 8200$, $M = 27900$ and $M = 655600$ respectively, these correspond to the number of entries in x_t for each timestep which resemble combinations of variables and lag lengths included in the model.

Dataset 2 The main dataset of interest is the cryptocurrency market dataset. There are two types of data, one labeled on-chain data is the data derived directly from the Bitcoin blockchain. In short, the blockchain can be described as a distributed ledger system in which a multitude of nodes receive and process transactions created by other actors. Every transaction leads from one public address to another. All nodes in the network then synchronise the state of the ledger to form a global consensus on which address owns how much Bitcoin. The network is pseudonymous: all addresses and their balance are

visible through time, although it is not directly visible by whom the address is controlled. Some researchers such as [13] created and verified heuristics in which it is possible to link addresses to entities such as exchanges and other services. Interested readers are referred to other sources such as [14] or [6]. Based on this heuristic the on-chain dataset consists of 121 exchanges which are partially identified on the blockchain by the authors of https://www.walletexplorer.com/. Given this we calculate its hourly balance and the number of inflows as well outflows of Bitcoins. The second part of that set which we label off-chain data consists of the prices as well as volume of Bitcoin on selected exchanges which were identified in the first dataset. These sets constitute matrices x_t and y_t in Eq. 5 whereas the latent state variable β_t give an economic interpretation of the relationship of the underlying dynamics of off-chain and on-chain data. Data is available up to minute frequency and spans from 2014 until early 2019. Due to the noisy nature of the data we focus on hourly frequency to show the scalability properties of our model. For all experiments we add constants for numerical stability in the algorithm and interpolate missing values to make forecasting results more comparable. Fig. 1 displays the on-chain Bitcoin balance as well as the off-chain trading volume of the Kraken.com exchange for a selected period.

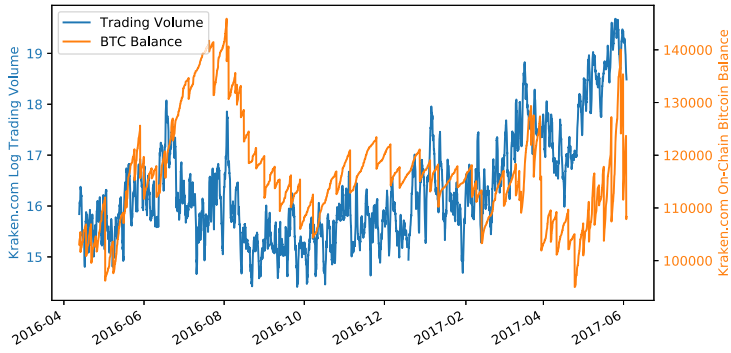


Figure 1. Example of on-chain and off-chain data: The amount of Bitcoin a set of addresses associated to an exchange on the blockchain hold as well as the trading volume on the exchange itself

5. Computational Time and Scalability

We evaluate the performance of Algorithm 1 on *Dataset 1* as we know this does not affect the generality of our study. We computed the values of execution time and we evaluated the scale-up factor. The scale-up factor for a problem decomposition of the function (10) is defined as [1]:

$$S_p(M, p_M) = \frac{T_{p_M}}{T_p}, \tag{17}$$

where p denotes the number of running processors, M denotes the problem size and p_M is the minimum number of processors used for the problem of size M. Table 1 shows the values of the execution time of Algorithm 1 for a problem of size $M = 1100, 8200, 27900, 65600$ created to match the real dataset which resemble combinations of variables and lag lengths included in the model. The experiments are run on a cluster with multiple 2.40GHz Intel Core i7-6700 CPUs and 256GB RAM available.

Problem Size	$M = 1100$	$M = 8200$	$M = 27900$	$M = 65600$
Processors	T_p (seconds)	T_p (seconds)	T_p (seconds)	T_p (seconds)
2	10.47×10^0	19.70×10^1	-	-
4	4.93×10^0	99.33×10^0	98.61×10^1	50.60×10^2
8	2.46×10^0	53.53×10^0	50.65×10^1	25.55×10^2
16	1.25×10^0	29.26×10^0	31.78×10^1	13.63×10^2
32	0.99×10^0	18.34×10^0	14.41×10^1	10.87×10^2

Table 1. Generated data table including stochastic volatility displaying computational time. Experiments ran on Imperial Cluster CX2

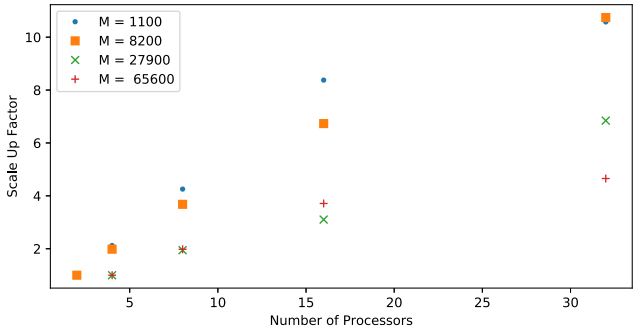


Figure 2. Values of scale-up factor for problem size $M = 1100, 8200$ with $p_M = 2$ and $M = 27900, 65600$ with $p_M = 4$

6. Forecast Comparison

This section evaluates the performance of the model in terms of forecasting. Mean squared forecast errors (MSFE) as well as mean absolute forecast errors (MAFE) are used to compare model quality in-sample. This is based on direct point forecasts, evaluating the residuals of predicted and realised values:

$$MSFE = \sum_{n=0}^{N} \left(\frac{\sum_{\tau=\tau_0}^{T-h} (y_{t,n}^r - \hat{y}_{t,n})^2}{T-h-\tau_0+1} \right) \quad (18) \qquad MAFE = \sum_{n=0}^{N} \left(\frac{\sum_{\tau=\tau_0}^{T-h} \left| y_{t,n}^r - \hat{y}_{t,n} \right|}{T-h-\tau_0+1} \right) \quad (19)$$

where $h = 1$ is the forecast horizon, evaluated before updating the state parameters and $\tau_0 = 1$, the starting date of the forecasting exercise. $y_{t,n}^r$ represents the actual realisation of a variable, while $\hat{y}_{t,n}$ represents the corresponding point forecast. The results in the tables are reported as averages over all included time series respectively, indicated by index n.

Fig. 3 shows the MSFE for one of the experiments using real data. Comparing both axes shows how after the TVP-VAR-DA assimilation the forecasting error decreases by multiple magnitudes, validating the predictive performance of the model.

The results for all experiments are reported in Table 2 and 3. We plot the ratio of forecasting errors before $\hat{y}_{t|t-1}$ and after $\hat{y}_{t|t}$ assimilation of observations, corresponding to forecasts generated via parameters in line 5 and 19 in Algorithm 1. The increase in forecasting accuracy is observable across all problem sizes for both generated and real

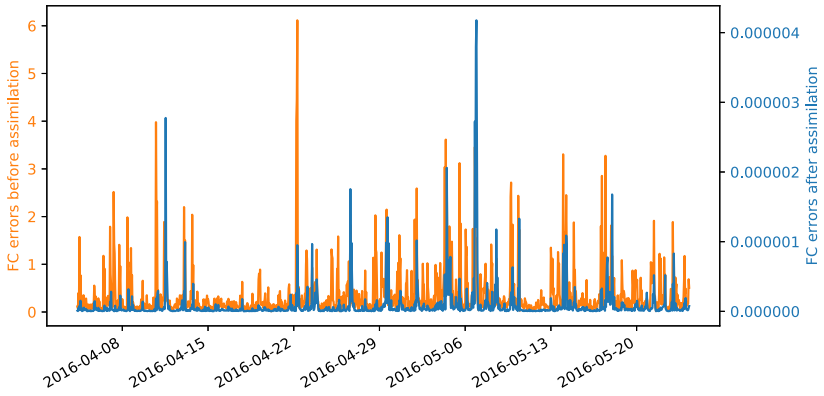


Figure 3. Comparison of forecasting errors before and after assimilation, depicting the average squared forecasting error at each timestep for problem size $M = 1100$

Problem Size	$M = 1100$		$M = 8200$		$M = 27900$		$M = 65600$	
Processors	MSFE	MAFE	MSFE	MAFE	MSFE	MAFE	MSFE	MAFE
2	$2.26e^{-5}$	$1.98e^{-4}$	$3.65e^{-6}$	$8.49e^{-5}$	-	-	-	-
4	$3.38e^{-5}$	$3.55e^{-4}$	$6.94e^{-6}$	$1.64e^{-4}$	$3.27e^{-6}$	$1.18e^{-4}$	$1.44e^{-6}$	$7.97e^{-5}$
8	$5.18e^{-5}$	$7.02e^{-4}$	$8.87e^{-6}$	$2.95e^{-4}$	$3.82e^{-6}$	$2.07e^{-4}$	$1.77e^{-6}$	$1.45e^{-4}$
16	$6.65e^{-5}$	$1.26e^{-3}$	$1.13e^{-5}$	$5.31e^{-4}$	$4.27e^{-6}$	$3.47e^{-4}$	$2.11e^{-6}$	$2.25e^{-4}$
32	$7.33e^{-5}$	$2.08e^{-3}$	$1.12e^{-5}$	8.02^{-4}	$3.97e^{-6}$	$4.95e^{-4}$	$2.24e^{-6}$	$3.35e^{-4}$

Table 2. Generated data table including stochastic volatility displaying accuracy ratios before and after assimilation

Problem Size	$M = 1100$		$M = 8200$		$M = 27900$		$M = 65600$	
Processors	MSFE	MAFE	MSFE	MAFE	MSFE	MAFE	MSFE	MAFE
2	$7.11e^{-6}$	$8.07e^{-4}$	$1.46e^{-5}$	$3.36e^{-3}$	-	-	-	-
4	$1.11e^{-5}$	$8.22e^{-4}$	$1.46e^{-5}$	$3.35e^{-3}$	$1.14e^{-5}$	$2.55e^{-3}$	$1.60e^{-6}$	$2.26e^{-3}$
8	$1.95e^{-5}$	$8.52e^{-4}$	$1.47e^{-5}$	$3.35e^{-3}$	$1.15e^{-5}$	$2.56e^{-3}$	$3.39e^{-6}$	$6.96e^{-3}$
16	$5.29e^{-5}$	$9.32e^{-4}$	$1.81e^{-5}$	$3.38e^{-3}$	$1.46e^{-5}$	$2.59e^{-3}$	$1.07e^{-6}$	$1.37e^{-2}$
32	$9.11e^{-5}$	$1.06e^{-3}$	$2.1e^{-5}$	$3.42e^{-3}$	$1.73e^{-5}$	$2.63e^{-3}$	$5.09e^{-6}$	$9.21e^{-3}$

Table 3. Real data table including stochastic volatility displaying accuracy ratios before and after assimilation

data, although more pronounced for the MSFE metric. By introducing the domain decomposition we see a slight decrease in accuracy when then number of processors increase but still improve forecasts by similar orders of magnitude. We use results of adjunct subdomains with no overlap to provide a lower bound for accuracy. The relative increase in errors due to domain decomposition is decreasing in larger scale problems compared to small ones. In Table 2 it is observable that, across all processors specifications, with increasing problem size the MSFE ratio decreases consistently, meaning that the larger the problem, the larger the increase in forecasting accuracy after assimilation. A similar patterns holds for the MAFE ratios. In contrast, Table 3 shows higher forecasting error ratios across all problem sizes except for the smallest. The forecasting ratio is performing better in the case of synthetic data since the data generation algorithm

is specified in such a way that it aligns with model assumptions of the TVP-VAR-DA, whereas using real cryptocurrency data, the errors are more pronounced due to the more erratic nature of the data.

7. Economic Results

As a case study, we analyse latent state parameters over the summer of 2015 and 2016 which represent the interaction of on-chain and off-chain movements of the Kraken.com exchange. Figure 4 displays selected entries from state vector β_t over time. In particular the predictive effect of price and trading volume changes on Bitcoin flows. In the top figure it is observable that in August 2015 changes in trading volume are associated with a decrease in bitcoin balance and thus outflow of bitcoins whereas in the same period for 2016 this relationship has nearly vanished. This is evidence that over time the on-chain activity has become more decoupled from actual price action on exchanges, which might be driven by other factors such as sentiment. It it also observable that around the change from August to September price changes have a significant positive effect on the inflow of Bitcoin, providing evidence for seasonal cycles in how the flow of Bitcoins affect exchanges.

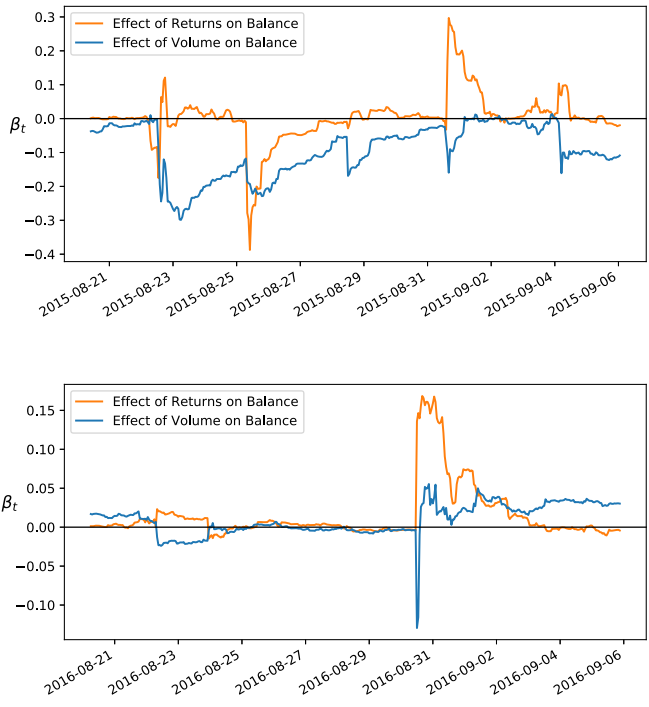


Figure 4. 2015 and 2016 hourly evolution of state variables of Kraken bitcoin blockchain balance and its relation to Kraken price and trading Volume changes.

8. Conclusion

We introduced a new model type that is capable of doing econometric inference at scale by leveraging a data assimilation approach. We show how already in the formulation of the problem we can take into account domain decomposition and parallelisation, showing how similar to computational sciences, economists can increase computational feasibility without sacrificing too much accuracy. We compared model performance and showed that the model generated latent economic variables which help to analyze economic phenomena such as the interaction of entities in cryptocurrency markets. Future work can include additional parallelisations of the algorithm or inference techniques which are unfeasible in standard environments, such as doing a fully Bayesian treatment of the TVP-VAR-DA model, as well as doing real-time forecasting and inference with high frequency data at scale.

References

[1] R. Arcucci, L. D'Amore, L. Carracciuolo, G. Scotti, and G. Laccetti. A decomposition of the tikhonov regularization functional oriented to exploit hybrid multilevel parallelism. *International Journal of Parallel Programming*, 45(5):1214–1235, 2017.

[2] M. Asch, M. Bocquet, and M. Nodet. *Data assimilation: methods, algorithms, and applications*, volume 11. SIAM, 2016.

[3] R. Bhar and D. Lee. Comparing estimation procedures for stochastic volatility models of short-term interest rates. *Available at SSRN 1160659*, 2009.

[4] R. Böhme, N. Christin, B. Edelman, and T. Moore. Bitcoin: Economics, technology, and governance. *Journal of Economic Perspectives*, 29(2):213–38, 2015.

[5] F. Canova and L. Gambetti. Structural changes in the us economy: Is there a role for monetary policy? *Journal of Economic dynamics and control*, 33(2):477–490, 2009.

[6] K. Christidis and M. Devetsikiotis. Blockchains and smart contracts for the internet of things. *Ieee Access*, 4:2292–2303, 2016.

[7] G. C. Fox, R. D. Williams, and G. C. Messina. *Parallel computing works!* Elsevier, 2014.

[8] A. G. Haldane. Will big data keep its promise? *Speech at the Bank of England Data Analytics for Finance and Macro Research Centre, King's Business School*, 2018.

[9] J. D. Hamilton. *Time series analysis*, volume 2. Princeton university press Princeton, NJ, 1994.

[10] A. Harvey, E. Ruiz, and N. Shephard. Multivariate stochastic variance models. *The Review of Economic Studies*, 61(2):247–264, 1994.

[11] S. J. Julier and J. K. Uhlmann. New extension of the kalman filter to nonlinear systems. In *Signal processing, sensor fusion, and target recognition VI*, volume 3068, pages 182–194. International Society for Optics and Photonics, 1997.

[12] R. E. Kalman. A new approach to linear filtering and prediction problems. *Journal of basic Engineering*, 82(1):35–45, 1960.

[13] S. Meiklejohn, M. Pomarole, G. Jordan, K. Levchenko, D. McCoy, G. M. Voelker, and S. Savage. A fistful of bitcoins: characterizing payments among men with no names. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 127–140. ACM, 2013.

[14] S. Nakamoto et al. Bitcoin: A peer-to-peer electronic cash system. 2008.

[15] G. E. Primiceri. Time varying structural vector autoregressions and monetary policy. *The Review of Economic Studies*, 72(3):821–852, 2005.

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200026

Cloud vs On-Premise HPC: A Model for Comprehensive Cost Assessment

Marco FERRETTI and Luigi SANTANGELO
Department of Electrical, Computer and Biomedical Engineering
University of Pavia, Italy

Abstract. Cloud Computing has emerged as an interesting alternative for running business applications, but this might not be true for scientific applications. A comparison between HPC systems and cloud infrastructure not always sees the latter winning over the former, especially when only performance and economical aspects are taken into account. But if other factors, such as turnaround time and user preference, come into play, the landscape of the usage convenience changes. Choosing the right infrastructure, then, can be essentially seen as a multi-attribute decision-making problem. In this paper we introduce an evaluation model, based on a weighted geometric aggregation function, that takes into account a set of parameters, among which job geometry, cost, execution and turnaround time. The notion of user preference modulates the model, and allows to determine which platform, cloud or HPC, might be the best one. The model has then been used to evaluate the best architecture for several runs of two applications, based on two different communication models. Results show that the model is robust and there is a not negligible number of runs for which a cloud infrastructure seems to be the best place for running scientific jobs.

Keywords. cloud computing, HPC, workload, cost-benefit analysis, turnaround time

1. Introduction

Cloud vs on-premise HPC for scientific applications is a long-standing debate [1–5], that has been tackled from many viewpoints, including the cost-perspective [6–8]. Contrary to widespread belief, a cost-benefit analysis comparing cloud infrastructures and HPC systems from the economical point-of view not always sees the cloud as the winner, unless applications allow for preemptible virtual instances. We got this result porting on the cloud two real applications (Cross Motif Search and BloodFlow) that have completely different patterns in their usage of computation and communication resources [9–12]. The former shows a simple master/worker communication model and is therefore less prone to the cloud inefficiency in message routing; the latter instead depends heavily on efficient point-to-point and collective communication primitives. Results show that neither from the performance perspective nor for the economical point-of-view, cloud seems to be a convenient place for running scientific application. But such a comparison is not fair as it does not take into account any factor which make Cloud so appealing. Indeed, just taking into account the turnaround time, the landscape of the usage convenience of the cloud computing changes. Building an evaluation model might help researchers to

understand which platform might be the best one depending on the user preference, the execution time, the cost for computing and the expected waiting time in the queue. To build such a model, a characterization of the workload of a real HPC system needs to be done in order to understand the job waiting time depending on the job geometry (job size, amount of memory, maximum runtime), job failure, setup time and maintenance time.

Many previous works [13–15] have already characterized the workload of the HPC systems, but most of them aimed at evaluating the resource utilization and improve the scheduling algorithms to get the highest system utilization possible. Many others instead tried to predict the waiting time using machine learning techniques [16–19]. In our work, instead, we want to characterize the workload of an HPC system, named Marconi, in order to assess the job waiting time. Such time is then introduced in a utility function which is used to evaluate the best infrastructure (between Marconi and Google Cloud) for running both target applications.

The paper is structured as follows: in Sec. 2, we describe Cross Motif Search and BloodFlow applications and their communication models. Section 3 summarizes the performance results we got running Cross Motif Search and BloodFlow on two similar architectures, Marconi, an HPC system, and Google Cloud Infrastructure, showing that both from the performance and from the economical perspective the cloud lags behind the HPC system, justifying the reason to introduce the turnaround time as a factor to make a fair comparison. In Sec. 4, we describe all the parameters, such as the job waiting time and the virtual instance startup time, that should be kept into account for making a better evaluation of both infrastructures. Section 5 shows a characterization of the jobs submitted on Marconi, with a focus on the job waiting time. Section 6 measures the virtual instance startup time on the Google Cloud Platform for different configurations. Section 7 puts the resulting job waiting time and virtual instance startup time into a decision-making model which uses the weighted geometric aggregation function to build a utility function. Such function also takes the elapsed time measured running Cross Motif Search and BloodFlow on the cloud and on Marconi, and makes an evaluation of both platform in order to understand which run is executed more conveniently on the HPC infrastructure and which one on the cloud, taking into account performance, cost and user preference. According to our utility function, the best infrastructure might not be the one which minimizes cost or maximizes performance, but that which optimizes the user expectation. Section 8 concludes the paper.

2. The target applications

An efficient interconnection network is of paramount importance for getting a high performance in many scientific applications. The communication model embedded in the application and the underlying network infrastructure, are two major factors that should not be ignored during transition toward the cloud. For this reason, to study whether or not cloud computing can be considered convenient for running scientific applications, from the performance perspective as well as the economical one, we selected two different applications, respectively Cross Motif Search and BloodFlow, which are based on two different communication models. The first application is based on a master/worker communication pattern and the time spent in communication is very small if compared with

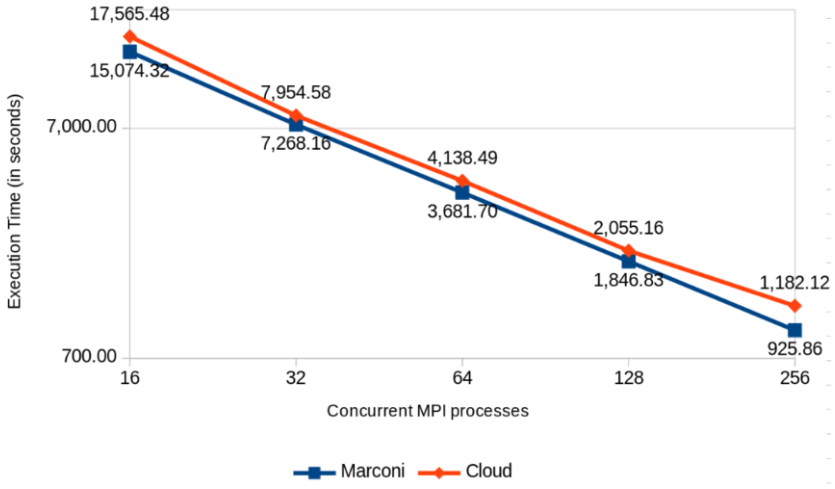


Figure 1. The CMS scalability on Marconi and on the cloud.

the whole elapsed time; the second one, instead, relies on a much more complex communication pattern, making extensive use of collective functions to scatter and gather data. Being based on two communication models which are opposite to each other, the two applications can be considered representative for a large subset of scientific applications. The following subsections describe both applications and their communication model.

2.1. Cross Motif Search

Cross Motif Search (CMS) [20] is a biological application which is able to look for recurring geometrical patterns in the secondary structures of proteins. The core algorithm relies on the generalized Hough transform [21] is used to find recurring geometrical patterns.

The last implementation of CMS [22] uses MPI standard to deliver messages across all processes. The communication model is very simple, as it is based on the traditional master/worker pattern. After starting the application, master and workers communicate to each other just using simple MPI primitives. Profiling activities [9] showed that the impact of the communication in the application performance is almost negligible if compared with the whole execution time. The last implementation of CMS [10, 23] was moved to the cloud in order to study its scalability and compare the cloud performance against the HPC performance. Figure 1 shows the application scalability for two similar architecture: an HPC system, named Marconi, and the cloud infrastructure provided by Google. The application showed a good scalability even on the cloud infrastructure as increasing the number of concurrent MPI processes, the time spent by the application is reduced, as on the HPC system. As the amount of messages exchanged among processes is not dependent on the number of concurrent processes, the scalability is good even increasing further the CPU number.

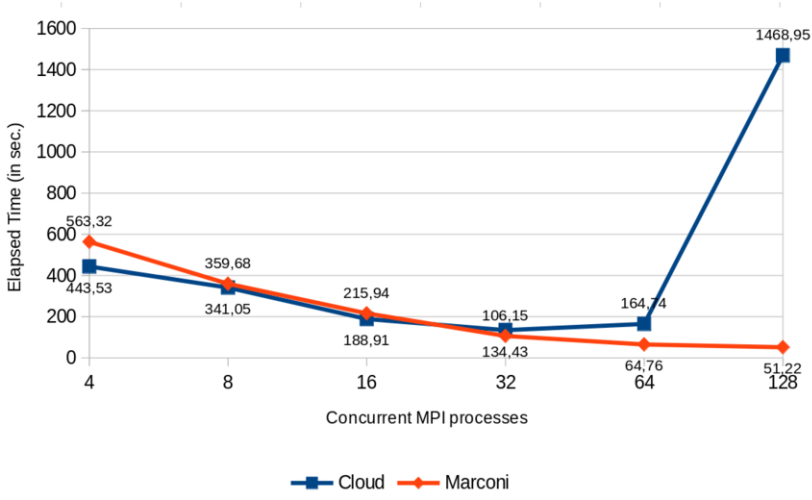


Figure 2. The BloodFlow scalability on Marconi and on the cloud.

2.2. BloodFlow

BloodFlow [24, 25] is a hemodynamics application which is able to run simulations of patient specific hemodynamics of an aorta through computational fluid dynamic analysis. The tool relies on a Navier-Stokes partial differential equation system, which is solved by using numerical approximations. A good description of BloodFlow can be found in [24, 26–28].

Profiling activities on BloodFlow [12] revealed that the application makes use of many MPI functions, point-to-point as well as collective, and that communication is a key factor which can affect application performance.

BloodFlow has been moved on the Cloud although the analysis we did on [10] revealed that BloodFlow might suffer if run on the cloud infrastructure, due to its huge amount of communication and the low network performance on the cloud system. Figure 2 compares the elapsed time measured running the application on both different architectures (Marconi and Cloud) and using different core numbers. The application seems to be able to perform well with a small number of concurrent processes, but when such number grows up the elapsed time becomes soon unmanageable and running the application on the cloud infrastructure is not convenient at all. A similar behaviour ensues even on Marconi, but at much higher core number (256-512).

3. Comparing performance and economical results

According to the results shown in figures 1 and 2, it is clear that CMS is able to scale very well even on the cloud but BloodFlow performs worse as it stops scaling at 32 cores, much before than on the HPC system. This comparison highlights that scientific applications based on a complex communication pattern, such as BloodFlow, might meet several troubles being run on the cloud while those applications based on simple communication model, like CMS, are good candidates to be executed on a cloud environment, because

of the small impact of the interconnection network. In conclusion, cloud computing does not seem to be yet a convenient place for running scientific application at least from the performance perspective.

To understand if Cloud Computing can be convenient at least from the economical perspective, we estimated the cost for running a virtual instance on three different cloud platforms provided respectively by Google, Amazon and Microsoft, in order to compare it with the cost of using a similar configuration in the HPC environment. Each virtual instance in the cluster runs a Red Hat Enterprise Linux distribution and is equipped with 8 cores, 16 GB of memory RAM and 100 GB of Hard Disk. All virtual instances have also been created in a physical cluster based in London. Our analysis revealsed that Microsoft is slightly more expensive (0.53 dollars per hour) than the other two providers (0.41 dollars per hour) but Google wins the comparison as the Amazon billing policy is less convenient because, for example, the cost is computed by hours and not by seconds as in Google. All costs have been computed using the calculator tool made available from all three providers [29–31] and are valid as of March 2019. Even though Google is the cheapest solution, it is still more expensive than Marconi, where the cost per hour for 8 cores is two times lower than Google. And even using preemptible instances (that is, instances that can be stopped if other tasks require access to those resources) the cost, which is dropped by half, stays still higher than on Marconi. In conclusion, not even from the economical perspective cloud seems to be convenient, in the economical setting available at the moment of the experiments (cost estimation on Marconi was based on billing for commercial user).

4. The evaluation model

Looking at the results showed in the Section 3, it might sound that, for scientific researchers, cloud is a burden rather than an opportunity. This might be true if only performance and cost are taken into account. But if other factors [32–35] come in, comparison might yield different results. For example, as the jobs on HPC systems are not usually executed on-the-fly but put in a queue, they might experience a not negligible waiting time. Then, introducing the turnaround time (which is the sum of execution time and job queue delay) as further factor to compare HPC and Cloud systems, the landscape of the usage convenience changes. In our vision, a fair comparison between cloud and HPC infrastructures should take into account not just performance and economical aspects but also waiting time, job failure, job setup time, maintenance time as well as the user preferences. A time-sensible user might be willing to pay a bit more for getting the results sooner and then the chosen architecture will be different according to its preferences. Choosing the right infrastructure can be essentially seen as a multi-attribute decision-making problem. A proper model based on all these attributes might help researchers to understand which platform might be the best one depending on the user preference, the execution time, the cost for computing and the expected waiting time in the queue. The selected architecture might not be the highest performing one, nor the most affordable, but that one which maximizes the utility function describing the model.

To measure the effectiveness of each platform using several attributes, we devised a utility function based on the weighted geometric aggregation function. The attributes taken into account by the formula are user preference, execution time, core hour cost,

expected waiting time in the queue for HPC system and virtual instance startup time for the cloud. Formulas 1 and 2 describe the utility function we adopt for measure the convenience to use the HPC infrastructure U_M or the Cloud system U_C for running any application.

$$U_M = \left(\frac{T_M + W_M}{\max(T_M + W_M, T_C + S_C)}\right)^{\lambda} * \left(\frac{C_M}{\max(C_M, C_C)}\right)^{(1-\lambda)} \quad (1)$$

$$U_C = \left(\frac{T_C + S_C}{\max(T_M + W_M, T_C + S_C)}\right)^{\lambda} * \left(\frac{C_C}{\max(C_M, C_C)}\right)^{(1-\lambda)} \quad (2)$$

In these formulas, T_M, W_M and C_M are respectively the elapsed time, the job waiting time and the cost for running the application on the HPC system, while T_C, S_C and C_C are the elapsed time, the virtual instance startup time and the cost for running the same application on the cloud. Parameter λ instead is the user preference. Its value ranges between 0 and 1. A value λ=0 means that the user is more sensible to the cost (and then the user would like to have the results at a lower cost, without being interested in the time to completion for getting such results); on the opposite side, λ=1 is the preference of a user who is mainly interested in minimizing the time to completion thus optimizing turnaround time. With this utility function, turnaround time comes in as a criterion for assessment. The model is validated by assessing all runs of CMS and BloodFlow applications on both Marconi and Cloud. If U_C is lower than U_M, users would choose Cloud as the preferred platform for running the applications, otherwise Marconi is the best choice.

As already mentioned, the utility function takes the time spent by the applications to be executed on the cloud and on Marconi and their relative costs. Furthermore, it requires the Waiting Time and the Virtual Instance Startup Time which need to be characterized. For this reason, in order to get both information, Sections 5 and 6 make a characterization of both times.

5. The workload analysis

The results described in this section refer to the jobs which have been successfully executed on Marconi A1 partition during eight months, from the 23rd of January up to the 26th of September 2018. During such observed period of time, the number of jobs submitted on Marconi A1 and successfully completed has been equal to 844,975. Half of all completed jobs terminated their execution in less than 43 seconds, while 80% stayed running for less than 1,400 seconds (almost 23 minutes). Only a negligible percentage of jobs (0.06%) took more than 24 hours to complete its execution, with a maximum elapsed time equals to 417,311.

5.1. Job Clusterization

Jobs on Marconi are submitted through Slurm [36]. Using Slurm, users can specify the task to run, the amount of required resources (number of cores and amount of memory), the wall time, which is the time the job might be left running the most, and finally the

queue where the job has to be put on. A queue is not handled like a pure FIFO queue. Each job is assigned a priority index which is computed with a complex formula taking into account many factors such as the waiting time in the queue, the size of the job (core number and amount of memory), the required wall time and furthermore a fair share factor which slows down jobs submitted by users who have almost spent their monthly hours. Because of this scheduling policy, the waiting time spent by a job cannot be computed in advance and might be highly variable. Furthermore, as the queue is chosen by the user at submission time, each queue contains jobs having highly variable geometries, making the queue-oriented classification not proper to be used as a way to classify jobs according to their geometry and the time spent running. For this reason, in order to get sets of more homogeneous jobs, we decided to use k-means as partitioning method for job clusterization. Instead of fixing a priori the number of clusters, we iterated k-means method several times, until the covariance coefficient was lower than 1.1 for all clusters. The covariance coefficient *cc* for the *i-th* cluster is computed as in formula 3:

$$cc^i = \frac{sd(J^i_{ElapsedTime}) + sd(J^i_{CPUNumber})}{mean(J^i_{ElapsedTime}) + mean(J^i_{CPUNumber})} \tag{3}$$

where *sd* is the standard deviation function, *mean* instead is the mean function and J^i is the set of jobs belonging to the *i-th* cluster.

Our test showed that the ideal cluster number is 16, because using a smaller clusterization number makes the covariance coefficient higher the 1.1 at least for one cluster.

5.2. Job Waiting time

5.2.1. A global perspective

An overview on the waiting time of all jobs started on A1 partition shows that the time spent by each job is not negligible as it can last even several days. Indeed, 5.56% of the jobs had to wait at least 24 hours before being run. Only 19.21% instead has been executed almost immediately, while almost fifty percent of the jobs waited at least two minutes before being run.

5.2.2. Waiting time by clusters

We also did a characterization of the clusters we found using k-mean technique. The analysis revealed that the median waiting time for all clusters ranges between 3 seconds (cluster 7) and 92,094 seconds (cluster 1).

5.2.3. Relative Waiting Time by Clusters

As the main aim of this work is to characterize the waiting time the jobs might experience on an HPC system like Marconi, we studied how the relative waiting time changes inside each cluster. We define the relative waiting time *RWT* as follows:

$$RWT = \frac{WaitingTime}{ElapsedTime} \tag{4}$$

Table 1. Relative Waiting Time for all clusters

cluster	CPU Number interval	Elapsed Time interval (sec.)	median	mean	3rd quartile	max
cluster 1	1 - 4752	34387 - 62258	2.17	5.50	8.15	42.96
cluster 2	1 - 4176	1864 - 8176	0.07	6.21	2.54	248.46
cluster 3	1 - 1872	735 - 1234	1.48	15.84	6.91	1,015.34
cluster 4	1 - 512	282 - 534	2.51	29.57	9.74	1,461.78
cluster 5	1 - 216	200 - 335	1.88	25.82	12.24	2,528.10
cluster 6	2048 - 7488	1 - 6643	708.14	2,341.14	3,387.19	69,066.80
cluster 7	1 - 34	1 - 46	0.50	409.09	11.00	313,432.00
cluster 8	1 - 5760	6960 - 19590	0.23	3.38	3.51	133.06
cluster 9	1 - 5760	62227 - 417311	0.06	1.33	1.28	20.95
cluster 10	1 - 5760	18892 - 34880	1.12	5.27	5.84	64.79
cluster 11	1 - 180	103 - 208	0.76	28.31	5.45	4,120.61
cluster 12	1 - 1872	1035 - 2593	0.47	14.39	2.51	507.77
cluster 13	162 - 2088	1 - 920	0.80	381.70	4.60	37,7678.50
cluster 14	1 - 1024	473 - 803	3.40	37.16	27.68	1,809.86
cluster 15	1 - 72	39 - 117	1.80	99.38	12.53	20,702.60
cluster 16	30 - 162	1 - 111	1.50	284.07	13.87	166,114.00

This value is always greater than or equal to 0. Better values are close to 0. The higher the relative waiting time, the greater the impact of the waiting time on the elapsed time. Table 1 shows the numerical values of such distribution. The mean value is almost always greater than the third quartile. This highlights that the distribution is badly affected by some outlier making the mean value and the maximum value much higher than the median value. For this reason, median value can better describe the waiting time because it is insensitive to the presence of outliers.

6. Virtual Instance Startup

Formulas 1 and 2 take into account not only the waiting time, which has been characterized in the previous section, but also the virtual instance startup time. Many works [37–39] have already studied Virtual Instance Startup Time and its relations with other factors such as the time of the day, operating system image size, instance type, data center location and number of instances requested at the same time. Nethertheless, we checked virtual instance startup time for a number of cluster configurations typically used for the benchmark suite of applications we are interested in. The analysis covers startup time measured when activating a cluster of three virtual instances on a physical infrastructure hosted in the West US (us-west2-a). Each virtual instance is equipped with CentOS 7, 50GB of virtual disk and using different virtual hardware configurations (from 1 to 8 cores, from 3.75 to 30 GB of RAM). Times have been measured starting virtual instance from a custom tool written using the Google SDK. Our tests revealed that the median startup time is about 10 seconds.

Table 2. Cluster distribution for CMS and BloodFlow runs

CPU Number	4 CPU	8 CPU	16 CPU	32 CPU	64 CPU	128 CPU	256 CPU
CMS			8	8	2	12	3
BloodFlow	14	4	5	11	15	16	

		CROSS MOTIF SEARCH				
		16	32	64	128	256
lower cost	0.0	MARCONI	MARCONI	MARCONI	MARCONI	MARCONI
user preference	0.1	MARCONI	MARCONI	MARCONI	MARCONI	MARCONI
	0.2	MARCONI	MARCONI	MARCONI	MARCONI	MARCONI
	0.3	MARCONI	MARCONI	MARCONI	MARCONI	MARCONI
	0.5	MARCONI	MARCONI	MARCONI	MARCONI	MARCONI
	0.6	MARCONI	MARCONI	MARCONI	MARCONI	CLOUD
	0.7	MARCONI	MARCONI	MARCONI	MARCONI	CLOUD
	0.8	MARCONI	MARCONI	MARCONI	CLOUD	CLOUD
	0.9	MARCONI	CLOUD	MARCONI	CLOUD	CLOUD
faster results	1.0	CLOUD	CLOUD	MARCONI	CLOUD	CLOUD

Figure 3. Preferred architecture for running CMS.

7. Applying the evaluation model on both applications

As described in Sec. 4, the utility function takes the time spent by the applications to be executed on the cloud and on Marconi and their relative costs. Furthermore, it requires the Virtual Instance Startup Time S_C and the Waiting Time W_M. The previous section gave us a measure of the time spent by a virtual instance to get ready to start the application, which can be fixed to 10. For the parameter W_M instead of using a single value on all runs, we decided to identify the cluster which each run might belong to, according to the job geometry. Then, the median relative waiting time *RWM* for the selected cluster is chosen as a factor to determine the waiting time used in the utility function. The job waiting time *WM* then can be easily computed as follows:

$$WM = RWM * TM \tag{5}$$

where *TM* is the job elapsed time. To be clearer, lets consider the first run of CMS. The application took 15,074.32 seconds using 16 cores. According to the job clusterization defined in table 1, the run might belong to the cluster number 8, where the median relative waiting time for all job in the cluster is 0.23. Then for this run, the estimated waiting time *WM* is equal to 15,074.32 * 0.23 = 3,467.09 seconds. The last run of BloodFlow, instead, took 51.22 seconds using 128 cores. Then this run belongs to the cluster number 16, having a relative waiting time equals to 1.50. Then for such run, the waiting time is equal to 76.83 seconds. Table 2 shows the cluster where each run for both applications belongs to.

Now, we have got all data needed to apply the utility function and assess the best platform. Figures 3 and 4 show which architecture might be the preferred for each run depending on the user preference. Light gray areas represent runs for which the cloud infrastructure is better. Looking at the table describing CMS, for 22% of all runs, the cloud Infrastructure seems to be the best architecture for running the application. For BloodFlow, instead, this percentage is higher, namely 48%.

		BLOODFLOW					
		4	8	16	32	64	128
lower cost	0.0	MARCONI	MARCONI	MARCONI	MARCONI	MARCONI	MARCONI
user preference	0.1	MARCONI	MARCONI	MARCONI	MARCONI	MARCONI	MARCONI
	0.2	MARCONI	MARCONI	MARCONI	MARCONI	MARCONI	MARCONI
	0.3	CLOUD	CLOUD	MARCONI	MARCONI	MARCONI	MARCONI
	0.5	CLOUD	CLOUD	CLOUD	MARCONI	MARCONI	MARCONI
	0.6	CLOUD	CLOUD	CLOUD	MARCONI	MARCONI	MARCONI
	0.7	CLOUD	CLOUD	CLOUD	MARCONI	MARCONI	MARCONI
	0.8	CLOUD	CLOUD	CLOUD	CLOUD	MARCONI	MARCONI
	0.9	CLOUD	CLOUD	CLOUD	CLOUD	MARCONI	MARCONI
faster results	1.0	CLOUD	CLOUD	CLOUD	CLOUD	CLOUD	MARCONI

Figure 4. Preferred architecture for running BloodFlow.

The results described so far are heavily dependent on the relative waiting time introduced in our model. Indeed, if we suppose to rise the relative waiting time of cluster 2 (which is the cluster where the 64-core CMS runs belong in) from 0.07 to 0.40, the cloud preference for CMS rises from 22% to 28%. It is worth noting that the increase we introduced changing the relative waiting time from 0.07 to 0.40, is not negligible. In fact, supposing to have an elapsed time of 3,681.70 seconds (which is the real elapsed time CMS took being run on Marconi using 64 cores), changing the relative waiting time from 0.07 to 0.40, the waiting time goes from 258 seconds to 1,473. According to this observation, we can state that our model is robust, since a high perturbation of the relative waiting time brings a small variation in the convenience to use the cloud infrastructure rather than the HPC system. The results presented above also show that although cloud computing might be more expensive and less powerful than the HPC system, when turnaround time becomes important, cloud computing can be a convenient alternative for running scientific applications.

8. Conclusion

Studying the convenience to use Cloud Infrastructures as alternative to HPC systems for running scientific application is not easy as it should take into account many factors, not only related to the performance and economical aspect. Even the user preference plays an important role as some users might prefer to have results as fast as possible, others instead might wish spending less. In this paper we introduced a new model for the cloud convenience evaluation which takes into account performance, cost, user preference and waiting time. The model has then been applied to study the best architecture to run two different applications, based on two different communication models. Results show that our model is robust as high perturbations in the relative waiting time bring small variation in the results. Furthermore, there is a not negligible number of runs of both applications for which Cloud seems to be the better place, according to our evaluation mode.

References

[1] Rashid Hassani, Md Aiatullah, Peter Luksch, Improving HPC Application Performance in Public Cloud, In IERI Procedia, Volume 10, 2014, Pages 169-176, ISSN 2212-6678.

[2] Napper, Jeffrey, and Paolo Bientinesi. ”Can cloud computing reach the top500?.” Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop. ACM, 2009.

[3] Ramakrishnan, L., Canon, R. S., Muriki, K., Sakrejda, I., & Wright, N. J. (2012). Evaluating interconnect and virtualization performance for high performance computing. ACM SIGMETRICS Performance Evaluation Review, 40(2), 55-60.

[4] Jackson, K. R., Ramakrishnan, L., Muriki, K., Canon, S., Cholia, S., Shalf, J., ... & Wright, N. J. (2010, November). Performance analysis of high performance computing applications on the amazon web services cloud. In Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on (pp. 159-168). IEEE.

[5] Zhai, Y., Liu, M., Zhai, J., Ma, X., & Chen, W. (2011, November). Cloud versus in-house cluster: evaluating Amazon cluster compute instances for running MPI applications. In State of the Practice Reports (p. 11). ACM.

[6] A. G. Carlyle, S. L. Harrell and P. M. Smith, "Cost-Effective HPC: The Community or the Cloud?," 2010 IEEE Second International Conference on Cloud Computing Technology and Science, Indianapolis, IN, 2010, pp. 169-176.

[7] T. Passerini, J. Slawinski, U. Villa and V. Sunderam, "Experiences with Cost and Utility Trade-offs on IaaS Clouds, Grids, and On-Premise Resources," 2014 IEEE International Conference on Cloud Engineering, Boston, MA, 2014, pp. 391-396.

[8] Nanath, K., & Pillai, R. (2013). A model for cost-benefit analysis of cloud computing. Journal of International Technology and Information Management, 22(3), 6.

[9] Ferretti, M., & Santangelo, L. (2018, September). Protein secondary structure analysis in the cloud. In Proceedings of the 6th International Workshop on Parallelism in Bioinformatics (pp. 63-70). ACM.

[10] Ferretti, M., & Santangelo, L. (2019, February). Profiling hemodynamic application for parallel computing in the cloud. In 2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP) (pp. 41-50). IEEE.

[11] Ferretti, M., & Santangelo, L. (2018, March). Hybrid OpenMP-MPI parallelism: porting experiments from small to large clusters. In 2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP) (pp. 297-301). IEEE.

[12] Auricchio, F., Fedele, M., Ferretti, M., Lefieux, A., Romarowski, R., Santangelo, L., & d Veneziani, A. (2018). Benchmarking a hemodynamics application on Intel based HPC systems. Paral Comput Everywhere, 32, 57.

[13] Dror G. Feitelson, Dan Tsafrir, David Krakov, Experience with using the Parallel Workloads Archive, Journal of Parallel and Distributed Computing, Volume 74, Issue 10, 2014, Pages 2967-2982, ISSN 0743-7315,

[14] Gonzalo P. Rodrigo, P.-O. stberg, Erik Elmroth, Katie Antypas, Richard Gerber, Lavanya Ramakrishnan, Towards understanding HPC users and systems: A NERSC case study, Journal of Parallel and Distributed Computing, Volume 111, 2018, Pages 206-221, SSN 0743-7315,

[15] S. Di, D. Kondo and W. Cirne, "Characterization and Comparison of Cloud versus Grid Workloads," 2012 IEEE International Conference on Cluster Computing, Beijing, 2012, pp. 230-238.

[16] Kianpisheh, Somayeh & Jalili, Saeed & Charkari, Nasrolah. (2012). Predicting Job Wait Time in Grid Environment by Applying Machine Learning Methods on Historical Information.

[17] Kumar R., Vadhiyar S. (2015) Prediction of Queue Waiting Times for Metascheduling on Parallel Batch Systems. In: Cirne W., Desai N. (eds) Job Scheduling Strategies for Parallel Processing. JSSPP 2014. Lecture Notes in Computer Science, vol 8828. Springer, Cham

[18] Andresen, D., Hsu, W., Yang, H., & Okanlawon, A. (2018). Machine Learning for Predictive Analytics of Compute Cluster Jobs. arXiv preprint arXiv:1806.01116.

[19] A. Matsunaga and J. A. B. Fortes, "On the Use of Machine Learning to Predict the Time and Resources Consumed by Applications," 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, Melbourne, VIC, 2010, pp. 495-504.

[20] Ferretti, M., Musci, M., & Santangelo, L. (2015). MPICMS: a hybrid parallel approach to geometrical motif search in proteins. Concurrency and Computation: Practice and Experience, 27(18), 5500-5516

[21] Ballard DH. Generalizing the Hough transform to detect arbitrary shapes. Pattern Recognition 1981; 13(2): 111-122

[22] Ferretti, M., Musci, M., & Santangelo, L. (2014, September). A hybrid OpenMP and OpenMPI approach to geometrical motif search in proteins. In 2014 IEEE International Conference on Cluster Computing (CLUSTER) (pp. 298-304). IEEE.

[23] Ferretti, M., & Santangelo, L. (2019). Optimized cloud-based scheduling for protein secondary structure analysis. The Journal of Supercomputing, 1-22.

[24] Quarteroni, A., & Valli, A. (2008). Numerical approximation of partial differential equations (Vol. 23). Springer Science & Business Media.
[25] Formaggia, L., Quarteroni, A., & Veneziani, A. (Eds.). (2010). Cardiovascular Mathematics: Modeling and simulation of the circulatory system (Vol. 1). Springer Science & Business Media.
[26] Bertagna, Luca & Deparis, Simone & Formaggia, Luca & Forti, Davide & Veneziani, Alessandro. (2017). The LifeV library: engineering mathematics beyond the proof of concept.
[27] M. A. Heroux et al., An overview of the trilinos project, ACM Trans. Math. Softw., vol. 31, no. 3, pp. 397-423, 2005
[28] Bertagna, L., Deparis, S., Forti, D., Formaggia, L., & Veneziani, A. (2016), The LifeV library: engineering mathematics beyond the proof of concept, Tech Report Dept. Math & CS, Emory University, TR2016-008, www.mathcs.emory.edu
[29] Amazon Calculator. Retrieved July 5, 2019, from https://calculator.s3.amazonaws.com/index.html
[30] Azure Pricing Calculator. Retrieved July 5, 2019, from https://azure.microsoft.com/it-it/pricing/calculator/
[31] Google Cloud Platform Princing Calculator. Retrieved July 5, 2019, from https://cloud.google.com/products/calculator/
[32] Costa, Pedro & Santos, Joao & Mira da Silva, Miguel. (2013). Evaluation Criteria for Cloud Services. IEEE International Conference on Cloud Computing, CLOUD. 598-605. 10.1109/CLOUD.2013.70.
[33] Geeta, Prakash S. (2018) A Review on Quality of Service in Cloud Computing. In: Aggarwal V., Bhatnagar V., Mishra D. (eds) Big Data Analytics. Advances in Intelligent Systems and Computing, vol 654. Springer, Singapore
[34] J. Singh, S. Agarwal, J. Mishra, A review: Towards quality of service in cloud computing, International Journal of Science and Research
[35] Kumar, Rakesh & Kumar, Chiranjeev. (2018). A Multi Criteria Decision Making Method for Cloud Service Selection and Ranking. International Journal of Ambient Computing and Intelligence. 9. 1-14. 10.4018/IJACI.2018070101.
[36] Overview Slurm Workload Manager. Retrieved July 5, 2019, from https://slurm.schedmd.com/overview.html
[37] M. Mao, H. Humphrey, A performance study on the VM startup time in the cloud, IEEE 5th International Conference on Cloud Computing, June, IEEE, 2012, pp. 423-430 (2012)
[38] Razavi K., Razorea L.M., Kielmann T. (2014) Reducing VM Startup Time and Storage Costs by VM Image Content Consolidation. In: an Mey D. et al. (eds) Euro-Par 2013: Parallel Processing Workshops. Euro-Par 2013. Lecture Notes in Computer Science, vol 8374. Springer, Berlin, Heidelberg
[39] Marathe, Aniruddha & Harris, Rachel & K. Lowenthal, David & R. de Supinski, Bronis & Rountree, Barry & Schulz, Martin & Yuan, Xin. (2013). A comparative study of high-performance computing on the cloud. HPDC 2013 - Proceedings of the 22nd ACM International Symposium on High-Performance Parallel and Distributed Computing.

GPU Computing Methods

This page intentionally left blank

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200028

GPU Architecture for Wavelet-Based Video Coding Acceleration

Carlos DE CEA-DOMINGUEZ [a,1], Juan C. MOURE [b], Joan BARTRINA-RAPESTA [a] and Francesc AULÍ-LLINÀS [a]

[a] *Department of Information and Communications Engineering, Universitat Autònoma de Barcelona, Spain*
[b] *Department of Computer Architecture and Operating Systems, Universitat Autònoma de Barcelona, Spain*

Abstract. The real time coding of high resolution JPEG2000 video requires specialized hardware architectures like Field-Programmable Gate Arrays (FPGAs). Commonly, implementations of JPEG2000 in other architectures such as Graphics Processing Units (GPUs) do not attain sufficient throughput because the algorithms employed in the standard are inherently sequential, which prevents the use of fine-grain parallelism needed to achieve the full GPU performance. This paper presents an architecture for an end-to-end wavelet-based video codec that uses the framework of JPEG2000 but introduces distinct modifications that enable the use of fine-grain parallelism for its acceleration in GPUs. The proposed codec partly employs our previous research on the parallelization of two stages of the JPEG2000 coding process. The proposed solution achieves real-time processing of 4K video in commodity GPUs, with much better power-efficiency ratios compared to server-grade Central Processing Unit (CPU) systems running the standard JPEG2000 codec.

Keywords. Wavelet-based video coding, high-throughput video coding, JPEG2000, GPU, CUDA.

1. Introduction

High definition video with resolutions ranging from 2K to 4K is nowadays common in devices such as TVs, digital cinema projectors, and mobiles. Among others, the JPEG2000 standard (ISO/IEC 15444) is employed for such video content in fields like TV production or digital cinema. This standard provides excellent coding performance and advanced features, such as quality progression, partial transmission, or error resilience [1]. Nonetheless, the algorithms employed to achieve them are very demanding computationally. They transform, code, and reorganize the data in three main stages that require multiple scans and coding operations. This results in long execution times and complex implementations. In the case of digital cinema, for instance, field-programmable gate arrays are required to achieve real-time decoding of 2K and 4K video.

[1]This work has been partially supported by the Spanish Ministry of Economy and Competitiveness and the European Regional Development Fund under Grants TIN2015-71126-R, TIN2017-84553-C2-1-R and RTI2018-095287-B-I00 (MINECO/FEDER, UE), and by the Catalan Government under Grants 2017SGR-463 and 2017SGR-313.

The first stage of most wavelet-based image/video codecs (including JPEG2000) reduces the image redundancy through the discrete wavelet transform (DWT). The operations performed by the DWT can be parallelized, so DWT implementations for parallel architectures have been widely studied in the literature [2–4]. The second stage employs bitplane coding together with arithmetic coding to reduce the statistical redundancy of wavelet coefficients. This stage scans the coefficients one-by-one, producing a result for each that can not be obtained without all the previous. This causality renders parallelism at the bitplane coding engine a very challenging task. Even though there are some works in the literature with this purpose [5–12], none of them is able to exploit the full potential of GPUs. The third stage of the pipeline reorganizes the data and forms the compressed file.

Aware of the high complexity of JPEG2000, the Joint Photographics Experts Group launched in June 2017 a call for proposals [13] aimed to develop an alternate algorithm for the bitplane and arithmetic coding stage that increases the throughput of the codec. This JPEG2000 part is described in [14, 15]. It increases performance about 10× though penalizes coding performance about 10%. Also, it sacrifices quality scalability, which may become an issue in image/video transmission scenarios.

In the same line of work but well before this call for proposals, we started a line of research whose final goal is to devise an end-to-end image/video codec that can exploit the fine-grain parallelism of GPUs while maintaining the same features of JPEG2000. For the bitplane and arithmetic coding engine, we introduced a bitplane coding strategy with parallel coefficient processing (BPC-PaCo) that does not hold dependencies among coefficients, allowing efficient implementations in GPUs [16–19]. This engine can effectively exploit the parallelism of SIMD architectures, which results in high speedup factors and lower power consumption with respect to the fastest implementations of JPEG2000, either executed in multi-core CPUs or in GPUs. Evidently, BPC-PaCo is not compliant with the standard, but it does *not* sacrifice quality scalability and it penalizes coding performance only about 2%. For the DWT, we also proposed an efficient architecture in [2] that achieves high performance in GPUs.

The first implementation of the end-to-end codec for GPUs was presented in [20]. Nonetheless, that implementation is only able to process individual high-resolution images. This paper presents a vastly improved implementation that processes video sequences in real-time thanks to the introduction of stream management with multiple CPU threads, a double-buffer strategy, and event handling to synchronize GPU and CPU operations. Experimental results achieved with consumer-grade GPUs suggest that the proposed codec achieves a throughput that allows encoding and decoding 4K video in real-time and yields highly better power consumption ratios than JPEG2000 codecs executed in CPUs.

The rest of the paper is structured as follows. Section 2 explains the basics of the Nvidia GPU architecture. Section 3 briefly describes the different parts of the JPEG2000 standard. Section 4 describes the proposed end-to-end codec in detail. Section 5 provides experimental results achieved when coding high resolution video in two GPUs and compares our results with Kakadu [21], one of the best multi-thread JPEG2000 implementations. The last section concludes summarizing this work.

2. Overview of Nvidia GPUs

Nvidia GPUs are hardware devices with tens of individual computing units called streaming multiprocessor (SMs). These SMs can work independently, allowing the GPU to process different sequences of operations, called streams, in parallel. This permits the execution of multiple algorithms in an interleaved fashion, which increases the opportunities for parallelism and thereby the throughput achieved. The SMs execute multiple 32-wide SIMD instructions (i.e., vector instructions) simultaneously.

GPUs from Nvidia employ the CUDA programming model, which defines a computation structure composed by (potentially) hundreds of thousands of threads grouped into warps (each with 32-threads), with each warp assigned to a thread block [22]. While the hardware device executes 32-wide SIMD instructions, a software CUDA thread is the virtualization of one of the lanes of the instruction. From the first CUDA-compatible architecture (v1.0) up to Pascal (v6.2), warps execute instructions in a lock-step synchronous fashion, featuring an implicit synchronization at the end of any divergence [23]. From Volta (v7.0) onward, the implicit synchronization is not included at the end of the branching instructions, and must be coded explicitly if needed [24]. Warps in a thread block are executed asynchronously and can cooperate via on-chip fast memories, using explicit synchronizing barrier instructions when required.

The CUDA memory model is hierarchically organized as follows: there is a space of local memory private to each thread, a shared memory private to each thread block, and a global memory public to all threads in the kernel application. From a microarchitecture point of view, the local memory reserved per thread is located either in the registers or the off-chip memory, depending on the available resources. In the proposed implementation, the host memory (CPU RAM) and the device memory (GPU VRAM) are accessed as different, non-unified memory regions, explicitly managing the moment and the amount of data which are copied between them.

3. Overview of JPEG2000

Our GPU codec carries out the same steps as JPEG2000, so they are briefly described below. Depending on the encoding mode employed, either lossy or lossless, the operations involved may be irreversible or reversible, respectively. Irreversible operations improve the compression ratio though they sacrifice image quality slightly.

The first stage of the coding pipeline is the DWT. Our implementation uses a lifting scheme approach [25] due to its low computational complexity. It applies a series of arithmetic operations first row by row and then column by column. The DWT outputs four different subbands, with three of them including smaller detail images and the fourth including the original image at lower resolution and higher energy. These operations are carried out (typically) 5 times within the fourth subband, with each iteration in a lower resolution subband. For lossy compression, the operations employ floating-point arithmetic, so the resulting data are converted to integers before the next stage. This conversion is performed via deadzone quantization [1].

The second stage applies bitplane coding with arithmetic coding. The wavelet coefficients are conceptually partitioned in small sets of typically 64×64 wavelet coefficients, called codeblocks. The strategy adopted to process each codeblock consists in coding

the most relevant information first. The data are divided in bitplanes, with each bitplane containing the set of bits from the same binary position of the unsigned binary representation of each coefficient. Encoding begins from the most significant bitplane (i.e., the one with the highest magnitude within the codeblock) to the lowest one. In JPEG2000, each bitplane is scanned in three coding passes. The first pass is called Significance Propagation Pass. This pass only processes the bits of those coefficients that have at least one significant neighbor. A coefficient is called significant from that bitplane that holds the first non-zero bit to the lowest. The second coding pass is called Magnitude Refinement Pass. It visits the coefficients that are significant in higher bitplanes. The Cleanup Pass processes the coefficients not visited in the previous passes. This coding strategy aims to code the information which holds more information first, effectively reducing distortion [26]. Each bit emitted by the bitplane coder is fed to the arithmetic coder along with its contextual information. The context considers the number of neighbors that are significant, employed to determine a probability for the processed bit. This probability is employed by the arithmetic coder to generate the final bitstream.

The compressed data of codeblocks can be truncated to fit a target bitrate. The method to carry out this optimization process is not defined in the standard, so each implementation may adopt its own solution. The final stage reorganizes the data and adds ancillary information needed by the decoder to decode the original image. The decoder carries out the same steps of the encoder in reverse order.

4. Proposed Codec

The proposed codec is implemented in CUDA. CUDA is employed instead of OpenCL because it provides the latest improvements in the newest architectures. JPEG2000 exposes fine-grain parallelism in all coding stages except for the bitplane coder. Except from the bitplane and arithmetic coder, our proposal produces the same output as JPEG2000 in each stage employing a parallel architecture that extracts most of the GPU performance. BPC-PaCo is employed in the bitplane coding engine [18, 19]. As previously stated, this engine is not compliant with the standard though it preserves the same features and allows parallelism at a fine-grain level.

Algorithm 1 describes the main steps of the proposed codec. Fig. 1 also illustrates its main stages. First, the required memory to process the entire video is allocated in the host CPU RAM (lines 1-2) and in the GPU DRAM, which are respectively referred to as M^H and M^D. Next, two CPU threads are created, denoted by t_1 and t_2 in Algorithm 1, to manage the input/output from/to the hard disk (lines 3-6 and 10-13, respectively). The codec utilizes a double-buffer strategy per stream. This double-buffer is employed for both reading the raw data and writing the compressed file, so four buffers are allocated for each stream. These buffers are referred to as $M^H[i]$ and $M^D[i]$ for the input, and $M^H[o]$ and $M^D[o]$ for the output, with $\{i, o\} \in \{1..2\}$. When reading, the data from one buffer are processed while the other is filled. For writing, the compressed data are transferred to the host from one GPU buffer while the other is already empty and can be filled with compressed data from the frame that is being processed. This buffer structure enables the parallelization of the processing task in two streams, removes the risk of a system memory overflow and increases the utilization of the system resources. Both threads are constantly checking the buffers to start data transfers as soon as possible.

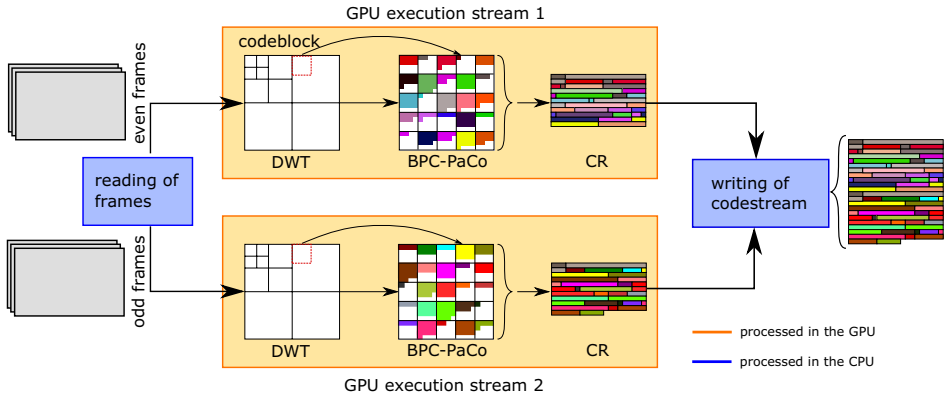


Figure 1. Illustration of the steps performed by the proposed video codec using two streams of execution.

Algorithm 1. Main routine of the codec

1: CPUMemoryAllocation()
2: GPUGlobalMemoryAllocation()
t_1
3: **for** each empty $M^H[i]$ **do**
4: $M^H[i] \leftarrow$ HDRead()
5: $M^D[i] \leftarrow M^H[i]$
6: **end for**
$S_{1,2}$
7: $D \leftarrow$ DWT_Q($M^D[i]$)
8: $\{B_l\} \leftarrow$ BPC_AC(D)
9: $M^D[o] \leftarrow$ CR($\{B_l\}$)
t_2
10: **for** each filled $M^D[o]$ **do**
11: $M^H[o] \leftarrow M^D[o]$
12: HDWrite($M^H[o]$)
13: **end for**

From lines 7 to 9 in Algorithm 1 the GPU functions, or kernels, to code a frame are called. The compression of each frame is carried out with three kernels. Two GPU streams, denoted by $S_{1,2}$, are employed to process a maximum of two frames simultaneously. Typically, each kernel transfers the data to be processed from the global memory $M^D[i]$ to the local memory R to accelerate memory accesses. The kernel DWT_Q$(\cdot)$ receives the original frame data as input and generates quantized wavelet coefficients that are the input of BPC_AC$(\cdot)$. BPC_AC$(\cdot)$ generates a bitstream per codeblock, referred to as B_l, with $l \in \{1..\widehat{L}\}$, $\widehat{L}$ being the number of codeblocks in each frame. This set of bitstreams is reorganized in the last kernel CR$(\cdot)$, which also adds ancillary information for decoding. This kernel does not transfer the compressed data to local registers since it only needs to reorganize the data in global memory.

As illustrated in Fig. 1, the use of two GPU streams allows the processing of two frames in parallel, increasing the throughput of the codec. Evidently, the three stages of the coding pipeline are carried out sequentially in each stream. Once a frame is coded, the resulting data are sent asynchronously to the host memory and the stream begins processing the next frame immediately. The three kernels are devised and implemented to extract fine-grain parallelism in the GPU. The proposed GPU-oriented architecture is able to process either high-resolution images or video in real time. Next, a brief description of each kernel is provided.

4.1. Discrete wavelet transform

The adopted DWT implementation [2] in our codec employs a register-based acceleration strategy [27] that transfers data from the global memory $M^D[i]$ to local registers, avoiding the use of shared memory. Threads communicate among them via shuffle instructions. This strategy allows data reuse and sharing, taking advantage of the fine-grain parallelism and data access locality of the algorithm. First, the image is conceptually divided in blocks that are independently processed by warps. This permits coarse-grain parallelism, populating more SMs of the GPU. The blocks take into account data dependencies of the transform, so they include some samples from adjacent blocks forming a halo. The halo is needed to obtain the same result as if the DWT was applied to the whole image at once. Within each block, the DWT is applied via the lifting scheme, which alternatively processes in the vertical and horizontal axis odd and even samples. If the compression mode is lossy, quantization is applied after the DWT since the next kernel requires integers.

4.2. Bitplane coding with parallel coefficient processing

The coefficients resulting from the DWT_Q$(\cdot)$ are conceptually partitioned in small sets called codeblocks. Typically, each codeblock contains 64×64 coefficients. They are transferred to the local memory to speed up the processing, like in the previous kernel. Codeblocks do not hold dependencies among them, so they are processed independently. The processing of codeblocks in parallel requires coarse-grain, control-divergent strategies that are employed in many implementations of the original JPEG2000 standard. In addition to this parallelism, the coding engine BPC-PaCo employed in this stage extracts fine-grain parallelism in the coding of the codeblock.

BPC-PaCo is based on bitplane coding, like JPEG2000. A particular feature of BPC-PaCo is that it conceptually divides the codeblock in 32 columns holding two coefficients each. Each codeblock is processed by a warp, and each 2-coefficient column is processed by a thread of the warp. Each thread carries out a modified version of significance coding that does not require cleanup, and the magnitude refinement pass. To employ only 2 coding passes instead of 3 like in JPEG2000 significantly increases the throughput [19]. Each emitted bit is coded via context-based arithmetic coding. To form the context of the coefficient, threads need to cooperate among them so that each coefficient can obtain information of all its adjacent neighbors. Again, this cooperation is performed via shuffle instructions. BPC-PaCo utilizes 32 arithmetic coders so that each thread in the warp can code all bits that it emits. The codewords generated by the arithmetic coders are interleaved in an optimal fashion in the final bitstream generated for the codeblock. The result of kernel BPC_AC$(\cdot)$ is the set $\{B_l\}$ that contains a bitstream per codeblock, with $l \in \{1..\widehat{L}\}$ and $\widehat{L}$ being the number of codeblocks per component. The probability model employed by the arithmetic coders is static, i.e., it employs pre-defined probabilities that are computed via a training set of images. This coding strategy permits the use of coarse- and fine-grain parallelism, since both the codeblocks and the coefficients within them are coded in parallel.

	SMs	*cores* × *SM*	*clock frequency*	*memory bandwidth*	*peak FP32 throughput*	*TDP*	*memory size*
GTX 1080 Ti	28	128	1923 MHz	484 GB/s	13.78 TFlops	250 W	11 GB
GTX 960M	5	128	1176 MHz	80 GB/s	1.5 TFlops	60 W	2 GB

Table 1. Features of the GPUs employed.

4.3. Codestream reorganization (CR)

The bitstreams produced for each codeblock have different size depending on the data coded, as depicted in Fig. 1. This results in data scattered in the output buffer of the global memory. The final stage of the coding process reorganizes these data putting them in a compact structure that can be transferred to the main memory of the host $M^H[o]$ and then written to the disk. This stage also includes auxiliary information in the final codestream for decoding.

When a warp compresses a codeblock, the lengths of the bitstreams are stored in a vector of integers $L = \{L_1, L_2, \cdots, L_{\widehat{L}}\}$. Then an aggregated list of lengths, i.e., $L' = \{0, L_1, L_1 + L_2, \cdots, L_1 + \cdots + L_{\widehat{L}}\}$ is generated via the Device Scan primitive from the Nvidia CUB framework [28]. To accelerate the access to this list, a fast lookup table, denoted by $\text{LUT}_{L'}$ is created. This LUT is generated applying a binary search over L' in which each position represents some positions of the original map. Our experience indicates that speedups about $2\times$ are achieved by using such a strategy. Kernel $\text{CR}(\cdot)$ then uses this LUT so that each thread transfers 2 bytes of the codeblock's data to the final output buffer.

5. Experimental Results

The throughput achieved by the proposed codec is compared with Kakadu v7.A.2 [21] in the experiments below. Kakadu is among the fastest implementations of JPEG2000, with multi-thread support for multi-core CPUs. It is programmed in C++ and is heavily optimized via assembly instructions. In the tests below, Kakadu is executed in a platform that has 4 AMD Opteron 6376 CPUs running at 2.3 GHz, employing a total of 32 threads of execution. Results from other JPEG2000 implementations in GPUs [11, 12] are not included herein because their throughput is similar or inferior to that of Kakadu, with the exception of Comprimato [10], which does not offer any option to test its implementation. Our codec is executed in the consumer-grade GPUs reported in Table 1, namely, the high-end GTX 1080 Ti for desktops, and the low-end GTX 960M for laptops. The tests code a video sequence of 948 frames of size 2048×832, gray-scale, and bit-depth of 8 bits per sample. The results shown below do not consider the I/O time needed to read/write the files from/to the disk since that may affect execution times significantly depending on the device employed. In all implementations, data are read from the host memory, where they are preloaded before starting the execution.

The first test evaluates the throughput achieved by our codec when using 1 or 2 GPU streams. The results are depicted in Fig. 2. The vertical axis reports the throughput achieved, in Mega samples per second (Msamples/sec.). The results for 1 and 2 streams are depicted for each GPU and for the encoding and decoding process. Using 2 streams provides a performance increase about 26% (7%) in the encoding process and 22% (9%)

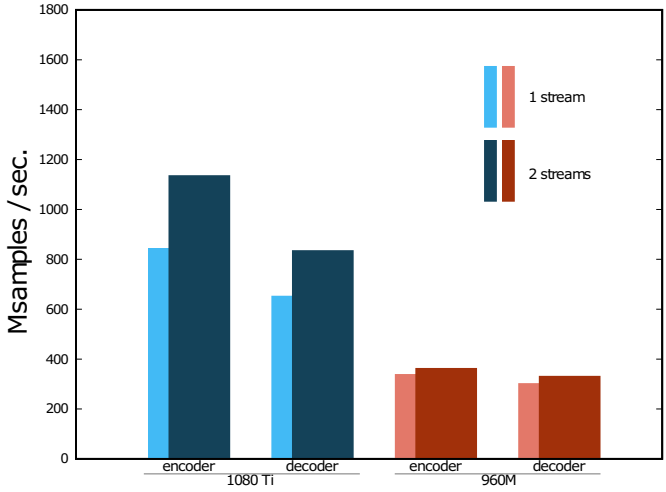


Figure 2. Evaluation of the throughput achieved when using 1 and 2 execution streams.

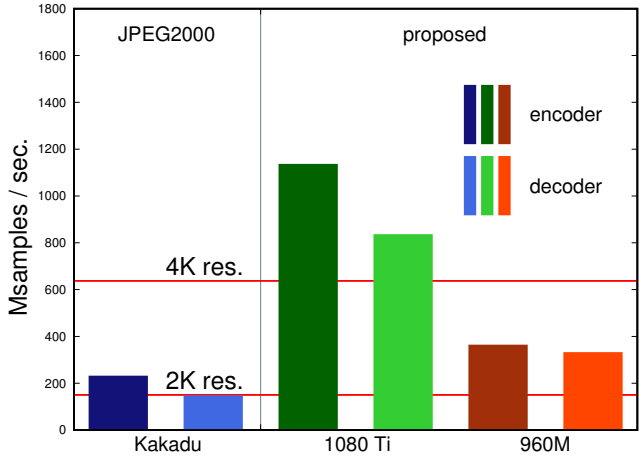


Figure 3. Evaluation of the throughput achieved by the proposed codec and Kakadu.

in the decoding process for the 1080 Ti (960M) GPU. The performance gain depends on the peak throughput of each GPU. The 1080 Ti has ample resources to process more than one frame whereas the 960M almost saturates its resources when coding a single frame (i.e., 1 stream).

The second test evaluates the throughput of the proposed codec running 2 streams and Kakadu. Fig. 3 depicts the obtained results. The proposed codec executed in the 1080 Ti yields a throughput about 5× higher than that of Kakadu. For the 960M, the throughput achieved is about 2× higher than that of Kakadu. Nonetheless, we recall that Kakadu is executed in an expensive multi-CPU platform, whereas the proposed codec employs commodity GPUs. Fig. 3 also depicts the throughput needed to process digital cinema video at resolutions of 2K and 4K in real time (straight horizontal lines). The results suggest that the proposed codec running in the 1080 Ti (960M) can process 4K

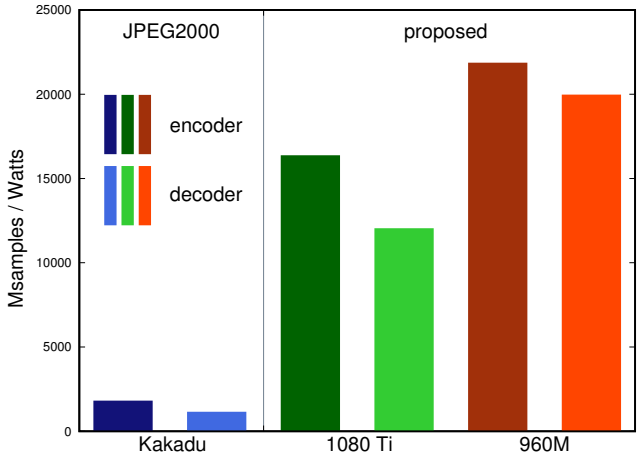


Figure 4. Evaluation of the power efficiency achieved by the proposed codec and Kakadu.

(2K) video in real time.

The third test evaluates power consumption. Fig. 4 depicts the results yield by Kakadu and our codec. In this case, the vertical axis reports Msamples processed per Watt consumed. Kakadu employs four high-end AMD Opteron processors, each with a thermal design power (TDP) of 115W, whereas the 1080 Ti and 960M GPUs have a TDP of 250W and 60W, respectively. The low power consumption of the 960M makes it the most efficient, being approximately 12× more power efficient than Kakadu for encoding and about 17× for decoding. The proposed codec executed in the 1080 Ti is less power-hungry than Kakadu too, with increases in efficiency about 9× and 10× for the encoder and decoder, respectively.

6. Conclusions

The JPEG2000 standard is mainly devised to exploit the coarse-grain parallelism provided in CPUs. When employed to code high-resolution video in scenarios such as TV production or digital cinema, implementations need specialized hardware or expensive computer platforms to meet real-time requirements. So far, implementations for cheaper devices such as GPUs are not able to achieve high throughput because the innermost algorithms of the coding system do not exhibit enough fine-grain parallelism. This paper introduces a fully parallel end-to-end codec that employs the framework of JPEG2000 but that provides –in all stages of the coding pipeline– distinct modifications that permits the use of fine-grain parallelism. This can be effectively exploited when executed in architectures that highly rely on SIMD parallelism such as those found in Nvidia GPUs. Experimental results coding high-resolution video indicates that the proposed codec is 5× faster than the most efficient implementations of JPEG2000 while reducing the power consumption more than 10×. None of the advanced features of JPEG2000 are sacrificed in the proposed codec, so it is ideal for scenarios that deal with massive data sets and/or power constraints.

References

[1] D. S. Taubman and M. W. Marcellin, *JPEG2000 Image compression fundamentals, standards and practice*. Norwell, Massachusetts 02061 USA: Kluwer Academic Publishers, 2002.

[2] P. Enfedaque, F. Auli-Llinas, and J. C. Moure, "Implementation of the DWT in a GPU through a register-based strategy," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 12, pp. 3394–3406, Dec. 2015.

[3] T. M. Quan and W.-K. Jeong, "A fast discrete wavelet transform using hybrid parallelism on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 11, pp. 3088–3100, Nov. 2017.

[4] W. J. van der Laan, A. C. Jalba, and J. B. Roerdink, "Accelerating wavelet lifting on graphics hardware using CUDA," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 1, pp. 132–146, Jan. 2011.

[5] S. Datla and N. S. Gidijala, "Parallelizing motion JPEG 2000 with CUDA," in *Proc. IEEE International Conference on Computer and Electrical Engineering*, Dec. 2009, pp. 630–634.

[6] R. Le, I. R. Bahar, and J. L. Mundy, "A novel parallel tier-1 coder for JPEG2000 using GPUs," in *Proc. IEEE Symposium on Application Specific Processors*, Jun. 2011, pp. 129–136.

[7] J. Matela, V. Rusnak, and P. Holub, "Efficient JPEG2000 EBCOT context modeling for massively parallel architectures," in *Proc. IEEE Data Compression Conference*, Mar. 2011, pp. 423–432.

[8] M. Ciznicki, K. Kurowski, and A. Plaza, "Graphics processing unit implementation of JPEG2000 for hyperspectral image compression," *SPIE Journal of Applied Remote Sensing*, vol. 6, pp. 1–14, Jan. 2012.

[9] M. Ciznicki, M. Kierzynka, P. Kopta, K. Kurowski, and P. Gepnerb, "Benchmarking JPEG 2000 implementations on modern CPU and GPU architectures," *ELSEVIER Journal of Computational Science*, vol. 5, no. 2, pp. 90–98, Mar. 2014.

[10] Comprimato. (2014, Apr.) Comprimato. [Online]. Available: http://www.comprimato.com

[11] (2016, Jun.) CUDA JPEG2000 (CUJ2K). [Online]. Available: http://cuj2k.sourceforge.net

[12] (2016, Jun.) GPU JPEG2K. [Online]. Available: http://apps.man.poznan.pl/trac/jpeg2k/wiki

[13] *High Throughput JPEG 2000 (HTJ2K): Call for Proposals*, ISO/IEC Std., 2017, document ISO/IEC JTC 1/SC29/WG1 N76037.

[14] D. Taubman, A. Naman, and R. Mathew, "FBCOT: a fast block coding option for JPEG 2000," in *Proc. SPIE Applications of Digital Image Processing*, vol. 10396, Sep. 2017, pp. 1–18.

[15] D. Taubman, A. Naman, R. Mathew, and M. D. Smith, "High throughput JPEG 2000 (HTJ2K): New algorithms and opportunities," *SMPTE Motion Imaging Journal*, vol. 127, no. 9, pp. 1–7, Oct. 2018.

[16] F. Auli-Llinas, "Stationary probability model for bitplane image coding through local average of wavelet coefficients," *IEEE Trans. Image Process.*, vol. 20, no. 8, pp. 2153–2165, Aug. 2011.

[17] F. Auli-Llinas and M. W. Marcellin, "Stationary probability model for microscopic parallelism in JPEG2000," *IEEE Trans. Multimedia*, vol. 16, no. 4, pp. 960–970, Jun. 2014.

[18] F. Auli-Llinas, P. Enfedaque, J. C. Moure, and V. Sanchez, "Bitplane image coding with parallel coefficient processing," *IEEE Trans. Image Process.*, vol. 25, no. 1, pp. 209–219, Jan. 2016.

[19] P. Enfedaque, F. Auli-Llinas, and J. C. Moure, "GPU implementation of bitplane coding with parallel coefficient processing for high performance image compression," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 8, pp. 2272–2284, Aug. 2017.

[20] C. de Cea-Dominguez, P. Enfedaque, J. Moure, J. Bartrina-Rapesta, and F. Auli-Llinas, "High throughput image codec for high-resolution satellite images," in *Proc. IEEE International Geoscience and Remote Sensing Symposium*, Jul. 2018, pp. 6524–6527.

[21] D. Taubman. (2018, Dec.) Kakadu software. [Online]. Available: http://www.kakadusoftware.com

[22] "CUDA, C Programming guide," Tech. Rep., Jan. 2015. [Online]. Available: http://docs.nvidia.com/cuda/cuda-c-programming-guide

[23] Nvidia. (2018, Jan.) Warp level primitives. [Online]. Available: https://devblogs.nvidia.com/using-cuda-warp-level-primitives/

[24] ——. (2019, Jun.) Nvidia Tesla V100 GPU architecture. [Online]. Available: http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[25] W. Sweldens, "The lifting scheme: A construction of second generation wavelets," *SIAM Journal on Mathematical Analysis*, vol. 29, no. 2, pp. 511–546, 1998.

[26] F. Auli-Llinas and M. W. Marcellin, "Scanning order strategies for bitplane image coding," *IEEE Trans. Image Process.*, vol. 21, no. 4, pp. 1920–1933, Apr. 2012.

[27] A. Chacon, S. Marco-Sola, A. Espinosa, P. Ribeca, and J. C. Moure, "Boosting the FM-index on the GPU: effective techniques to mitigate random memory access," *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, vol. 12, no. 5, pp. 1048–1059, Sep. 2015.

[28] Nvidia. (2018, Dec.) CUB framework. [Online]. Available: https://nvlabs.github.io/cub/

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200029

GPGPU Computing for Microscopic Pedestrian Simulation

Benedikt Zönnchen [a,b,1], Gerta Köster [a]
[a] *Munich University of Applied Sciences, Lothstrae 64, 80335 Munich, Germany*
[b] *Technische Universität München, Boltzmannstrasse 3, 85748 Garching, Germany*

Abstract. GPGPU computation of microscopic pedestrian simulations has been largely restricted to Cellular Automata and differential equations models, leaving out most agent-based models that rely on sequential updates. We combine a linked-cell data structure to reduce neighborhood complexity with a massive parallel filtering technique to identify agents that can be updated in parallel, thus extending GPGPU computation to one such model, the Optimal Steps Model. We compare two different OpenCL implementations: a parallel event-driven update scheme and a parallel update scheme that violates the event order for the sake of parallelism. We achieve significant speed ups for both in two benchmark scenarios making faster than real-time simulations possible even for large-scale scenarios.

Keywords. discrete event simulation, agent-based simulation, pedestrian dynamics, GPGPU, linked cell algorithm

1. Introduction

Modelling of crowd dynamics has become an important area of research. It helps to understand the interaction of large crowds on a macroscopic level. Results of reliable crowd simulations support safety managers, engineers, event managers and security staff in their decisions. Off-line simulations allow testing of architectural solutions for buildings or facilities for mass events. Nowadays, the application of pedestrians simulations goes beyond off-line simulations. There is a growing interest in and a need for on-line data-driven simulations. Such simulations can predict the future — but only if the computation is faster than real-time. Since microscopic crowd simulations are computationally expensive, they must be accelerated to enable predictive simulations on a large scale.

With the breakdown of Dennard scaling, clock frequencies of single Central Processing Units (CPUs) no longer increase significantly. As a consequence, manufacturers turned their attention towards multi-core processors. In contrast to CPUs, the hardware architecture of Graphics Processing Units (GPU) is designed for massive parallelism. Since GPUs are part of many current and upcoming supercomputers, efficient exploitation of GPUs has become essential in scientific computing. Additionally, GPUs offer thousands of cores inside affordable off-the-shelf workstations making general-purpose Graphics Processing Units (GPGPUs) a source of cheap and efficient computational power. In crowd dynamics thousands or even millions of virtual pedestrians move

[1]Corresponding Author: Benedikt Zönnchen: *zoennchen.benendikt@hm.edu*

simultaneously. At the same time, they are spatially separated, which implies that there is a chance for parallel updates. Consequently, we consider how to exploit GPUs for large-scale crowd simulations.

To simulate thousands of pedestrians in real-time, Cellular Automata (CA) based models are an attractive choice. Space is discretized by a regular and fixed grid of cells and agents are usually represented by occupied cells. This regularity induces efficiency with respect to computational complexity even without parallelism but it also causes inflexibility and inaccuracy in terms of modelling. Motion is restricted to the grid making CA unsuitable for scenarios with high pedestrian densities or fine spatial granularity. (GPGPU) for CA modelling has been explored and successfully applied by many researchers [1,2,3,4,5,6].

Other wide-spread classes of microscopic pedestrian models are force- and velocity-based models where a set of ordinary differential equations (ODEs) describes motion. In contrast to Cellular Automata, agents move in continuous space. Discretization of the continuous model is necessary to numerically solve the equations. Especially for crowded scenarios, accurate results imply small time steps and thus a lot of computational power. In [3] the authors discuss GPU implementations of a CA model, the Social Distance Model (SDM), and the force-based Social Force Model (SFM). They achieve a speed up factor of approximately 4 by exploiting GPGPU.

Almost all microscopic models are, in fact, agent-based models (ABMs). There is a lot of research on using hardware accelerators for ABMs but mostly outside the field of pedestrian dynamics. An extensive overview can be found in [7].

In this contribution we extend massive parallelism through GPUs to another class of agent-based pedestrian models represented by the well validated Optimal Steps Model (OSM) [8,9,10]. In the OSM each agent steps ahead in two dimensional space, driven by its individual pace. The agent's next position is found by solving an optimization problem. Thus the OSM is discrete in time and continuous in space. We present and compare two implementations of the OSM which differ in their update schemes: an inherently sequential event-driven update scheme, which is the OSM's original update scheme, and a newer parallel update scheme.

2. The Optimal Steps Model

The OSM combines aspects from both, CA and differential equation models. It inherits its rule-based discrete stepping events from CA and motion in continuous space from differential equation models. Furthermore, it can be classified as an agent-based model (ABM), since each agent is individualized by its unique free-flow speed v_{free} and stride length λ. The principle idea behind the OSM is that the natural stepwise movement of pedestrians leads to a spatial discretization within the simulation [11,9]: Let Ω be the simulated spatial domain and $\Omega_{\text{out}} \subset \Omega$ the obstacle domain, that is, all space covered by obstacles such as walls. Let $\Omega_{\text{in}} = \Omega \setminus \Omega_{\text{out}}$ be the walkable part of the scenario. Furthermore, let $\partial\Omega_{\text{out}}$ be the scenario boundary. Pedestrians are represented by circular shaped agents of radius $r_p = 0.195$ meters. They move inside Ω_{in}. Agents can step forward in any direction by choosing a position inside their stepping circle. See Fig. 1a. The radius of an agent's stepping circle is derived from the experimentally observed linear dependency of the stride length on the free-flow speed presented in [8]. That is, the radius is given by

$$\lambda = \beta_0 + \beta_1 \times v_{\text{free}} + \varepsilon, \tag{1}$$

where v_{free} is the agent's free-flow speed and ε is a normally distributed error term, $\varepsilon \sim \mathcal{N}(0, \sigma)$. The intercept β_0 and slope β_1 stem from a regression through experimental data [8]. Therefore, λ represents the natural stride length of the the modelled pedestrian. To obtain a heterogeneous population, the free-flow speed is chosen from a truncated normal distribution. Let x_k be the current position of agent i and τ_i be the event time of its next step, then the next position x_{k+1} is found by optimizing a utility function Φ within the stepping circle around the agent:

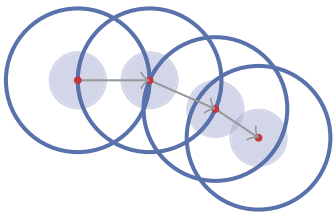
$$x_{k+1} = \arg\min_{y \in P_i} \Phi_i(y), \quad \text{with} \quad P_i = \{y : \|y - x_k\| \leq \lambda_i\} \tag{2}$$

The Optimal Steps Model is event driven. In fact, the linear dependency between free-flow speed and step length also uniquely determines each agent's pace: While the model moves the agent to x_{k+1} in an instant, its next footstep event occurs at $\tau_i + \lambda_i / v_{i,\text{free}}$. In our free and open implementation of the OSM [10], the optimization problem is currently solved either by the Nelder-Mead method or by a brute force evaluation on a discretization of P_i [12] visualized in Fig. 1.

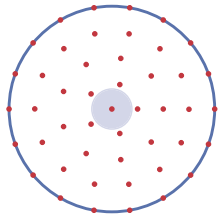
The utility function Φ, which is often interpreted as a potential field, ensures that the destination is reached while skirting obstacles and other agents. We consider it more closely, because calculating Φ contains computationally expensive steps. Let Φ_i be the utility function, or potential field, of agent i. It is given by a sum of sub-utilities or sub-potentials: $\Phi_i = \Phi_{t,\Gamma} + \Phi_o + \Phi_{p,i}$.

- $\Phi_{t,\Gamma}$: contributes attraction to a target Γ and is given by the solution of the eikonal equation. $\Phi_{t,\Gamma}(x)$ encodes the travel time required to reach Γ starting from x. All agents approaching the same target share a common target potential field.
- Φ_o: contributes repulsion caused by obstacles and depends on the distance $d_\Omega(x) = \min_{y \in \partial\Omega} \|x - y\|$, which is the shortest distance to the closest obstacle.
- $\Phi_{p,i}$: is the sum of local repulsion terms caused by other agents and depends on the distance to these agents.

The target and obstacle potential fields are static but Φ_p changes dynamically with the movement of agents. Both repulsive potentials increase with decreasing distance to ob-



a) Ilustration of footsteps of agent i. The circles (blue) indicate the actual step radius λ_i and the shaded area (blue) represents the agent torso of radius r_p.



b) Approximation of P_i by equidistant points inside the step circle.

Figure 1. Computation of the next agent position.

stacles and other agents, respectively. Each sub-potential of Φ_p is realized by a function that has compact support, that is, it is zero outside a small area of influence. For a more detailed description of the modelling aspects we refer to [11,9].

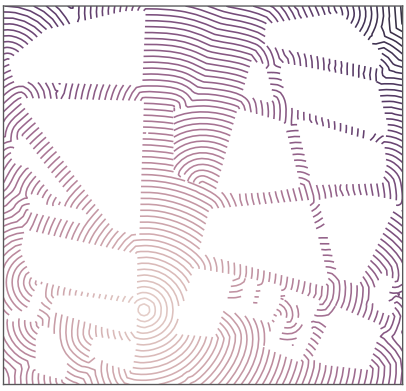
Regarding computational complexity Φ_p is the crucial part. Each sub-potential includes the evaluation of a square root and an exponential function. Therefore, we aim at computing as many sub-potentials as possible in parallel. One essential property to work with is that Φ_p is a local function. More precisely, if x is the position of agent j with $j \neq i$ then its contribution to $\Phi_{p,i}$ at y is zero if and only if $\|x-y\| > w_p$. The locality property and the fact that agents are spatially separated imply that footstep events of agent are not likely to interfere with each other if they are close in time but distant in space. This permits us to exploit parallelism.

2.1. The Event-driven Update Scheme

The orginal OSM is event-driven. The event-driven update scheme processes events in their natural order, that is, the way they occur. In terms of the OSM this ensures that for the choice of the next footstep at time t all footstep events which starts at $\tau < t$ are already processed. From a modelling perspective this implies that pedestrians can anticipate currently processed footsteps of nearby pedestrians. Therefore, Φ_p actually depends on agents' positions in the very near future. By using the event-driven update scheme the OSM becomes a discrete event simulation (DES) model. Note that even though pedestrians only anticipate the movement of nearby pedestrians, this can lead to a chain of navigation adaptations propagating through the whole spatial domain.

2.2. The Parallel Update Scheme

An alternative implementation of the OSM presented in [13] suggest a parallel update scheme. The parallel update scheme uses a global synchronizing clock. An increase of the clock by a fixed time step Δt processes all footstep events within $(t; t+\Delta t]$ in parallel.



a) A target potential $\hat{\Phi}_{t,\Gamma}$ spreads out like wave from a target on the bottom left.

b) Distance function $\hat{d}_\Omega$ which gives the minimal distance to the nearest obstacle.

Figure 2. Plot of solutions of the Eikonal equation of a real world scenario of size 450×400 square meters. White areas are contained in Ω_{out} and therefore not walkable.

This means that we need to deal with potential collisions. The parallel update scheme consist of the following steps:

seek: parallel computation of the next desired positions
move: parallel movement of agents and adjustment of their event time if their event time is the smallest among all competing agents

Agents are competing if their bodies overlap with respect to their desired position. These two steps are repeated until all event times are greater than $t + \Delta t$. It is important to notice that Φ_p changes with each repetition. For the first **seek** call Φ_p depends on the agents' positions at time t.

2.3. Parallel versus Event-driven Update Scheme

Currently, all OSM parameters are calibrated for the event-driven update scheme. If one wants to use the parallel update for predictive simulations, the parameters must be re-calibrated. The parallel update scheme produces the same result as the event-driven update scheme if **move** only effects one agent, that is, if Δt is sufficient small. In [13] the authors showed that, otherwise, the parallel update scheme produces larger evacuation times. This indicates that agents use sub-optimal paths to their destinations because they lose some of their ability to anticipate other agent's motion. We decided to compare computation times and estimate speed-ups for both schemes.

3. OpenCL Implementations of the Optimal Steps Model

We base our implementation on OpenCL to support a broad range of hardware accelerators. It is integrated in our open source framework Vadere [10] which is written in Java. To call our OpenCL kernels within Java, we use the Lightweight Java Game Library 3.2.3 [14].

The OSM is a model on the operational level. It executes locomotion when each agent's destination is known. Route choice, or selection of the destination, is part of the tactical level, which is, in principle, a decision making process. As such its implementation consists of divergent code paths which does not lend itself to execution on the GPU. Consequently, we focus on the operational level and keep the execution of the tactical level on the CPU.

At the start of the simulation the host (CPU) writes the necessary data (all required agent information, $\Phi_{t,\Gamma}$ and d_Ω) to the device (GPU). The host defines how much time the simulation should be stepped forward by the device. After the device has finished its computation the result is transferred back to the host. This allows us to incorporate the tactical level if necessary.

For the sake of simplicity we assume a constant number of n agents during the simulation which are numbered from 0 to $n-1$ having the same target Γ. To compute the target and obstacle potentials on the GPU, $\Phi_{t,\Gamma}$ and the distance function d_Ω are required. We approximate both by $\hat{\Phi}_{t,\Gamma}$ and $\hat{d_\Omega}$, receptively. They are depicted in Fig. 2. $\hat{\Phi}_{t,\Gamma}$ and $\hat{d_\Omega}$ are solutions of the eikonal equation computed by the Fast Marching Method [15] for a regular grid. In case of the target potential the initial wave front of the Fast Marching Method starts at the target boundary $\partial\Gamma$, i. e., $\hat{\Phi}_{t,\Gamma}(x) = 0$ if $x \in \Gamma$. In case of the distance

function, it starts at $\partial\Omega$, i. e., $\hat{d}_\Omega(x) = 0$ if $x \in \Omega_{\text{out}}$. The computation is done on the host. Both grids are transferred into the global memory of the GPU. Values in between grid points are bilinearly interpolated. Furthermore, an approximation of the possible next footstep positions P is computed using a unit circle depicted in Fig. 1 and transferred into cached constant memory. This means that we are using optimizing by "brute force". The possible next positions for agent i at position x_i are given by

$$P_i = \{q \mid p \times \lambda_i + x_i, p \in P\}, \tag{3}$$

where P are the points inside a unit. All required constants such as the domain size and the grid size of $\hat{\Phi}_{t,\Gamma}$ and $\hat{d}_\Omega$ are also transferred to constant device memory. To make use of beneficial coalesce memory, we convert the arrays of structure (AoS), used by the CPU code of Vadere, into a structure of arrays (SoA). Listings 1 and 2 depicts the difference and list all required agent information.

```
class Agent {
  float x;
  float y;
  float eventTime;
  float speed;
  float strideLength;
}
```

Listing 1: Arrays of structure used in object oriented programming.

```
class Agents {
  float[] x;
  float[] y;
  float[] eventTime;
  float[] speed;
  float[] strideLength;
}
```

Listing 2: Structure of arrays used in GPGPU programming.

3.1. The Linked Cell Algorithm

In order to avoid the $\mathcal{O}(n^2)$ complexity of the neighbours search we exploit the locality of agent potentials. Dynamic data structures are difficult to manage on the GPU. The linked cell data structure is a well-known technique to deal with this. We adopt it for our purposes. Let w_Ω, h_Ω be the width and height of a tight bounding rectangle enclosing the whole simulation domain Ω and let c be the cell size of the linked cell data structure, then we divide the space into

$$w_c \times h_c = l, \text{with } w_c = \lceil w_\Omega / c \rceil, h_c = \lceil h_\Omega / c \rceil \tag{4}$$

cells uniquely numbered from 0 to $l-1$. We choose c such that for a given cell, it suffice to consider only agents in its Moore neighborhood to compute the next position of any agent within the cell. Let $v_{\max}, s_{\max}$ be the maximum speed and stride length over all agents. Then a cell size

$$c = \max\{s_{\max}, v_{\max} \times \Delta t\} + r_p + w_p, \tag{5}$$

is sufficient, if we reconstruct the data structure every Δt seconds. Before each update cycle (simulating Δt seconds) we construct the data structure by using an OpenCL implementation of the algorithm described in [16,17] which consist of three steps:

hash: for each agent with position (x,y) its cell id (hash) is computed by $h(x,y) = w_c \times \lfloor y/c \rfloor + \lfloor x/c \rfloor$

sort: agents ids are sorted by a bitonic sort according to their cell id

ordering: agents, i. e., all arrays of the SoA depicted in Listing 2 are reordered according to the sort result

find: cell start indices and cell end indices are detected by unequal consecutive cell ids

The construction is depicted in Fig. 3. The reordering does not only simplify the access to nearby agents but additionally increases the cache hit rate during the following computation steps of the cycle.

3.2. The Parallel Update Scheme

Implementing the parallel update scheme for the GPU is straightforward. One update cycle is realized by invoking multiple OpenCL kernel functions which steps the simulation time from t to $t + \Delta t$. A cycle is completed if there exist no more agent with an event time smaller or equal to $t + \Delta t$. For each agent we have to remember two positions: its actual position and its next possible position. Therefore, we extend the SoA by two additional floating point arrays.

3.2.1. Seek

After the linked cell date structure is constructed, we compute the agents' next possible position in parallel. Each agent is assigned to a different work item (thread) executing the **seek** kernel. If the agents' next footstep happens before $t + \Delta t$, that is, if $\tau \leq t + \Delta t$, the next possible best position is computed. The work item reduces all possible positions to the best one by solving Eq. (2). Finally, the resulting position is saved in global memory.

3.2.2. Move

For each agent the **move** kernel is executed on a different work item (thread). This kernel tests if there are any collisions with respect to the possible next positions (calculated by

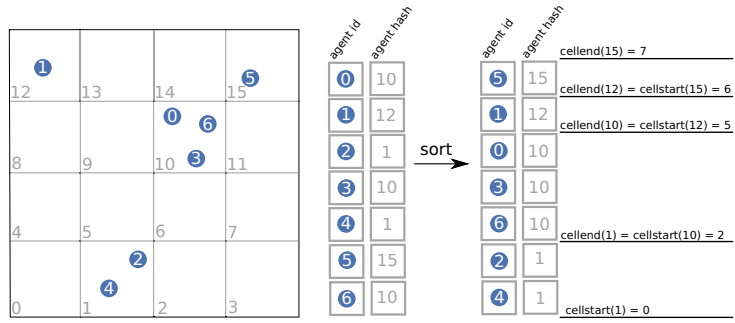


Figure 3. Construction of the linked cell data structure with a cell width w_c and height h_c equal to 4 and $n = 7$ agents. The spatial domain is covered by the rectangle on the left. Agents numbered from 0 to $n-1$ are depicted in blue. After the sorting based on the agent's hash, the start and end, for example, cell 10 are the two array positions at which the agent hash changes to or from 10, i. e., cellstart(10) = 2 and cellend(10) = 5.

the **seek** kernel) agents within the Moore neighbourhood of the linked cell data structure. If there is none, we update the agents event time

$$\tau \leftarrow \tau + (\lambda_i / v_{i,\text{free}}) \tag{6}$$

and position accordingly. Otherwise, we mark the cycle as conflicted. We repeat the cycle, that is, the **seek** and **move** operation until there is no collision detected and all event times are greater than $t + \Delta t$.

3.3. The Event-driven Update Scheme

The number of agents the event-driven update can update in parallel is greatly reduced compared to the parallel update. An upper bound is given by the number of cells l. And in the worst we can only update a single agent. Therefore, we split the computation of the next agent position into $|P|$ tasks, where $|P|$ is the number of possible next positions of agent i. Let $\mathbf{M}$ contain the agent ids of all agents we can update in parallel. Then we evaluate

$$\Phi_i(x_i + z \times \lambda_i), \text{ for } i \in \mathbf{M}, z \in P \tag{7}$$

in parallel. Beforehand, we have to efficiently compute $\mathbf{M}$ which is realized by the following three kernel functions.

3.3.1. Cellfilter

We implement two filters which are processed consecutively. The first **cellfilter** is invoked for each cell of the linked cell data structure. It iterates over all agents of a specific cell and filters the agent with the smallest event time $\tau \leq t + \Delta t$. Its id is written into an array $\mathbf{M}'$ of size $w_c \times h_c$. If no agent was found, which happens if the cell is empty, -1 is written instead. Compare Fig. 4.

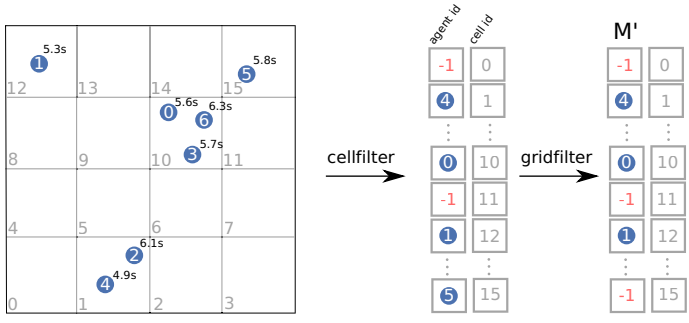


Figure 4. Construction of $\mathbf{M}'$ using the situation depicted in Fig. 3 by invoking **cellfilter** and **gridfilter** consecutively. Blue highlighted numbers represent agent ids and the time represent their event time λ_i. The first array is constructed by **cellfilter**. For each cell the agent with the shortest event time is written into the array. In this example **gridfilter** filters the agent 5 because of the smaller event time of agent 0.

3.3.2. Gridfilter

The second kernel **gridfilter** is also invoked for each cell and filters the left-over agents. It replaces the id by -1 if there is an agent in the Moore neighbourhood with a smaller event time. Note that each work item only has to test 8 agents due to the first filter.

3.3.3. Align

The result of the filtering is a large integer array $\mathbf{M}'$ of size $w_c \times h_c$ containing some agent ids and a lot of negative ones. To compute $|\mathbf{M}|$ we use a modified prefix sum using the algorithm presented in [18]. Instead of summing everything up, we only add 1 if the array value is none negative. Additionally, we compute a second prefix sum array $\mathbf{K}$ ignoring all positive values. Since all agent ids are non negative and all other array entries are set to -1, $-\mathbf{K}[j]$ gives the number of cells with an id smaller than j that are unaffected by any movement update. It follows that $j + \mathbf{K}[j]$ is equal to the number of cells with an id smaller than j which are affected by changes. The **align** kernel is invoked for each cell. Let i be the id of a work item (thread), then all work items generates the aligned array $\mathbf{M}$ of $|\mathbf{M}|$ agent ids by executing the following assignment in parallel:

$$\mathbf{M}[i + \mathbf{K}[i]] \leftarrow \mathbf{M}'[i], \text{ if } \mathbf{M}'[i] \geq 0. \tag{8}$$

After executing the **align** kernel, $\mathbf{M}$ only contains agent ids of the agent which can be updated in parallel. Compare Fig. 5.

3.3.4. Move

To use fast shared memory the **move** kernel is executed by $|\mathbf{M}| \times |P|$ work items grouped into $|\mathbf{M}|$ work groups. The i-th work item (thread) of the j-th work group (thread group) computes $\Phi_{\mathbf{M}[j]}(x_i)$ where x_i is the i-th possible next position. All immediate results are saved into shared memory. Therefore, each work group requires $|P| \times 3 \times 4$ bytes local memory, i. e., 2×4 bytes for each point in P and four bytes to save each evaluation of Φ. After all work items complete their task, the final next position is computed by a parallel reduction using $\lceil |\mathbf{P}|/2 \rceil$ work items which finally solves Eq. (2). The first work item of each work group writes the resulting next position back to global memory. We repeat the cycle, i. e., **cellfilter**, **gridfilter**, **align**, **move** until $\mathbf{M}$ is empty.

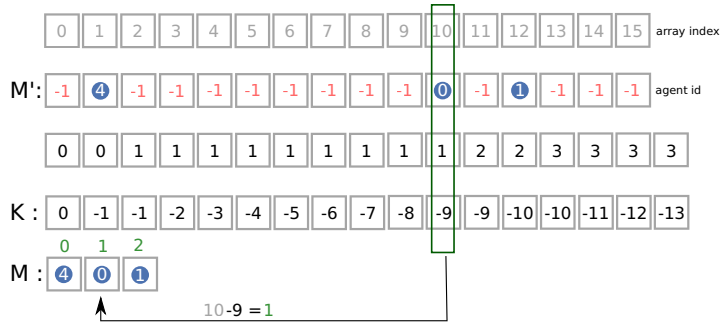


Figure 5. Construction of $\mathbf{M}$ using the situation depicted in Fig. 3 by invoking **align** after the kernel function **gridfilter** has finished. The third and forth line represent the prefix sum arrays. Blue highlighted numbers represent agent ids.

	parallel update scheme			event-driven update scheme		
	OpenCL (GPU)	**OpenCL (CPU**	**Java (CPU)**	**OpenCL (GPU)**	**OpenCL (CPU)**	**Java (CPU)**
10k	3	15	80	20	99	140
100k	13	119	1190	60	546	2100
500k	74	1348	12137	160	2875	30096

Table 1. Average computation time in milliseconds of $\Delta t = 0.4$ seconds simulation time of the open space scenario for $10 \times 10^3, 100 \times 10^3$ and 500×10^3 agents.

4. Comparison of Computation Times

In order to compare computation times of all implementations, we carry out a series of tests. The parallel event-driven update scheme is expected to perform best for evenly distributed and well-separated agents because in this case their footstep events are likely to be independent from each other. It should perform worst if cells are either empty or highly populated. Therefore, we use two benchmark scenarios. The first one consist of multiple bottlenecks which yield high local densities. Even the multi-bottleneck scenario is simple, it imitates more complex geometries and situations by generating a wide range of densities, i. e., from low densities at the start of the simulation to high densities at the time of congestion. For the second scenario we evenly distribute agents inside a large rectangle at the bottom and place the target at the top. Both scenarios are depicted in Fig. 6. For all tests $|P|$ is approximated by 32 points and Δt is set to 0.4 seconds. Note that our OpenCL implementation uses single precision and the existing Java implementation double precision.

Tests were carried out on the following hardware platform: Intel i5-7400 Quad-Core (3.50 GHz), 8 GB DDR4 SDRAM and a graphics card NVIDIA GeForce GTX 1050 Ti / 4 GB GDDR5 VRAM.

In open space, i. e., for the second scenario, using GPGPU computation over the existing Java implementation speeds up the simulation by multiple order of magnitude, i. e., the simulation runs more than 100 times faster. Running the same OpenCL code on the CPU is $5 - 18$ times slower compared to the GPU. The GPU scales much better for a growing number of agents. Compare Table 1. Furthermore, during the simulation the computation times do not significantly fluctuate. The multi-bottleneck scenario

a) Bottleneck scneario. **b)** Open space scenario.

Figure 6. Illustration of both benchmark scenarios. All agents are uniformly distributed inside the green rectangle at $t = 0$ seconds. They walk towards their orange target at the top. The blue trajectories reveals the agents' movement through one of the 6 bottlenecks (left) and straight towards their top target (right).

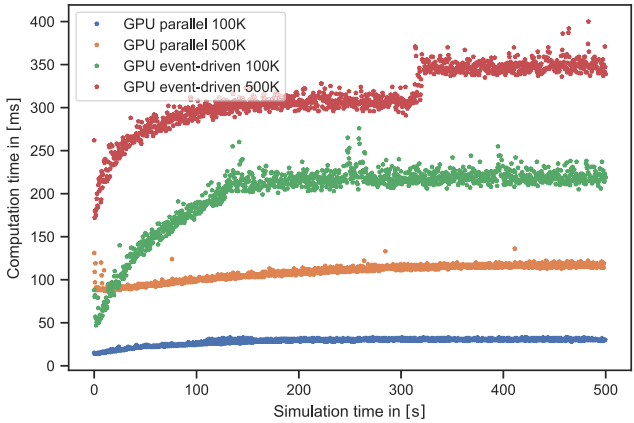


Figure 7. Comparison of computation times over a simulation run of the multi bottleneck scenario for 100 and 500 thousand agents using the parallel and event-driven update scheme. The computation time is required to simulate $\Delta t = 0.4$ seconds.

benchmark reveals that computation times do fluctuate during the simulation run, if the event-driven update scheme is used. As expected, the computation slows down because agents approach the bottlenecks, and thus move closer together. After approximately 100 simulated seconds, the computation time reach a plateau because more and more agents passed the bottleneck. Figure 7 illustrated this phenomenon. The jump at 300 seconds for 500 thousand simulated agents might be the result of some caching effect but further investigations are required.

5. Conclusion

We proposed mechanisms to enable GPU computation for the agent-based Optimal Steps Model which simulates pedestrian dynamics. We presented two implementations: One relied on a parallel update scheme thus modifying the original model. The other parallelized an inherently sequential event-driven update scheme by efficiently identifying independent events and by splitting the event computation into multiple independent tasks. For this we combined a linked cell data structure with massive parallel filtering. We achieved speed-ups of multiple order magnitude for both update schemes compared to the single threaded Java version. Using the same code base but different devices shows that for the chosen hardware setup, the GPU outperforms the CPU by a factor up to 18 for both update schemes. For our specific hardware setup and two non-trivial benchmark scenarios we were able to simulate up to half a million agents faster than real-time. Our techniques can be carried over to any model where the agents' influence remains local and where agents are spatially spread. This is true for many models. Thus we showed that there is great potential in using GPGPU for pedestrian dynamics beyond CA models or differential equation models.

6. Acknowledgement

We thank the research office (FORWIN) of the Munich University of Applied Sciences and the Faculty Graduate Center CeDoSIA of TUM Graduate School at Technical University of Munich for their support. The authors are supported by the German Federal Ministry of Education and Research through the project S2UCRE to study the acceleration of microscopic pedestrian simulations by designing efficient and parallel algorithms (grant no. 13N14463).

References

[1] S. Rybacki, J. Himmelspach, and A. M. Uhrmacher. Experiments with single core, multi-core, and gpu based computation of cellular automata. In *Advances in System Simulation, 2009. SIMUL '09. First International Conference on*, pages 62 –67, Sept 2009.

[2] Q. Miao, Y. Lv, and F. Zhu. A cellular automata based evacuation model on gpu platform. In *2012 15th International IEEE Conference on Intelligent Transportation Systems*, pages 764–768, Sep. 2012.

[3] Hubert Mroz and Jaroslaw Was. Discrete vs. continuous approach in crowd dynamics modeling using gpu computing. *Cybernetics and Systems*, 45(1):25–38, 2014.

[4] Jarosław Was, Hubert Mroz, and Pawel Topa. Gpgpu computing for microscopic simulations of crowd dynamics. *COMPUTING AND INFORMATICS*, 2015.

[5] Adrian Kulusek, Pawel Topa, and Jarosław Was. Towards effective gpu implementation of social distances model for mass evacuation. Working paper, 2016.

[6] Adrian Kłusek, Paweł Topa, and Jarosław Was. An implementation of the social distances model using multi gpu-systems. Working paper, 2016.

[7] Jiajian Xiao, Philipp Andelfinger, David Eckhoff, Wentong Cai, and Alois Knoll. A survey on agent-based simulation using hardware accelerators. *ACM Comput. Surv.*, 51(6):131:1–131:35, January 2019.

[8] Michael J. Seitz and Gerta Köster. Natural discretization of pedestrian movement in continuous space. *Physical Review E*, 86(4):046108, 2012.

[9] Isabella von Sivers and Gerta Köster. Dynamic stride length adaptation according to utility and personal space. *Transportation Research Part B: Methodological*, 74:104–117, 2015.

[10] Benedikt Kleinmeier, Benedikt Zönnchen, Marion Gödel, and Gerta Köster. Vadere: An open-source simulation framework to promote interdisciplinary understanding. *Collective Dynamics*, 4, 2019.

[11] Michael J. Seitz, Felix Dietrich, and Gerta Köster. The effect of stepping on pedestrian trajectories. *Physica A: Statistical Mechanics and its Applications*, 421:594–604, 2015.

[12] Isabella von Sivers and Gerta Köster. How stride adaptation in pedestrian models improves navigation. *arXiv*, 1401.7838(v1), 2014.

[13] Michael J. Seitz and Gerta Köster. How update schemes influence crowd simulations. *Journal of Statistical Mechanics: Theory and Experiment*, 2014(7):P07002, 2014.

[14] Lightweight java game library 3. https://www.lwjgl.org/.

[15] J. A. Sethian. A fast marching level set method for monotonically advancing fronts. *Proceedings of the National Academy of Sciences*, 93(4):1591–1595, 1996.

[16] U. Erra, B. Frola, V. Scarano, and I. Couzin. An efficient gpu implementation for large scale individual-based simulation of collective behavior. In *High Performance Computational Systems Biology, 2009. HIBI '09. International Workshop on*, pages 51–58, Oct 2009.

[17] Simon Green. Particle simulation using cuda, May 2010.

[18] M. Harris, S. Sengupta, and J.D. Owens. Parallel prefix sum (scan) with cuda. *GPU Gems*, 3(39):851–876, 2007.

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200030

High Performance Eigenvalue Solver for Hubbard Model: Tuning Strategies for LOBPCG Method on CUDA GPU

Susumu YAMADA [a,1], Masahiko MACHIDA [a] and Toshiyuki IMAMURA [b]
[a] *Center for Computational Science & e-Systems, Japan Atomic Energy Agency*
[b] *RIKEN Center for Computational Science*

Abstract. The exact diagonalization is the most accurate approach for solving the Hubbard model. The approach calculates the ground state of the Hamiltonian derived exactly from the model. Since the Hamiltonian is a large sparse symmetric matrix, we usually utilize an iteration method. It has been reported that the LOBPCG method is one of the most effectual solvers for this problem. Since most operations of the method are linear operations, the method can be executed on CUDA GPU, which is one of the mainstream processors, by using cuBLAS and cuSPARSE libraries straightforwardly. However, since the routines are executed one after the other, cached data can not be reused among other routines. In this research, we tune the routines by fusing some of their loop operations in order to reuse cached data. Moreover, we propose the tuning strategies for the Hamiltonian-vector multiplication with shared memory system in consideration of the character of the Hamiltonian. The numerical test on NVIDIA Tesla P100 shows that the tuned LOBPCG code is about 1.5 times faster than the code with cuBLAS and cuSPARSE routines.

Keywords. LOBPCG method, CUDA GPU, CUDA Fortran, cuSPARSE, cuBLAS, Hubbard model, quantum lattice systems

1. Introduction

The Hubbard model[1][2] has attracted a tremendous number of physicists since the model exhibits a lot of interesting phenomenon such as High-T_c superconductivity. When we solve the ground state (the smallest eigenvalue and the corresponding eigenvector) of the Hamiltonian derived from the model, we can understand its properties. Therefore, a lot of computational methods for solving this problem have been proposed. The most accurate one is the exact diagonalization which directly solves the ground state of the Hamiltonian derived exactly from the model. Since the Hamiltonian is a huge sparse symmetric matrix, we usually solve the eigenvalue problem with an iteration method, such as the Lanczos method[3], the LOBPCG method[4][5], and so on.

[1]Corresponding Author: Japan Atomic Energy Agency, 178-4 Wakashiba, Kashiwa, Chiba, 277-0871, Japan; E-mail: yamada.susumu@jaea.go.jp.

The graphics processing unit (GPU), which is one of the mainstream processors, achieves an excellent performance with regular data access pattern. Since most of operations of the LOBPCG method are linear ones, the method can be executed on CUDA GPU by using cuBLAS routines[6]. Moreover, since the Hamiltonian can be decomposed into three matrices, whose non-zero elements are arranged regularly, by considering its physical property[7][8], we have proposed the code for the Hamiltonian-vector multiplication using the non-zero patterns in the three matrices. We reported in [8] that the code was faster than cuSPARSE routines[9] on CUDA 4.0. However, cuSPARSE routines have been tuned, and then, nowadays they are faster than our proposed codes (see Table 1).

In this research, we focus on the shared memory whose access speed is much faster than the local global memory's. And then, we propose new strategy to store the matrix data in the shared memory when performing Hamiltonian-vector multiplication. Moreover, we fuse some linear operations, which can be calculated using cuBLAS routines, to improve the cache performance.

The rest of the paper of structured as follows. In Section 2, we briefly introduce the algorithm of the Hamiltonian-vector multiplication and its conventional calculation strategy. And we propose the tuning strategies for the multiplication using the shared memory system on CUDA GPU. Section 3 presents the tuning strategies for other operations of the LOBPCG method. Section 4 shows the result of numerical test on NVIDIA Tesla P100. A summary and conclusion are given in Section 5.

2. Tuning strategy for Hamiltonian-vector multiplication

2.1. Hamiltonian-vector multiplication

The Hamiltonian of the Hubbard model is given as

$$H = -t \sum_{i,j,\sigma} c^{\dagger}_{j\sigma} c_{i\sigma} + \sum_{i} U_i n_{i\uparrow} n_{i\downarrow}, \tag{1}$$

where t is the hopping parameter from a site to another one and U_i is the repulsive energy for one-site double occupation of two fermion the i-th site. Quantities $c_{i,\sigma}$, $c^{\dagger}_{i,\sigma}$ and $n_{i,\sigma}$ are the annihilation, the creation, and the number operator of a fermion with pseudo-spin σ on the i-th site, respectively. When we solve the ground state of the Hamiltonian, we can understand the property of the model.

Here, the Hamiltonian is a huge sparse symmetric matrix, therefore, we usually utilize an iteration method, such as the Lanczos method, the LOBPCG method, and so on. Since the most time-consuming operation of the solvers is the matrix-vector multiplication, it is crucial to tune the Hamiltonian-vector multiplication. Therefore, the storage formats of the Hamiltonian and the vector are crucial for high performance computing. Here, the multiplication Hv can be split as

$$Hv = Dv + (I_{\downarrow} \otimes A_{\uparrow})v + (A_{\downarrow} \otimes I_{\uparrow})v, \tag{2}$$

where $I_{\uparrow(\downarrow)}$, $A_{\uparrow(\downarrow)}$, and D are the identity matrix, a sparse symmetric matrix derived from the hopping of an up-spin (a down-spin), and a diagonal matrix from the repulsive energy, respectively[10]. The multiplication (2) can be represented as

```
      i=(blockIdx%x-1)*blockDim%x+threadIdx%x
      j=(blockIdx%y-1)*blockDim%y+threadIdx%y
      ix=threadIdx%x; iy=threadIdx%y
      V_s(ix,iy)=V_new(i,j)
      call syncthreads()
!!
      i=(blockIdx%y-1)*blockDim%x+threadIdx%x
      j=(blockIdx%x-1)*blockDim%y+threadIdx%y
      ix=threadIdx%x; iy=threadIdx%y
      do k=iru(i), iru(i+1)-1
        V_s(iy,ix)=V_s(iy,ix)+Au(k)*V_r(j,icu(k))
      enddo
      call syncthreads()
!!
      i=(blockIdx%x-1)*blockDim%x+threadIdx%x
      j=(blockIdx%y-1)*blockDim%y+threadIdx%y
      ix=threadIdx%x; iy=threadIdx%y
      V_new(i,j)=V_s(ix,iy)
```

(a) $A_{\uparrow}V$

```
      i=(blockIdx%y-1)*blockDim%x+threadIdx%x
      j=(blockIdx%x-1)*blockDim%y+threadIdx%y
      do k=ird(j), ird(j+1)-1
        V_new(i,j)=V_new(i,j)+V(i,icd(k))*Ad(k)
      enddo
```

(b) $VA_{\downarrow}^{T}$

Figure 1. Schematic CUDA Fortran code of $A_{\uparrow}V$ and $VA_{\downarrow}^{T}$. The data of V and `V_r` are stored in column-major order and row-major one, respectively. Here, `V_s` is the shared memory array. The non-zero elements of the matrices $A_{\uparrow}$ and $A_{\downarrow}$ are stored in the CRS format, that is, the vectors `A*`, `ic*`, and `ir*` store the the values of non-zero elements, the column indexes of the elements, and the indexes where each row starts.

$$V_{i,j}^{new} = \bar{D}_{i,j}V_{i,j} + \sum_{k=1}^{m} A_{\uparrow i,k}V_{k,j} + \sum_{k=1}^{n} V_{i,k}A_{\downarrow k,j} \tag{3}$$

where the subscript i,j of the matrix is represented as the (i,j)-th element and V and $\bar{D}$ are constructed from the elements of the vector v and the diagonal elements of the matrix D in consideration of the physical property of the Hubbard model, respectively[10].

2.2. Conventional multiplication strategy

When the data of the matrix V are stored in column-major order, we can execute the multiplication (2) with contiguous memory access on CUDA Fortran by the following:

1. $V_{new} \leftarrow$ elementwise product of $\bar{D}$ and V,
2. $V_r \leftarrow V$ (row-major $\leftarrow$ column-major (transpose)),
3. $V_{new} \leftarrow V_{new} + A_{\uparrow}V_r$, (see Fig.1 (a))
4. $V_{new} \leftarrow V_{new} + VA_{\downarrow}^{T}$. (see Fig.1 (b))

On the other hand, we can execute the multiplication (2) with cuSPARSE routines as follows:

1. $V_1 \leftarrow$ elementwise product $\bar{D}$ and V

```
      real(8), shared :: au_s(ndim)
      integer, shared :: icu_s(ndim)
!
      i =(blockIdx%x-1)*blockDim%x + threadIdx%y
      i0=(blockIdx%x-1)*blockDim%x + 1
      k =iru(i)-1
      k0=k-iru(i0)
      k1=iru(i+1)-iru(i)
      do l=0,k1-1,blockDim%x
        if (threadIdx%x+l.le.k1) then
          au_s(k0+l+threadIdx%x)=Au(k+l+threadIdx%x)
          icu_s(k0+l+threadIdx%x)=icu(k+l+threadIdx%x)
        endif
      enddo
!
      call syncthreads()
```

Figure 2. Schematic CUDA Fortran code for storing the data of the matrix $A_\uparrow$ the shared memory. Since we store the matrix using the CRS format, therefore, the target vectors are `Au` and `icu`, which store the values of non-zero elements and the column indexes of the elements. In this code, the built-in variable `blockDim%y` should be equal to `blockDim%x`. Moreover, we set the value `blockDim%x` so that the coalescing access can be realized for not only this operation but also the matrix-vector multiplication $A_\uparrow V$.

2. $V_1 \leftarrow V_1 + A_\uparrow V$ (using "cusparseDcsrmm"),
3. $V_2 \leftarrow A_\downarrow V^T$ (using "cusparseDcsrmm2"),
4. $V_{new} \leftarrow V_1 + (V_2)^T$ (transpose and addition).

It was reported in [8] that the former algorithm (our algorithm) was faster than the latter (cuSPARSE) on CUDA 4.0. However, the cuSPARSE routines have been tuned, consequently, they are nowadays faster than our conventional method (see Table 1).

2.3. Tuning strategies by considering memory access

When the codes shown in Fig. 1 are executed on GPU, all threads in a block requires the same data of the matrices $A_\uparrow$ and $A_\downarrow$. In the conventional strategy, since the data are stored in the global memory, each thread has to access them individually. On the other hand, GPU has the shared memory system which can be accessed by all threads even faster than the global one. And, the target data can be stored in the shared memory just by accessing the data stored in the global memory by any one thread, not all threads. Therefore, it is expected that the performance improves by storing data of the matrices on the shared memory. However, since the size of the shared memory on GPU is very small, all data can not be stored. Then, when executing a thread block, we store only data required for the calculation in the shared memory (see Fig. 2).

Moreover, we fuse a do-loop of the elementwise product of $\bar{D}$ and V with that of $VA_\downarrow^T$ to improve the cache performance. Table 1 shows the elapsed time for the multiplication on GPU system in Japan Atomic Energy Agency (see Table 2). The result indicates that the tuned code is about 1.3 times faster than cuSPARSE routines on a recent GPU system.

Table 1. Elapsed time of Hamiltonian-vector multiplication on GPU system in Japan Atomic Energy Agency (see Table 2). The target Hamiltonian is derived from 2-dimensional (4×4-site) Hubbard model with 7 up-spins and 7 down-ones. In the tuned code, the elementwise product of $\bar{D}$ and V is fused with the multiplication $VA_{\downarrow}^{T}$, therefore the table indicates the sum of their elapsed times.

	Elapsed time (msec)		
	Conventional	cuSPARSE	Tuned
Elementwise product of $\bar{D}$ and V	5.66	5.66	13.44
$VA_{\downarrow}^{T}$	28.85	9.06	
$A_{\uparrow}V$	33.70	19.07	13.11
Transpose (and addition)	4.50	7.93	4.50
Total	72.71	41.72	31.05

Dimension of $A_{\uparrow}(A_{\downarrow})$: 11440
Number of non-zero elements of $A_{\uparrow}(A_{\downarrow})$: 144144
Dimension of Hamiltonian: 130873600

Table 2. Details of GPU system in Japan Atomic Energy Agency.

Processor	Intel Xeon E5-2680 v4
GPU	NVIDIA Tesla P100
Fortran Compiler	pgfortran 17.1-0
CUDA Version	8.0
Compile option	`-O3 -Mcuda=6.0 -lcublas -lcusparse -llapack -lblas`

3. Tuning strategies for other operations of LOBPCG method

In this section, we propose the tuning strategies for operations other than the matrix-vector multiplication of the LOBPCG method shown in Fig. 3.

First, we focus on the two 3×3-dimensional symmetric matrices (S_A and S_b in Fig. 3). In order to construct them, we have to calculate ten inner product operations using six vectors[2]. These operations can be realized by executing cuBLAS routine `cublasddot` ten times. However, since these operations are executed one after the other, we can not reuse cached data which were used in other operations. Therefore, we fuse ten operations and store the data in the shared memory to improve cache performance. The most important operation in an inner product on GPU is sum-reduction. When the shared memory system is used appropriately, the operation can be executed efficiently on GPU[11]. However, the shared memory is too small to store all data. Therefore, we decompose the vectors so that we can store the decomposed data in the shared memory, and we calculate partial sums of ten inner products using the decomposed vectors (see Fig. 4). After that, we calculate the global sums of partial sums using the shared memory system. The elapsed time using cuBLAS and our proposed code for the inner product operations on NVIDIA Tesla P100 for the same problem in Table 1 are 33.29 msec and 25.53 msec, respectively.

Moreover, Fig. 5 shows the operations to update the vectors x, p, X, P, and w in the LOBPCG method. Each operation can be realized using a cuBLAS routine. On the other

[2]Twelve inner products are required to construct the two matrices. However, since two vectors w and p are normalized, there is no need to calculate the two inner products (w,w) and (p,p).

$\boldsymbol{x}_0 :=$ an initial guess, $\boldsymbol{p}_0 := 0$
$\boldsymbol{x}_0 := \boldsymbol{x}_0/\|\boldsymbol{x}_0\|$, $X_0 := A\boldsymbol{x}_0$, $P_0 := 0$, $\mu_{-1} := (\boldsymbol{x}_0, X_0)$, $\boldsymbol{w}_0 := X_0 - \mu_{-1}\boldsymbol{x}_0$
do k=0, ... until convergence
$W_k := A\boldsymbol{w}_k$
$S_A := \{\boldsymbol{w}_k, \boldsymbol{x}_k, \boldsymbol{p}_k\}^T \{W_k, X_k, P_k\}$
$S_B := \{\boldsymbol{w}_k, \boldsymbol{x}_k, \boldsymbol{p}_k\}^T \{\boldsymbol{w}_k, \boldsymbol{x}_k, \boldsymbol{p}_k\}$
Solve the smallest eigenvalue μ and the corresponding vector $\boldsymbol{v}$,
$S_A\boldsymbol{v} = \mu S_B\boldsymbol{v}$, $\boldsymbol{v} = (\alpha, \beta, \gamma)^T$.
$\mu_k := (\mu + (\boldsymbol{x}_k, X_k))/2$
$\boldsymbol{x}_{k+1} := \alpha\boldsymbol{w}_k + \beta\boldsymbol{x}_k + \gamma\boldsymbol{p}_k$, $\boldsymbol{x}_{k+1} := \boldsymbol{x}_{k+1}/\|\boldsymbol{x}_{k+1}\|$, $\boldsymbol{p}_{k+1} := \alpha\boldsymbol{w}_k + \gamma\boldsymbol{p}_k$, $\boldsymbol{p}_{k+1} := \boldsymbol{p}_{k+1}/\|\boldsymbol{p}_{k+1}\|$
$X_{k+1} := \alpha W_k + \beta X_k + \gamma P_k$, $X_{k+1} := X_{k+1}/\|\boldsymbol{x}_{k+1}\|$, $P_{k+1} := \alpha W_k + \gamma P_k$, $P_{k+1} := P_{k+1}/\|\boldsymbol{p}_{k+1}\|$
$\boldsymbol{w}_{k+1} := X_{k+1} - \mu_k\boldsymbol{x}_{k+1}$, $\boldsymbol{w}_{k+1} := \boldsymbol{w}_{k+1}/\|\boldsymbol{w}_{k+1}\|$
enddo

Figure 3. Algorithms of LOBPCG method for the matrix A. Here, X, P, and W mean the vectors multiplied by the matrix A, that is, Ax, Ap, and Aw, respectively.

```
real(8), shared :: V_s(128,3)
!!
i=2*(blockIdx%x-1)*blockDim%x+threadIdx%x
ith=threadIdx%x
ibl=blockIdx%x
!!
V_s(ith,1)=x(i)*x(i)+x(i+128)*x(i+128)
V_s(ith,2)=x(i)*w(i)+x(i+128)*w(i+128)
V_s(ith,3)=w(i)*w(i)+w(i+128)*w(i+128)
call syncthreads()
!!
do k=1,3
 do j=6,0,-1
  n=2**j
  if (ith.le.n) V_s(ith,k)=V_s(ith,k)+V_s(ith+n,k)
  call syncthreads()
 enddo
enddo
v(ibl,1)=V_s(1,1);v(ibl,2)=V_s(1,2);v(ibl,3)=V_s(1,3);
```

Figure 4. Schematic CUDA Fortran code for calculating partial sums ($v(*,1)$, $v(*,2)$, and $v(*,3)$) of three inner products (x,x), (x,w), and (w,w) from two vectors x and w using the shared memory array `V_s`. Here, the code is executed using partitioned vectors whose length is 256, that is, the built-in variable `blockDim%x` is set as 128. After this calculation, the global sum is calculated using the partial ones. In actual execution of the LOBPCG method, we use the code extended with a similar strategy for calculating ten inner products using six vectors.

hand, when we consider the data dependencies, we can replace all loops of the operations with one loop as shown in Fig. 6. In addition, it is necessary to normalize vectors p, P, w and W using $p_{norm}(= ||p||)$ and $w_{norm}(= ||w||)$ before the inner product operations mentioned above, because the more the iteration converges, the smaller the norms of p and w become[3]. The normalization can be executed using a routine `cublasdscal`, but their loops can be also combined with the fused loop of the inner product operations

[3]It is also necessary to normalize the vectors x and X. However, the norm of the vector x is theoretically 1, therefore, these normalization are executed after calculating S_A and S_B. And we correct the corresponding elements of S_A and S_B in accordance with these normalization.

```
p ← γp (cublasdscal);      P ← γP (cublasdscal)
p ← p + αw (cublasdaxpy); P ← P + αW (cublasdaxpy)
x ← βx (cublasdscal);      X ← βX (cublasdscal)
x ← x + p (cublasdaxpy);   X ← X + P (cublasdaxpy)
w ← X (cublasdcopy);       w ← w − λx (cublasdaxpy)
p_norm ← ||p|| (cublasdnrm2); w_norm ← ||w|| (cublasdnrm2)
```

Figure 5. Operations to update vectors and calculate norm of vectors in LOBPCG method. All loops of the operations, which can be realized using cuBLAS routines, can be fused into one loop by considering data dependencies.

```
real(8), shared :: V_s(128,2)
!!
i=2*(blockIdx%x-1)*blockDim%x+threadIdx%x
ith=threadIdx%x
ibl=blockIdx%x
!!
p(i)=γ*p(i)+α*w(i); p(i+128)=γ*p(i+128)+α*w(i+128);
P(i)=γ*P(i)+α*W(i); P(i+128)=γ*P(i+128)+α*W(i+128);
x(i)=β*x(i)+p(i); x(i+128)=β*x(i+128)+p(i+128);
X(i)=β*X(i)+P(i); X(i+128)=β*X(i+128)+P(i+128);
w(i)=X(i)-λx(i); w(i+128)=X(i+128)-λx(i+128);
V_s(ith,1)=p(i)*p(i)+p(i+128)*p(i+128)
V_s(ith,2)=w(i)*w(i)+w(i+128)*w(i+128)
call syncthreads()
!!
do k=1,2
 do j=6,0,-1
  n=2**j
  if (ith.le.n) V_s(ith,k)=V_s(ith,k)+V_s(ith+n,k)
  call syncthreads()
 enddo
enddo
v(ibl,1)=V_s(1,1);v(ibl,2)=V_s(1,2);
```

Figure 6. Schematic CUDA Fortran code for fusing all operations shown in Fig. 5. We calculate the partial sums ($v(*,*)$ and $v(*,2)$) of two inner products (p,p), and (w,w) using the shared memory array `V_s`. Here, we set the built-in variable `blockDim%x` as 128. After this calculation, we calculate the two global sums using the partial ones, and then, the square root of them.

mentioned above. It is expected that these loop fusion operations improve the cache performance and realize speedup.

4. Numerical test

In this section, we examine the performance of the LOBPCG method for the Hubbard model. We solve the ground state (the minimal eigenvalue and the corresponding eigenvector) of the eigenvalue problem derived from a 2-dimensional (4×4-site) model with 7 up-spins and 7 down-ones using the LOBPCG method on the GPU system, whose details are shown in Table 2. Table 3 shows the number of the iterations, the elapsed time, and the performance. The result indicates that the conventional method is slower than that using cuSPARSE routines. Moreover, it is confirmed that the performance improves by par-

Table 3. Elapsed time and performance for exact diagonalization on NVIDIA Tesla P100. The target Hamiltonian is the same as Table 1. Here, the Hamiltonian-vector multiplication is executed using the conventional code, cuSPARSE, and the tuned one. Moreover, other operations are execute using cuBLAS and the tuned code.

Multiplication	Conventional	cuSPARSE	Tuned	Tuned
Others	cuBLAS	cuBLAS	cuBLAS	Tuned
Number of iterations	164	164	164	164
Elapsed time (sec)	30.24	24.90	23.12	16.57
Performance (GFLOPS)	69.0	83.8	90.3	125.9

tially storing the matrix elements on the shared memory and its performance is superior to cuSPARSE's one. And then, when other operations are also tuned, the code achieves speedup of 1.5 times faster than the code using cuBLAS and cuSPARSE routines.

5. Conclusions

We have proposed the tuning strategy using the shared memory for Hamiltonian-vector multiplication on the exact diagonalization method for the Hubbard model for the CUDA GPU. Since the size of the shared memory is very small, we store only the matrix data required by the executing block in the shared memory. The numerical result shows that the matrix-vector multiplication using the strategy is about 1.3 times faster than that using the cuSPARSE routines. Moreover, we also tuned other linear operations of the LOBPCG method in order to reuse more cached data. Therefore, we fused some loops into one loop by considering data dependencies. At a result, it is confirmed that the LOBPCG method using the proposed tuning strategies is about 1.5 times faster than that using cuBLAS and cuSPARSE routines.

In future work, in order to examine the physical property of a large Hubbard model, we aim to realize the high performance exact diagonalization on multi-GPU systems. For this aim, we plan to investigate the tuning strategies in consideration of the effects of the data communication between GPUs and/or CPUs.

Acknowledgment

This research was partially supported by JSPS KAKENHI Grant Number 18K11345. Computations in this work were performed on GPU system in Japan Atomic Energy Agency.

References

[1] M. Rasetti, ed., The Hubbard Model: Recent Results, World Scientific, Singapore (1991)
[2] A. Montorsi, ed., The Hubbard Model, World Scientific, Singapore (1992)

[3] J.K. Cullum and R.A. Willoughby, Lanczos Algorithms for Large Symmetric Eigenvalue Computations, Vol.1: Theory, Philadelphia: SIAM, (2002)
[4] A. V. Knyazev, Preconditioned eigensolvers - An oxymoron?, Electronic Transactions on Numerical analysis 7, 104-123 (1998)
[5] A. V. Knyazev, Toward the optimal eigensolver: Locally optimal block preconditioned conjugate gradient method, SIAM J. Sci. Comput.,23, 517-541 (2001)
[6] cuBLAS. URL: https://developer.nvidia.com/cublas.
[7] T. Siro and A. Harju, Exact diagonalization of the Hubbard model on graphics processing units,Comp. Phy. Comm., 183, 1884-1889 (2012).
[8] S. Yamada, T. Imamura, and M. Machida, High Performance Eigenvalue Solver in Exact-diagonalization Method for Hubbard Model on CUDA GPU, Parallel Computing: On the road to Exascale (G. R. Joubert, et. al, Ed.), IOS Press, 361-369 (2016).
[9] cuSPARSE. URL: https://developer.nvidia.com/cusparse.
[10] S. Yamada, T. Imamura, and M. Machida, 16.447 TFlops and 159-Billion-dimensional Exact-diagonalization for Trapped Fermion-Hubbard Model on the Earth Simulator, *Proc. of SC05* (2005).
[11] M. Harris, Optimizing parallel reduction in CUDA, NVIDIA Developer Technology (2007).

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200031

Parallel Smoothers in Multigrid Method for Heterogeneous CPU-GPU Environment

Neha IYER and Sashikumaar GANESAN [1]
Department of Computational and Data Sciences,
Indian Institute of Science, Bangalore 560012, India

Abstract. Modern-day supercomputers are equipped with sophisticated graphics processing units (GPUs) along with high-performance CPUs. Adapting existing algorithms specifically to GPU has resulted in under-utilization of CPU computing power. In this respect, we parallelize Jacobi and successive-over relaxation (SOR), which are used as smoother in multigrid method to maximize the combined utilization of both CPUs and GPUs. We study the performance of multigrid method in terms of total execution time by employing different hybrid parallel approaches, viz. accelerating the smoothing operation using only GPU across all multigrid levels, alternately switching between GPU and CPU based on the multigrid level and our proposed novel approach of using combination of GPU and CPU across all multigrid levels. Our experiments demonstrate a significant speedup using the hybrid parallel approaches, across different problem sizes and finite element types, as compared to the MPI only approach. However, the scalability challenge persists for the hybrid parallel multigrid smoothers.

Keywords. Parallel multigrid method, multi-GPU, multi-core, hybrid CPU-GPU

1. Introduction

Supercomputers today are equipped with multi-core CPU and multi-GPU to gain maximum performance. A single node of the top supercomputers supports up to 64 CPU cores with multiple GPUs. To utilize such massive computing power, there has been a significant effort to adapt existing algorithms onto the GPU architecture. In general, the compute extensive tasks are off-loaded to GPU while the CPU acts as a mediator performing data transfer, launching kernel call to the GPU or waiting idly in cases of blocking device API calls. Such practices have resulted in under-utilization of available CPU cores. The pressing need to utilize the combined computing capability of both GPUs and CPUs is even more relevant in case of small scale systems like personal computers that can support up to two GPU cards and up to eight CPU cores.

One of the centrepiece tasks that demand acceleration in the scientific computing community is solving a linear system of equations that generally arise from discretization of partial differential equations (PDEs) using a numerical method such as finite element. The multigrid method is considered to be the most efficient solver for such large

[1]Corresponding Author.
E-mail addresses: iyermohan@iisc.ac.in (N. Iyer), sashi@iisc.ac.in (S. Ganesan)

Table 1. Time taken by different operations at each level of multigrid V-cycle

Level	N	Pre-Smoothing Time (msec)	Post-Smoothing Time (msec)	Restriction (msec)	Prolongation (msec)
6	2,146,689	980.0	980.0	768.0	725.0
5	274,625	119.0	118.0	97.0	92.0
4	35,937	14.0	14.0	12.0	11.0
3	4,913	1.6	1.6	1.3	1.1
2	729	0.1	0.1	0.1	0.1
1	125	0.04			

sparse system of equations, with $O(N)$ computational complexity where N is the number of unknowns. Among the key operations of multigrid method, viz. smoothing, restriction and prolongation, smoothing is significantly time-consuming. Table 1 shows the time taken by different operations in a six-level geometric multigrid V-cycle. There is a notable difference in time taken by smoothing compared to other operations for fine mesh. In this respect, our main contribution is to improve the performance of geometric multigrid solver by developing hybrid parallel smoothers that concurrently utilizes all available computing resources in heterogeneous distributed systems. The hybrid parallel smoothers are implemented in our in-house finite element package ParMooN [1].

The rest of the paper is organized as follows: Section 2 discusses relevant work to accelerate the multigrid method. Section 3 gives a brief introduction to the geometric multigrid solver, the framework of ParMooN package and the model equation used for experiments. Section 4 describes the three different hybrid parallel approaches for smoothers. The experimental results and analysis are presented in Section 5 and Section 6 concludes the paper with key takeaways and future work.

2. Related Work

The problem of concurrent utilization of different computing resources has been previously studied in the literature. In [2], a parallel Jacobi iterative algorithm has been developed to exploit the computing capability of CPU along with the accelerators Xeon-Phi and GPU on a single node. In the case of multigrid method, existing approaches have adapted the key operations of smoothing and grid transfer to GPU architecture. The performance effect of combining the MPI only implementation of smoothers and grid transfer operators with either OpenMP or accelerators has been investigated in [3]. Another approach of mapping the fine level operations of geometric multigrid V-cycle to GPU and coarse level operations to CPU has been studied in [4]. The challenges of integrating existing MPI-based finite element package FEAST with GPU accelerated multigrid solvers has been presented in [5].

3. Background and Model Problem

3.1. Geometric Multigrid Method

Geometric multigrid (GMG) method is the most efficient iterative technique for solving a system of equations derived from a structured mesh. It operates on a hierarchy of meshes

ranging from coarse to fine level l, where $l = 0, ..., L$. The fine mesh is obtained by successively refining the coarse mesh L times uniformly. A typical multigrid γ-cycle is shown in Algorithm 1. For $\gamma = 1$ and $\gamma = 2$, the cycle becomes a V-cycle and W-cycle respectively. Classical iterative methods such as Jacobi or Successive-over relaxation (SOR) is used as a smoother due to their property of quick dampening of oscillatory modes. Multigrid method further leverages this property by recursively projecting the residual onto a coarser mesh where the smooth modes appear oscillatory. Few iterations of Jacobi or SOR work effectively on the coarse mesh and the computed correction is projected back to the fine mesh and used to update the original solution.

Algorithm 1 Multigrid γ-Cycle

1: Procedure MG-CYCLE(l)
2: **if** $l == 0$ **then**
3: Solve $A_l u_l = f_l$ exactly {coarsest level}
4: **else**
5: Apply pre-smoothing α times on $A_l u_l = f_l$ with an initial guess for u_l
6: Restrict the residual to the next coarse level $f_{l-1} = R_l^{l-1}(f_l - A_l u_l)$, where R is the restriction operator from l to $(l-1)$ level
7: Set $u_{l-1} = 0$
8: **for** $j = 0$ to γ_l **do**
9: MG-CYCLE($l-1$)
10: **end for**
11: Prolongate u_{l-1} to next fine level as $\tilde{u_l} = u_l + P_{l-1}^l u_{l-1}$, where P is the prolongation operator from $(l-1)$ to l level
12: Apply post-smoothing α times on $A_l u_l = f_l$ with an initial guess as $\tilde{u_l}$
13: **end if**

3.2. Parallel Framework of ParMooN

In the parallel framework of ParMooN package [1], the input mesh is partitioned using METIS [6] software and the collection of cells is distributed across the MPI processes. Each process is allocated a sub-domain of cells on which it performs computations. Discretization of the domain leads to defining the degrees of freedom (DOFs) that constitute the unknowns. For a 3D mesh geometry, DOFs may be defined on vertices, edges, faces and in the interior of the cell based on the type of finite element used. Further, there are three types of finite element, viz. conforming, nonconforming and discontinuous type and we have considered conforming Q_1, Q_2 and nonconforming Q_1^{nc} type of finite elements as shown in Figure 1.

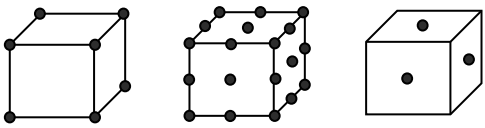


Figure 1. Conforming Q_1, Q_2 and nonconforming Q_1^{nc} finite element

Each MPI process classifies the DOFs into different types to facilitate communication of the solution with the neighbouring processes. The marking of DOFs in an individual process for a 2D domain is shown in Figure 2. The DOFs defined on the sub-domain boundaries are called Interface DOFs. These DOFs are shared by neighbouring MPI processes and the process that computes the solution at these DOFs mark the DOFs as Master DOF. All other processes sharing this DOF, mark it as Slave DOF. The DOFs that belong to the same process and are connected to Interface DOFs are called as Dependent DOFs. The Dependent DOF connected to a Slave DOF is called as Dependent1 DOF, otherwise it is called as Dependent2 DOF. The DOFs that belong to neighbouring processes but are connected to Interface DOFs are tagged as Halo DOFs. Further, Halo DOF connected to Master DOF is marked as Halo1, otherwise it is marked Halo2 DOF. The remaining DOFs owned by the process are defined as Independent DOFs.

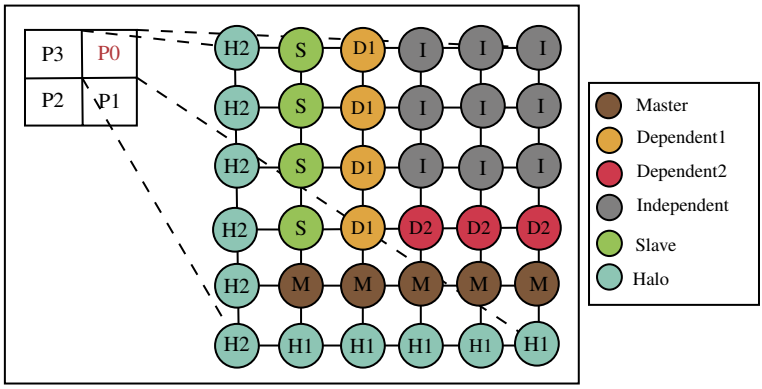


Figure 2. DOF Nomenclature using Q_1 finite element in a 2D domain for P0 process

The Master DOF in one process corresponds to Slave DOF in neighbouring processes. Similarly, Dependent1 and Dependent2 DOF correspond to Halo1 and Halo2 DOF. Hence, during each smoothing iteration, it is sufficient for each process to communicate the Master and Dependent1 DOFs with the neighbouring processes while the Dependent2 DOFs are communicated for every restriction and prolongation operations. The smoothing operation at each multigrid level including the coarsest level is performed using either Jacobi and SOR. The restriction and prolongation operators in ParMooN are as per [7] that proposed a general grid transfer operator between arbitrary finite element spaces.

3.3. Model Problem

We consider the steady-state Poisson equation with Dirichlet boundary condition on domain $\Omega \subseteq \mathbb{R}^3$ given by,

$$\begin{aligned} -\Delta u &= f \quad \text{in} \quad \Omega \\ u &= 0 \quad \text{on} \quad \partial\Omega. \end{aligned} \tag{1}$$

Here, u is the unknown scalar quantity and the source term f is chosen in such a way that the analytical solution $u = \sin(\pi x)\sin(\pi y)\sin(\pi z)$ satisfies equation (1). The equation is

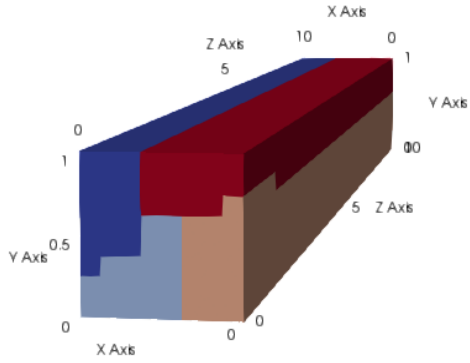


Figure 3. Cuboid domain partitioned among four MPI processes

solved in parallel by multiple MPI processes in ParMooN. The input domain is considered as a cuboid, shown in Figure 3 and is meshed using hexahedral cells. Equation (1) is discretized using standard Galerkin finite element method and subsequently the system of linear equations is solved up to a fixed precision using geometric multigrid V-cycles.

4. Implementation of Hybrid Parallel Smoother in Multigrid Method

4.1. DOF colouring

In SOR, each MPI process computes and communicates the DOFs in a pre-defined order based on the DOF type as shown in Algorithm 2. To perform the SOR iteration on GPU, the independent sets of DOFs that can be updated simultaneously must be identified. The colouring algorithm assigns a colour to each DOF such that no two connected DOFs of the same type have the same colour. A maximum of $O(d_m)$ colours will be used, where d_m is the maximum number of neighbours of the same type of DOF. The maximum neighbours of a DOF depend on the mesh structure and on the type of finite elements.

Algorithm 2 SOR Iteration at each MPI process

```
for j = 0 to n_smooth do
    Compute Master DOF
    Communicate Master DOF
    Compute Dependent1 DOF
    Communicate Dependent1 DOF
    Compute Dependent2 DOF
    Compute Independent DOF
end for
```

Once the DOFs of all types are coloured, a CUDA kernel is launched to compute the DOFs that are assigned the same colour. Kernels are launched sequentially for each type of DOF as shown in Algorithm 2. DOF colouring step is not required in case of Jacobi iterations as the updated DOFs are computed using old DOF values. Hence, all DOFs can be updated in parallel.

4.2. Sparse Matrix-Vector Multiplication on GPU

The smoothing iterations in both Jacobi and SOR, involve repeated sparse matrix-vector multiplication (SpMV). The global stiffness matrix in ParMooN is stored in compressed sparse row (CSR) format. The CUDA kernels, CSR Scalar and CSR Vector proposed in [8] for performing SpMV in CSR format on GPU, are modified to perform the smoothing iterations. The CSR Scalar approach assigns a single CUDA thread whereas CSR Vector approach assigns a warp (32 threads) to perform a matrix row and vector multiplication. The performance benefits of both approaches are studied in our experiments.

4.3. Hybrid Parallel Approaches

We have designed three major approaches that decides whether the smoothing iteration is to be performed on GPU or CPU. The approaches are described as follows.

4.3.1. GPU only

In this approach, the smoothing iterations are performed on GPU for all types of DOFs and across all levels of multigrid. The iteration proceeds in the same manner as in Algorithm 2. For each type of DOF, a CUDA kernel is launched for each colour in case of SOR otherwise a single CUDA kernel is launched in case of Jacobi.

4.3.2. CPU-GPU non-overlapping

In this approach, the smoothing iterations are performed on GPU or CPU based on the level of the multigrid. A threshold multigrid level is empirically chosen such that the iterations on and above the chosen multigrid level are performed on GPU whereas the iterations below the threshold level are performed on CPU for all types of DOFs. As the system size is large on fine levels of multigrid, the ratio of number of DOFs to the total number of colours is high, thus allowing us to exploit fine-grained parallelism on GPU. At coarse levels of multigrid, the small system size makes CPU more suitable solver as it is better optimized for memory access.

4.3.3. CPU-GPU overlapping

In the case of SOR, each iteration is divided between CPU and GPU based on the type of DOF. The Independent DOFs constitute the major chunk of the total DOFs. Also, the Independent and Dependent2 DOFs need not be communicated during an iteration and hence these two types of DOFs are offloaded to GPU. The host process concurrently computes Master and Dependent1 DOFs and handles communication with other processes. The computation of boundary Independent DOFs require the updated Dependent1 DOF values and hence it is transferred from CPU to GPU and merged with GPU solution. Similarly, the Master DOFs are transferred to GPU and merged with GPU solution. The Independent and Dependent2 DOFs are transferred to CPU and merged with CPU solution at the end of the iteration. All merging operations are performed on the GPU and transferred back to CPU whenever required. Figure 4 shows a schematic representation of the various concurrent operations on CPU and GPU for a single SOR iteration.

In the case of Jacobi, the total DOFs are partitioned between GPU and CPU using an empirically chosen ratio of 4:1. At the end of each iteration, the GPU solution is merged with CPU solution.

The kernel calls to compute Independent DOFs are performed based on the ratio of the number of Independent DOFs to colours. If the ratio is greater than the empirically chosen value then a single kernel is launched per colour else the kernel calls for different colours are merged, thus trading off synchronous update to avoid kernel launch overhead.

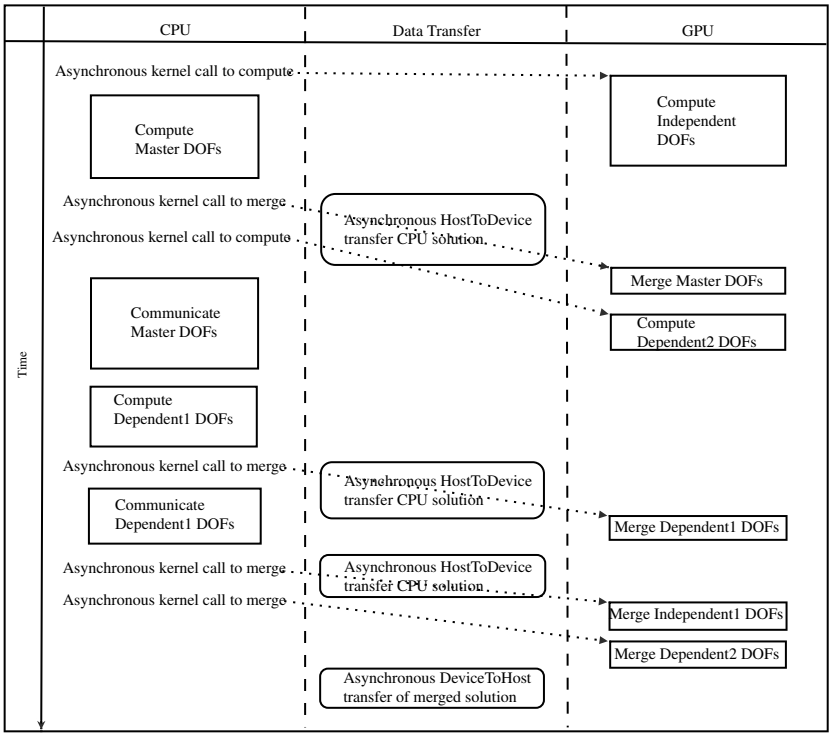


Figure 4. Schematic representation of CPU-GPU overlapping algorithm for a single SOR iteration

4.4. CUDA optimizations

Different aspects of CUDA programming optimizations incorporated are as follows:

1. Minimum data transfer between CPU and GPU: The required data structures are transferred only once before the start of the smoothing step. During smoothing, only the solution array is transferred to merge the solution.
2. Maximum shared memory usage: The CUDA kernels use two separate shared memory arrays, one to perform parallel warp-wide reduction while computing the matrix row and vector dot product and the other to store the diagonal element of each row which is needed to update the DOF during the iteration.
3. Maximum CUDA occupancy: The threads per block value is set to maximize the occupancy value of each streaming multiprocessor (SM).
4. Implicit synchronization using warp-based operation: Since the CSR Vector approach uses warp-based approach to perform SpMV, no explicit synchronization is required within the CUDA kernel call.
5. Use of Multi-Process Service (MPS): We use MPS that allows the kernel and data transfer operations from multiple MPI processes to overlap on a single GPU.

5. Experimental Results

Experiments are performed to compare the strong scaling performance of the different hybrid parallel approaches with the existing MPI only approach of ParMooN for the following three variants:

1. Types of smoother: Jacobi and SOR
2. Size of problem: Small (100 K), medium (1000 K) and large (10000 K)
3. Types of finite element: conforming Q_1, Q_2 and nonconforming Q_1^{nc}

The variable multigrid parameters used for different experiments are listed in Table 2. The experiments are executed on CRAY XC40 machine at SERC, Indian Institute of Science, Bangalore [9]. A single node in the cluster has an Intel IvyBridge 2.4 GHz based single CPU socket with 12 cores along with an NVIDIA Tesla K40 GPU card with 2880 cores and 12GB device memory. The small and medium size problem experiments are performed using four nodes with up to eight CPU cores per node and large size problems are performed using up to eight nodes with eight CPU cores per node.

Table 2. Multigrid Solver Parameters

Smoother	FE type	Levels	N	n_{pre}	n_{post}	n_{coarse}	$\omega_{smoother}$
Jacobi	Q_2	4	2000 K	5	5	10	0.67
	Q_1^{nc}	5	6000 K				
	Q_1	4	2000 K				
SOR	Q_2	4	2000 K	5	5	10	1.00
	Q_1^{nc}	5	6000 K				
	Q_1	6	17000 K				
		5	2000 K				
		4	300 K				

5.1. Scaling performance of hybrid parallel smoothers using different finite elements

Figure 5 shows the total time taken by the geometric multigrid solver using different hybrid parallel approaches for the smoother. We use CSR Vector approach for smoothing iteration on GPU. The total time includes solving as well as communication time between neighbouring processes of the multigrid solver. At low scale, the hybrid parallel approaches applied to smoothers have reduced the solving time significantly compared to MPI only approach and that too, across different finite elements. The average reduction in solving time across different finite elements for two MPI processes is 39% and 77% using Jacobi and SOR, respectively. The hybrid parallel approaches however, are not able to perform consistently at higher scales. The poor scaling performance of GPU approaches may be attributed to the decrease in problem size per process causing reduced parallelism and significant increase in data transfer and kernel launching overheads.

Among the three hybrid parallel approaches, the performance of GPU only and CPU-GPU non-overlapping are quite comparable. The GPU only performs better at low scale whereas the performance of CPU-GPU non-overlapping gets better at higher scales because of the involvement of CPU at coarse levels of multigrid. CPU-GPU overlapping approach performs slightly poorer at low scales since the CPU workload is higher and hence takes more time to complete the computation as compared to GPU. However, the performance matches with other approaches as the scale increases.

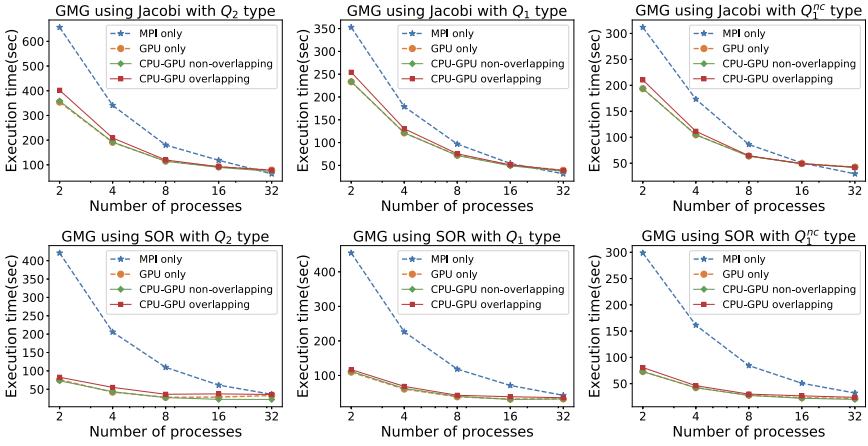


Figure 5. Scaling performance of GMG solver using Jacobi and SOR smoothers for conforming Q_1, Q_2 and nonconforming Q_1^{nc} finite elements for medium size problems (1000 K)

5.2. Performance of hybrid parallel smoothers across different problem sizes

The performance trend of the hybrid parallel approaches is tested across different problem sizes. Table 3 shows the speedup obtained across three different problem sizes of 300 K(small), 2000 K(medium) and 17000 K(large) with SOR smoother and Q_1 finite element. For small problem size, the performance slowly degrades with an increasing number of processes using hybrid parallel smoother. The speedup achieved using each of the approaches increases on increasing the problem size. The speedup trend for GPU only and CPU-GPU non-overlapping is almost comparable. The CPU-GPU non-overlapping results in better speedup compared to GPU only, as the scale of processes increases.

Table 3. Speedup of hybrid parallel approaches to MPI only approach across different problem sizes

N	**GPU only**				**CPU-GPU overlapping**				**CPU-GPU non-overlapping**			
	Number of processes				Number of processes				Number of processes			
	4	8	16	32	4	8	16	32	4	8	16	32
300 K	1.97	1.26	0.72	0.31	1.20	1.15	0.71	0.31	1.38	1.47	0.93	0.42
2000 K	3.73	3.09	2.25	1.31	3.17	2.62	1.80	1.16	3.61	2.99	2.32	1.32
17000 K	3.89	4.17	3.77	3.05	3.90	3.69	2.96	2.41	4.30	4.19	3.80	3.12

5.3. Performance gain using CSR Vector

The CSR Scalar and CSR Vector approaches are tested for conforming Q_1, Q_2 and nonconforming Q_1^{nc} finite elements using SOR smoother. Figure 6 shows the speedup achieved using CSR Vector compared to CSR Scalar using GPU only approach.

CSR Vector performs better across all the considered finite elements. The higher-order finite element particularly Q_2 type benefit more ($\sim$ 3 times) using the CSR Vector approach as there are more number of non-zeroes in each matrix row thus exploiting fine-grained parallelism. This reduction in solving time can be leveraged to compensate for higher communication time observed in general for higher-order finite elements.

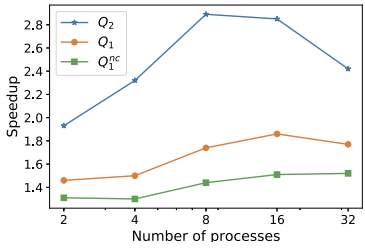


Figure 6. CSR Vector and CSR Scalar comparison for different finite elements

6. Conclusions

We have implemented and analyzed three different hybrid parallel approaches for multigrid smoother. The GPU only and CPU-GPU non-overlapping approaches give better speedup in certain scales compared to MPI only, but fail to utilize all computing resources simultaneously in a heterogeneous distributed system. The proposed novel CPU-GPU overlapping approach overcomes this drawback and performs comparably to both GPU only and CPU-GPU non-overlapping, provided the ratio of computing speed to workload is balanced between CPU and GPU. Individually, the studied approaches shows poor scalability. The GPU only gives good performance benefits at fine mesh having large problem size. The CPU-GPU overlapping works well on the intermediate mesh where there is better load balancing. On the coarse mesh, MPI only works best with small problem size. This leads us to explore the use of a combination of different approaches to further optimize the performance of the multigrid method. In future, we have planned to improve the scaling behaviour by deriving a heuristic to switch between the hybrid parallel approaches based on the mesh size and the number of MPI processes.

References

[1] S. Ganesan, V. John, G. Matthies, R. Meesala, A. Shamim, and U. Wilbrandt, "An object oriented parallel finite element scheme for computations of PDEs: Design and implementation," in *2016 IEEE 23rd International Conference on High Performance Computing Workshops (HiPCW)*, pp. 106–115, IEEE, 2016.

[2] S. Contassot-Vivier and S. Vialle, "Algorithmic scheme for hybrid computing with CPU, Xeon-Phi/MIC and GPU devices on a single machine," *Parallel Computing: On the Road to Exascale*, vol. 27, p. 25, 2016.

[3] M. Wlotzka and V. Heuveline, "Energy-efficient multigrid smoothers and grid transfer operators on multicore and GPU clusters," *Journal of Parallel and Distributed Computing*, vol. 100, pp. 181–192, 2017.

[4] N. Sakharnykh, "High-Performance Geometric Multi-Grid with GPU Acceleration," Feb. 2016. https://devblogs.nvidia.com/high-performance-geometric-multi-grid-gpu-acceleration/.

[5] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. S. McCormick, H. Wobker, C. Becker, and S. Turek, "Using gpus to improve multigrid solver performance on a cluster," *International Journal of Computational Science and Engineering*, vol. 4, no. 1, p. 36, 2008.

[6] G. Karypis and V. Kumar, "METIS – Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0," tech. rep., 1995.

[7] F. Schieweck, "A general transfer operator for arbitrary finite element spaces," 2000.

[8] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," tech. rep., Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008.

[9] "SahasraT." http://www.serc.iisc.ac.in/cray-xc40-named-as-sahasrat/.

This page intentionally left blank

Load Balancing Methods

This page intentionally left blank

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200033

Progressive Load Balancing in Distributed Memory

Mitigating Performance and Progress Variability in Iterative Asynchronous Algorithms

Justs ZARINS [a,1], Michèle WEILAND [b]
[a] *School of Informatics, University of Edinburgh, UK*
[b] *EPCC, University of Edinburgh, UK*

Abstract. System performance variability is a significant challenge to scalability of tightly-coupled iterative applications. Asynchronous variants perform better, but an imbalance in progress can result in slower convergence or even failure to converge, as old data is used for updates. In shared memory, this can be countered using progressive load balancing (PLB). We present a distributed memory extension to PLB (DPLB) by running PLB on nodes and adding a balancing layer between nodes. We demonstrate that this method is able to mitigate system performance variation by reducing global progress imbalance 1.08x–4.05x and time to solution variability 1.11x–2.89x. In addition, the method scales without significant overhead to 100 nodes.

Keywords. asynchronous algorithm, load balancing, performance variability, iterative algorithm, system noise

1. Introduction

With the ever increasing scale of high performance computing systems, there comes an array of new challenges. Technical, architectural and economic hurdles need to be overcome in order to build and deploy an exascale machine. However, creating the right hardware is only half the answer; software that can run on it efficiently is an essential part.

Any algorithm or application aiming to run on millions of parallel threads, which may be running at varying speeds, must be able to cope with performance variability. In such a scenario, tightly synchronising algorithms are not a suitable choice. Instead, we consider asynchronous or "chaotic" algorithms [1] which can make progress with stale data if some other thread has stalled. This allows for greater flexibility in adjusting to performance variability. Nevertheless this does not mean variability can be ignored completely. It still affects time required to reach the solution, but in a more complex manner than with synchronous algorithms. Therefore it is interesting to consider load balancing in the context of asynchronous algorithms.

[1]Corresponding Author: E-mail: j.zarins@ed.ac.uk

Note that in this paper we are *not* considering task based parallelism, where asynchrony refers to replacing global synchronisation with point-to-point synchronisation to satisfy data dependencies.

A recent approach [2] called *progressive load balancing* (PLB) was introduced to address the asynchronous context. The method was shown to be able to effectively mitigate the effect of a slow core in a shared memory environment. PLB achieves this by periodically moving work between CPU cores, not in order to equalise iteration rates, but to bound progress imbalance; it is balancing load over time, not instantaneously. In the present paper we build upon PLB and extend it to the distributed memory setting. Specifically, this paper makes the following contributions:

- A description of an implementation strategy extending PLB to distributed memory.
- An evaluation of the application of distributed PLB (DPLB) to balance the load in an iterative asynchronous algorithm.

2. Background

As we move towards exascale computing, synchronisation in applications is an increasingly important issue. Performance of individual cores, sockets and entire nodes that, on paper are identical, is in fact variable. This is due to factors such as energy efficiency and temperature management [3–5], random OS noise [6], and network latency variation [7]. Increasing the number of components that an application is run on also increases the likelihood that performance variation will be encountered; this is an increasing issue when considering exascale [8]. Synchronisation in applications within this hardware context results in large loss of efficiency, even running at the rate of the slowest component. While static load balancing can help, ultimately it is of limited use because the machine's performance can change at runtime.

Given the efficiency limitations of large scale synchronisation, it is natural to consider algorithms that do not rely on strict synchronous execution. In asynchronous algorithms, data synchronisation points are removed to allow the use of stale data if other workers in the system have not made progress due to stalls. There exist various iterative convergent algorithms where this is possible [1, 9–12]. Performance in asynchronous algorithms is generally dictated by a tradeoff between iteration rate and convergence rate. The former is usually improved by going asynchronous, but the latter may suffer because of the use of stale values, at the extreme resulting in failure to converge [13]. It follows that system performance variability is still a concern because it results in "progress variability".

The issue of progress variability in asynchronous algorithms has previously been tackled using a load balancing approach called progressive load balancing [2]. It is framed as a load balancing method specific to asynchronous algorithms since they do not require balance to be instantaneous. Instead, balancing is done over time by effectively swapping iteration rates of different problem subdomains. In other words, balance is in a state of dynamic (not static) equilibrium; update rates of problem subdomains keep changing but the difference in number of updates between subdomains is bounded. This leads to a dynamically controlled level of asynchrony in the system, which was shown to be effective at dealing with performance variation in shared memory.

In more detail, the global problem domain (e.g. a 2D grid for solving Jacobi's algorithm in 2 dimensions) can be split into more subdomains than there are threads. Then each thread will have multiple subdomains to continually update. The update rate per subdomain is inversely proportional to the number of subdomains that a thread owns. Thus the update rate of a particular subdomain can be increased or decreased by removing from or adding to the owning thread's workload respectively. PLB uses this mechanism to periodically move subdomain ownership between threads in order to limit update staleness without wasting CPU time spent on waiting for stragglers. In the resulting pattern some subdomains get updated quicker for a time, using stale values, but later the update rates are changed so that the subdomain that had raced ahead begins to iterate slower and eventually falls behind at which point the process is reversed again. In this paper we will later see how the scheme can be extended to distributed memory with a separate layer to move subdomains between nodes and the PLB mechanism to integrate the subdomains within nodes.

2.1. Related Work

The area of load balancing is an active field of research, however the majority of techniques are developed for, and applied to, synchronous algorithms and so may not transition well to asynchronous algorithms or require significant changes to the techniques. For example, work stealing is a popular and scalable method [14]. Workers process local queues of tasks and when they run out, more work is stolen from work queues of other workers. In this form it cannot be applied to asynchronous algorithms because workers in principle never run out of work and always appear busy, so they would just use continuously more stale values. Work stealing can be applied to semi-synchronous algorithms, where a maximum staleness bound is enforced so the amount of work available per worker does have a limit. However, our experiments showed that the method does not work well in the semi-synchronous case because the system soon reaches a state where there are many starved workers and not much work to steal. Hence we are focusing on techniques that have been shown to be applicable to asynchronous algorithms, which is a key criterion for us.

A few examples of load balancing of asynchronous algorithms in distributed memory exist. For instance, Bahi et al. show a load balancing algorithm applied to a 1D stencil application [15]. This algorithm sends parts of the working array from one worker to a less loaded neighbour. The algorithm is similar to PLB, however it seeks a static load balance while PLB aims to create dynamic equilibrium, which can result in good balance with coarser work adjustments. Additionally, the proposed algorithm is presented in 1 dimension only; an extension to multiple dimensions would be difficult to design and implement.

A more passive balancing approach has been applied to large scale deep learning [10]. The application uses asynchronous stochastic gradient descent as the core algorithm. Groups of synchronous workers are linked together asynchronously to update parameter servers for the model under training. This hybrid asynchronous-synchronous scheme balances statistical and hardware efficiency by tweaking the sizes and the number of synchronous groups, but cannot deal with performance variation changes at runtime.

Another strategy is to ignore performance variance itself and rather deal with the resulting staleness. There are various examples of such algorithmic corrections for stale

values applied to asynchronous stochastic gradient descent [16,17]. A limitation of these approaches is a lack of generalisation to other applications.

Our test problem in this work is asynchronous Jacobi's algorithm (see Sec. 4). This problem has been previously examined by Bethune et al. at large scale [18]. They observed large variations in numbers of iterations completed by different processes. This resulted in a significant increase in the number of iterations taken to converge. They also document a case where a single core running at half speed doubled the runtime of a 32k core synchronous run.

3. Extending PLB to Distributed Memory

While PLB was shown to be successful in a shared memory setting, for it to be truly valuable it needs to be able to scale further. In this paper we extend the method and evaluate its effectiveness in a distributed memory setting.

3.1. DPLB

To extend PLB we add a layer that moves work between nodes. This method is referred to as *distributed progressive load balancing* (DPLB). In DPLB we run PLB on each node, and add infrequent work movements across nodes. This extension is important for situations where whole nodes are affected by noise and are significantly slower than others.

The main steps in the algorithm are as follows:

1. Periodically, with a set frequency, nodes find out the average number of updates performed on other nodes.
2. The difference between the highest and lowest averages are compared to a set threshold.
3. If the difference is larger than the threshold, the least progressed node sends a randomly chosen problem subdomain to the node that has advanced the most.
4. The node that has received the subdomain assigns it one of its cores initially, but, since PLB is running on every node, the subdomain gets moved between cores as is required to balance progress on the node.

The implementation details of these steps will vary based on the problem that is being solved and the programming techniques and libraries used, but we will next explain some of the most important implementation considerations for our example case.

3.2. Implementation

In our implementation we target iterative convergent algorithms that can be parallelised by splitting the global problem domain into smaller subdomains. We also assume that data is being exchanged between the subdomains using "halos". Stencil applications match this pattern closely, however the balancing principles presented here are not limited to this class of applications.

Distributed communications are implemented mainly using MPI single sided calls. This communication paradigm is well suited to asynchronous algorithms, since it min-

Table 1. "Cirrus" test system details (www.cirrus.ac.uk)

System type	SGI ICE XA	Topology	Hypercube, 282 nodes
CPU Sockets	2	L3 cache	45 MB
CPU	Intel E5-2695	RAM per CPU	128 GB
Core count per CPU	18	Compiler	GCC 6.2
Clock	2.1 GHz	MPI library	Intel 17.0
Interconnect	FDR Infiniband	Main compilation flags	-O2

imises the need for global synchronisation. Also, the application can be more dynamic because there is no need to match specific sends and receives. Some two sided communication still exists, but only where the matching does not interfere with asynchrony. On node we use OpenMP threading.

Information gathering about work progress of nodes is done using a reduction implemented using RMA operations. Every node publishes a small data structure containing the average progress of its problem subdomains. Other nodes can query these structures with a get operation when global balance is being checked.

An important part of the implementation is moving subdomains between nodes dynamically and adjusting communication targets. To ensure scalability, it is important to avoid introducing a global bottleneck here, for example by using a centralised table of subdomain physical locations. Instead, in our implementation subdomains keep track of just their neighbours' locations. When a subdomain moves, it leaves behind a message with its new host rank (i.e. MPI rank). When its neighbours perform halo exchange, as part of the halo they also receive the message that the subdomain has moved and which is the new rank that should be queried for the desired halos.

The main component facilitating this interaction is metadata appended to halos, specifically an ID and owner rank. Upon retrieval of a halo, the metadata is checked to make sure it is as expected (initial locations of subdomains are known). If the metadata rank is not the same as the rank the halo was received from, the halo and associated subdomain have moved (the rank that does the moving changes the halo metadata to reflect the rank to which it has migrated). Once the new rank is known, an array of halo displacements is retrieved from the target rank. The array is searched to find the physical memory location of the target halo. The halo can now be retrieved and the ID checked to make sure they are correct. Only the communicating neighbours were involved in this transaction, which makes it scalable.

4. Experiments

As our test application we use Jacobi's algorithm applied to the diffusion problem in 2 dimensions. This is an iterative convergent stencil application which is often used when testing asynchronous algorithms due to its simplicity and numerical stability [1]. We use a Gaussian shaped function as the boundary condition along one edge, the others being set to zero. The problem domain is distributed across nodes in 1 dimension, along the x axis.

We used the HPC system Cirrus for our experiments. Hardware and compiler details are listed in Table 1.

While there is inherent noise and imbalance in the system, for some experiments we inject artificial noise to simulate particular scenarios in order to have repeatable experiments. Noise is generated by running an additional background thread that sleeps and busy-waits for set amounts of time. Additionally, the workers' *niceness* is set to a high value so that they have lower priority, thus yielding to the noise generating threads when active.

We chose a noise level of 40% per CPU socket (i.e. the CPU effectively runs at 60% of its normal clock frequency). This value is the mean of worst case clock frequency variations due to manufacturing variability observed in [4] when limiting node power - a factor to consider in future exascale systems with global power constraints. Also, Chunduri et al. report application runtime variability between 1.18x and 1.74x (38% on average) related to network congestion on a production system [19]. For experiments where we slow down a whole node, we chose the same level to make comparisons between experiments more direct. This can happen if both sockets are slow, the node is hot from a previous job or if there is significant network congestion.

Each experiment was repeated multiple times on different sets of nodes. Where possible, a series of experiments with differing settings (e.g. normal, normal plus balancer, normal plus balancer plus noise etc.) were repeated on the same node set so that differences between the experiments would be mainly due to algorithmic differences, instead of node conditions.

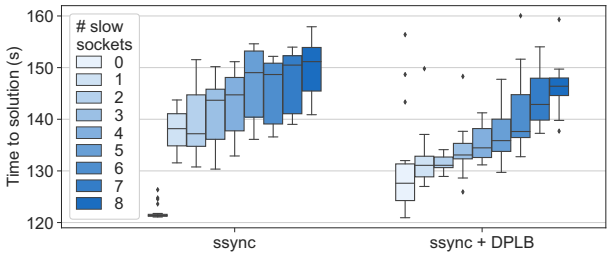
In *time to solution* (TTS) experiments the application runs until the global l_2-norm of the residual, normalised by its initial value, reaches a threshold. We set this threshold at 10^{-3}. Generally the threshold is smaller in real applications, however here we wanted to limit the total execution time and focus on performance metrics rather than the final solution.

We use a problem size of 1000 by 1000 values per core. Thus, in the 15 node experiments the global problem size is $6k$ by $90k$ and in the 100 node experiments it is $6k$ by $600k$.

5. Evaluations

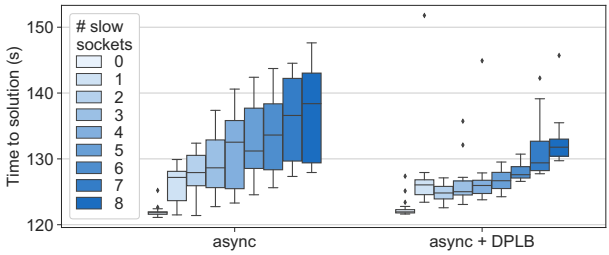
In this section we present an experimental evaluation of DPLB acting on semi-synchronous and asynchronous Jacobi. Figures 1 to 3 show time to solution results comparing performance before and after applying DPLB (each bar represents 9 to 20 data points). Less time and smaller variance is better. These figures include tables of iteration rates (in units of 1000 iterations per second per node) and staleness (most stale halo encountered). The specific settings of PLB parameters were chosen to be the same as in [2] because these were found to give good performance ($nPairs = 6$, $lowThresh = 2$, $highThresh = 6$). DPLB performs balancing across nodes every 0.5 seconds with PLB balancing each CPU socket separately every 0.001 seconds. The semi-synchronous staleness bound is set to 30 and in both the semi-synchronous and the asynchronous experiments each core initially holds 4 problem subdomains.

To test the method we run Jacobi on 15 nodes while growing the number of nodes with a slow CPU socket. In order to survey the range of possible noise scenarios, a portion of the experiments has noise placed randomly and at fixed locations. For the latter we picked "worst case" and "best case" noise placement, based on the problem that is being



	sync(4)		ssync(4) + DPLB	
# slow	stale	iter rate	stale	iter rate
0	31	10.80	34	10.75
1	31	9.86	39	10.44
2	31	9.34	39	10.38
3	31	9.25	39	10.30
4	31	9.20	40	10.18
5	31	8.97	40	10.01
6	31	9.02	39	9.96
7	31	8.90	40	9.70
8	31	8.76	39	9.49

Figure 1. Effect of DPLB on semi-synchronous Jacobi running on 15 nodes. Color indicates the number of CPU sockets running 40% slower. The table shows median values.



	async(4)		async(4) + DPLB	
# slow	stale	iter rate	stale	iter rate
0	1860	10.85	175	10.86
1	3122	10.75	2587	10.78
2	3339	10.65	2070	10.75
3	3493	10.55	2732	10.69
4	3628	10.45	2507	10.62
5	3593	10.35	2725	10.53
6	3657	10.26	2876	10.45
7	3723	10.16	3455	10.33
8	3731	10.06	3152	10.22

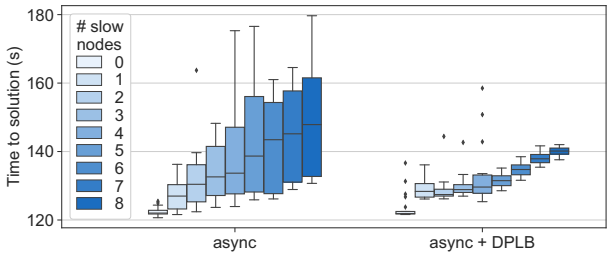
Figure 2. Effect of DPLB on asynchronous Jacobi running on 15 nodes. Color indicates the number of CPU sockets running 40% slower. The table shows median values.

solved. The initial conditions put a Gaussian shaped source in the middle of the problem domain, so updates in the middle contribute more towards reducing the residual than the edges. Thus we add noise to components that are initially responsible for the middle of the problem domain to get worst case performance and add noise to edges to get best case performance.

Figure 1 shows results for the semi-synchronous version. Without balancing, the time to solution gradually increases; we also observed instances of 200%–260% slowdown when 6, 7 or 8 sockets were noisy. DPLB mitigates the noise noticeably for all noise counts, and avoids the large outliers at higher noisy socket counts. Since progress imbalance is capped, the performance difference comes from DPLB sustaining a higher iteration rate. The balanced version's median TTS is reduced by 3–10%, except for the noiseless case where the unbalanced version is 5% faster on average. We note that the current implementation allows the staleness bound to be overstepped slightly due to subdomain updates occurring while some subdomains are being transferred between nodes.

Results for the asynchronous version can be seen in Figure 2. In the table it can be seen that iteration rate is not affected adversely by DPLB and halo staleness is reduced 1.08x–1.61x. As a result, the balanced asynchronous version converges quicker for every noise setting, with a median reduction of up to 6%. TTS of the balanced version is larger than that of the noiseless case, but this is to be expected even with perfect balancing since slow components take away the total amount of available compute power in the system. Furthermore, the worst case TTS grows at a higher rate without DPLB, which implies reduced scalability. With DPLB the worst case TTS remains mostly flat until noise is added to 5 or more sockets.

Because the asynchronous version shows better performance than the semi-synchronous version overall, we test it further by slowing down whole nodes, not just



# slow	async(4) stale	async(4) iter rate	async(4) + DPLB stale	async(4) + DPLB iter rate
0	1880	10.85	181	10.86
1	3794	10.65	3051	10.64
2	4755	10.46	1272	10.53
3	4617	10.26	1257	10.45
4	5377	10.07	1782	10.29
5	5591	9.87	1680	10.13
6	5575	9.68	1559	9.90
7	5581	9.47	1377	9.67
8	5709	9.33	1707	9.47

Figure 3. Effect of DPLB on asynchronous Jacobi running on 15 nodes. Color indicates the number of nodes running 40% slower. The table shows median values.

individual CPU sockets. This can occur if a job is assigned a hot node or if there is a lot of network communication from other jobs going through the node's links. These results can be seen in Figure 3. The overall patterns are similar to the previous case, but more pronounced. Balancing reduces median TTS by up to 6% again, but the reduction in worst case TTS and staleness is significantly higher at 1.24x–4.05x.

An important feature to emphasise is the excellent reduction in performance variability due to DPLB. Table 2 shows, for each noisy component count, the ratio of the unbalanced version's spread of TTS (distance between the boxplots' whiskers) against that of the balanced version. The last column of the table shows this ratio applied to the spread of TTS across all counts of noisy components, i.e. between the highest top whisker and lowest bottom whisker in each category. The change for the semi-synchronous code varies between 0.06x (the balanced version is more variable) and 4.00x (the balanced version is less variable). However, for the asynchronous code, balancing always reduces variance, ranging from 1.14x and 11.07x. If the number of noisy components is not set at any particular value, the balanced versions range from 1.11x to 2.89x less variant. This increased consistency in runtime is crucial for time sensitive applications, e.g. predicting the path of a hurricane using weather simulation. It is also important in cases such as application scheduling on shared compute resources, benchmarking and keeping within budget of HPC resources.

As a final test, we ran our code on 100 nodes (3600 cores) with highly variable noise settings from run to run in order to simulate a real life scenario. For each individual run we selected a random set of nodes to be noisy; the size of the set was also chosen randomly between 0 and 15. The level of slowdown on each node in the set was chosen randomly between 15% and 40%. We obtained 42 data points with the asynchronous Jacobi code and another 42 with asynchronous Jacobi plus DPLB. The results can be seen in Table 3. Both versions performed very similarly. The test problem, when increased in size, proved to be highly resilient to random noise so adding load balancing in this case did not reduce time further. However, other inputs can be more sensitive to noise and this experiment shows that DPLB has no significant overhead in this setting and it scales perfectly well.

On the whole, the results of the asynchronous algorithm with DPLB show greatly reduced variance in TTS and variability in update progress of problem subdomains. In addition, the worst case noise scenario TTS is less when DPLB is added while the best case noise scenario is slightly higher. These observations taken together indicate that smoothing noise is beneficial in the majority of the time. While reducing progress imbalance occurring in a less critical part of the problem domain results in a small increase in TTS,

Table 2. Runtime variability ratios

# slow	ssync, socket	async, socket	async, node
0	0.06	1.14	8.12
1	1.21	1.86	1.47
2	4.00	2.42	3.50
3	2.19	3.55	3.88
4	1.81	4.24	6.26
5	1.03	3.39	7.65
6	0.83	4.37	5.06
7	0.89	1.51	5.71
8	1.71	3.41	11.07
extremes	1.11	1.51	2.89

Table 3. Runtime comparison on 100 nodes

(seconds)	mean	min	max	std. dev.
async	118.1	118.2	120.9	1.3
async + DPLB	118.1	115.2	120.7	1.4

not reducing imbalance in a more critical part results in a much larger increase in TTS. On average, the risk of excessive runtime and progress imbalance of an asynchronous algorithm can be noticeably reduced with DPLB.

6. Conclusions

We have presented a method (DPLB) for applying progressive load balancing to an asynchronous algorithm in a distributed memory setting by adding periodic movement of work between nodes and running PLB on the nodes. Evaluation of DPLB showed that it is able to mitigate system performance variation by a reduction of 1.08x–4.05x in global progress imbalance and by 1.11x–2.89x in time to solution variability. We did not observe any significant overheads even when running on 100 nodes. In future work we plan to apply DPLB to other asynchronous iterative algorithms where there is scope for splitting the problem domain and moving it between computing units, for example the Schwarz method or stochastic gradient descent (SGD). This technique improves the resilience of asynchronous algorithms to noise and hence increase their value as components for meeting the exascale challenge.

Acknowledgments

This work was supported by grant EP/L01503X/1 for the University of Edinburgh School of Informatics Centre for Doctoral Training in Pervasive Parallelism (pervasiveparallelism.inf.ed.ac.uk) from the UK Engineering and Physical Sciences Research Council (EPSRC). This work used the Cirrus UK National Tier-2 HPC Service at EPCC (www.cirrus.ac.uk) funded by the University of Edinburgh and EPSRC (EP/P020267/1).

References

[1] D. Chazan and W. Miranker, "Chaotic relaxation," *Linear Algebra and its Applications*, vol. 2, no. 2, pp. 199–222, apr 1969.

[2] J. Zarins and M. Weiland, "Progressive load balancing of asynchronous algorithms," in *Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA3'17. ACM, 2017, pp. 5:1–5:9.

[3] D. Hackenberg *et al.*, "An energy efficiency feature survey of the intel Haswell processor," in *International Parallel and Distributed Processing Symposium Workshop*. IEEE, May 2015, pp. 896–904.

[4] Y. Inadomi *et al.*, "Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2015, p. 78.

[5] A. Porterfield *et al.*, "Application runtime variability and power optimization for exascale computers," in *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*. ACM, 2015, p. 3.

[6] F. Petrini, D. J. Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q," in *Supercomputing Conference*. ACM/IEEE, 2003, pp. 55–55.

[7] R. Underwood, J. Anderson, and A. Apon, "Measuring network latency variation impacts to high performance computing application performance," in *Proceedings of the International Conference on Performance Engineering*. ACM/SPEC, 2018, pp. 68–79.

[8] R. Lucas *et al.*, "DOE ASCAC Subcommittee Report February 10, 2014," 2014.

[9] H. Anzt *et al.*, "A block-asynchronous relaxation method for graphics processing units," *Journal of Parallel and Distributed Computing*, vol. 73, no. 12, pp. 1613–1626, dec 2013.

[10] T. Kurth *et al.*, "Deep learning at 15PF: Supervised and semi-supervised classification for scientific data," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. ACM, 2017, pp. 7:1–7:11.

[11] D. A. Donzis and K. Aditya, "Asynchronous finite-difference schemes for partial differential equations," *Journal of Computational Physics*, vol. 274, pp. 370–392, oct 2014.

[12] F. Magoulès, D. B. Szyld, and C. Venet, "Asynchronous optimized Schwarz methods with and without overlap," *Numerische Mathematik*, vol. 137, no. 1, pp. 199–227, Sep 2017.

[13] D. P. Bertsekas and J. N. Tsitsiklis, "Some aspects of parallel and distributed iterative algorithms – a survey," *Automatica*, vol. 27, no. 1, pp. 3–21, 1991.

[14] J. Dinan *et al.*, "Scalable work stealing," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, Nov 2009, pp. 1–11.

[15] J. M. Bahi, S. Contassot-Vivier, and R. Couturier, "Dynamic load balancing and efficient load estimators for asynchronous iterative algorithms," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 4, pp. 289–299, Apr. 2005.

[16] I. Mitliagkas *et al.*, "Asynchrony begets momentum, with an application to deep learning," in *54th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE, 2016, pp. 997–1004.

[17] S. Zheng *et al.*, "Asynchronous stochastic gradient descent with delay compensation," in *International Conference on Machine Learning*, 2017, pp. 4120–4129.

[18] I. Bethune *et al.*, "Performance analysis of asynchronous Jacobi's method implemented in MPI, SHMEM and OpenMP," *Int. J. High Perform. Comput. Appl.*, vol. 28, no. 1, pp. 97–111, Feb. 2014.

[19] S. Chunduri *et al.*, "A generalized statistics-based model for predicting network-induced variability," in *10th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS'19)*, ser. SC'19, November 2019.

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200034

Learning-Based Load Balancing for Massively Parallel Simulations of Hot Fusion Plasmas

Theresa POLLINGER [a,1] and Dirk PFLÜGER [a]
[a] *Institut für Parallele und Verteilte Systeme, Universität Stuttgart*

Abstract The sparse grid combination technique can be used to mitigate the curse of dimensionality and to gain insight into the physics of hot fusion plasmas with the gyrokinetic code GENE. With the sparse grid combination technique, massively parallel simulations can be performed on target resolutions that would be prohibitively large for standard full grid simulations. This can be achieved by numerically decoupling the target simulation into several smaller ones. Their time dependent evolution requires load balancing to obtain near optimal scaling beyond the scaling capabilities of GENE itself. This approach requires that good estimates for the runtimes exist.

This paper revisits this topic for large-scale nonlinear global simulations and investigates common machine learning techniques, such as support vector regression and neural networks. It is shown that, provided enough data can be collected, load modeling by data-driven techniques can outperform expert knowledge-based fits – the current state-of-the-art approach.

Keywords. load balancing, gyrokinetics, exascale, machine learning, sparse grid combination technique, machine learning

1. Introduction

The research on hot fusion plasmas remains a pressing topic, and understanding the relevant processes through simulation is necessary to optimize large experimental reactors such as ITER. While such devices in fusion research are being built, simulation results should always be one step ahead to assist in understanding and planning experiments [1]. This does not simply happen due to Moore's Law (and successors), because the gyrokinetic formulation of the Vlasov-Maxwell equations is at least five-dimensional, and numerical discretizations in higher dimensions suffer the so-called "curse of dimensionality". We have proposed the sparse grid combination technique to mitigate the curse of dimensionality and to gain insight into the physics of hot fusion plasmas [2] with the gyrokinetic code GENE [1]. It replaces the computationally infeasible target solution by a combination of many smaller solutions. Using our parallel combination technique framework, one can compute the partial solutions in parallel process groups, where each

[1] Corresponding Author: Theresa Pollinger, Institut für Parallele und Verteilte Systeme, Universität Stuttgart, Universitätsstr. 38, 70569 Stuttgart, Germany; Email: theresa.pollinger@ipvs.uni-stuttgart.de

process group is assigned multiple tasks, i.e., partial solutions, to solve [3]. At this point, load balancing becomes crucial, since imbalanced task distributions will lead to unnecessary idle times for thousands of processors. Previous work investigated load balancing for linear local initial-value computations with the combination technique [4]. For simulations that fully capture the global non-stationary behavior of the plasma, this is not sufficient any more. We therefore revisit this topic, in order to facilitate computations at scales that have not been attempted with the combination technique before.

Accordingly, this paper first gives a comprehensive introduction to the scientific background, i.e., global nonlinear gyrokinetic simulations with GENE and decoupling via our sparse grid combination technique framework. We discuss how load imbalances can arise by mis-estimating simulation runtimes. As the main contribution of this paper, we improve on the state of the art by introducing machine learning methods for load balancing. We follow Heene et al. [4] and try to find a good a-priori estimate of the runtime of each task. Runtime data on varying tasks is collected depending on simulation parameters and the degree of parallelization. We have compared different methods of predicting the runtimes on previously unseen grids: nearest neighbor interpolation, support vector regression and neural networks, as well as the state-of-the-art model based on expert knowledge. The latter is still a reasonable approach for our purposes, but can be outperformed by a neural-network based prediction. It follows that by collecting data early on, we can save on efficiency losses and re-initialization overhead for large simulations, while being able to extend our models as we collect more runtime data.

2. Scientific Background

2.1. Gyrokinetic Simulations with the GENE Code

The GENE code is a state-of-the-art solver for the Maxwell-Vlasov system of equations, consisting of the Maxwell equations and the Vlasov equation

$$\frac{\partial F_\sigma}{\partial t} + \frac{\mathrm{d}\mathbf{X}}{\mathrm{d}t} \cdot \nabla F_\sigma + \frac{\mathrm{d}v_\parallel}{\mathrm{d}t} \frac{\partial F_\sigma}{\partial v_\parallel} + \frac{\mathrm{d}\mu}{\mathrm{d}t} \frac{\partial F_\sigma}{\partial \mu} = 0, \tag{1}$$

which connects the plasma particle distribution function f to the electromagnetic field [5]. While f is actually located in six-dimensional phase space, the gyrokinetic transform reduces these to five, by integrating out the gyration direction of the plasma particles. The remaining cartesian dimensions for GENE's Eulerian approach are denoted by $x, y, z, v_\parallel, \mu$. The time step integration is performed through an explicit fourth-order Runge-Kutta scheme.

Even though the complexity of solving is drastically reduced by the gyrokinetic transform, the computational work required for solving the integro-differential equations still suffers the curse of dimensionality – simulations are unfeasible for high resolutions.

A simplified way of looking at the equations that GENE solves is splitting into a linear and nonlinear part

$$\frac{\partial f}{\partial t} = \mathscr{L}(f) + \mathscr{N}(f). \tag{2}$$

While previous work looked at times measured for executing only the linear part of the simulation – and for relatively low resolutions – this topic needs revisiting, now that we are dealing with nonlinear, global large-scale simulations. The nonlinear part of the model becomes dominant, and to capture the chaotic behavior it produces, higher minimal resolutions are required. As the maximum resolution is further increased, more features can be resolved [5].

At this point, we need to clearly distinguish two different effects that affect the runtime of a GENE simulation: the *time per time step* is the time needed to process one single explicit time step, which we assume to stay constant during the course of a given simulation, and *adaptive time-stepping* needed to ensure stability, which will change nonlinearly with the simulated fields [6].

This paper will focus on the *time per time step* needed for a given simulation grid. This is a reasonable restriction for scenarios where we assume that a combination takes place after all grids have progressed by one time step – more on this in the next section, which focuses on the combination technique for sparse grids.

2.2. Massively Parallel Computation with the Sparse Grid Combination Technique

Our approach to break the curse of dimensionality is the use of the sparse grid combination technique [7]. The basic idea is that we can run the simulation on many relatively coarse anisotropic grids in the index set I; the d-dimensional *level vector* $\vec{\ell} = [\ell_1, \ldots, \ell_d]$ defines each grid's resolution as $2^{\ell_i} + 1$ in dimension i. I contains all $\vec{\ell}$ in the convex hull of the simplex spanned by the minimum level $\vec{\ell}_{min}$ and the corners of the cartesian hypercube between $\vec{\ell}_{min}$ and the maximum level $\vec{\ell}_{max}$. The combination results in a sparse grid representation $f^{(c)}$

$$f^{(c)} = \sum_{\vec{\ell} \in I} c_{\vec{\ell}} f_{\vec{\ell}} \qquad , \qquad c_{\vec{\ell}} = \sum_{\vec{z} \leq \vec{1}} (-1)^{|\vec{z}|_1} \chi_I(\vec{\ell} + \vec{z}). \tag{3}$$

of the solution, defined on a finer (target) grid of resolution $\vec{\ell}_{max}$. χ_I is the characteristic function of I. For a thorough description of sparse grids and the combination technique please refer to [8].

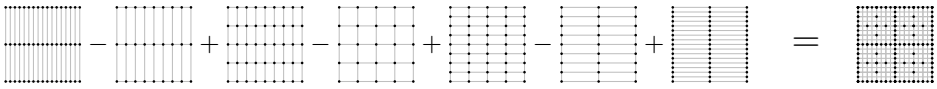


Figure 1. Schematic of the standard sparse grid combination technique with $\vec{\ell}_{min} = [2,2]$ and $\vec{\ell}_{max} = [4,4]$

Existing legacy solvers for cartesian grids can be used in a black-box fashion, and they do not need to be refactored to implement the numerical operators on the sparse grid itself. With respect to the GENE code, this means that one can start multiple instances of GENE to compute the solution to the same physical problem on one of these grids respectively [2]. We will call this combination of simulation parameters and grid resolution $\vec{\ell}$ a *task*. After the simulation has progressed by a defined time interval for each task, the solution is recombined by the use of the combination technique, cf. Fig. 1, such that the values match on all points which are shared between grids. Note that the simulation step for one task is independent of other tasks, meaning that the tasks can be processed in an embarrassingly parallel manner [3].

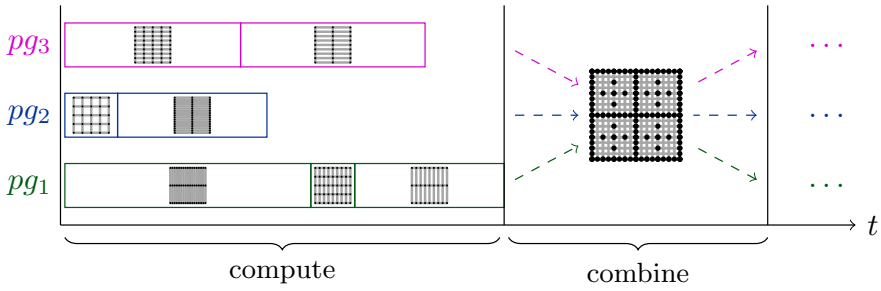


Figure 2. Possible parallel computation scheme of the grids in Fig. 1

Since the GENE simulations considered here are far too large to fit on a single node – usually requiring 20 to hundreds of GB in main memory – our C++ framework for the sparse grid combination technique employs a manager-worker scheme, where a worker consists of a whole process group using up to thousands of cores. The manager distributes *tasks* to the process groups. Currently, the framework is restricted to process groups of the same size only. Each process group will execute the assigned tasks one after the other, each until the simulation time of the next combination step is reached. Then, the grids of the tasks are updated with the results of the other tasks by way of the combination technique. This usually means that some of the process groups will have to wait for the longest-running one to finish, as illustrated by the gap in Fig. 2. If we assume the simplest set-up where every task uses the same time step, the optimization reduces to finding the best possible assignment of tasks to the process groups.

But *what is a well-balanced assignment of tasks with respect to the process groups?* This is the core question of this paper, and the following sections will present an approach to answer it.

3. Data-driven Load Modeling

Load balancing in HPC is often implemented by sophisticated domain decomposition schemes [9], or by reserving resources to different parts of the algorithm, such as different solvers [10]. We however are concerned with a particular set-up in which balance of load is asserted at a larger scale by an approach that is offered by the sparse grid combination technique. We estimate the runtime of single grids beforehand, and assign them to process groups from "longest" to "shortest", filling up idle times, cf. Fig. 2. At the same time, the black box solver may use additional load balancing internally. To the best of our knowledge, modeling the runtime of simulation time steps via machine learning techniques has not been implemented for the combination technique before.

The problem considered here is closely connected to basic scheduling algorithms. Let us assume, however, that re-assignment of a task to another process group is a costly operation that should be avoided. In our set-up, this is based on the fact that, in addition to the data on the grid, large amounts of internal simulation data are required per task – e.g., the GENE gyro matrix. If a task were to be moved, the internal data would have to be explicitly transferred or recomputed on the target process group, and the initialization of GENE can take over one hundred regular time step lengths. This is in addition to the

time required to transfer the current grid (which for high resolutions may contain a large amount of data, up to several GB).

Note that the runtime of GENE is determined by many factors which impede the use of a simple linear or performance model normal form ansatz: adding to the usual caching and communication effects, the sparsity pattern of the gyro matrix, an update algorithm with access complexity in $\mathcal{O}(n_x^2)$, and the fast fourier transform applied in y direction influence the run time heavily. But since GENE is run on many different machines, runtime data may be collected across many different machines and physical problems, in order to obtain a transferable load model at no additional computational cost.

3.1. Data Acquisition

Data was collected on the Hazel Hen supercomputer, a Cray XC40 system with Intel Haswell processors, two sockets per node, each at 12 cores. GENE was started with parameters for a scenario with adiabatic electrons at different discretizations, i.e., the level vector $\vec{\ell}$ that lives in the gyrokinetic dimensions $x, y, z, v_\parallel, \mu$. Samples were randomly selected between $\vec{\ell}_{min} = [6,4,3,4,3]$ and $\vec{\ell}_{max} = [14,8,6,8,6]$, leaving out grids that are too small to represent the underlying physics, and also those that would be larger than 2^{29} degrees of freedom in total. This was done for different levels of parallelization, for power-of-two node counts from 2^5 to 2^{15}. We will denote the processor count for each sample by 2^p. For 2^{15}, only about half as many samples were taken as for the other 2^p, since this data was quite costly to obtain. Note that previous work [4] considered grid sizes small enough to fit on $2^5 = 32$ cores on Hazel Hen, such that the higher degrees of parallelization were not a matter of discussion then.

As a simplification, the parallelization was constrained to a specific strategy depending on the level vector $\vec{\ell}$. Domain knowledge by the GENE developers is leveraged to always set a close-to-optimal parallelization: parallelize from the outer to the inner loops, i.e., first in μ direction, then in $v_\parallel$, and so on. This approach was validated on a small subset of the space (31 samples) where exhaustive parallelization tests were run: comparing the optimal runtime to the runtime obtained by way of this heuristic gave an average runtime penalty of $\approx 15\%$. The prediction of optimal parallelizations was beyond the scope of this work but would be interesting for future work into data-driven methods for GENE.

To summarize, our inputs x are the level vector $\vec{\ell}$ and the degree of parallelization p

$$x_i = (\ell_{x,i}, \ell_{y,i}, \ell_{z,i}, \ell_{v_\parallel,i}, \ell_{\mu,i}, p_i), \tag{4}$$

which means that sample i had a resolution of $2^{\ell_{x,i}}$ grid points in x direction, and was run on 2^{p_i} processes. On these samples x_i, GENE was run to find the resulting runtime t^* (averaged over 20 time steps). Out of the 2048 randomly sampled tasks, 1837 fit into the main memory of the assigned processes. They constitute our training/validation data set (80% or 1470 samples) and the test set, on which the comparisons in Section 4 will be based. The corresponding outputs – the actual wall clock times – are distributed unevenly: the mean is at 0.957, while the median runtime is only 0.268. The long-tail shape of the distribution is similar for the whole data set and the separate parallelizations, as well as the test schemes discussed in Section 4.1.

In the following, we investigate how the runtime can be estimated based on this data.

3.2. Data-Driven Methods for Load Balancing

For our purposes, *data driven* means that, apart from the input data, no further domain knowledge about the physical or computational properties of the tasks need to be known – they should be represented by the learned model that we generate from data.

We first predict the runtimes of the tasks. Based on these estimates, a descendingly ordered list is created, and the corresponding tasks are distributed to the process groups accordingly. Note that a good ordering is more important than an accurate prediction of the actual runtimes. Heene et al. [4] discussed the differences between static and (initial) dynamic load balancing. Whereas in static load balancing, the full task assignment is given at the beginning of the simulation, the dynamic variant will wait for the currently running task to finish before assigning the next in a work-stealing fashion. This dynamic assignment is done for the first time step only. Here, we will focus on the dynamic variant. We start by discussing the anisotropy-based model currently in use for the application. It is then compared to three different machine learning approaches.

3.2.1. Anisotropy-based Fits for Runtime Estimation based on Expert Knowledge

Following up on previous work by Heene et al. [4], we tested model-based fits to predict the runtime of tasks based on the resolutions in the different directions. The dependence on the overall number of points $r(N)$ is modeled based on expert knowledge by an exponential fit for each degree of parallelization p individually. These estimations are then enriched with another least-squares fit h on the anisotropy s of the level vector $\vec{\ell}$

$$r(N) := mN^k + c, \qquad h(\vec{s}_{\vec{\ell}}) = c + \sum_{i=1}^{d-1} c_i s_{\vec{\ell},i}, \qquad \vec{s}_{\vec{\ell},i} = \frac{\vec{\ell}_i}{|\vec{\ell}|_1} \tag{5}$$

to give the overall runtime estimate

$$t(N, \vec{s}_{\vec{\ell}}) = r(N) \cdot h(\vec{s}_{\vec{\ell}}). \tag{6}$$

The model is based on the observation that higher resolutions in some directions lead to substantially higher runtime and memory footprints. This least-squares fit on the runtime thus constitutes an expert knowledge-based baseline against which we can compare.

3.2.2. Nearest Neighbor

The nearest neighbor estimator stores all the data in the training data set. To predict the runtime, it takes the features of the test data and returns the value of the closest training point (in Euclidean norm). If it is queried for a data point that has the same distance to multiple known points, it will randomly return any of the neighbors' values.

We can think of the nearest neighbor estimation as "zero-th order extrapolation", the best guess we can make without actually doing any computation on the data.

3.2.3. Support Vector Regression

Support vector regression (SVR) is an application of support vector machines to regression problems [11]. The regressor is defined by learned weights w, which are determined by minimizing $\|w\| = \langle w, w \rangle$ subject to linear constraints. The constraints are designed to make sure that predictions which fall within an error of ε of the true value will not contribute to the loss; the inner product $\langle \cdot, \cdot \rangle$ is approximated by kernel functions k using a regularization parameter C [11]. For our tests, the (Gaussian) radial basis function kernel $\exp(-\gamma \|\vec{x}_i - \vec{x}_j\|^2)$ was employed. The SVR parameters were optimized by a grid search algorithm employing five-fold cross validation. Results are shown in Table 1.

For both SVR and Nearest Neighbor the `scikit-learn` Python library [12] was used. It was also used to perform standard feature scaling on the input data x for the SVR and the neural network regression, which the next section is going to discuss.

Regularization / error weight C	Soft margin width ε	RBF kernel "pointiness" γ
450	0.01	0.05

Table 1. Optimal SVR parameters on our training / validation data set

3.2.4. Neural Network / Multi-Layer Perceptron Regression

In a feed-forward artificial neural network (ANN), a function is modelled by matrix and bias vector weights, which transform the input linearly, followed by the application of a (usually) nonlinear activation function ϕ. This is done successively, layer by layer, to return the output, which is the modelled value. The transition of data y from layer l to layer $l+1$ may be described by the matrix and vector weights w and b as

$$y_{l+1,j} = \phi\left(b_{l+1,j} + \sum_{k}^{n_k} w_{l+1,k,j} \cdot y_{l+1,k}\right), \qquad \vec{y_0} = \vec{x} \tag{7}$$

where n_k is the width of layer l. The training of the network – i.e., fitting the weights – is done by the backpropagation algorithm, using stochastic optimization heuristics. For a thorough description of neural networks, please refer to [13].

Hyperparameter Fitting by Genetic Algorithms We used the `Tensorflow` package [14] to learn the training data set with five-fold cross-validation, using the robust Huber regression loss function, cf. [13]. The training was run for 100 epochs, with no batch processing. Since choosing the optimal hyperparameters for ANNs by hand is notoriously difficult, genetic techniques to select the network architecture have been shown to work in many settings [15,16]. Here, genetic selection was applied on layer depth $(1-9)$, layer width $(1-11)$, activation function, and optimizer. The fitness was the negative validation root-mean-square error (RMSE). To prevent overfitting, only those configurations that had a maximum of 700 weights to be adjusted were considered, which amounts to about half the size of the training / validation data set. The results are shown in Table 2.

# hidden layers	# nodes / layer	activation function	optimizer
6	7	`tf.nn.elu`	`tf.keras.optimizers.Adam`

Table 2. Optimal neural network parameters on our training / validation data set

4. Results

The standard machine learning metric – error on previously-unseen test data ("zero-shot test") – returned the results shown in Table 3. The neural network was randomly initialized before training. As additional set-up, one out of five trained networks was selected by lowest validation error; only this one was used to capture the error on the test sets in Section 4.1. We can see by comparing the two rightmost bars that this is not cherry-picking good random results but that the test errors are low on every trained network.

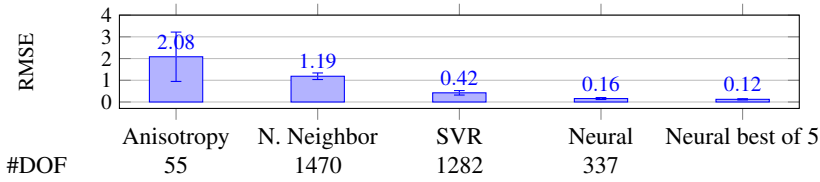


Table 3. Test errors (RMSE), based on 64 different train / test splits on the data

We observe that the estimation accuracy on the runtimes is seemingly quite good for the learning algorithms. But note that in comparison to the output distribution discussed in Section 3.1, which has a median runtime of 0.268, the estimation errors are still relatively high. Also, the estimation accuracy of the expected runtime is only suitable for evaluation to some extent. After all, a runtime estimate may be arbitrarily bad, but still help us achieve optimal scheduling if the relative ordering between the tasks is correctly represented. We will see in the next section that this leads to different outcomes if the models are applied to an actual combination scheme data set.

4.1. Results on Full Scenarios

Let us consider a test scenario, which is an actual standard combination scheme at physically relevant scales: The scheme consists of 124 grids, with level vectors between $[7,4,3,4,3]$ and $[11,8,6,8,6]$; we are approximating a full grid with 2^{39} unknowns with grids between 2^{21} and 2^{25} unknowns. All of the grids could be processed on 256 processors ($p = 8$) respectively. We also conducted a larger test case which could only be executed on bigger process groups, yielding similar results.

The resulting graph, Fig. 3, shows the parallel efficiency of the task assignment obtained depending on which trained model is used, how large the process groups are chosen (p), and how many of them there are. To make this more visually graspable, they are also averaged by method, displayed on the bottom.

We see that there are overlaps, but also a clear tendency: the neural network predicts the ordering often nearly-optimal, followed by the anisotropy-based model. The nearest-neighbor heuristic and the SVR still return reasonable results, considering we mostly get parallel efficiencies above 80%. The fact that they are lower for SVR than for the other approaches may be due to the already moderate number of input dimensions.

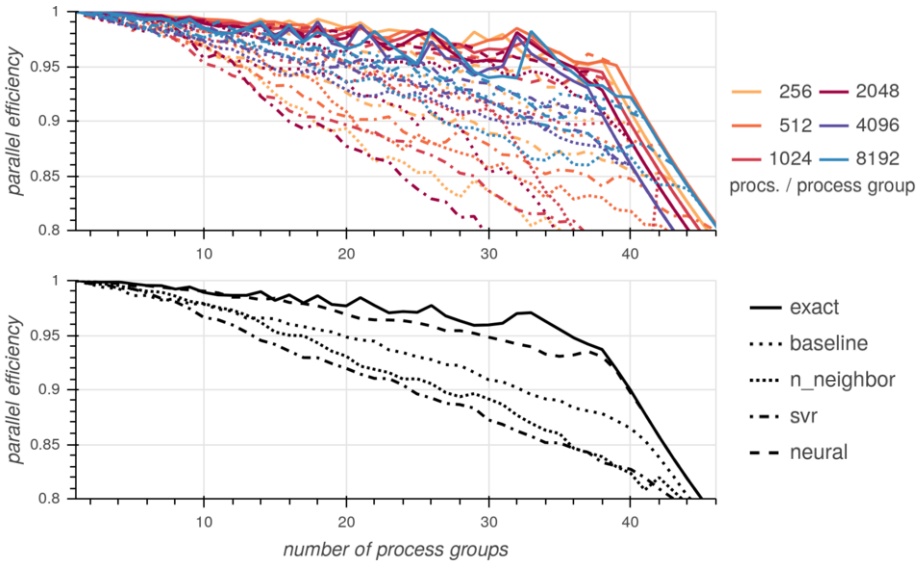


Figure 3. Parallel efficiencies for the test scenario. Note that data is included for $p = 8$ to 13.

Note that this experiment was done analytically, by adding up the exact runtimes off-line; actually running all these simulations would have been too costly. Accordingly, the true memory requirements are not represented here, but it is safe to assume that

1. the curves for low levels of parallelization will only be valid for the higher numbers of processes (as otherwise there will not be enough total memory available), and
2. high memory footprints strongly correlate with high runtimes, such that balancing runtimes will also balance memory usage to some extent.

We can observe that loads can be estimated well by using data-driven techniques for load balancing, despite the rather data-scarce setting. Furthermore, the data-driven approaches outperform the baseline based on expert knowledge not only in regions with plenty of data points, but also in those parameter regions where extrapolation dominates: for large numbers of processes. Still, the data-driven approaches excel only if enough data is at hand. It is therefore essential to keep track of GENE runtimes and the simulation parameters and metadata, such as system architecture and GENE version.

This could potentially pay off even more when using process groups of different sizes [17], which could be an interesting subject of study in the future.

5. Conclusion

In this paper, we studied the data-based prediction of runtimes for load balancing. This enabled us to obtain good load balances for the massively parallel sparse grid combination technique with GENE. While the expert knowledge-based model used until now is reasonable, it can be outperformed by purely data-driven methods such as neural networks, given enough data and automated selection of network parameters.

Based on this insight, it is now feasible to collect run time data for GENE simulations on the job, improving data-driven load models along the way. Especially with respect

to different resolutions, at least knowledge about a good ordering between tasks should be possible even across compute systems. Let us note that data-driven approaches for load balancing are most suited for situations where the concrete code is considered a black-box. If, however, the behavior of the compute and communication systems, as well as the algorithm and its implementation are well-understood and stable, it will be more beneficial to use model-based approaches. In all other cases, one should use carefully selected data-driven approaches, especially when a lot of compute time is at stake – such as with the massively parallel sparse grid combination technique employed with GENE.

Acknowledgements We would like to thank Raphael Leiteritz and Tilman Dannert for suggestions and fruitful discussions. This work was supported by the German Research Foundation (DFG) through the Priority Programme 1648 Software for Exascale Computing (SPPEXA).

References

[1] F. Jenko, D. Told, T. Görler, et al. Global and local gyrokinetic simulations of high-performance discharges in view of ITER. *Nucl. Fusion*, 53(7):073003, 2013.

[2] M. Heene, A. P. Hinojosa, M. Obersteiner, H.-J. Bungartz, and D. Pflüger. EXAHD: An exa-scalable two-level sparse grid approach for higher-dimensional problems in plasma physics and beyond. In *High Performance Computing in Science and Engineering '17*, pages 513–529. Springer, 2018.

[3] M. Heene. *A Massively Parallel Combination Technique for the Solution of High-Dimensional PDEs*. PhD thesis, Universität Stuttgart, 2018.

[4] M. Heene, C. Kowitz, and D. Pflüger. Load balancing for massively parallel computations with the sparse grid combination technique. In *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*, volume 25 of *Advances in Parallel Computing*, pages 574–583, 2014.

[5] T. Görler. *Multiscale effects in plasma microturbulence*. PhD thesis, Universität Ulm, 2009.

[6] H. Doerk and F. Jenko. Towards optimal explicit time-stepping schemes for the gyrokinetic equations. *Computer Physics Communications*, 185(7):1938–1946, 2014.

[7] M. Griebel, W. Huber, U. Rüde, and T. Störtkuhl. The combination technique for parallel sparse-grid-preconditioning or -solution of PDEs on workstation networks. In *Parallel Processing: CONPAR 92 VAPP V*, 1992.

[8] H.-J. Bungartz and M. Griebel. Sparse grids. *Acta Numerica*, 13:147–269, 2004.

[9] S. Hirschmann, C. W. Glass, and D. Pflüger. Enabling unstructured domain decompositions for inhomogeneous short-range molecular dynamics in ESPResSo. *The European Physical Journal Special Topics*, 227(14):1779–1788, 2019.

[10] A. Totounferoush, N. Ebrahimi Pour, J. Schröder, S. Roller, and M. Mehl. A new load balancing approach for coupled multi-physics simulations. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2019.

[11] A. J. Smola and B. Schölkopf. A tutorial on support vector regression. *Statistics and Computing*, 14(3):199–222, 2004.

[12] F. Pedregosa, G. Varoquaux, et al. Scikit-learn: Machine Learning in Python. *JMLR*, 12:2825–2830, 2011.

[13] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer-Verlag, 2 edition, 2009.

[14] M. Abadi, A. Agarwal, P. Barham, et al. TensorFlow: Large-scale machine learning on heterogeneous systems. URL: https://www.tensorflow.org/, 2015.

[15] J. D. Schaffer, D. Whitley, and L. J. Eshelman. Combinations of genetic algorithms and neural networks: a survey of the state of the art. In *COGANN-92: International Workshop on Combinations of Genetic Algorithms and Neural Networks*, pages 1–37, 199206.

[16] R. Miikkulainen, J. Liang, E. Meyerson, et al. Evolving deep neural networks. In *Artificial Intelligence in the Age of Neural Networks and Brain Computing*, pages 293–312. Academic Press, 2019.

[17] M. Molzer. Implementation of a parallel sparse grid combination technique for variable process group sizes. Bachelor's thesis, TU München, 2018.

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200035

Load-Balancing for Large-Scale Soot Particle Agglomeration Simulations

Steffen HIRSCHMANN [a,1], Andreas KRONENBURG [b], Colin W. GLASS [c] and Dirk PFLÜGER [a]

[a] *Institute for Parallel and Distributed Systems, University of Stuttgart, Germany*
[b] *Institute for Combustion Technology, University of Stuttgart, Germany*
[c] *Department of Mechanical Engineering, Helmut Schmidt University Hamburg, Germany*

Abstract. In this work, we combine several previous efforts to simulate a large-scale soot particle agglomeration with a dynamic, multi-scale turbulent background flow field. We build upon previous simulations which include 3.2 million particles and implement load-balancing into the used simulation software as well as tests of the load-balancing mechanisms on this scenario. We increase the simulation to 109.85 million particles, superpose a dynamically changing multi-scale background flow field and use our software enhancements to the molecular dynamics software ESPResSo to simulate this on a Cray XC40 supercomputer. To verify that our setup reproduces essential physics we scale the influence of the flow field down to make the scenario mostly homogeneous on the subdomain scale. Finally, we show that even on the homogeneous version of this soot particle agglomeration simulation, load-balancing still pays off.

Keywords. molecular dynamics, short-range, dynamic load-balancing, soot-particle agglomeration, domain decomposition

1. Introduction

Short-range molecular dynamics (MD) [1] is an important field in Computational Sciences. One particular example of a real-world application is the simulation of soot particle agglomeration, which, for example, is relevant for the efficiency of industrial processes. In these processes, particles collide and link irreversibly. Of particular interest is the morphology of the resulting agglomerates. Because results of a computer simulation allows the examination of morphology of agglomerates over time, computer simulation plays an important role in this area.

The approach we use has been described in [2,3]: Agglomeration processes are simulated in a precomputed, turbulent flow field. Clustering of particles is driven by Brownian motion as well as the background flow, which gets more important as the particle density decreases. The influence of a turbulent background flow field is of particular interest in particle agglomeration simulations. While small turbulence scales can be resolved in

[1]Corresponding Author: Steffen Hirschmann, Institute for Parallel and Distributed Systems, University of Stuttgart, Universitätsstraße 38, 70569 Stuttgart; Email: steffen.hirschmann@ipvs.uni-stuttgart.de

small simulations, the question remains if large, multi-scale turbulence flows critically influence the results.

In order to get to more realistic agglomeration simulations we use a larger and dynamically changing flow field that covers more scales of turbulence as well as a larger setup including the number of particles. Starting from the largest simulation in [3] ("Case 6"), we increase the domain size by a factor of 3.25. This allows us to cover more realistic scales of turbulence. We keep the original particle loading. Hence, we increase the number of primary particles from 3.2 million in [3] by a factor of $3.25^3 \approx 34.33$ to 109.85 million.

This scenario is very large considering the elaborate physical bonding model used. In fact, to the best of our knowledge it is the largest simulation with ESPResSo so far. And, because we simulate an agglomeration process, the simulation naturally gets more and more heterogeneous over time. Simulations of these sizes and types pose two major challenges: (1) We need large-scale parallelism to cope with a simulation of this size, and (2) we must dynamically adapt the domain decomposition to the changing particle distribution in order to cope with load-imbalances arising from heterogeneity.

These challenges require us to combine this large-scale real world scenario with our previous efforts to bring dynamic load-balancing to the MD software ESPResSo. In particular, we need to make use of dynamic load-balancing at runtime, the newly created, non-regularly partitioned grids and their associated asynchronous communication [4,5] as well as other contributions to the MD software at hand, like parallel input and output using MPI-IO.

In order to validate the physical correctness of the scenario and assess the applicability of our load-balancing methods, we use a rather homogeneous version of the scenario. A more homogeneous scenario allows us to first focus on physical correctness of the setup while not crucially depending on the best possible load-balancing. Following the simulation, we can test the applicability of our load-balancing methods for this scenario.

The remainder of this work is structured as follows: In Section 2 we report on related research. In Section 3 we elaborate on the numerical simulation models as well as the used code and our load-balancing methodology. We describe our simulation setup in Section 4. Subsequently, in Section 5 we show the physical results and our assessment of load-balancing for this setup. Finally, in Section 6 we summarize our work and conclude with a note on further topics to investigate.

2. Related Work

At the core of our modeling are Langevin-based agglomeration processes. These have, e.g., been studied in [6]. We are, however, interested in agglomeration processes that are subject to a turbulent background flow field and that links particles irreversibly. This linking process should completely prohibit rotation and sliding of the particles. To this end, the Langevin-based model has been augmented in [2,3] to include a coupling to a static flow field as well as dynamic bonding to link particles irreversibly at runtime. In [2] several bonding models that effectively prohibit sliding are proposed and evaluated. We use the so-called "AB" (*all-bonds*) model from this work, which has the advantage, that it does not require the addition of virtual particles.

A different approach to tackle the upscaling of agglomeration simulation is coarse-graining, i.e. aggregating whole clusters into one "super particle" (during the simulation)

and, thus, reducing the total computational burden. This is still a field of active research, as the involved modeling is complex. For example, coarse grained particle need to accurately resolve the collision probabilities of the underlying "real" cluster. Algorithmically, this kind of dynamic coarse graining leads to larger cell sizes and, thus, likely to less parallelism and more load-imbalances. Studies and first results for this approach are, e.g., presented in [7]. However, the technique described there is not ready yet for large-scale simulations such as ours.

For load-balancing several heuristics are used in existing MD (and other) software. We have discussed details on the most commonly used ones in [8] and have implemented some of them in the MD software ESPResSo in [4,9,5]. In this work, we focus on the partitioner based on Space-filling curves (SFC) leveraging the well-known and scalable library `p4est` [10,11], which in turn uses the Z-curve [12]. The actual partitioning for SFC-based algorithms is performed using so-called chain-on-chain partitioning [13]. Several studies find graph partitioning performs best because of its superior model while SFC-based partitioning is fast and consumes less memory [14,15]. A more theoretical review of several partitioning algorithms can be found in [16] listing important properties like speed, memory usage, etc. Eibl and Rüde [17] inspect different partitioners for the discrete element method and find that there is a trade-off between scalability and quality of partitioning and recommend an SFC-based strategy for small and mid-sized scenarios. Particularly for MD, several methods are implemented in the simulation software "ls1 mardyn" and compared in [18,19,20]. These studies also present cost heuristics to estimate the load of individual subdomains. They give us enough reason to focus on SFC-based partitioning first for our current work.

3. Methodology

Our main methodological approach is two-part. First, we explain what numerical models we use to simulate a soot particle agglomeration process within the molecular dynamics framework. Second, we explain what parallelization and load-balancing approaches we use for the implementation of the numerical models. Additionally, we briefly introduce the relevant quantities for analyzing the shape of agglomerates.

3.1. Numerical Models

We model intermolecular interactions with the well-known Lennard-Jones-12-6 potential which consists of an attractive and a repulsive part,

$$U_{LJ}(r) = 4\varepsilon \left(\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right),$$

where σ and ε are properties of the modeled primary particles. Particles can be bound together with distance-based and angular harmonic bonds. Their associated potentials are given as

$$U_{distance}(r) = \frac{1}{2} k_h (r - r_0)^2 \qquad \text{, and} \qquad U_{angular}(\phi) = \frac{1}{2} k_a (\phi - \phi_0)^2,$$

Figure 1. Visualization of bonding at runtime. Left: If two particles are closer than the collision distance, the algorithm aims to finds a third particle within the collision distance to establish a triangular bonding structure. Right: The bonding structure consists of three angular bonds (indicated as $\theta_1, \theta_2, \theta_3$) as well as three distance-based bonds (indicated as l_1, l_2, l_3) between the particle pairs. If no third particle could be found, only the distance bond l_1 is created.

where r_0 and ϕ_0 are the equilibrium distance and angle, respectively, and k_h and k_a spring constants. These bonds are established in groups at runtime for each triple of particles that collides. This so-called "AB" model is adapted from [2] and depicted in Figure 1. Note, that the equilibrium angle ϕ_0 is not constant across bonds but rather different for each one. It is chosen as the angle between the particles at collision time.

In order to model frictional influence from a fluid and Brownian motion, we use Langevin Dynamics. A study of purely Langevin-driven agglomeration processes without turbulent background flow can be found in [6]. The equation of motion is given as:

$$m\ddot{\vec{x}} = \vec{f} - \gamma\left(\dot{\vec{x}} - \vec{u}_{flow}(t,\vec{x})\right) + \vec{R}(t), \tag{1}$$

where $\vec{f}$ are the forces given by the intermolecular potentials described above. Additionally, $\vec{R}(t)$ is a random noise, which, together with the frictional term $\gamma(\dot{\vec{x}} - \vec{u}_{flow}(t,\vec{x}))$ models Brownian motion and the temperature, as well as the frictional influence of the fluid. The velocities $\vec{u}_{flow}(t,\vec{x})$ stem from the fluid (external flow field) at time t and position $\vec{x}$. Analogously to [3] we model the friction between the fluid and the particles by Stokes' law, so $\gamma = 3\pi\mu\sigma/C_c$ where μ is the viscosity of the fluid, σ the particle diameter and C_c being the Cunningham correction factor. In ESPResSo, Langevin Dynamics is implemented with a Velocity Verlet integrator, see e.g. [21], combined with a so-called Langevin thermostat [22] that applies the frictional and random forces given the temperature and γ.

3.2. Parallelization and Load-Balancing

We use the simulation software ESPResSo[2] [23,22], which covers all the relevant physics involved. Relevant parts of the dynamic bonding mechanisms have been implemented in the course of [2,3] and are also described in [23]. ESPResSo uses the Linked-Cell algorithm [24] in combination with Verlet lists to calculate forces stemming from short-range potentials, like the Lennard-Jones potential, in linear time. Based on the Linked-Cell discretization, ESPResSo defines a uniform spatial domain decomposition to allow for MPI-based parallelization [22]. While the simulation core of ESPResSo is implemented in C++, it exposes a Python-based front-end for setting up the scenario and controlling the simulation [25].

[2]Extensible Simulation Package for Research on Soft Matter, http://www.espressomd.org

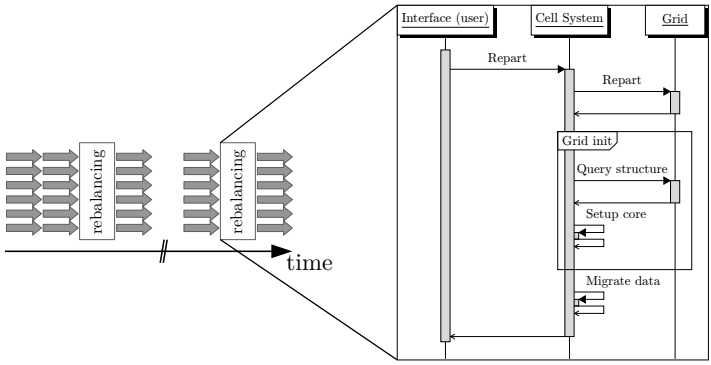


Figure 2. Left: A sketch of a segment of an MD simulation. The gray arrows depict regular time steps. From time to time the domain decomposition has to be adapted to the underlying scenario (indicated by the "rebalancing" boxes). We outline our implementation on the right: The user calls a function "repart" from their script. The linked cell grid ("cell system") internally asks the grid to repartition itself and then sets up the established partitioning in the core of the simulation software. Afterwards, cell payload (particles) is migrated transparently for the user.

Our adaptions keep the MPI-only parallelization and its 1:1 mapping of subdomains to processes. In [4] we devised a general scheme to change this fixed decomposition, allowing for arbitrary ones. Based on this work, we have implemented different decompositions. We make the load-balancing mechanism available to the user, so they can conveniently implement strategies for partitioning scenarios in the simulation scripts themselves. Note that this load-balancing mechanism is not constrained to the application presented in this work. It can be employed to any heterogeneous simulation in ESPResSo. We depict this ability to do dynamic repartitioning and sketch the underlying implementation in Figure 2.

Given a function m that defines a suitable load metric or measurement of execution time for every process $p \in \{1,\dots,P\}$, we partition based on the imbalance $\mathscr{I}(m)$. The imbalance is defined as the maximum over average load measurement: $\mathscr{I}(m) = \frac{P \max\{m(p)\}}{\sum_p m(p)}$. In the current setup, we use $\mathscr{I}(m) > 1.1$ as criterion with the load metric $m(p)$ as the number of particles of process p and partition at most every 1000 time steps. In [4] we have shown for a smaller agglomeration scenario that choosing the number of particles as metric $m(p)$ performs best among a range of different choices.

3.3. Analysis

An important characterization of particle clusters is their fractal dimension D_f [26]. It is the power law relationship of the number of particles to their radius, see e.g. [27], and calculated as

$$N = \left(\frac{r_g}{d}\right)^{D_f},$$

with N the number of particles in the cluster, $d = \frac{\sigma}{2}$ and r_g the radius of gyration, which is the standard deviation of the particle positions $\vec{r}_i$ in a cluster:

$\hat{n}$	T	σ	l_0
$6.25 \cdot 10^{-3}$	600 K	20 nm	$2600\,\sigma$

Table 1. Basic MD simulation parameters.

σ	ε^*	t^*
20 nm	$1.25 \cdot 10^{-20}$ J	14 ns

Table 2. Reference values used for nondimensionalization of physical quantities in the MD simulation.

$$r_g = \sqrt{\frac{1}{N}\sum_{i=1}^{N} ||\vec{r} - \vec{r}_i||^2} \quad \text{, with} \quad r = \frac{1}{N}\sum_{i=1}^{N} \vec{r}_i.$$

4. Simulation Setup

The basic simulation parameters, which will be explained in the following, can be found in Table 1. The reference length, energy and time used for nondimensionalization of the MD simulation can be found in Table 2. The simulation comprises $109.85 \cdot 10^6$ particles, the largest simulation with ESPResSo so far. Each of these primary particles has a diameter of $\sigma = 20\,\text{nm}$. Initially, they are placed in a simulation box of size $l_0 = 2600\,\sigma$, uniformly randomly distributed. We employ periodic boundary conditions in all dimensions. The particle loading is $\hat{n} = 6.25 \cdot 10^{-3}$. The temperature of the solvent is $T = 600\,\text{K}$.

The flow field is primarily characterized by its kinematic viscosity ν and its dissipation rate ε. Based on these, we can derive the characteristic time scales, namely Brownian diffusion time t_{BM} and the Kolmogorov time scale t_k. These allow us to define the nondimensional Péclet number $\text{Pe} = \frac{t_{\text{BM}}}{t_k}$ that describes the relative importance of turbulence over Brownian motion. The second nondimensional quantity that is used in [3] to describe a scenario is the Knudsen number $\text{Kn} = \frac{l_{mfp}}{\sigma/2}$, where l_{mfp} is the mean free path length of the flow.

Our goal is to reproduce a larger version with more turbulent flow scales of "Case 6" from [3]. This setup uses $\text{Kn} = 11$ and $\text{Pe} = 1$. To achieve that, we generate the external background flow field in a separate pre-processing step using a Direct Numerical Simulation (DNS) of homogeneous, isotropic forced turbulence in a box of length $L = 2\,\pi\,l_0 \approx 326.7\,\mu\text{m}$. It solves the incompressible Navier-Stokes equations with periodic boundary conditions, discretized on a $64 \times 64 \times 64$ mesh. This mesh is unrelated to the linked cell grid and only defines the resolution of the flow field in the later MD simulation. The viscosity is $\nu = 5.13 \cdot 10^{-5}\,\text{m/s}^2$ and the dissipation rate $\varepsilon = 1.25 \cdot 10^{10}\,\text{m}^2/\text{s}^3$, which equals the desired values of Kn and Pe. These, however, lead to very heterogeneous particle distributions as we have shown in [4,5] on basis of "Case 6" from [3]. For first experiments at scale, we keep the particle distribution mostly homogeneous on the subdomain scale by reducing the influence of the flow field in the transport equation. Therefore, we scale down $\vec{u}_{flow}(t, \vec{x}_i)$ in Equation 1 by about 1/3. This rescales the gradients of the velocity by an equal factor, and, thus, also the dissipation rate. So in our current simulation the Péclet number is roughly a third of the intended target value.

We compute $6 \cdot 10^7$ iterations, each having a length of $dt = 10^{-4}\,t^* = 1.4\,\text{fs}$. Thus, the end of the simulation is at $t_{end} = 8.4\,\mu\text{s}$. The bonding constants $\tilde{k}_a$ and $\tilde{k}_h$ are estimated according to [3] to $\tilde{k}_a = \tilde{k}_h = 1000\,\varepsilon^*$. As collision distance, we use $r_{col} = \sigma$.

As mentioned above, we have extended and use the simulation software ESPResSo as it implements all relevant numerical models, especially dynamic bonding at runtime. We perform the scenario setup as follows: (1) Setup random particles with zero velocities, (2) initialize all required potentials, (3) equilibrate the system using steepest descend

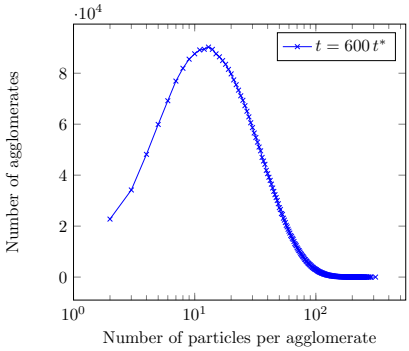


Figure 3. Histogram of the number of agglomerates per size. Size is in number of particles per agglomerate. The location of the maximum indicates that the agglomeration process has left its initial state where growth is mainly driven by collisions of primary particles.

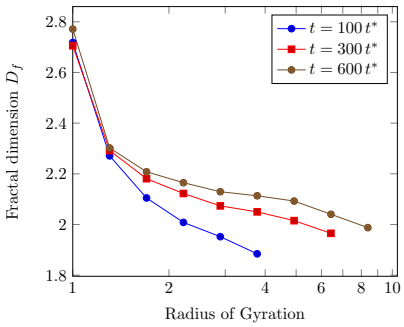


Figure 4. Average fractal dimension D_f of agglomerates of all agglomerates of a certain radius of gyration at different time steps. While $t = 100t^*$ is still early in the simulation, the average D_f does not vary much in later time steps.

integration [25], (4) reset velocities and forces to zero, (5) setup the thermostat as well as collision detection and dynamic bonding, and (6) start the simulation.

5. Results

One indicator of how much the agglomeration process has progressed in time, is the number of agglomerates of a certain size. We present a histogram of the sizes of the agglomeration for $t_{end} = 600t^*$ in Figure 3. We clearly see that the most prevalent cluster size is about one order of magnitude higher than the smallest possible (2 primary particles). This means that the simulation has left the initial state where the growth of agglomerates is mainly driven by primary particle collisions. At t_{end} process is primarily driven by cluster-cluster collisions for significant growth of the agglomerates.

In Figure 4 we plot the average D_f of all agglomerates with more than 15 particles at $t = 100t^*$, $300t^*$, and $600t^*$ depending on the radius of gyration of the agglomerates. This relationship enables scientists to understand the agglomeration process and to determine its influence on larger industrial processes and products. Therefore, the criterion for stopping the simulation in [3] is when the individual lines converge. While the process clearly has not converged yet at $t = 100t^*$ in Figure 4, the difference in D_f for the same r_g between $t = 300t^*$ and $600t^*$ gets significantly smaller, indicating a possible convergence. Since the agglomerates are still rather small ($r_g \leq 10$), the average fractal dimension is in the range of $1.9 \leq D_f \leq 2.2$. These D_f are a bit higher than the ones published by [3] using the same ansatz, as well as experimental data obtained from soot aggregates in (turbulent) flames, see e.g. [28]. However, the smaller the velocities of the superposed, turbulent background flow field in Equation 1, the higher the influence of Brownian motion. In this case, [3] also states, that with a higher influence of Brownian motion, D_f approaches 2 for larger agglomerates. Beyond that we can see a trend for agglomerates with larger r_g to have a smaller D_f. In conclusion, we can say, that we are able to reproduce physical results for the simulation of soot particle agglomeration, and we assume that the mentioned differences stem from the smaller flow field velocities.

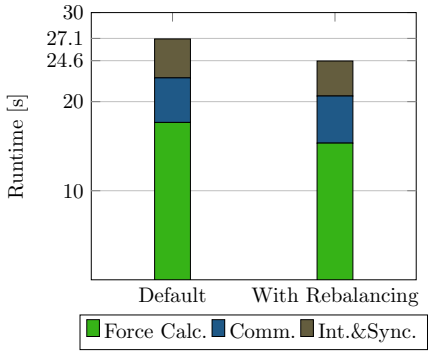


Figure 5. Runtime of 1000 time steps beginning at $t_0 = t_{end} = 600t^*$ for the default, unbalanced paratitioning ("Default") and our SFC-based partitioning ("With Rebalancing") for 7200 processes. For each of the three components (force calculation with the Linked-Cell/Verlet-list method, communication and integration including synchronization) we plot the maximum runtime of any process. Even though the setup is quite homogeneous, we can reduce the runtime by about 10 %.

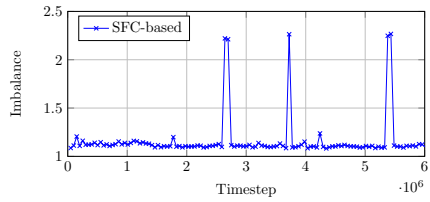


Figure 6. Imbalance in runtime per 1000 time steps between the different processes.Rebalancing with the SFC-based method is performed if the number of particles diverges by 10 % on any process from the average. The number of particles per cell is also used as weights for repartitioning. The spikes are still under investigation. Their occasional occurrences *could* simply be due to an influence of non-deterministic components like hardware, communication, general network load on the HPC system etc.

5.1. Load-balancing

Although we enforce a more or less homogeneous particle distribution among the subdomains, we still use load-balancing to counteract smaller heterogeneity that evolves over time. For unscaled versions of this scenario, good load-balancing is crucial as our studies on the smaller 3.2 million particle setup [4,5] have shown. Therefore, we also test and show load-balancing results here. The simulation ran on "Hazel Hen" at the High Performance Computing Center Stuttgart (HLRS). It is a Cray XC40 machine with 2 Intel Xeon E5-2680v3 ("Haswell" microarchitecture) per node, each of which has 12 cores (not counting Simultaneous Multi-Threading) and a Cray Aries interconnect.

We use the snapshot at $t_{end} = 600t^*$ for our test. We run the test on 300 nodes, i.e. 7200 processes. The imbalance in the number of particles for a decomposition into equally sized boxes (`MPI_Dims_create`) is about 18.4 %, which is quite homogeneous. The runtimes for the default parallelization and our load-balanced one can be found in Figure 5. We plot the relevant runtimes in the following way: Let

$$f_1 = \max\left\{t_{force}(p)\right\}_{p=1}^{P},$$

$$f_2 = \max\left\{t_{force}(p) + t_{comm}(p)\right\}_{p=1}^{P}, \text{ and}$$

$$f_3 = \max\left\{t_{force}(p) + t_{comm}(p) + t_{int}(p) + t_{sync}(p)\right\}_{p=1}^{P},$$

where "force", "comm", "int" and "sync" refer to the individual components: force calculation, communication, integration and synchronization. Then, we plot f_1, $f_2 - f_1$ and $f_3 - f_2 - f_1$, i.e. the difference of the individual maxima runtimes of the phases. We can see, that despite the homogeneous particle distribution, we achieve a runtime reduction of about 10 %. Additionally, in Figure 6 we can see, that the load-balancing is capable of

keeping the imbalance at about the desired level of 1.1 during almost the entire 6 million time steps. The occasional spikes are still being investigated. Given the overall behavior, however, it is likely, that the spikes stem from runtime noise.

6. Conclusion

We successfully combined several previous works into one large-scale, load-balanced soot particle agglomeration simulation. We set up the simulation with a complex physical bonding model, as well as a dynamically changing, multi-scale turbulent background flow field. We increased the simulation to over 100 million particles, which is over 30 times larger than previous works with ESPResSo and, to the best of our knowledge, the largest simulation ever with ESPResSo.

In order to assess the physical correctness of the new setup, we kept the simulation rather homogeneous by rescaling the velocities stemming from the superposed flow field. This way, we did not have to deal with large heterogeneity, and we were able to reproduce previous results. We showed that the resulting fractal dimensions of the clusters seem reasonable and consistent with previous results. Also, we showed, that even though we keep heterogeneity low, load-balancing is still able to reduce the runtimes by about 10 % and to consistently keep the imbalance in runtime low throughout the simulation.

6.1. Future Work

There are two paths that we will pursue further. As we have verified that the setup is physically correct and that load-balancing pays off, the first path is to use a physically correct flow field with $\mathrm{Pe} \approx 1$. This will let us study the impact of the multi-scale turbulent and dynamic background flow field on agglomeration processes at physically relevant scales.

Second, we have implemented different kinds of load-balancing methods in previous work [4,9,5], some of which might be more suitable for the heterogeneous version of the simulation. We intend to test different methods as well as different metrics and relate them to scenario properties in order to gain deeper insight into the problem of balancing heterogeneous simulations with a low average fractal dimension of the clusters.

Acknowledgements

The authors gratefully acknowledge financial support provided by the German Research Foundation (DFG) as part of the former Collaborative Research Center (SFB) 716, and the computing time on “Hazel Hen” granted by the High Performance Computing Center Stuttgart (HLRS). Special thanks go to Rudolf Weeber from the ICP, University of Stuttgart, for his expertise and fruitful discussions.

References

[1] M. P. Allen and D. J. Tildesley. *Computer Simulation of Liquids*. Oxford University Press, 1989.
[2] Gizem Inci et al. Modeling nanoparticle agglomeration using local interactions. *Aerosol Science and Technology*, 48(8):842–852, July 2014.

[3] Gizem Inci et al. Langevin dynamics simulation of transport and aggregation of soot nano-particles in turbulent flows. *Flow, Turbulence and Combustion*, pages 1–21, January 2017.

[4] Steffen Hirschmann et al. *Load Balancing with p4est for Short-Range Molecular Dynamics with ESPResSo*, volume 32 of *Advances in Parallel Computing*, pages 455–464. IOS Press, 2017.

[5] Steffen Hirschmann, Colin W. Glass, and Dirk Pflüger. Enabling unstructured domain decompositions for inhomogeneous short-range molecular dynamics in ESPResSo. *The European Physical Journal Special Topics*, 227(14):1779–1788, March 2019.

[6] Lorenzo Isella and Yannis Drossinos. Langevin agglomeration of nanoparticles interacting via a central potential. *Physical Review E*, 82:011404, July 2010.

[7] Milena Smiljanic et al. Developing coarse-grained models for agglomerate growth. *The European Physical Journal Special Topics*, 227(14):1515–1527, March 2019.

[8] Steffen Hirschmann, Dirk Pflüger, and Colin W. Glass. Towards understanding optimal Load-Balancing of heterogeneous Short-Range molecular dynamics. In *Workshop on High Performance Computing and Big Data in Molecular Engineering 2016 (HBME 2016)*, Hyderabad, India, December 2016.

[9] Steffen Hirschmann et al. *Load-Balancing and Spatial Adaptivity for Coarse-Grained Molecular Dynamics Applications*. Springer, 2018.

[10] Carsten Burstedde et al. p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, January 2011.

[11] Tobin Isaac, Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. Recursive algorithms for distributed forests of octrees. *SIAM Journal on Scientific Computing*, 37(5):C497–C531, January 2015.

[12] G. M. Morton. A computer oriented geodetic data base; and a new technique in file sequencing. Technical report, IBM Ltd., 1966.

[13] Ali Pınar and Cevdet Aykanat. Fast optimal load balancing algorithms for 1D partitioning. *Journal of Parallel and Distributed Computing*, 64(8):974–996, 2004.

[14] William F. Mitchell. A refinement-tree based partitioning method for dynamic load balancing with adaptively refined grids. *Journal of Parallel and Distributed Computing*, 67(4):417–429, April 2007.

[15] S. Schambeger and J. M. Wierum. Graph partitioning in scientific simulations: Multilevel schemes versus space-filling curves. In *International Conference on Parallel Computing Technologies*, volume 2763 of *LNCS*, pages 165–179, 2003.

[16] Bruce Hendrickson and Tamara G. Kolda. Graph partitioning models for parallel computing. *Parallel Computing*, 26(12):1519 – 1534, 2000.

[17] Sebastian Eibl and Ulrich Rüde. A systematic comparison of runtime load balancing algorithms for massively parallel rigid particle dynamics. *Computer Physics Communications*, 2019.

[18] Martin Buchholz. *Framework zur Parallelisierung von Molekulardynamiksimulationen in verfahrenstechnischen Anwendungen*. Verlag Dr. Hut, 2010.

[19] M. Buchholz, H.-J. Bungartz, and J. Vrabec. Software design for a highly parallel molecular dynamics simulation framework in chemical engineering. *Journal of Computational Science*, 2(2):124–129, 2011.

[20] Christoph Niethammer et al. ls1 mardyn: The massively parallel molecular dynamics code for large systems. *Journal of Chemical Theory and Computation*, 10(10), October 2014.

[21] Benedict J. Leimkuhler et al. Integration methods for molecular dynamics. In *Mathematical Approaches to Biomolecular Structure and Dynamics*, volume 82 of *The IMA Volumes in Mathematics and its Applications*, pages 161–185. Springer New York, 1996.

[22] Hans Jörg Limbach et al. ESPResSo – an extensible simulation package for research on soft matter systems. *Computer Physics Communications*, 174(9):704–727, May 2006.

[23] Axel Arnold et al. ESPResSo 3.1: Molecular dynamics software for coarse-grained models. In *Meshfree Methods for Partial Differential Equations VI*, volume 89 of *Lecture Notes in Computational Science and Engineering*, pages 1–23. Springer Berlin Heidelberg, September 2013.

[24] R. W. Hockney and J. W. Eastwood. *Computer Simulation Using Particles*. Taylor & Francis, Inc., Bristol, PA, USA, 1988.

[25] Florian Weik et al. Espresso 4.0 – an extensible software package for simulating soft matter systems. *The European Physical Journal Special Topics*, 227(14):1789–1816, March 2019.

[26] Benoit B. Mandelbrot. *The fractal geometry of nature*, volume 173. W. H. Freeman New York, 1983.

[27] R. J. Samson et al. Structural analysis of soot agglomerates. *Langmuir*, 3(2):272–281, 1987.

[28] Ümit Ö. Köylü, Yangchuan Xing, and Daniel E. Rosner. Fractal morphology analysis of combustion-generated aggregates using angular light scattering and electron microscope images. *Langmuir*, 11(12):4848–4854, 1995.

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200036

On the Autotuning of Task-Based Numerical Libraries for Heterogeneous Architectures

Emmanuel AGULLO [a], Jesús CÁMARA [b,1], Javier CUENCA [b] and Domingo GIMÉNEZ [c]

[a] *HiePACS Team, Inria Bordeaux Sud Ouest, France*
[b] *Department of Engineering and Technology of Computers, University of Murcia, Spain*
[c] *Department of Computing and Systems, University of Murcia, Spain*

Abstract. A roadmap for autotuning task-based numerical libraries is presented. Carefully chosen experiments are carried out when the numerical library is being installed to assess its performance. Real and simulated executions are considered to optimize the routine. The discussion is illustrated with a task-based tile Cholesky factorization, and the aim is to find the optimum tile size for any problem size, using the Chameleon numerical linear algebra package on top of the StarPU runtime system and also with the SimGrid simulator. The study shows that combining a smart exploration strategy of the search space with both real and simulated executions results in a fast, reliable autotuning process.

Keywords. autotuning, linear algebra, task-based programming, heterogeneous computing, simulation

1. Introduction

The complexity of modern computers makes the design of high performance numerical libraries extremely challenging. Task-based programming paradigms have been proved to alleviate the exercise, as part of the burden is delegated to a third party software, commonly referred to as a runtime system. Nonetheless, the resulting libraries are often left with one or more parameters to be carefully set up in order to achieve high performance. This work describes an approach on how to use autotuning techniques to select the best values for some algorithmic parameters of the linear algebra routines of these kinds of libraries.

The proposed approach is applied to routines of Chameleon [1]. The computational kernels of the library are used as building blocks for higher-level routines designed for heterogeneous platforms composed of multicore CPUs with one or more GPUs. This dense linear algebra library is derived from PLASMA [2] and internally uses StarPU [3], a runtime system which enables us to express parallelism through sequential-like code

[1]Corresponding Author: J. Cámara, Department of Engineering and Technology of Computers, University of Murcia, 30100 Espinardo, Murcia, Spain; E-mail: jcamara@um.es.

and which schedules the different tasks over the hybrid processing units. These tasks are executed by using optimized implementations of linear algebra libraries, such as Intel MKL [4] for multicore CPU and MAGMA [5] or cuBLAS [6] for GPU. In previous works, several frameworks have been developed which focus on how to optimize linear algebra kernels on heterogeneous platforms [7,8]. Other approaches are proposed to predict the performance of a dynamic task-based runtime system for heterogeneous multicore architectures [9]. In contrast with those previous works, we propose the application of tuning strategies to obtain the best value of the algorithmic parameters of the routines for an efficient use of the hybrid components in the computational node. The application of these strategies is illustrated with the Cholesky routine, a fundamental and representative linear algebra algorithm, with the focus on the selection of the value for the tile size.

The rest of the paper is organized as follows. Section 2 introduces the Cholesky routine of Chameleon and how it is executed by using the StarPU runtime system. Section 3 describes the training strategies proposed for selecting the best values for the tile size. Experimental results are shown in Section 4 for a heterogeneous platform. Possible extensions of the methodology are discussed in Section 5. Section 6 concludes the paper.

2. Cholesky Routine of Chameleon

The Cholesky factorization (or Cholesky decomposition) of an $n \times n$ real symmetric positive definite matrix A has the form $A = LL^T$, where L is an $n \times n$ real lower triangular matrix with positive diagonal elements. This factorization is mainly used as a first step for the numerical solution of linear equations $Ax = b$, where A is a symmetric, positive definite matrix.

The reference implementation of the Cholesky factorization for machines with hierarchical levels of memory is part of the LAPACK library [10]. It consists of a succession of panel (or block column) factorizations followed by updates of the trailing submatrix.

In the Chameleon library, the Cholesky routine follows a tile-based scheme in which the $n \times n$ matrix to be factorized is split in multiple submatrices, or tiles, of size $nb \times nb$ [1]. To enable the concurrent use of all the computational units on a heterogeneous platform, the Chameleon library splits the work into smaller tasks, which correspond to the computational kernels involved in performing the decomposition: `potrf`, `trsm`, `gemm` and `syrk`. The complexity of scheduling these tasks, solving data dependencies and of data consistency is delegated to StarPU [3]. By default, it uses the *lws* scheduler, because it provides correct load balancing and locality, and also takes into account priorities, although different scheduling policies can be selected, such as *eager, prio, ws,...* However, none of them considers the selection of the best value to use for the tile size, *nb*, in the Cholesky routine. Therefore, it is necessary to develop optimization strategies to suitably select the best value for *nb*.

Figure 1 shows the steps for executing a linear algebra routine of Chameleon (such as the Cholesky decomposition) using the StarPU runtime system. Each routine is computed following a tile-based algorithm. Then, a direct acyclic graph is created with the dependencies between tasks and, finally, these tasks are scheduled using the StarPU runtime system, which executes each of the tasks in the different computational units with the use of optimized implementations of the basic linear algebra routines.

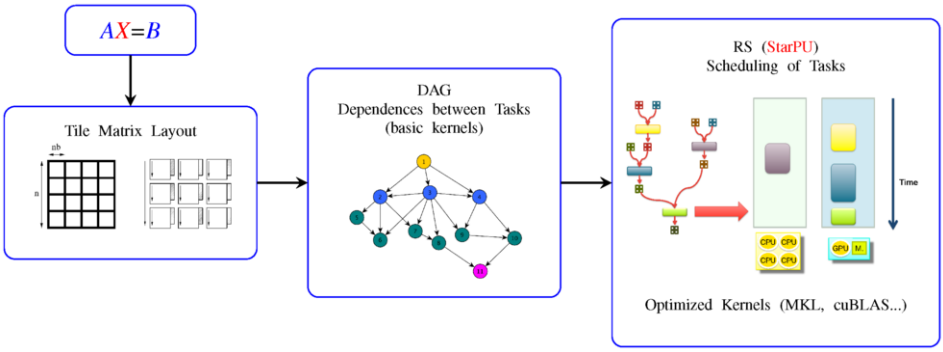


Figure 1. Execution of a linear algebra routine in Chameleon.

3. Training Strategies

Our study focuses initially on applying training strategies to select the value for the tile size, *nb*, of the Cholesky routine for a number of selected and representative problem sizes. Empirical and simulated approaches are combined with exhaustive and pruned searching methods. So, four resulting training strategies are considered:

S1: Empirical+Exhaustive

A naive approach for tuning the library would be to collect empirical data exhaustively from a large set of experiments for representative problem sizes. The routine is experimentally executed on the heterogeneous platform using a set of tile sizes for each selected problem size, n. As a result, the performance for each pair (n, nb) is obtained and stored for further use.

S2: Empirical+Pruned

Usually the time employed by *S1* approach is very high. So, to reduce the experimentation time, ensuring at the same time results close to the exhaustive ones, a pruned strategy of the search space can be used. It exploits the fact that the tile size *nb* trades off the performance of an individual task (the higher the *nb*, the higher the performance of the task) with the concurrency between tasks (the smaller the *nb*, the wider the DAG of tasks). We therefore consider a strategy similar to the one employed on multicore-only platforms [11]. In the proposed approach, the search starts with the lowest problem size (e.g, $n = 2000$) and seeks the optimum tile size *nb*. Once a given problem size n has been explored, the next problem size ($n = 4000, 8000, \ldots$, in that order) is investigated. The key idea is that the search continues with the next problem size using as its starting-point the best tile size selected for the previous problem size. For instance, if the optimum tile size for $n = 4000$ is $nb = 256$, the pruned strategy directly assesses $nb = 256$ (not evaluating the previous value for the tile size) for $n = 8000$ and, then, the next tile size (in increasing order) is considered until it reaches a tile size with which the performance is not improved. Then, the process continues with the next problem size using as starting-point the best *nb* obtained for the previous problem size, and so on.

S3: Simulated+Exhaustive

The *S1* and *S2* strategies require access to a heterogeneous platform for the experiments. Instead, we can use a simulator and apply an off-line training strategy on a separate laptop. For this purpose, we use the SimGrid simulator [9]. During an empirical phase, for each tile size and set of problem sizes, a very quick sampling of data is collected for each of the routine kernels and the generated information is stored in files called *codelets*. The information stored is based on performance models of the execution time, which can be history-based or regression-based, and is used by the simulator to estimate the duration of a task. After that, the simulator could be used over these *codelets* on a personal laptop to estimate the performance for each pair (n, nb), so reducing the experimentation time with respect to the empirical approaches.

S4: Simulated+Pruned

This strategy is applied in the same way as *S2*, but using the information collected after applying the *S3* strategy. The goal is to further reduce the search time required to obtain the tile size for each problem size while maintaining a good performance estimation.

4. Experimental Results

The experiments were carried out on a heterogeneous node with 12 CPU cores (2 hexa-core) and 6 NVIDIA GPUs (4 GeForce GTX590 and 2 Tesla K20c) using the set of problem sizes $\{2000, 4000, 8000, \ldots, 32000\}$ and a fixed set of tile sizes $\{208, 256, 288, 320, 384, 448, 512, 576\}$.

4.1. Searching the Tile Size

The results obtained for the Cholesky routine of Chameleon using the exhaustive strategies (*S1* and *S3*) are shown in Figure 2. Figure 2a shows the results obtained with the empirical *S1* strategy. The performance (y-axis) for each problem size significantly depends on the tile size *nb*, reaching the asymptotic value when using the highest tile sizes in larger problem sizes. Figure 2b, instead, shows the results when using the simulated *S3* strategy. The performance achieved for each problem size is very similar to that obtained with the empirical strategy, especially for large matrix and tile sizes.

If we consider the empirical and simulated approaches using the pruned strategy, satisfactory results are obtained. Figure 3 shows that the performances obtained with the *S1* and *S2* strategies perfectly overlap, but both approaches use the actual platform to perform the search for the tile-size values. The simulated *S3* and *S4* strategies, however, return very decent tuning without (almost) using the actual compute node during the training phase, achieving performance results similar to the empirical ones.

The results obtained with the four strategies are similar in terms of performance, but not in terms of the search time required. Table 1 compares the *nb* value selected for each problem size using each one of the strategies (values also shown in Figure 3) and the time employed in finding each value during the search process. In the empirical approaches (*S1* and *S2* strategies) the selected values for the tile size are identical, but the search time employed is lower when using the pruned strategy. The *S1* strategy uses 30 minutes of

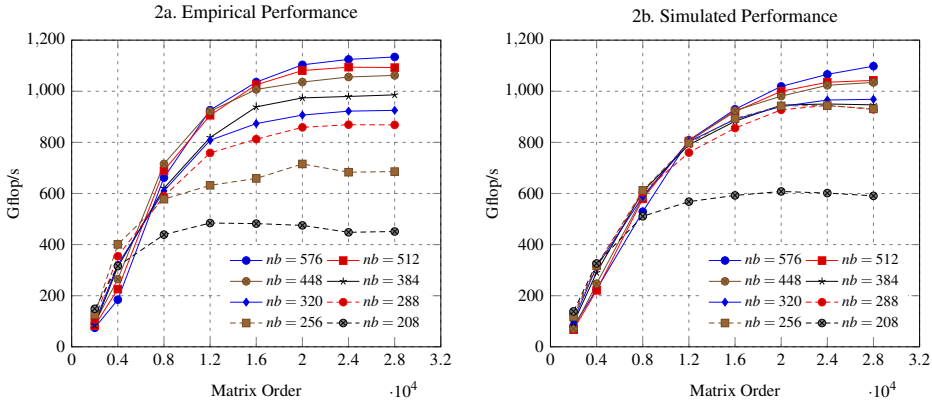


Figure 2. Empirical (2a) and simulated (2b) performance of the Cholesky routine of Chameleon using a fixed set of *nb* values for each problem size.

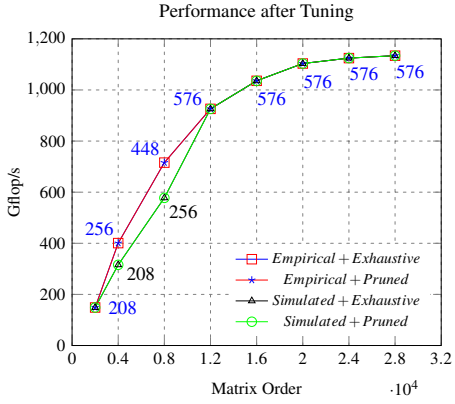


Figure 3. Performance of the Cholesky routine of Chameleon after applying each of the four considered strategies (the selected *nb* value by each strategy is also displayed).

platform time to find the *nb* values. However, by using the *S2* pruned strategy, the time is reduced to 154 seconds. With the simulated approaches (*S3* and *S4* strategies), the values obtained for the tile size are similar to those obtained with the empirical approaches, but with much less search time, and the search can be done on a separate laptop. For small and medium problem sizes, the *nb* values differ slightly from those obtained with the empirical approaches due to small variations in the performance estimated by the simulator, but the time employed in searching for the *nb* values is reduced. With the *S3* strategy, the simulation time is about 30 minutes and with the *S4* pruned strategy it is reduced to only 84 seconds. There are some entries in the table for the *S2* and *S4* pruning strategies for which the time is not displayed (represented as '-'). This is because when the search reaches the highest value considered for *nb*, this value is used for higher problem sizes, so no time is spent searching for them.

Table 1. Value for the tile size (*nb*) for each problem size using the different strategies and execution time (in seconds) employed during the search for each strategy and problem size.

	*S*1		*S*2		*S*3		*S*4	
n	*nb*	time	*nb*	time	*nb*	time	*nb*	time
2000	208	67	208	17	208	2	208	1
4000	256	71	256	25	208	5	208	3
8000	448	89	448	67	256	24	256	13
12000	576	125	576	45	576	67	576	67
16000	576	184	576	-	576	143	576	-
20000	576	273	576	-	576	321	576	-
24000	576	400	576	-	576	470	576	-
28000	576	581	576	-	576	753	576	-

4.2. Testing the Training Strategies

Once the routine has been trained for a set of problem sizes and different values for the tile size, we test the validity of the training with a set of experiments for an intermediate set of problem sizes. This testing process consists of applying an interpolation process to the information stored for the tile size during the search process performed by each of the training strategies. The goal of this tuning strategy is to analyze how far the results obtained (in terms of Gflops) with the tile size selected are with respect to the experimental optimum. Table 2 compares the results obtained. In general, the selected *nb* value (*nb* column) differs slightly from the optimum (*nb_opt* column). Furthermore, the deviation of the performance with the tuning strategy with respect to the experimental optimum (*dev* column) is quite small, mainly for large problem sizes (between 1% and 5%). Nevertheless, this interpolation process can be considered a valid tuning strategy because it allows fast prediction of a good value for *nb* for a given problem size.

Table 2. Comparison of the tile size (*nb*) selected by applying an interpolation process with respect to the experimental optimum. The *dev* column shows the deviation of the performance (in %) obtained with each tuning strategy with respect to the highest experimental performance.

		*S*1		*S*2		*S*3		*S*4	
n	*nb_opt*	*nb*	*dev*	*nb*	*dev*	*nb*	*dev*	*nb*	*dev*
3000	240	232	5	232	5	208	2	208	2
6000	288	352	12	352	12	232	15	232	15
10000	512	512	0	512	0	416	5	416	5
14000	672	576	3	576	3	576	3	576	3
18000	672	576	3	576	3	576	3	576	3
22000	896	576	1	576	1	576	1	576	1
26000	896	576	1	576	1	576	1	576	1

5. Extensions to the Experimental Study

So far, the experiments have been carried out considering a fixed set of values for the tile size and using all the computing units of the heterogeneous node. Results are satisfactory for the proposed training strategies, but when an intermediate set of problem sizes is used

(e.g, by a user), the decisions in the selection of the value for the algorithmic parameters are not always the best ones [12,13]. In this section we analyze how to improve the tuning process, either by adjusting the search for the tile size or by selecting the appropriate number of computing units to use.

5.1. Other Values for the Tile Size

Experimental results show that when an interpolation process is applied for some problem sizes, the deviation in Gflops with the selected *nb* is a little too far from the optimum. Rather than searching for the optimum value for *nb*, we can consider neighboring values and analyze the variability obtained in terms of performance in order to decide the best value for *nb*. Table 3 shows the deviation obtained for each one of the intermediate problem sizes when considering three values to the left and to the right of the interpolated one (*nb* column). The value used to obtain the next (or previous) neighbor is set according to the problem size. For $n \leq 5000$ a value of 8; for $5000 < n < 10000$ a value of 16 and for $n \geq 10000$ a value of 32. Therefore, the distance value used for *nb* could be automatized according to the range of problem sizes considered. A positive value in the deviation means an increase in performance over that achieved with the selected *nb* by the interpolation process, and a negative value means a decrease in the performance. In general, when $n \leq 10000$, the lowest deviation (or best improvement) is achieved with the immediately previous neighbor to *nb*. Instead, when $n > 10000$, the best neighbor is usually the third in increasing order with respect to *nb*. Therefore, the interpolation process could be slightly adjusted for a better selection of the tile size to use for a given problem size. For that, a search process could be applied, starting from the interpolated values for *nb* and considering both the distance value for *nb* in function of the problem size, and a percentage value for cases where an extreme value for *nb* is reached, in order to continue exploring in that direction until a new value decreases the performance.

Table 3. Comparison of the performance variability (in %) obtained with several neighbors with respect to the selected tile size (*nb*).

n	nb_1	dev	nb_2	dev	nb_3	dev	nb	nb_4	dev	nb_5	dev	nb_6	dev
3000	208	+4	216	+2	224	+3	232	240	+5	248	-10	256	-6
6000	304	+6	320	+7	336	+12	352	368	-4	384	+2	400	+1
10000	416	-5	448	-2	480	-1	512	544	-6	576	-3	608	-4
14000	480	0	512	+1	544	-2	576	608	-1	640	+1	672	+3
18000	480	-3	512	-4	544	-7	576	608	-3	640	-3	672	+3
22000	480	-6	512	-3	544	-8	576	608	-3	640	-2	672	-1
26000	480	-7	512	-3	544	-8	576	608	-3	640	-3	672	-1

5.2. Other Algorithmic Parameters

As mentioned, the StarPU runtime system is able to efficiently schedule the kernels among the available computational units of the system, but it tends to execute them using all the devices of the node. Our proposal is to analyze whether an appropriate selection of the number of computational units to use for each problem size allows better performances with an efficient use of the computational resources. We apply a selective search process which consists of successively adding computing units (CPU and each

GPU), following an increasingly powerful order. It is important to notice that the current version of StarPU does not support the data-transfer model between GPUs implemented on the latest NVIDIA devices. So, the process starts by searching for the best tile size for the current problem size and platform configuration (the initial device considered is the CPU). Then, the search continues by adding the most powerful GPU, and the best value for the tile size is searched for by applying a bi-directional guided search, using as starting-point the best value obtained for the previous platform configuration. When the process finishes, both the best platform configuration and tile size for each problem size are obtained. Table 4 shows the results of applying this tuning process for a set of problem sizes on the heterogeneous node considered (12 CPU cores and 6 NVIDIA GPUs: 4 GeForce GTX590, numbered 0, 2, 3 and 4, and 2 Tesla K20c, numbered 1 and 5). It is important to note that StarPU only uses physical cores of the CPU (without hyper-threading), therefore, the number of CPU cores is adjusted depending on the number of GPUs used, since one CPU core is intended to manage one GPU. For the set of problem sizes considered, the search process takes about 185 minutes, but each experiment is performed 10 times in order to obtain representative means for the Gflops. For small problem sizes, a subset of the computing units of the node is selected, but when the problem size increases it tends to use all the computing units. Column 'Tuned_Gflops' shows the performance obtained with the configuration selected by the tuning process, and 'Cham_Gflops' shows the performance of the routine when it is executed with the same tile size but using the default platform configuration. For all problem sizes, the best performance is obtained in the tuned case even when the whole platform is used, since by default StarPU schedules the tasks among workers (each of the GPUs) based on data dependencies, but does not take into account the computational power of the computing units. Therefore, despite the search time employed, this tuning process is a good strategy to consider if we want to efficiently use the computing units of the node.

Table 4. Performance obtained for each problem size with the best configuration selected (*Tuned_Gflops*) and using the default platform configuration (*Cham_Gflops*).

		Computing Units			
n	*nb*	*CPU_Cores*	*GPU_IDs*	*Tuned_Gflops*	*Cham_Gflops*
1000	112	12	{−}	76	46
2000	192	9	{1,5,0}	164	117
3000	192	8	{1,5,0,2}	285	196
4000	240	7	{1,5,0,2,3}	412	352
5000	256	6	{1,5,0,2,3,4}	545	465
6000	256	6	{1,5,0,2,3,4}	626	554
7000	320	6	{1,5,0,2,3,4}	687	608
8000	320	6	{1,5,0,2,3,4}	753	682
9000	304	6	{1,5,0,2,3,4}	791	713
10000	304	6	{1,5,0,2,3,4}	834	758
11000	512	6	{1,5,0,2,3,4}	880	803
12000	576	6	{1,5,0,2,3,4}	909	896

6. Conclusions

Task-based libraries allow us to efficiently schedule and execute linear algebra kernels on heterogeneous platforms, but they are not able to decide the best values for some algorithmic parameters of the routines, such as the tile size *nb*. In this work we propose some tuning strategies for selecting satisfactory values for the tile size on tile-based routines. We also analyze the best number of computing units to use for each problem size on a heterogeneous platform. The Cholesky routine is considered as proof of concept, using highly optimized implementations of the Cholesky factorization both for multicore and GPU. We focus on the tile size as the algorithmic parameter to optimize because this routine is executed in the Chameleon library by following a tile-based algorithm. The experimental results obtained are satisfactory, showing that the pruning strategies (both with empirical and simulated approaches) are good options to select the value for the tile size for each problem size in a short time, allowing us to obtain performances close to the experimental optima. Also, we show that a good selection of the computing units of the node for each problem size (mainly for medium problem sizes) is paramount if we want to efficiently use the computational resources with a better exploitation of the system. Our aim is to apply the proposed methodology to other linear algebra routines (such as LU or QR factorization) and to integrate the tuning process inside the Chameleon library, extending the study of selecting which computing units to use to bigger heterogeneous platforms (with a large number of computational resources).

Acknowledgment

This work was supported by the Spanish MCIU and AEI, as well as European Commission FEDER funds, under grant RTI2018-098156-B-C53.

References

[1] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault, and Stanimire Tomov. Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In *GPU Computing Gems*, volume 2. Morgan Kaufmann, September 2010.

[2] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38 – 53, 2009.

[3] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[4] Intel MKL Web Page. `https://software.intel.com/en-us/mkl`. [Online; accessed 29.07.2019].

[5] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180:012037, jul 2009.

[6] CUDA Toolkit Documentation. `https://docs.nvidia.com/cuda/cublas/index.html`. [Online; accessed 29.07.2019].

[7] Hartwig Anzt, Blake Haugen, Jakub Kurzak, Piotr Luszczek, and Jack J. Dongarra. Experiences in autotuning matrix multiplication for energy minimization on GPUs. *Concurrency and Computation: Practice and Experience*, 27(17):5096–5113, 2015.

[8] Brice Videau, Kevin Pouget, Luigi Genovese, Thierry Deutsch, Dimitri Komatitsch, Frdric Desprez, and Jean-Franois Mhaut. BOAST: A Metaprogramming Framework to Produce Portable and Efficient Computing Kernels for HPC Applications. *The International Journal of High Performance Computing Applications*, 32(1):28–44, 2018.

[9] Luka Stanisic, Samuel Thibault, Arnaud Legrand, Brice Videau, and Jean-François Méhaut. Faithful Performance Prediction of a Dynamic Task-Based Runtime System for Heterogeneous Multi-Core Architectures. *Concurrency and Computation: Practice and Experience*, 27(16):4075–4090, 2015.

[10] Ed Anderson, Zhaojun Bai, Christian H. Bischof, L. Susan Blackford, James Demmel, Jack J. Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, A. McKenney, and Danny C. Sorensen. *LAPACK Users' Guide, Third Edition*. Software, Environments and Tools. SIAM, 1999.

[11] Emmanuel Agullo, Jack Dongarra, Rajib Nath, and Stanimire Tomov. A Fully Empirical Autotuned Dense QR Factorization for Multicore Architectures. In *Euro-Par 2011 Parallel Processing*, volume 6853 of *Lecture Notes in Computer Science*, pages 194–205, Aug 2011.

[12] Jesús Cámara, Javier Cuenca, Domingo Giménez, Luis Pedro García, and Antonio M. Vidal. Empirical installation of linear algebra shared-memory subroutines for auto-tuning. *International Journal of Parallel Programming*, 42(3):408–434, Jun 2014.

[13] Gregorio Bernabé, Javier Cuenca, Luis-Pedro García, and Domingo Giménez. Auto-tuning techniques for linear algebra routines on hybrid platforms. *Journal of Computational Science*, 10:299 – 310, 2015.

Parallel Algorithms

This page intentionally left blank

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200038

Batched 3D-Distributed FFT Kernels Towards Practical DNS Codes

Toshiyuki IMAMURA, [a,b,1] Masaaki AOKI [b], Mitsuo YOKOKAWA [b]
[a] *RIKEN, Center for Computational Science*
[b] *Kobe University*

Abstract. This work introduces a new idea of batched 3D-FFT with a survey of data decomposition methods and a review of the states-of-arts high performance parallel FFT libraries. Besides, it is argued that the particular usage of multiple FFTs has been associated with the batched execution. The batched 3D-FFT kernel, which is performed on the K computer, shows 45.9% speedup when N and P are 2048^3 and 128, respectively. The batched FFT allows the developer to take advantage of a flexible internal data layout and scheduling to improve the total performance.

Keywords. 3DFFT, parallel FFT library, Batched execution, Optimization of invocation of multiple FFTs, FFTE-C

Introduction

Three-dimensional Fast Fourier Transform (hereafter 3DFFT) is widely used in the field of computational science, such as the direct numerical simulation (DNS) of incompressible flow turbulence. A lot of studies claim the necessity of a high performance 3DFFT library for large scale and high-resolution simulations (for example, see [1]).

The 1D-FFT is naturally one of the numerical kernels influenced by memory bandwidth, which has a computational complexity of $O(N \log N)$ and memory transfer with $O(N)$ data length. Practical usage of memory, such as the cache blocking and a higher radix FFT kernel, was investigated to optimize the single-node performance. For the multi-process parallelization of the 3DFFT, we have two technical issues: i) the short of parallelism, and ii) the significant communication overhead. One conventional solution to them is to select a higher dimensional data decomposition. It is known that the 2D and 3D decomposition yield a higher degree of parallelism than the simple 1D decomposition. To select the appropriate communication scheme is significant for the multiple-node operation as well. On the other hand, there is no much room to improve the performance of the single FFT kernel itself. However, we can apply more advanced parallel optimization techniques on an application with multiple independent FFTs being frequently called. For instance, the implementation technique and internal advanced task scheduling may result in significant performance improvement when a large number of FFTs are called.

[1] 7-1-26 Minatojima-minami-machi, Chuo-ku, Kobe, Hyogo 650-0047, Japan;
E-mail: imamura.toshiyuki@riken.jp

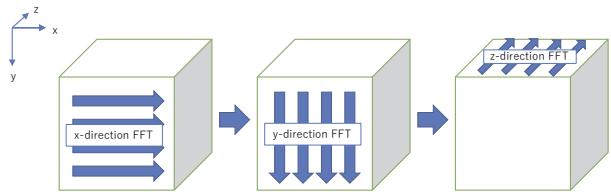


Figure 1. Schematic diagram of a 3DFFT

Our contributions in this study are mainly summarized in two parts. First, we develop a high performance three-dimensional FFT library with several possible data-distribution. Second, we introduce the idea of the batched execution, which exploits a flexible internal data layout and scheduling. In particular, the latter works overlapping with communication and computation for several independent 3DFFTs. Eventually, we obtain an excellent speedup on the K computer by the overlap technique.

The rest of this paper consists of an overview of 3DFFT, methodology, implementations, and states-of-arts libraries. Then, a new idea of Batched execution to reduce several performance bottlenecks is introduced in section 2. In section 3, performance investigation and benchmark on the K computer are demonstrated. Section 4 concludes this study and shows some future works untouched in this work.

1. Methodology and Implementation of 3DFFT

Most of the recent advanced FFT libraries support 2D and 3D parallel FFT routines, for example, FFTW[2], FFTE[3], 2decomp[4], P3DFFT[5], et al. These libraries exploit many optimizing features such as on SIMD, thread parallelism, and MPI parallel as well. Since supercomputer is composed of interconnected computing nodes, we must divide the technical issues into optimal performance on a single node, then optimize them. In this work, we make use of the FFTE implementation [3] as a portable and high performance 1DFFT kernel.

1.1. Implementation of 3D FFT

The 3D DFT is defined as

$$Y(\beta_x,\beta_y,\beta_z)=\sum_{\alpha_x=0}^{n_x-1}\sum_{\alpha_y=0}^{n_y-1}\sum_{\alpha_z=0}^{n_z-1}X(\alpha_x,\alpha_y,\alpha_z)\omega_{n_x}^{\alpha_x\beta_x}\omega_{n_y}^{\alpha_y\beta_y}\omega_{n_z}^{\alpha_z\beta_z} \tag{1}$$

where, X and Y refer to the three-dimensional input and output data with an $n_x \times n_y \times n_z$ rectangular coordinate, respectively. The 3D-DFT consists of $n_y{\cdot}n_z$, $n_z{\cdot}n_x$ or $n_x{\cdot}n_y$ one-dimensional FFTs in each direction as shown in Fig. 1. For the 1D-FFT in each direction, they must be realigned or redistributed so that the data are obligatory arranged in a consecutive manner on the memory hierarchy.

Parallel FFT has another issue due to the data decomposition across multiple processors. We must *re-distribute* the data so that the one-dimensional target data to be transformed is gathered on a local memory of a particular process. In the case that the process

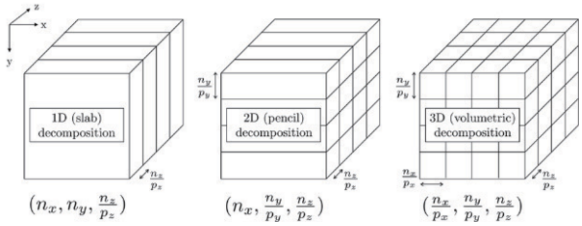


Figure 2. 1D-, 2D-, 3D data decompositions

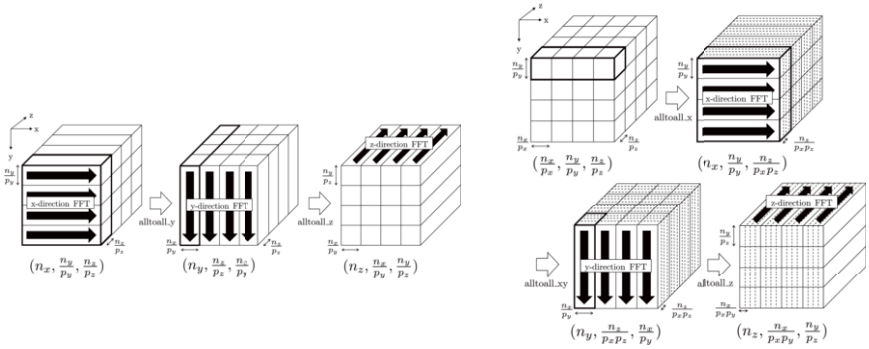


Figure 3. Data redistribution (left: pencil decomposition, right: volumetric decomposition)

grid is associated with the regular spatial grid, all-to-all personalized communication [6], which is defined as a collective communication `MPI_Alltoall` in the MPI specification, is required. Generally, we consider three ways to distribute 3D data: the slab decomposition, the pencil decomposition, and the volumetric decomposition, as presented in Fig. 2. What is more, for the volumetric decomposition, we can select an appropriate communication algorithm either a naïve one or the Jung's algorithm [7] according to the network configuration.

1.2. Pencil Decomposition versus Volumetric Decomposition

The pencil decomposition is used in major FFT libraries, such as FFTE[3] and 2decomp[4]. In the pencil decomposition, two alltoall communications and four transpositions are essential (Fig.3, left). On the other hand, in the volumetric decomposition, each dimensional FFT on a single process must be done due to an obligation of a consecutive data arrangement. This data retrieval on a single process results in worse performance than the pencil decomposition even if we expect a higher degree of parallelism. The naïve communication pattern, which is done in an alternating direction fashion, needs five alltoall collectives. This method is called as **1d-alltoall**. An advanced method can be applied within three times of communication by combining the 1D grouping and the 2D grouping, which is Jung's algorithm and we call it **2d-alltoall** (Fig.3, right). When we suppose to use a 3D or higher dimensional torus network with two types of alltoall communications based on pair-wise protocols and 3D-Torus protocols presented in [6], the communication costs for each decomposition are estimated as in Table 1 (T_{1D}: slub,

Table 1. All-to-All cost for multi-dimensional decomposition FFTs

	Pairwise	3D-Torus
T_{1D}	$(\alpha_0+\alpha P)+\beta NP^{-1}$	$3\alpha\sqrt[3]{P}+(3/2)\beta NP^{-2/3}$
T_{2D}	$2(\alpha_0+\alpha\sqrt{P})+2\beta NP^{-1}$	$4\alpha\sqrt[4]{P}+2\beta NP^{-3/4}$
T_{2D+1d}	$5(\alpha_0+\alpha\sqrt[3]{P})+5\beta NP^{-1}$	$5\alpha\sqrt[3]{P}+(5/2)\beta N^{-2/3}$
T_{3D+2d}	$3\alpha_0+\alpha(2\sqrt[3]{P}+P^{2/3})+3\beta NP^{-1}$	$4\alpha\sqrt[3]{P}+2\beta NP^{-2/3}$

T_{2D}: pencil, T_{3D+1d}: volumetric+1d-alltoall, and T_{3D+2d}: volumetric+2d-alltoall). Here, P and N represent the number of processes and the global problem size (N^3), respectively. Also, "α" (or "α_0") and β refer to the communication overhead [sec] and the reciprocal of throughput [sec/Byte], respectively. This analytic result suggests that the pencil decomposition is likely to minimize the communication cost in most cases.

On the contrary to previous discussions, it is natural for practical applications to arrange the volumetric data distribution because of its simple and straightforward mapping of the volume of data onto a processor grid. There is no clear answer about this inconsistent issue, but it should be distinguishable from the cases where spatial division and FFTs are separable issues either when we call FFTs or investigate complex numerical integration by the spectral method. Moreover, both 3D+1d and 3D+2d include redistribution from the volumetric decomposition to the pencil decompositions, essentially. We recognize a room of optimization in the internal selection of spatial data division and management of invoking multiple FFTs. The next section follows this argument and shows our implementation of the Batched 3D-FFT routine, which is intended to reduce the cost of the communication overhead.

2. Batched FFT

2.1. Idea of general Batched execution

The idea of Batch processing is one of the task-oriented programming methodologies and task scheduling where independent task invocations are queued on a many-core processor or GPU accelerators. A typical example of Batch processing is multitasking/multithread parallel processing by using Batched BLAS [8,9]. One of the primary purposes of Batched BLAS is to increase the processing efficiency of all tasks by allocating extremely coarse-grained tasks to the appropriate granularity of core groups and increase system availability by an efficient parallel task scheduling. Furthermore, for small-scale problems, data structures can be internally converted, and hardware-specific techniques, such as SIMD and cache lines, can be applied accordingly. Due to this, smaller problems can take advantage of SIMD and cache lines as well.

Many FFTs are frequently applied to calculate convolutions in frequency space instead of calculating nonlinear terms in the spectral method. The idea of a Batched execution is practical to reduce the overall execution time because the expensive startup cost is concealed behind the processing time of Batched components. In fact, a typical DNS code such as in [1,10] has six inverse-FFTs and twelve FFTs without dependency, thus overlapping is available in any intra-group FFT between the FFT+transposition operations performed internally and the communication as shown in Fig. 4.

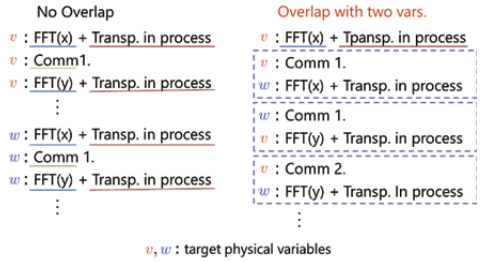


Figure 4. Simple sketch of overlapping data transposition and a 1D-FFT

2.2. *Preliminary Implementation of Batched FFT APIs*

FFTW and p3dfft define 'many' API, which is designated to simplify thread parallelization in units of plans, but the part that passes multiple plans collectively to functions can be classified as a Batch processing. The 'many' API functions of FFTW are implemented as a serialized version; however, an MPI parallel version performs only for a one-dimensional FFT. On the other hand, the latest p3dfft also supports multi-dimensional FFT. Cufft for CUDA-GPUs provides a batched FFT functionality for small data in order to accelerate the deep learning computing.

Suppose that we have independently computable variables, and those array sizes and distribution methods are the same. As shown in the previous section, the parallel 3DFFT operates a 1DFFT on a local variable, and the communication operation for another variable works independently. Therefore, it is possible to overlap both operations. Ultimately, we aim to conceal the full part of the calculation (local 1DFFTs) behind the communication and increase the total system utilization. Fig. 5 illustrates the overlapping of communication and calculation when eight variables are divided into four groups, with each containing two FFT tasks. When we have n independent 3DFFT tasks and $T_{\text{FFT-x}} = T_{\text{FFT-y}} = T_{\text{FFT-z}} \ll T_{\text{AlltoAll}}/2$ holds, the optimal time is smaller than $2T_{\text{FFT-*}} + nT_{\text{Alltoall}}$. In general, the cost of successive or combined AlltoAll operations is degraded. Thus, this approach is more effective if no significant overhead occurs in the actual implementation (overlap mechanism and other operations such as data copy, merging, sorting, and so on).

We devised two different implementations, i) ALLTOALLW + derived data types, ii) buffer reordering, as well as using synchronous and asynchronous ALLTOALL. Specifically, in the implementation i), we introduce a user-derived MPI data type for the sender and receiver to invoke only one `MPI_Alltoallw` for the transposition of two three-dimensional arrays. The implementation ii) introduces a send buffer and a receive buffer to realign two array data into one serialized array. The scheduling of the FFT tasks includes another grouping parameter, which specifies how many tasks are packed. As shown in Fig. 5, the grouping parameter affects parallelism and concurrency of the Batch processing. Thus, the scheduling algorithm is tunable, and we can optimize the Batched FFT by choosing an appropriate grouping parameter.

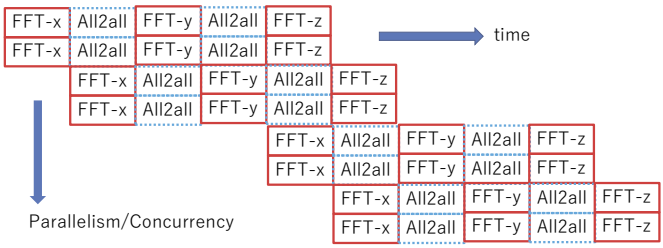


Figure 5. Advanced task scheduning on a Batched execution

3. Numerical Experiment and Discussions

The benchmarks were executed on several processors and a supercomputer as follows,

1. K: The K Computer, Fujitsu Sparc64 XIIIfx (2GHz, 8cores),
2. FX100: Fujitsu Sparc64 IXfx (1.975GHz, 32cores),
3. KNL: Intel Xeon Phi 7250 (Knight Landing) (1.4GHz, 68cores), and
4. Skylake: Intel Xeon Gold 6148 (2.4GHz, 20cores, 2sockets).

The execution time of data-decompositions of a 3-dimensional real FFT was measured on the K computer with a large number of computational nodes, from 64 to 1,024. The number of threads was 8 and fixed as the same as the number of physical cores. On each system, vendor-supplied compiler, for example, Fujitsu C/C ++ compiler version 1.2.0 or Intel C/C++ compiler version 18.0.1 was used with the most reasonably optimization options, `-Kfast -Kopenmp -Kmfunc=2`, or `-O3 -fma -xHOST -axMIC-AVX512`, respectively. The system specification of the K computer is shown in the Appendix.

3.1. A 1DFFT kernel (FFTE-C)

FFTE-C is a C language version of FFTE [3]. FFTE comprises radix 2, 3, 4, 5, and 8 kernels, and is capable of doing an FFT for 1-, 2-, and 3-dimensional data of length $n = 2^p \cdot 3^q \cdot 5^r$. Since the source code was written in the conventional Fortran77 (partly in Fortran90 or PGI CUDA Fortran), we were motivated to remake a C version in order to perform pointer-oriented data handling for a Batch processing and more advanced task scheduling. Furthermore, we newly introduced a radix 16 FFT kernel.

The roofline model defined by $\min\left\{F_{\text{peak}}, B_{\text{peak}} \times \text{Operational intensity}\right\}$ reflects the performance estimation of a target program that is supposed to be possibly bounded by the memory bandwidth [11]. Here, F_{peak}, B_{peak} refer to the theoretical peak performance of floating-point calculation and the theoretical peak of memory bandwidth, respectively. Table 2 summarizes the values of parameters (the number of issues of loads, stores, real additions, and real multiplications) and Byte/Flop ratio for the innermost loop of the target 1DFFT code. The table shows that a higher radix kernel has a more significant operational intensity. Thus, the performance upper bound of higher radix FFT raises according to the roofline model analysis.

We measured the performance of 1D FFTE-C kernel for a 16,777,216(=16^6) dimensional data on each of four processors, and Table 3 presents their roofline performance

Table 2. The number of LDs, STs and floating-point operations and byte/flop ratio for the innermost loop of radix-2, 4, 8, 16 FFT kernels

	Radix-2	Radix-4	Radix-8	Radix-16
Loads	4	8	16	32
Stores	4	8	16	32
Multiplications	4	12	32	84
Additions	6	22	66	174
Byte/Flop ratio	6.400	3.765	2.612	1.984

Table 3. Roofline Performance Evaluations by using four types of processors

	K	FX100	KNL	Skylake
F_{peak} (1CPU) [GFlop/s]	128	1011	3046	1536
B_{peak} (1CPU) [GB/s]	46.6	302	490	87
Flop counts [GFlop]	1.42	1.42	1.42	1.42
Total memory access [GB]	2.15	2.15	2.15	2.15
Roofline [GFlop/s]	34.09	232.02	421.06	82.11
Roofline [ms]	57.20	8.52	4.85	25.61
Experiment [ms]	288.82	16.55	7.48	29.43
Ratio Experiment/Roofline	0.20	0.52	0.65	0.87

evaluations. Since our FFTE-C kernel was mainly developed on an Intel Xeon Gold (Skylake) and tuned up thoughtfully, 87% of the roofline performance is acceptable as well as Intel Xeon Phi (KNL) showed similar roofline performance.

On the other hand, on Fujitsu processors, especially, Sparc64 VIIIfx, the same code has achieved only 20% of the roofline performance. In the case that the radix is a power of two, the six-step FFT, which is employed in the FFTE library, also folds the array internally with a stride of multiply of the power of two. In such data arrangement, multiple FFTs processed simultaneously conflict resources of the 2 way L1 cache frequently. On another Fujitsu processor, FX100 (Sparc64 IXfx), because the L1 cache is doubled in set associativity, the penalty resulting from the cache thrashing is dismissed and overhead has been relaxed.

3.2. Performance improvement of Batched 3DFFT

As a result of numerical experiments (Fig.6) on the K computer, the pencil decomposition (indicated as 2decomp) requires more time for calculations other than communication; in other words, the communication time achieves about half of the total execution time. Also, the pencil decomposition shows the highest performance regardless of the problem size and the number of nodes. When the problem size is $1,024^3$, the calculation time is more significant other than communication for 2D decomposition. When the problem size is $2,048^3$, the communication time occupies about half of the total execution time. The performance per compute node is generally low from the performance analysis of 1D-FFTE, but nevertheless, the ratio of the communication time to the total execution time degrades because the network performance of Torus interconnects is relatively higher.

Fig.7 demonstrates that the execution time of the Batched-FFT is reduced by more or less 30% compared with that without overlap. Even if `MPI_Ialltoall` does not work

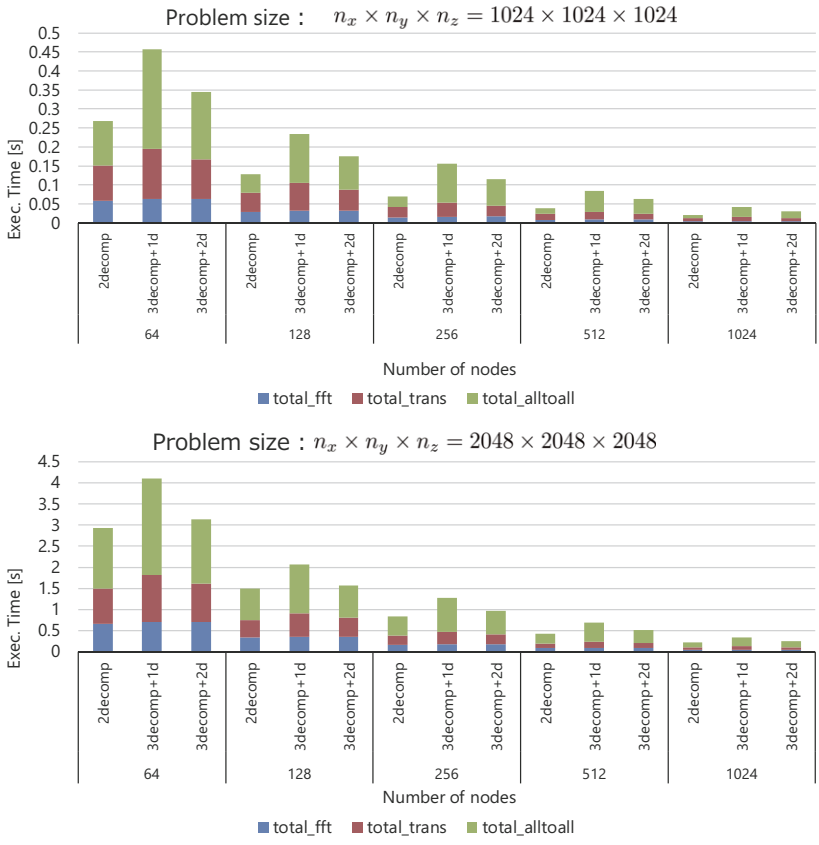


Figure 6. Performance comparison among 1D-, 2D-, 3D decomposition on K (Upper: $1{,}024^3$, Bottom: $2{,}048^3$)

as an expected non-blocking collective function on the K computer, we achieved a 45.9% speedup when P=128 with an expense of one physical processing core out of 8 for communication. It concludes a remarkable performance improvement by the Batch processing with a simple scheduling mechanism.

Fig. 8 shows the experimental result of the advanced Batch processing, in which 12 variables were targeted, and grouping and pipeline scheduling presented in Fig. 5 were performed. Although the two methods shown in the previous section were implemented, the advantages of grouping has not been confirmed, instead, the startup overhead was slightly increased though it was initially expected to be reduced. Therefore, it is necessary to optimize the data arrangement and the alltoall function itself by using other derived types.

4. Summary

In this study, we surveyed data decomposition methods of the 3D-FFT and developed a high performance parallel FFT library based on the pencil decomposition. Besides, a

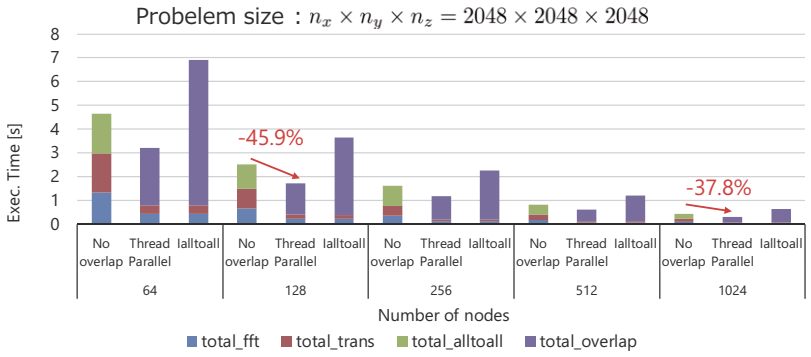


Figure 7. Benchmark result of a Batched 3DFFT for two variables on K

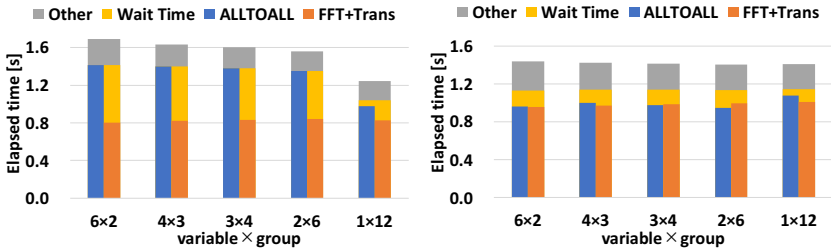


Figure 8. Preliminaly result of Advanced Batched 3D-FFT implementations on K (left: derived type, right: reordering)

particular usage of multiple FFTs has been associated with the idea of the Batched execution. The Batched 3D-FFT kernel performed on the K computer gains 45.9% speedup in performance when $N = 2,048^3$ and $P = 128$. The Batched FFT allows the developer to take advantage of a flexible internal data layout and pipeline to improve the total performance. Currently, the supported functions and possible optimization are limited. More advanced optimization by handling abstracted multiple FFT operations and performance improvement are our future works as well as evaluation of real application codes such as DNS codes on the supercomputer Fugaku, which is the successor of the K computer.

Acknowledgments

The authors would like to thank RIKEN technical staff, Mr. Takuma Kano, for his sincere supports. The results of the present study were obtained in part by using the K computer at RIKEN CCS, Oakforest-PACS at the University of Tokyo, and Hokusai GreateWave/GreatFall at RIKEN . The present study was also supported in part by JSPS KAKENHI (JP19H04127).

References

[1] M. Yokokawa, K. Itakura, A. Uno, T. Ishihara, Y. Kaneda, and Y. Hasegawa. 16.4-Tflops direct numerical simulation of turbulence by a fourier spectral method on the earth simulator. In *Proc. 2002 ACM/IEEE conference on Supercomputing (SC'02)*, pages 1–17. IEEE Computer Society Press, 2002.

[2] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. In *Proceedings of the IEEE*, volume 93, pages 216–231, 2005.

[3] D. Takahashi. FFTE: A fast fourier transform package. http://www.ffte.jp/.

[4] N. Li and S. Laizet. 2DECOMP&FFT a highly scalable 2D decomposition library and FFT interface. Cray User Group 2010 conference, Edinburgh, 2010.

[5] D. Pekurovsky. P3DFFT: a framework for parallel computations of fourier transforms in three dimensions. *SIAM Journal on Scientific Computing*, 34(4):C192–C209, 2012.

[6] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing, 2nd edition*. Pearson Education Limited, 2003.

[7] J. Jung, C. Kobayashi, T. Imamura, and Y. Sugita. Parallel implementation of 3D FFT with volumetric decomposition schemes for efficient molecular dynamics simulations. *Computer Physics Communications*, 200:57–65, 2016.

[8] J. Dongarra, I. Duff, M. Gates, A. Haidar, S. Hammarling, N. J. Higham, J. Hogg, P. V. Lara, M. Zounon, S. D. Relton, and S. Tomov. A proposed API for Batched Basic Linear Algebra Subprograms, May 2016.

[9] J. Dongarra, I. Duff, M. Gates, A. Haidar, S. Hammarling, N. J. Higham, J. Hogg, P. V. Lara, P. Luszczek, M. Zounon, S. D. Relton, S. Tomov, T. Costa, and S. Knepper. Batched BLAS (Basic Linear Algebra Subprograms) 2018 Specification. https://www.icl.utk.edu/files/publications/2018/icl-utk-1170-2018.pdf.

[10] M. Yokokawa, K. Morishita, T. Ishihara, A. Uno, and Y. Kaneda. Performance of a two-path aliasing free calculation of a spectral dns code. In *Rodrigues J. et al. (eds) Computational Science ICCS 2019, Lecture Notes in Computer Science*, volume 11539, pages 587–595, 06 2019.

[11] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communication ACM*, 52, 2009.

A. The hardware specification of the K computer

Total peak performance		10.62 PFlops
Number of nodes		82,994
Node	Socket (CPU+ICU)	1
	Performance	128 GFLOPS
	Memory	16GB
	Memory throughput	64GB/s (8 cores)
Interconnect	Product	Tofu Interconnect
	Topology	6D-Mesh/Torus
CPU	Product	SPARC64^{TM} VIII fx
	Cores	8
	Operating frequency	2.0 GHz
	L1 cache (each core)	L1I: 32KB/2way
		L1D: 32KB/2way
	L2 cache	6MB/12way
	SIMD	2DFPs(FMA)×2pipes

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200039

On Superlinear Speedups of a Parallel NFA Induction Algorithm

Tomasz JASTRZĄB [a,1]
[a] *Institute of Informatics, Silesian University of Technology, Gliwice, Poland*

Abstract. The parallel induction algorithm discussed in the paper finds a minimal nondeterministic finite automaton (NFA) consistent with the given sample. The sample consists of examples and counterexamples, i.e., words that are accepted and rejected by the automaton. The algorithm transforms the problem to a family of constraint satisfaction problems solved in parallel. Only the first solution is sought, which means that upon finding a consistent automaton, the remaining processes terminate their execution. We analyze the parallel algorithm in terms of achieved speedups. In particular, we discuss the reasons of the observed superlinear speedups. The analysis includes experiments conducted for the samples defined over the alphabets of different sizes.

Keywords. parallel algorithm, superlinear speedup, nondeterministic automata induction, constraint satisfaction

1. Introduction

Deterministic and nondeterministic finite automata play a crucial role in various practical applications, including artificial intelligence, grammatical inference, and bioinformatics [1,2,3]. The last field of application is particularly interesting, as also stated in [4], since the automata can be used to detect patterns hidden in bioinformatics data. In this context, automata can act as classifiers for previously unseen sequences, or as generators, producing new sequences that may bear some biological meaning.

A nondeterministic finite automaton (NFA) is given by a tuple $A = (Q, \Sigma, \delta, q_0, Q_F)$, where Q is a finite set of states, Σ is an alphabet, $\delta : Q \times \Sigma \to 2^Q$ is a transition function, $q_0 \in Q$ is an initial state and $Q_F \subseteq Q$ is a set of final states [5]. A sample $S = (S_+, S_-)$ consists of two sets of words, where a word w is a finite sequence of symbols defined over the alphabet Σ, set S_+ contains examples, while set S_- contains counterexamples.

The aim of the parallel induction algorithm is to find a minimal NFA consistent with the given sample S. The automaton is *consistent* with S iff it accepts all the examples and rejects all the counterexamples. A word w is accepted by the automaton A iff there exists a sequence of transitions between state q_0 and at least one state $q \in Q_F$ on which the word is read. Otherwise, the word is rejected. The automaton is consistent and *minimal* iff no two states can be merged together without losing the consistency.

[1]Corresponding Author: Tomasz Jastrząb, Institute of Informatics, Silesian University of Technology, ul. Akademicka 16, 44-100 Gliwice, Poland; E-mail: Tomasz.Jastrzab@polsl.pl.

The induction of minimal consistent NFA is known to be hard. It was shown that NFA minimization is impossible from polynomial time and data [6]. It was also shown that even if the sample is given in the form of a deterministic finite automaton, the problem is PSPACE-complete [7]. Hence, the use of parallel computing is vital for efficient solving of the problem at hand.

In the paper, we study a parallel algorithm for solving the minimal NFA induction problem. The algorithm constructs first a set of independent constraint satisfaction problems (CSP), described in detail in Section 2, and then solves them in parallel. We also consider a modified version of the algorithm in which each CSP is solved independently according to a number of different variable orderings. Furthermore, we discuss the implications of the use of shared and distributed memory models, including the issues of distributed computation termination and the overhead of interprocess communication.

The main contribution of the paper is the analysis of the achieved speedups. We focus in particular on the superlinear speedups[2] observed for certain samples. We provide explanations of the anomalies. We analyze the speedups in the function of the number of processes, but also with respect to the input samples. They differ in the sizes of the alphabets, the lengths of the examples and counterexamples and the sizes of the resulting automata. We consider different samples including the ones presented in the literature [9] and randomly generated based on the publicly available resources [10].

The rest of the paper is organized as follows. In Section 2 we present the problem formulation considered in the paper. In Section 3 we discuss the basic and modified parallel induction algorithms. Section 4 contains the results of the experiments and the discussion of the superlinear speedups. Finally, in Section 5 we present the conclusions.

2. Problem Formulation

The problem of minimal consistent NFA induction can be viewed from two different perspectives. Namely:

1. It is an *optimization problem*, if we first induce any consistent NFA, and later reduce it by merging redundant states.
2. It is a *decision problem*, if we first fix the number of states, and later search for a consistent automaton with the given number of states.

Note that in the first case, the final size of the automaton depends on the order in which the merges are performed. As a consequence, the resulting automaton need not be minimal. With the second approach, by taking the number of states to be $k = 1, 2, \ldots$, we not only find the consistent automaton for the given k, but we can also prove that it is indeed minimal, if no consistent NFA exists for $k-1$ states. However, even for the decision problem, the solution (i.e., the induced NFA) does not have to be unique.

There exists a number of algorithms following the first approach towards NFA induction mentioned above. They include the *DeLeTe2* algorithm [11], Nondeterministic Regular Positive Negative Inference (NRPNI) [12], and the state merging algoriths based on the notions of unambiguous [13] or universal [14] automata. The algorithm discussed in [14] has been extended in [15], to produce an algorithm that is independent of the or-

[2] A superlinear speedup occurs when the achieved speedup is greater than the number of used processes. For more information see [8].

der in which the merges are performed. Yet another approach was taken in [16], in which subautomata consistent with the set S_- were generated for each member of the set S_+, and were later removed when an example was accepted by a different subautomaton.

The decision problem formulation was pursued in [17], in which the basic encoding of the induction problem as a CSP was proposed. The encoding was later improved in [18,19], which allowed for a significant reduction of the solution space size. The impact of the selected variable ordering schemes on the performance of parallel induction algorithms was also investigated in [20,4]. Finally, some considerations related to the possibility of using multiple variable orderings at the same time were presented in [21]. In the current paper, we further elaborate on this possibility in terms of achieved speedups.

Let us now recall the CSP-based formulation of the induction problem solved by the parallel algorithms described in Section 3. The description is based on [19] and corresponds to the decision problem stated before. Let k be the given number of states and l be the size of the alphabet. We assume two types of binary variables y and z. Variables y_i, $i = 0,1,\ldots,k^2l-1$, denote the elements of the transition function δ, and variables z_j, $j = 0,1,\ldots,k-1$, mark the states as final or non-final. Let Σ be ordered lexicographically and let $loc(a)$ denote the zero-based position of a symbol a within Σ. Then each index i of a variable y_i, corresponding to a transition $q_m \xrightarrow{a} q_n$, $q_m, q_n \in Q$, is given by [17]:

$$i = k^2 \cdot loc(a) + k \cdot m + n. \tag{1}$$

Given the variables defined above, the consistency of the automaton with the sample $S = (S_+, S_-)$ is defined as follows:

1. If set S_+ or set S_- contains the empty word λ, then $z_0 = 1$, for $\lambda \in S_+$ (the empty word is accepted), and $z_0 = 0$, for $\lambda \in S_-$ (the empty word is rejected).
2. For all examples, the word w is accepted by the NFA iff there exists a sequence of transitions over which word w is spelled out, provided that this sequence ends in a final state. Therefore, for each $w \in S_+ \setminus \{\lambda\}$, it holds that:

$$\bigvee_{j=0..k-1} \left(\bigvee_{1..k^{|w|-1}} (y_{i_1} \wedge y_{i_2} \wedge \ldots \wedge y_{i_{|w|}}) \right) \wedge z_j = 1, \tag{2}$$

   where $i_1, i_2, \ldots, i_{|w|}$ are the indices of y_i variables computed according to Eq. (1), for $0 \leq m,n < k$ and $a \in \Sigma$ appearing in word w.
3. For all counterexamples, the word w is rejected by the NFA iff no sequence of transitions over which word w is spelled out exists, or such a sequence ends in a non-final state. Therefore, for each $w \in S_- \setminus \{\lambda\}$, it holds that:

$$\bigvee_{j=0..k-1} \left(\bigvee_{1..k^{|w|-1}} (y_{i_1} \wedge y_{i_2} \wedge \ldots \wedge y_{i_{|w|}}) \right) \wedge z_j = 0, \tag{3}$$

   where $i_1, i_2, \ldots, i_l$ are defined as before.

Example 1. To clarify Eqs. (2) and (3) let us consider the following example. Let the sample be $S = (\{a, aa, ba, bba\}, \{\lambda, b, ab\})$ and let $k = 2$. Since $\lambda \in S_-$ we have $z_0 = 0$.

Since $z_1 = 0$ cannot lead to a valid solution (no word would be accepted), we set $z_1 = 1$. Equations (2) and (3), after applying values of z_0 and z_1, take the following form:

for word a:	$y_1 = 1$
for word aa:	$y_0 \wedge y_1 \vee y_1 \wedge y_3 = 1$
for word ba:	$y_4 \wedge y_1 \vee y_5 \wedge y_3 = 1$
for word bba:	$y_4 \wedge y_4 \wedge y_1 \vee y_4 \wedge y_5 \wedge y_3 \vee y_5 \wedge y_6 \wedge y_1 \vee y_5 \wedge y_7 \wedge y_3 = 1$
for word b:	$y_5 = 0$
for word ab:	$y_0 \wedge y_5 \vee y_1 \wedge y_7 = 0$

After solving the above equations we get that $y_1 = 1$, $y_4 = 1$, $y_0 \vee y_3 = 1$, $y_5 = 0$, and $y_7 = 0$. It means that the resulting automaton contains the transitions $q_0 \xrightarrow{a} q_1$, $q_0 \xrightarrow{b} q_0$ and at least one of the transitions $q_0 \xrightarrow{a} q_0$ or $q_1 \xrightarrow{a} q_1$. Moreover, the automaton cannot contain the transitions $q_0 \xrightarrow{b} q_1$ and $q_1 \xrightarrow{b} q_1$. The existence of transitions related to variables y_2 (transition $q_1 \xrightarrow{a} q_0$) and y_6 (transition $q_1 \xrightarrow{b} q_0$) cannot be determined based on the given sample S. The example solutions are shown in Figure 1.

Figure 1. Automata consistent with sample S, in which $y_0 = 1, y_1 = 1, y_4 = 1$ (left), and $y_1 = 1, y_3 = 1, y_4 = 1$

3. Parallel Algorithms

Let us now discuss the basic parallel algorithm for solving the induction problem [21]. As already stated, it aims at solving independent CSPs in parallel to speed up the computation. Note that the algorithm, shown in Figure 2, searches for one solution only.

The algorithm BASICPARINDUCTION starts by checking if the value of variable z_0 can be established based on the presence of the empty word (line 2). Depending on the outcome of this check, it sets the number of possible CSPs n as follows: (i) $n = 2^k - 1$, for $\lambda \notin (S_+ \cup S_-)$, (ii) $n = 2^{k-1}$, for $\lambda \in S_+$, (iii) $n = 2^{k-1} - 1$, for $\lambda \in S_-$. These CSPs are then distributed among processes (line 3). Each process employs a backtracking procedure (lines 5–10), to find the assignments of values to y variables.

There are a few points about the algorithm shown in Figure 2 that are worth mentioning. First of all, structure Z_s is a k-element vector of z_j variables' values. Based on these values, Eqs. (2) and (3) are simplified by removing the terms for which $z_j = 0$ (see Example 1). Secondly, the way in which the CSPs are distributed among processes in the **parfor** loop (line 3) depends on the used memory model. For a shared memory model, new processes may be forked by a master process, while for a distributed memory model, the processes may be assigned to the Z_s vectors based on their ranks. In either case the interprocess communication overhead at this point is minimal. Thirdly, for each CSP both the y_i variables and their values are selected according to the given ordering scheme (line 6), which is the same for each CSP (see [21] for a discussion of other possibilities). Finally, since we search for the first solution, upon finding it the computation

```
1:  procedure BASICPARINDUCTION(S, k)
2:      if λ ∈ (S+ ∪ S−) then set z0 accordingly
3:      parfor s ← 1 to n do
4:          Zs ← assignment of values to zj variables, 0 ≤ j < k
5:          ▷ start backtracking procedure
6:          select next yi and assign value according to the given ordering
7:          evaluate Eqs. (2) and (3)
8:          if contradition found then change value or return to the previous yi
9:          if solution found then notify other processes and terminate
10:         ▷ end backtracking procedure
11:     end parfor
12: end procedure
```

Figure 2. The basic parallel induction algorithm

terminates. The way in which the termination procedure is realized, depends again on the memory model used. In case of the shared memory model, it is enough to use a global Boolean flag protected against simultaneous read-write access by a mutex. In case of the distributed memory model, a message has to be sent to other processes, indicating that they may terminate their execution. However, to receive the message, each process has to periodically check for message arrival. Hence, the distributed memory model incurs some time overhead resulting from channel probing and interprocess communication.

The modified version of the parallel induction algorithm is shown in Figure 3. It applies multiple ordering schemes to each of the analyzed CSPs. The intuition behind this approach is that the "best" ordering is not known in advance, and it may differ between respective CSPs. Thus, to increase the chances for efficient computation, we employ multiple orderings to the same instance of the CSP. This way we also capitalize on the negative results, i.e., when the process using some ordering determines that no solution exists for the given CSP (given Z_s), it notifies the other processes working on the same CSP, that they should terminate their execution. This way, the time to solve a given CSP is shorter, and equal to the run time of the process using the "best" ordering.

Let us discuss the effects of the memory models on the MULTIVOPARINDUCTION algorithm. The distribution of computation ocurrs in lines 3 and 5. Let n be the number of CSPs and m be the number of ordering schemes. Then in the shared memory model, we can fork nm processes, divide them into n groups working on the Z_s vectors and for each process in the given group apply a different ordering scheme for the same Z_s. In case of the distributed memory model, we can still use the process ranks, but this time we have to group the processes working on the same vector Z_s. As to the termination procedure, for the shared memory model, we need a set of global Boolean flags, one for each group of processes working on the same CSP, to indicate negative results. For the distributed memory model, we need to introduce a different message type for each group of processes, to indicate group termination, as opposed to global termination when the solution is found. Therefore, the overhead of interprocess communication does not change for the distributed memory model, while it increases for the shared memory model, due to the need for access synchronization to the group termination flag.

Figures 4 and 5 show the work distribution and interprocess communication related to the termination procedure for the two parallel algorithms. For the basic algorithm, we

```
1: procedure MULTIVOPARINDUCTION(S, k)
2:     if λ ∈ (S+ ∪ S−) then set z0 accordingly
3:     parfor s ← 1 to n do
4:         Zs ← assignment of values to zj variables, 0 ≤ j < k
5:         parfor t ← 1 to m do
6:             ▷ start backtracking procedure
7:             select next yi and assign value according to the given ordering t
8:             evaluate Eqs. (2) and (3)
9:             if contradition found then change value or return to the previous yi
10:            if solution found then notify other processes and terminate
11:            ▷ end backtracking procedure
12:        end parfor
13:        if solution not found then notify other processes working on Zs and terminate
14:    end parfor
15: end procedure
```

Figure 3. The modified parallel induction algorithm

assumed that the number of processes is equal to n, while for the modified version, this number is equal to nm.

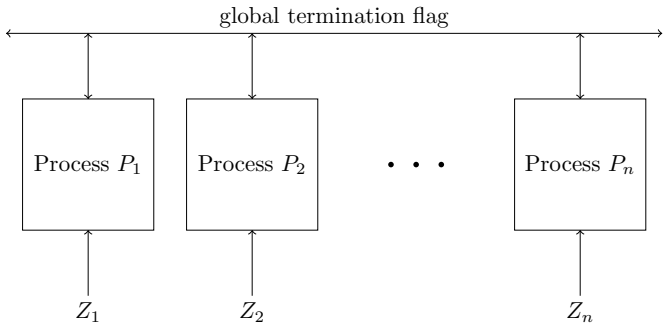


Figure 4. Work distribution and computation termination procedure for BASICPARINDUCTION algorithm

4. Experiments

The parallel algorithms were implemented in Java and executed on a pair of 12-core Intel Haswell 2.3 GHz processors with 128 GB RAM. The read-write access to the shared memory was protected using the `synchronized` keyword. The time measurements were performed using `System.nanoTime()` function.

The experiments were conducted for the selected Tomita languages [9] and for the samples built from the peptides listed in WALTZ-DB database [10]. The Tomita languages are defined over the alphabet $\{0,1\}$, while the peptides are based on an alphabet of up to 20 symbols, representing amino acids. The summary of the differences between these two sample sources is shown in Table 1.

The experiments aimed at observing the speedups obtained by the basic and modified parallel algorithms. The algorithms used three different ordering schemes, namely

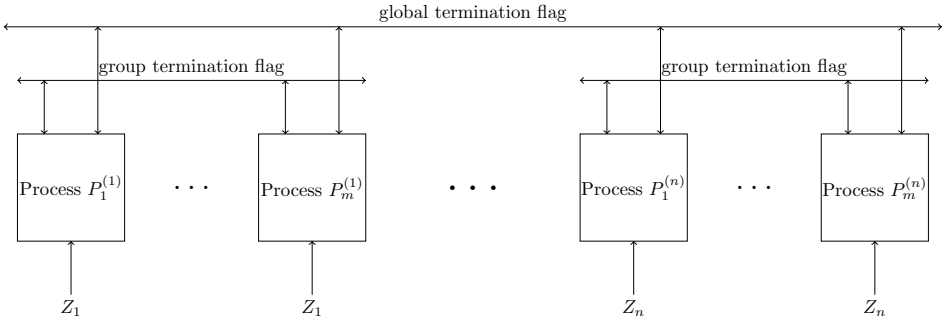


Figure 5. Work distribution and computation termination procedure for MULTIVOPARINDUCTION algorithm

Table 1. Comparison of sample characterisitcs based on Tomita languages and WALTZ-DB samples

Sample characteristic	Tomita languages	WALTZ-DB samples
Number of samples N	10	50
Number of states k	3–4	2–3
Alphabet size $\|\Sigma\|$	2	18–20
Sample size $\|S_+\|+\|S_-\|$	20–25	50
Word length $\|w\|$	0–18	5–6
Contains empty word $\lambda \in (S_+ \cup S_-)$	yes	no

the *deg* scheme [22], as well as the *min-max-ex* and *min-max-cex* schemes [4]. The *deg* scheme uses static ordering based on variable degree, while the other two schemes use dynamic ordering based on the examples and counterexamples, respectively.

In the first experiment we compared the basic parallel algorithm executed by a single process and by the number of processes corresponding to n. The distribution of obtained speedups, for different variable orderings, is shown in Figure 6. The box plots show the minimum and maximum speedup values (marked by the lines extending from the box), together with the first, second, and third quartile (marked by the box itself).

Based on the results shown in Figure 6 we noticed two kinds of anomalies. On the one hand, we observed slowdowns present mostly for the Tomita languages. On the other hand, we noted the superlinear speedups (up to 8500) in case of WALTZ-DB samples.

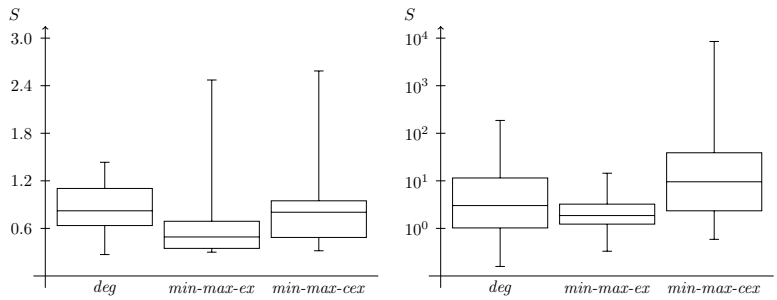


Figure 6. Speedups achieved by the BASICPARINDUCTION algorithm for the Tomita languages (left) and WALTZ-DB samples

The reason for the negative anomalies is that when the solution found during the sequential and parallel execution of the algorithm is the same, the latter approach introduces an overhead resulting from parallelism. Since the processes read from and write to the shared memory, access synchronization occurs. Furthermore, the NUMA architecture of the processors also affects the distribution of memory access times. This in turn introduces certain delays to the overall execution time. The reasons for positive anomalies are two-fold. Either there exists more than one solution for the given sample, or the solution is found for a CSP that is not the first one analyzed. In the former case, the algorithm executed in parallel allows to find the "simplest" solution, i.e., the solution that can be found in the shortest time. In the latter case, the parallel algorithm is able to bypass the "hard" CSP instances that have to be solved during the sequential run of the algorithm.

Example 2. Let us assume that a two-state automaton is sought. We consider three different CSPs resulting from the pairs of assignments $Z_0 = \langle 0,1\rangle$, $Z_1 = \langle 1,0\rangle$, and $Z_2 = \langle 1,1\rangle$. Let us assume that the execution times are $\tau_0 = 10$ s, $\tau_1 = 1$ s, and $\tau_2 = 25$ s, and that a solution exists for the cases Z_0 and Z_1. The sequential execution takes 10 s (solution for Z_0 found), while the parallel execution for $n = 3$ processes takes only 1 s (solution for Z_1 found), which gives a speedup of 10.

Example 3. Let us assume that a two-state automaton is sought. We consider three different CSPs resulting from the pairs of assignments $Z_0 = \langle 0,1\rangle$, $Z_1 = \langle 1,0\rangle$, and $Z_2 = \langle 1,1\rangle$. Let us assume that the execution times are now $\tau_0 = 25$ s, $\tau_1 = 1$ s, and $\tau_2 = 10$ s, and that a solution exists for the case Z_1. The sequential execution takes 26 s (cases Z_0 and Z_1 considered), while the parallel execution for $n = 3$ processes takes only 1 s (after solution for Z_1 is found all processes terminate), which gives a speedup of 26.

In the experiments performed for the WALTZ-DB samples, we counted 25, 34, and 34 cases in which a different solution was found by the sequential and parallel algorithm using *deg*, *min-max-ex*, and *min-max-cex*, respectively. Out of these cases, there were 12, 19, and 11 cases which resulted in superlinear speedups. Additionally, for the cases in which the sequential and parallel execution provided the same solution, there were 13, 14, and 5 cases, in which we observed superlinear speedups.

In the second experiment we used the MULTIVOPARINDUCTION algorithm to observe how the use of multiple ordering schemes affects the execution times. In particular, we compared the run times of the modified algorithm with the sequential executions of the basic algorithm. The summary of obtained speedups is shown in Table 2.

Based on the results shown in Table 2, we note that the use of multiple orderings sometimes fails to bring any improvement in the execution time, regardless of the type of sample (see min columns). It is caused by even more frequent synchronization between processes, occurring also within the groups solving the same CSP. However, we observe that the MULTIVOPARINDUCTION algorithm allows also for large superlinear speedups

Table 2. Speedups achieved by the MULTIVOPARINDUCTION algorithm with respect to the basic algorithm

	deg			*min-max-ex*			*min-max-cex*		
Sample source	min	max	avg	min	max	avg	min	max	avg
Tomita	0.1	2.8	0.7	0.2	17.6	2.8	0.2	23.1	3.8
WALTZ-DB	0.1	115.1	13.6	0.9	38101.7	2688.8	0.5	96.8	11.6

(see max columns in WALTZ-DB row). These speedups are observed for the ordering schemes different than the one that found the solution. It is so because, the "best" ordering can produce the solution in much shorter time than the other orderings, bypassing also their problems in solving certain CSPs.

We noted that in case of the WALTZ-DB samples, the *deg* ordering scheme was usually the one that allowed to find the solution in the shortest time (for 30 out of 50 samples). The same trend was also preserved for Tomita languages, for which the *deg* ordering scheme was the fastest in 7 out of 10 cases.

5. Conclusions

We analyzed the speedups obtained by the basic and modified parallel algorithms for NFA induction. For the Tomita languages, defined over two-symbol alphabet, we usually observed negative anomalies, i.e., the algorithms slowed down with the increase of the number of processes. Furthermore, these samples turned out to be easy enough to be solved efficiently even by a single process. For the peptide-based samples of WALTZ-DB database, the parallelism was expolited to a larger extent. Firstly, there were 3 cases in which the sequential execution of the algorithm failed to find the solution within the time limit of 8 hours. And secondly, we observed superlinear speedups of up to 8500 for no more than 7 processes, and over 38 000, for up to 21 processes.

To explain the differences between the two kinds of samples, let us note that the solution space is given by $2^{k^2 l}$, where k is the number of states and l is the alphabet size. Therefore, for the Tomita languages we need to consider at most $2^{32} \approx 4 \cdot 10^9$ different assignments of values to y variables. For the WALTZ-DB samples, we get $2^{180} \approx 10^{18}$ possible assignments. Therefore, the bigger solution space allows for better use of the parallelism and increases also the probability that more than one solution exists. Hence, it allows to achieve superlinear speedups in the cases discussed in Examples 2 and 3.

In the future, we plan to investigate the performance of the algorithms in the cases in which more than one solution is sought. In particular, we are interested in analyzing how the number and types of used variable ordering schemes would affect the speedups. Moreover, we plan to investigate deeper the reasons for the observed slowdowns.

Acknowledgments

The research was supported by National Science Centre Poland, project registration no. 2016/21/B/ST6/02158. The computations were carried out using the computer cluster Tryton at the Academic Computer Center in Gdańsk.

References

[1] C. de la Higuera. A bibliographical study of grammatical inference. *Pattern Recognition*, 38(9):1332–1348, 2005.

[2] W. Wieczorek and O. Unold. Induction of directed acyclic word graph in a bioinformatics task. In A. Clark, M. Kanazawa, and R. Yoshinaka, editors, *Proceedings of the 12th International Conference on Grammatical Inference*, volume 34 of *JMLR Workshop and Conference Proceedings*, pages 207–217. Proceedings of Machine Learning Research, 2014.

[3] W. Wieczorek and O. Unold. Use of a novel grammatical inference approach in classification of amyloidogenic hexapeptides. *Computational and Mathematical Methods in Medicine*, 2016:1–10, 2016.

[4] T. Jastrząb. A comparison of selected variable ordering methods for NFA induction. In J.M.F. Rodrigues et al., editors, *Computational Science – ICCS 2019*, volume 11540 of *LNCS*, pages 741–748. Springer, Cham, 2019.

[5] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Publishing Company, 1979.

[6] C. de la Higuera. Characteristic sets for polynomial grammatical inference. *Machine Learning*, 27:125–138, 1997.

[7] T. Jiang and B. Ravikumar. Minimal NFA problems are hard. *SIAM Journal on Computing*, 22(6):1117–1141, 1993.

[8] S. Ristov, R. Prodan, M. Gusev, and K. Skala. Superlinear speedup in HPC systems: why and when? In *Proceedings of FEDCSIS*, pages 889–898. IEEE, 2016.

[9] M. Tomita. Dynamic construction of finite automata from examples using hill-climbing. In *Proceedings of the 4th Annual Conference of the Cognitive Science Society*, pages 105–108. University of Michigan, USA, 1982.

[10] J. Beerten, J. Van Durme, R. Gallardo, E. Capriotti, L. Serpell, F. Rousseau, and J. Schymkowitz. WALTZ-DB: a benchmark database of amyloidogenic hexapeptides. *Bioinformatics*, 31(10):1698–1700, 2015.

[11] F. Denis, A. Lemay, and A. Terlutte. Learning regular languages using RFSAs. *Theoretical Computer Science*, 313(2):267–294, 2004.

[12] G.I. Alvarez, J. Ruiz, A. Cano, and P. Garcia. Nondeterministic regular positive negative inference NRPNI. In J.F. Diaz, C. Rueda, and A. Buss, editors, *Proceedings of the XXXI Latin American Informatics Conference*, pages 239–249. 2005.

[13] F. Coste and D. Fredouille. Unambiguous automata inference by means of state merging methods. In N. Lavrac et al., editors, *Proceedings of the 14th European Conference on Machine Learning*, volume 2837 of *LNAI*, pages 60–71. Springer-Verlag, Berlin, Heidelberg, 2003.

[14] P. Garcia, M. Vazquez de Parga, G.I. Alvarez, and J. Ruiz. Universal automata and NFA learning. *Theoretical Computer Science*, 407(1–3):192–202, 2008.

[15] P. Garcia, M. Vazquez de Parga, G.I. Alvarez, and J. Ruiz. Learning regular languages using nondeterministic finite automata. In O.H. Ibarra and B. Ravikumar, editors, *Proceedings of the 13th International Conference on Implementation and Application of Automata*, volume 5148 of *LNCS*, pages 92–101. Springer-Verlag, Berlin, Heidelberg, 2008.

[16] M. Vazquez de Parga, P. Garcia, and J. Ruiz. A family of algorithms for non deterministic regular languages inference. In O.H. Ibarra and H.-C. Yen, editors, *Proceedings of the 11th International Conference on Implementation and Application of Automata*, volume 4094 of *LNCS*, pages 265–274. Springer-Verlag, Berlin, Heidelberg, 2006.

[17] W. Wieczorek. Induction of non-deterministic finite automata on supercomputers. In J. Heinz, C. de la Higuera, and T. Oates, editors, *Proceedings of the 11th International Conference on Grammatical Inference*, volume 21 of *JMLR Workshop and Conference Proceedings*, pages 237–242. Proceedings of Machine Learning Research, 2012.

[18] T. Jastrząb, Z.J. Czech, and W. Wieczorek. Parallel induction of nondeterministic finite automata. In R. Wyrzykowski et al., editors, *Proceedings of the 11th International Conference on Parallel Processing and Applied Mathematics*, volume 9573 of *LNCS*, pages 248–257. Springer, Cham, 2016.

[19] T. Jastrząb. On parallel induction of nondeterministic finite automata. In I. Altintas et al., editors, *Proceedings of the International Conference on Computational Science*, volume 80 of *Procedia Computer Science*, pages 257–268. Elsevier, 2016.

[20] T. Jastrząb. Performance evaluation of selected variable ordering methods for NFA induction. In *Proceedings of the 14th International Conference on Grammatical Inference – Extended Abstracts*. 2018.

[21] T. Jastrząb. Two parallelization schemes for the induction of nondeterministic finite automata on PCs. In R. Wyrzykowski et al., editors, *Proceedings of the International Conference on Parallel Processing and Applied Mathematics*, volume 10777 of *LNCS*, pages 279–289. Springer, Cham, 2017.

[22] R. Dechter and I. Meiri. Experimental Evaluation of Preprocessing Techniques in Constraint Satisfaction Problems. In *Proc. of IJCAI'89*, pages 271–277, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200040

A Domain Decomposition Reduced Order Model with Data Assimilation (DD-RODA)

Rossella Arcucci [a], César Quilodrán Casas [a], Dunhui Xiao [b,c,a], Laetitia Mottet [c], Fangxin Fang [c,a], Pin Wu [d], Christopher Pain [c,a] and Yi-Ke Guo [a]

[a] *Data Science Institute, Department of Computing, Imperial College London, UK*
[b] *ZCCE, College of Engineering, Swansea University, UK*
[c] *Department of Earth Science & Engineering, Imperial College London, UK*
[d] *School of Computer Science and Engineering, Shanghai University, Shanghai, China*

Abstract. We present a Domain Decomposition Reduced Order Data Assimilation (DD-RODA) model which combines Non-Intrusive Reduced Order Modelling (NIROM) method with a Data Assimilation (DA) model. The NIROM is defined on a partition of the domain in sub-domains with overlapping regions and the DA is defined on a partition of the domain in sub-domains without overlapping regions. This choice allows to avoid communications among the processes during the Data Assimilation phase. However, during the balance phase, the model exploits the domain decomposition implemented in DD-NIROM which balances the results among the processes exploiting overlapping regions. The model is applied to the pollutant dispersion within an urban environment. Simulations are performed using the open-source, finite-element, fluid dynamics model Fluidity.

Keywords. Numerical simulations, Reduced Order Models, Data Assimilation, Domain Decomposition

1. Introduction

It is estimated that by 2050, around four-million deaths per year will be attributable to outdoor air pollution (twice the current mortality rate) [1]. This mandates the development of techniques that can be used for emergency response, real-time operational prediction and management. Numerical simulations are extensively used as a predictive tool to better understand complex air flows and pollution transport on the scale of individual buildings, city blocks and entire cities [2]. Fast-running Non-Intrusive Reduced Order Model (NIROM) for predicting the turbulent air flows has been proved to be an efficient method to provide numerical forecasting results [3]. However, due to the reduced space on which the model operates, the solution includes uncertainties that are somewhat ambiguous [3]. Additionally, any computational methodology contributes to uncertainty due to finite precision and the consequent accumulation and amplification of round-off errors. Taking into account these uncertainties is essential for the acceptance of any numerical simulation. The main question is how to incorporate data (e.g. from physical

measurements) in models in a suitable way, in order to improve model predictions and quantify prediction uncertainty.

Here, the focus is on the prediction of nonlinear dynamical systems: the classical application example being weather forecasting. In this paper, we combine a Domain Decomposition NIROM (DD-NIROM) method [4] with Data Assimilation (DA). DA is an uncertainty quantification technique used to incorporate observed data into a prediction model in order to improve numerical forecasted results. The DD-RODA (Domain Decomposition Reduced Order Data Assimilation) model we propose in this paper achieves both efficiency and accuracy by including Variational DA (VarDA) into DD-NIROM.

The DD-NIROM can be constructed by a combination of proper orthogonal decomposition (POD) and machine learning methods or interpolation methods. The key idea of the DD-NIROM is that it constructs a set of hypersurfaces representing the reduced system (including linear and non-linear processes). The novelty of the NIROM and DD-NIROM, presented in [3], lies in how they are generated, i.e. how the hypersurfaces are calculated using a machine learning method. The model we introduce in this paper combines the state of the art of domain decomposition reduced order models with an efficient variational DA model defined on an optimal reduced space [5,6,7,8] and on a decomposition of the domain in sub-domains named (DD-DA). Even if the DD-DA method we employ is efficient, it lacks of efficiency in the pre-processing phase which mainly consists in evaluating and computing the background error covariance matrices. Modelling and specification of the covariance matrix of background error constitute important components of any data assimilation system [9]. The main attributes of the background error covariance matrix are: to spread out the information from the observations; to provide statistically consistent increments at the neighboring grid points; and to ensure that observations of one model variable produce dynamically consistent increments in the other model variables. The use of DD-NIROM for the pre-processing phase of the DD-DA process can improve the efficiency of the whole prediction-correction cycle with a consequent improvement of the operational prediction model fidelity.

In summary, in this paper we combine a Domain Decomposition Non-intrusive Reduced Order Modelling method [4] with Domain Decomposition Data Assimilation [8,7] in a Domain Decomposition Reduced Order Data Assimilation (DD-RODA) model in order to achieve both accuracy and efficiency in our simulations. An important advantage of the DD-RODA approach is that once the DD-NIROM model is obtained, there is no need to refer to the full model while performing DD-DA. With this approach, in fact, we improve

- the accuracy of the DD-NIROM model by introducing information from observed data using the variational DD-DA process.
- the efficiency of the DD-DA process in the pre-processing phase: we use the DD-NIROM results to train our background error covariance matrices resulting in a strong reduction of the overall execution time.

We demonstrate the accuracy and the scalability of our approach. A mathematical formulation of the model is provided.

The model is tested on the pollutant dispersion within an urban environment. Simulations are performed using the open-source, finite-element, fluid dynamics software Fluidity (http://fluidityproject.github.io/). The details of the equations solved and their

implementations can be found in [10,11,12]. In this paper, the state variable consists of values of pollution concentration. However, the algorithm and numerical methods proposed in this work can be applied to other physical problem involving other equations and/or state variables.

2. Reduced Order Assimilation model

Let $\mathbf{u}$ be a state variable and let f represent a full physical system:

$$\dot{\mathbf{u}} = f(\mathbf{u}, t) \tag{1}$$

where t denotes the time.
Let Ω be the discrete spatial domain and let $\mathscr{P}(\Omega) = \{\Omega_j\}_{j=1,\dots,s}$ be a partition of Ω in s sub-domains. Let $\mathbf{u}_j$ be the restriction of the state variable $\mathbf{u}$ on the sub-domain Ω_j. In this work, a Domain Decomposition Non-intrusive Reduced Order Modelling (DD-NIROM) method is used to enhance the computational efficiency. The reduced order model projects the sub-domains of the full physical system with a big dimensional size onto a reduced space sub-domains with a much smaller dimensional size, therefore it is faster to solve.
Let n denotes a time level, the DD-NIROM uses a Proper Orthogonal Decomposition (POD) method and Gaussian Process Regression (GPR) method to approximate the solutions of equation (1).

In DD-NIROM based on the POD method, any variables $\mathbf{u}_j^n$ (for example, the velocity or tracers) at time level n can be expressed by the expansion,

$$\mathbf{u}_j^n = \overline{\mathbf{u}}_j + \sum_{i=1}^{M} \alpha_{ji}^n \phi_{ji}, \tag{2}$$

where α_{ji}^n ($i \in \{1,2,\dots,M\}$) denotes the POD coefficients of the POD basis functions at the time level n. ϕ_{ji} are the POD basis functions. M is the number of POD basis functions ($M << N$) which can represent most (99% for example) of energy within the chosen solution snapshots. $\overline{\mathbf{u}}_j$ represents the mean of the snapshots.

Let n be a fixed time level and let $\mathbf{u}_j^n$ be a state variable expressed by DD-NIROM as described in equation (2). Let $\mathbf{e}_j^n = \mathbf{u}_j - \mathbf{u}_j^n$ be the error introduced by replacing the full physical system in (1) by the NIROM model (2). We introduce a DD-Reduce Order Assimilation process by which the DD-NIROM model in (2) is combined with a DD-Data Assimilation method in order to improve the accuracy of the solution $\mathbf{u}_j^n$ (i.e. reduce $\mathbf{e}_j^n$) introducing information by observation of the state variable $\mathbf{u}_j$.

Let $\mathbf{v}_j^n$ be an observation of the state variable at time n, the aim of DD-Reduced Order Data Assimilation (DD-RODA) problem is to find an optimal trade-off between the prediction made based on the DD-NIROM system state $\mathbf{u}_j^n$ (background) defined in (2) and the available observation $\mathbf{v}_j^n$. For a fixed time step n, given $\mathbf{u}_j^n$, $\mathbf{v}_j^n$ and a mapping

$$H_j : \mathbf{u}_j^n \mapsto \mathbf{v}_j^n, \tag{3}$$

the DD-RODA process consists in finding $\mathbf{u}_j^{DD-RODA}$ as inverse solution of

$$\mathbf{v}_j^n = H_j\left(\mathbf{u}^{DD-RODA}\right), \tag{4}$$

subject to the constraint that $\mathbf{u}_j^{DD-RODA} = \mathbf{u}_j^n$, i.e.:

$$\mathbf{u}_j^{DD-RODA} = \overline{\mathbf{u}}_j + \sum_{i=1}^{M} \alpha_{ji}^n \phi_{ji}. \tag{5}$$

where ϕ_{ji} denotes the POD basis functions. Since H_j is typically rank deficient, equation (4) is an ill-posed inverse problem [13,14]. The Tikhonov formulation [15,16] leads to an unconstrained least squares problem, where the term in (5) provided by DD-NIROM ensures the existence of a unique solution in (4). The DD-RODA process can then be described as follows:

$$\mathbf{u}_j^{DD-RODA} = argmin_{\mathbf{u}_j}\left\{\|\mathbf{u}_j - \mathbf{u}_j^n\|^2_{B_j^{-1}} + \|\mathbf{v}_j^n - H\left(\mathbf{u}_j\right)\|^2_{R_j^{-1}}\right\} \tag{6}$$

where R_j and B_j are the observation and model error covariance matrices respectively, defined on each subdomain Ω_j, $j = 1,\ldots,s$:

$$R_j := \sigma_0^2 I_j, \tag{7}$$

with $0 \leq \sigma_0^2 \leq 1$ and I_j the identity matrix,

$$B_j = V_j V_j^T \tag{8}$$

where V_j is the deviance matrix [6]. If equation (6) is linearised around the background state [17], we have:

$$\mathbf{u}_j = \mathbf{u}_j^n + \delta\mathbf{u}_j \tag{9}$$

where $\delta\mathbf{u}_j = \mathbf{u}_j - \mathbf{u}_j^n$ denotes the increments. The DD-RODA problem is formulated by the following form:

$$\delta\mathbf{u}_j^{DD-RODA} = argmin_{\delta\mathbf{u}_j} J_j(\delta\mathbf{u}_j) \tag{10}$$

where

$$J_j(\delta\mathbf{u}_j) = \frac{1}{2}\delta\mathbf{u}_j^T\mathbf{B}_j^{-1}\delta\mathbf{u}_j + \frac{1}{2}(\mathbf{H}_j\delta\mathbf{u}_j - \mathbf{d}_j^{DD-NIROM})^T\mathbf{R}_j^{-1}(\mathbf{H}_j\delta\mathbf{u}_j - \mathbf{d}_j^{DD-NIROM}) \tag{11}$$

and

$$\mathbf{d}_j^{DD-NIROM} = [\mathbf{v}_j^n - H_j\left(\mathbf{u}_j^n\right)] \tag{12}$$

is the misfit between the observation and the solution computed by DD-NIROM (see Algorithm 1) and

$$H_j(\mathbf{u}_j) \simeq H_j\left(\mathbf{u}_j^n\right) + \mathbf{H}_j \delta \mathbf{u}_j \tag{13}$$

denotes the linearised observational and model operators evaluated at $\mathbf{u}_j = \mathbf{u}_j^n$ where $\mathbf{H}_j$ is the Hessian of H_j. In equation (10), the minimisation problem is defined on the field of increments [18]. In order to avoid the inversion of $\mathbf{B}_j$, as $\mathbf{B}_j = \mathbf{V}_j \mathbf{V}_j^T$, the minimisation can be computed with respect to a new variable [17] $\mathbf{w}_j = \mathbf{V}_j^+ \delta \mathbf{u}_j$ and $\mathbf{V}_j^+$ denotes the generalised inverse of $\mathbf{V}_j$:

$$\mathbf{w}_j^{DD-RODA} = argmin_{\mathbf{w}_j} J_j(\mathbf{w}_j) \tag{14}$$

where

$$J_j(\mathbf{w}_j) = \frac{1}{2}\mathbf{w}_j^T \mathbf{w}_j + \frac{1}{2}(\mathbf{H}\mathbf{V}_j \mathbf{w}_j - \mathbf{d}_j^{DD-NIROM})^T \mathbf{R}_j^{-1} (\mathbf{H}\mathbf{V}_j \mathbf{w}_j - \mathbf{d}_j^{DD-NIROM}) \tag{15}$$

In the next section, DD-RODA is applied to improve the pollutant dispersion prediction within an urban environment. Simulations are performed using the open-source, finite-element, fluid dynamics model Fluidity.

3. Numerical example

The capability of DD-RODA has been estimated using an urban environment located in London South Bank University (LSBU) area (London, UK) shown in Figure 1. The computational domain has a size of $[0,2041] \times [0,2288] \times [0,250]$ (metres). This work uses the 3D non-hydrostatic Navier-Stokes equations as the full physical system,

$$\nabla \cdot \mathbf{u} = 0, \tag{16}$$

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\nabla p + \nabla \cdot \boldsymbol{\tau}, \tag{17}$$

where $\mathbf{u} \equiv (u,v,w)^T$ is the velocity, $p = \tilde{p}/\rho_0$ is the normalised pressure ($\tilde{p}$ being the pressure and ρ_0 the constant reference density) and $\boldsymbol{\tau}$ denotes the stress tensor.

Simulations were carried out for the study area using Fluidity, an open-source, finite-element, fluid dynamics model [12]. The dispersion of pollutant is described by the classic advection-diffusion equation with the pollutant concentration treated as a passive scalar. A source term was added to the advection-diffusion equation to mimic a constant release of pollutant generated by traffic in a busy intersection for example. The location of the point source is depicted by the red sphere in Figure 1(a). The time step was adaptive based on the Courant (CFL) number defined by the user, and the Crank-Nicholson scheme was used for the time discretization [11,10]. The mesh is shown in Figure 1(c). The outlet boundary condition was defined by a zero-pressure (no-stress) condition; perfect slip boundary conditions were applied at the top and on the sides of the domain and no-slip boundary conditions were applied on all building facades and the bottom surface of the domain. A synthetic incoming-eddy method was used at the inlet [20] to mimic the behaviour of the boundary layer. The mean velocity profile was prescribed as in equation (18):

Algorithm 1 DD-RODA algorithm on each sub-domain Ω_j, $j = 1, ..., s$

▷ The following are known: $\{f_i\}_{i=1}^m$, $\{\boldsymbol{\phi}_i\}_{i=1}^m$, H_j and R_j

$\boldsymbol{\alpha}_j^0 = \boldsymbol{\alpha}_j(t_0)$ ▷ Initialisation of POD coefficients

for $n = 1$ *to* $\mathcal{N}_t$ **do**
  $t = t_0 + n\Delta t$ ▷ Current time

  ▷ Step (a): calculate the POD coefficients, $\boldsymbol{\alpha}_j^n$, at the current time step:

  for $i = 1$ *to* m **do**
    $\alpha_{ji}^n = f_{ji}(\alpha_{j1}^{n-1}, \alpha_{j2}^{n-1}, \ldots, \alpha_{jm}^{n-1})$
  endfor

  ▷ Step (b): obtain the solution $\mathbf{u}_j^n$ in the full space at the current time, t, by projecting α_{ji}^n onto the full space using equation (2):

  $\mathbf{u}_j^n = \mathbf{0}$
  for $i = 1$ *to* m **do**
    $\mathbf{u}_j^n = \mathbf{u}_j^n + \alpha_{ji}^n \boldsymbol{\phi}_{ji}$
  endfor

  ▷ Step (c): compute the optimal background error covariance matrix:

  $\mathbf{V}_j^n = \mathbf{u}_j^n - \bar{\mathbf{u}}_j$
  $\mathbf{V}_j = \{\mathbf{V}_j, \mathbf{V}_j^n\}$

endfor

$\mathbf{V}_{j\tau} = TSVD(\mathbf{V}, \tau)$. ▷ Truncated SVD regularised matrix [6,19]

▷ Step (d): solve the reduced order assimilation process (6):

$\mathbf{d}_j^{DD-NIROM} \leftarrow \mathbf{v}_j^n - H_j \mathbf{u}_j^n$ ▷ Compute the misfit

$\mathbf{G}_j \leftarrow \mathbf{H}_j \mathbf{V}_{j\tau} \mathbf{w}_j - \mathbf{d}_j^{DD-NIROM}$

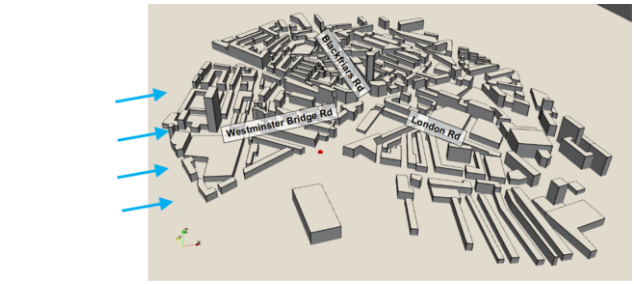
$\mathbf{w}_j^{DD-RODA} = argmin_{\mathbf{w}_j} \left\{ \frac{1}{2} \mathbf{w}_j^T \mathbf{w}_j + \frac{1}{2} \mathbf{G}_j^T \mathbf{R}_j^{-1} \mathbf{G}_j \right\}$ ▷ Compute the minimum

$\delta \mathbf{u}_j^{DD-RODA} \leftarrow \mathbf{V}_{j\tau} \mathbf{w}_j$ ▷ From the reduced to physical space

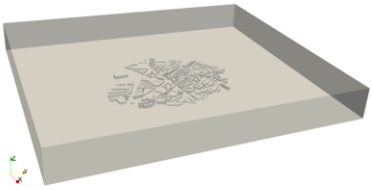
$\mathbf{u}_j^{DD-RODA} \leftarrow \mathbf{u}_j^n + \delta \mathbf{u}_j^{DD-RODA}$

$$(u, v, w) = \left(0.97561 \ln\left(\frac{z}{0.01}\right), 0, 0\right) \tag{18}$$

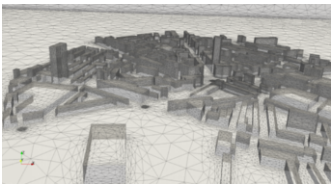
where z denotes the height (in m). The inlet length-scale $\boldsymbol{L}$ and Reynolds stresses $\boldsymbol{Re}$ are prescribed constant and equal to 100 m and 0.8 respectively, for the diagonal components, and zero elsewhere. Zero velocity is prescribed on the bottom and on the wall boundaries. Zero stress conditions is set to be $p = 0$ at the outlet boundary and a perfect slip condition is specified on the vertical lateral boundaries. Experiments have been implemented and tested on 3 high performance nodes equipped with bi-Xeon E5-2650 v3 CPU and 250GB of RAM with Python 3.5



(a)



(b)



(c)

Figure 1. (a) London South Bank University (LSBU) test case area. The red sphere denotes the location of the pollution source and the blue arrows denote the wind direction. (b) 3D computational domain and (c) surface mesh of the test site.

The accuracy of the DD-RODA results is evaluated by the mean squared error on each sub-domain:

$$MSE(u_j) = \frac{\|u_j - u_j^C\|_{L^2}}{\|u_j^C\|_{L^2}} \tag{19}$$

computed with respect to a control variable u_j^C provided by observed data, for $j = 1,\ldots,s$ and s denotes the number of sub-domains. Figure 2 shows the values of $MSE(u^{DD-NIROM})$ and $MSE(u^{DD-RODA})$ for a decomposition made of $s = 16$ sub-domains running on $p = 16$ processors. We can observe that the error decreases for each sub-domain. We observe a bigger gain in terms of accuracy reduction in sub-domains where the pollution concentration is more diffused. Fox example, the sub-domain number 11 presents a bigger gain as shown in Figure 2, this sub-domain is the central sub-domain in Figure 4 (orange colour).

We evaluated the execution time needed to compute the solution of the DD-RODA model using Algorithm 1. Let T_s denote the execution time of Algorithm 1 for a domain decomposition made of s sub-domains. We assume that $p = s$, where p denotes the number of processors and we pose:

$$T_s = max\{T_{s_i}\}_{i=1,\ldots,s} \tag{20}$$

where T_{s_i} denotes the execution time for each processor on each sub-domain. The total execution time is shown in Figure 3(a). There is a clear decreasing trend in the total execution time with the increase of number of processors. Figure 3(b) shows the values of execution time of DD-NIROM and Fluidity on $p = 4, 16, 32$ processors for a decomposition of $s = 4, 16, 32$ sub-domains. The gain in terms of execution time provided by using

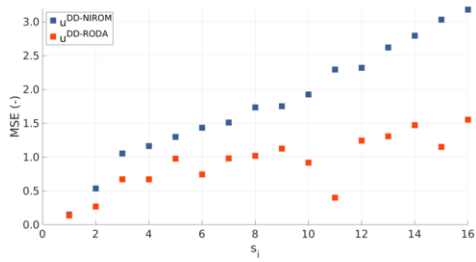
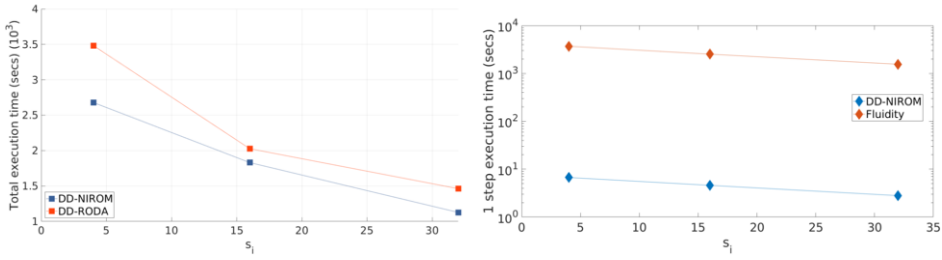


Figure 2. Values of $MSE(u^{DD-NIROM})$ and $MSE(u^{DD-RODA})$ for a decomposition made of $s = 16$ sub-domains running on $p = 16$ processors

DD-NIROM instead of Fluidity strongly impact on the efficiency of the pre-processing to the Data Assimilation phase (Step (c) of Algorithm 1) for computing the covariance matrices V_j for each time step n.



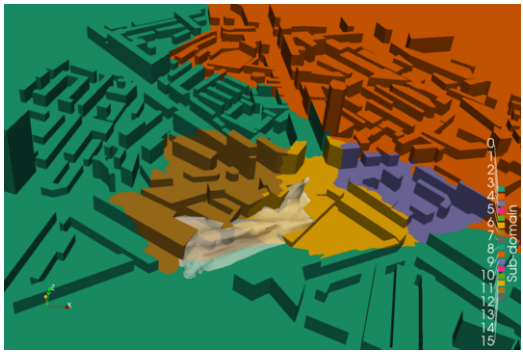
(a) (b)

Figure 3. (a) Values of execution times of DD-NIROM and DD-RODA for a number of sub-domains $s = 4, 16, 32$ running on $p = 4, 16, 32$ processors (plot in linear scale) (b) Values of execution time for running 1 time step of DD-NIROM and Fluidity on $p = 4, 16, 32$ processors (plot in log scale).

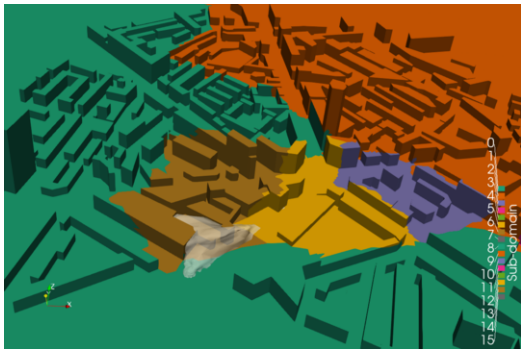
Figure 4 shows the impact of DD-RODA on the iso-surface of the pollutant concentration for $5.10^{-1} kg/m^3$ computed in parallel with $p = 16$ processors and generated by a point source. Figure 4(a) shows the results predicted by DD-NIROM, i.e. $u^{DD-NIROM}$, while Figure 4(b) shows the observed data, i.e. v. Values v are assimilated in parallel by DD-RODA to correct the forecasting data $u^{DD-NIROM}$. The assimilated data after the DD-RODA process, i.e. $u^{DD-RODA}$, are then obtained (Figure 4(c)).

4. Conclusions

In this paper, we have presented a Domain Decomposition Reduced Order Data Assimilation (DD-RODA) model which is a fusion of the Non-Intrusive Reduced Order Modelling method with a 3D Data Assimilation both defined on a decomposition of the domain in sub-domains. We proved that our approach improves both accuracy of the DD-NIROM model and efficiency of the DA process. The accuracy of the DD-NIROM model is improved by introducing information from observed data exploiting the 3D variational DA process. The efficiency of the DA process is mainly improved in the pre-processing phase. In fact, we used the DD-NIROM results to train our background error covariance



(a) $u^{DD-NIROM}$: predicted pollutant iso-surface by DD-NIROM.



(b) v: observed pollutant iso-surface.

(c) $u^{DD-RODA}$: assimilated pollutant iso-surface by DD-RODA.

Figure 4. Iso-surface, in white, of the pollutant concentration for $5.10^{-1} kg/m^3$ computed in parallel with $p = 16$ and generated by a point source.

matrices and we have shown that this implies a strong reduction of the overall execution time. The efficiency and the accuracy of our model were discussed and tested using a 3D case of air flows and pollution transport in an urban environment. The algorithms and the method proposed are, however, enough generic and can be used easily for other physical problems.

Acknowledgments

This work is supported by the EPSRC Grand Challenge grant "Managing Air for Green Inner Cities" (MAGIC) EP/N010221/1 and by the EPSRC Centre for Mathematics of Precision Healthcare EP/N0145291/1.

References

[1] Jos Lelieveld, John S Evans, Mohammed Fnais, Despina Giannadaki, and Andrea Pozzer. The contribution of outdoor air pollution sources to premature mortality on a global scale. *Nature*, 525(7569):367, 2015.

[2] Bert Blocken. Computational fluid dynamics for urban physics: Importance, scales, possibilities, limitations and ten tips and tricks towards accurate and reliable simulations. *Building and Environment*, 91:219–245, 2015.

[3] D Xiao, CE Heaney, L Mottet, F Fang, W Lin, IM Navon, Y Guo, OK Matar, AG Robins, and CC Pain. A reduced order model for turbulent flows in the urban environment using machine learning. *Building and Environment*, 148:323–337, 2019.

[4] D Xiao, CE Heaney, F Fang, L Mottet, R Hu, DA Bistrian, E Aristodemou, IM Navon, and CC Pain. A domain decomposition non-intrusive reduced order model for turbulent flows. *Computers & Fluids*, 2019.

[5] R. Arcucci, C. Pain, and Y. Guo. Effective variational data assimilation in air-pollution prediction. *Big Data Mining and Analytics*, 1(4):297 – 307, 2018.

[6] R. Arcucci, L. Mottet, C. Pain, and Y. Guo. Optimal reduced space for variational data assimilation. *Journal of Computational Physics*, 2018.

[7] R. Arcucci, L. DAmore, L. Carracciuolo, G. Scotti, and G. Laccetti. A decomposition of the tikhonov regularization functional oriented to exploit hybrid multilevel parallelism. *International Journal of Parallel Programming*, 45(5):1214–1235, 2017.

[8] R. Arcucci, L. Carracciuolo, and R. Toumi. Toward a preconditioned scalable 3dvar for assimilating sea surface temperature collected into the caspian sea. *Journal of Numerical Analysis, Industrial and Applied Mathematics*, 12(1-2):9–28, 2018.

[9] D.G. Cacuci, I. M. Navon, and M. Ionescu-Bujor. Computational methods for data evaluation and assimilation. *CRC Press*, 2013.

[10] R. Ford, C. C. Pain, M. D. Piggott, A. J. H. Goddard, C. R. E. de Oliveira, and A. P. Umpleby. A nonhydrostatic finite-element model for three-dimensional stratified oceanic flows. part i: Model formulation. *Monthly Weather Review*, 132:2816–2831, 2004.

[11] E. Aristodemou, T. Bentham, C. Pain, and A. Robins. A comparison of mesh-adaptive les with wind tunnel data for flow past buildings: Mean flows and velocity fluctuations. *Atmospheric Environment*, 43:6238–6253, 2009.

[12] Imperial College London AMCG. Fluidity manual v4.1.12. 4 2015.

[13] H. K. Engl, M. Hanke, and A. Neubauer. Regularization of inverse problems. *Kluwer*, 1996.

[14] N. Nichols. Mathematical concepts in data assimilation. *W. Lahoz, et al. (Eds.), Data Assimilation, Springer*, 2010.

[15] P.C. Hansen. Rank deficient and discrete ill-posed problems. *SIAM, Philadelphia*, 1998.

[16] Y. Wang, I. M. Navon, X. Wang, and Y. Cheng. 2D Burgers equation with large Reynolds number using POD/DEIM and calibration. *International Journal for Numerical Methods in Fluids*, 82(12):909–931, 2016.

[17] A.C. Lorenc. Development of an operational variational assimilation scheme. *Journal of the Meteorological Society of Japan*, 75:339–346, 1997.

[18] JP. Courtier. A strategy for operational implementation of 4d-var, using an incremental approach. *Q J R Meteorol Soc*, 120(519):1367–1387, 1994.

[19] R. Arcucci, L. D'Amore, J. Pistoia, R. Toumi, and A. Murli. On the variational data assimilation problem solving and sensitivity analysis. *Journal of Computational Physics*, pages 311–326, 2017.

[20] D. Pavlidis, G.J. Gorman, J.L.M.A. Gomes, C. Pain, and H. ApSimon. Synthetic-Eddy Method for Urban Atmospheric Flow Modelling. *Boundary-Layer Meteorology*, 136:285–299, 2010.

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200041

Predicting Performance of Classical and Modified BiCGStab Iterative Methods

Boris KRASNOPOLSKY [a,1]
[a] *Institute of Mechanics, Lomonosov Moscow State University, Russia*

Abstract. The paper focuses on comparison of the efficiency of various formulations of BiCGStab iterative methods in the parallel computations. The analytical execution time model, based on the volume of data transfers with the memory, is presented. The corresponding model predictions are validated with results of the calculations. The proposed model is used to compare the performance of the classical BiCGStab and Pipelined BiCGStab method formulations. The obtained model predictions and simulation results are compared with results of other authors.

Keywords. Krylov subspace iterative methods, analytical execution time model, BiCGStab, Pipelined BiCGStab, scalability

1. Introduction

Improving efficiency of Krylov subspace iterative methods on high performance compute systems is among the topics of active research for decades. These methods are widely used to solve large sparse systems of linear algebraic equations occurred as a result of spatial discretization of the corresponding differential equations. The Krylov subspace methods typically consist of three types of algebraic operations: vector updates, dot products and matrix-vector multiplications. The first ones are the local operations, the second ones require global reduction and the third ones require the corresponding communications with neighbour processes. While the time for local communications with neighbours is typically independent of the number of compute processes (at least in case of using graph partitioning methods to minimize the number of neighbours and the corresponding volume of communications), the time to perform global communications grows with increasing the number of compute processes.

Multiple attempts have been performed to reformulate the corresponding methods in order to change the sequence of calculations to minimize the time spent on communications. Among the unpreconditioned BiCGStab methods, the Improved BiCGStab [1] and Pipelined BiCGStab [2] are the non-limiting examples of the corresponding modified methods. The Improved BiCGStab method combines together multiple dot products, thus allowing to reduce three global reductions of classical BiCGStab method per each iteration to only one. The Pipelined BiCGStab needs two global reductions, but the algorithm allows to overlap the corresponding communications by calculations of matrix-

[1]Corresponding Author: Boris Krasnopolsky, Institute of Mechanics, Lomonosov Moscow State University, 119192, Michurinsky ave. 1, Moscow, Russia; E-mail: krasnopolsky@imec.msu.ru.

vector multiplications. The proposed optimizations, however, require some extra vector operations, thus making the range of applicability and optimality for each of the methods a priori not evident. The authors have shown [2] that the Pipelined BiCGStab starts to outperform the classical BiCGStab at the scales of several compute nodes. The presented results, however, were not analysed in terms of the achieved performance, thus leaving some doubts when interpreting these results.

The current paper discusses the formulation of the analytical execution time model and shows detailed comparison of the classical BiCGStab and reformulated Pipelined BiCGStab methods with both analytical model predictions and calculation results. The presented results lead to a different conclusion compared the one stated in [2]. Possible reasons for the observed deviations are also discussed.

The rest of the paper is organized as follows. The second section provides description of the analytical execution time model and corresponding estimates for the classical and Pipelined BiCGStab methods. The third section contains description of the testing methodology used for benchmarking. The fourth section presents the analytical model validation results and thorough comparison of the classical and Pipelined BiCGStab methods. The possible reasons of deviations with results of [2] are also highlighted. The conclusion section finalizes the paper.

2. Analytical Execution Time Model

Accurate analysis of the time measurements and the achieved performance requires comparison with the calculation results by the other authors or some analytical models justifying the obtained results. The current paper provides the details of the analytical execution time model, allowing to predict the calculation times for the Krylov subspace iterative methods. The proposed expressions deal with the BiCGStab methods, however, the same methodology can be applied to other Krylov subspace methods.

Opposite to the models discussed in [4,5], which are based on the number of floating point operations to perform the computations, the current model is based on the volume of data transfers with the memory. The Krylov subspace iterative methods comprise of the vector operations and matrix-vector multiplications, and performance of these operations is limited by the memory bandwidth of the compute system [6]. This fact makes the volume of data transfers the more natural choice as a basis of the performance model than the number of floating point operations.

The proposed analytical model accounts the times to perform the vector operations, T_{vec}, matrix-vector multiplications, T_{mul}, and the time spent on communications. The last one includes the times for local communications (exchanges with local neighbours when performing matrix-vector multiplications) and global reductions (global operations when calculating dot products). The local communications are typically overlapped by the calculations with the local matrix blocks, while the possibility to hide the global communications by the calculations depends on the specific method formulation. The classical BiCGStab method requires 22 vector read/write operations, 2 matrix-vector multiplications and 3 global reductions per each iteration; global reductions are not overlapped by calculations and act as global synchronization points. This leads to the corresponding expression for the execution time:

$$T^{BiCGStab}(p) = 22T_{vec}(p) + 2T_{SpMV}(p) + 3T_G(p), \quad (1)$$

where

$$T_{vec}(p) = \frac{8N}{b\,p}, \tag{2}$$

$$T_{SpMV}(p) = \max\left(T_{mul}(p), T_L(l)\right), \tag{3}$$

$$T_{mul}(p) = \frac{N\left(8(C+1)+4(3C+1)\right)}{b\,p}. \tag{4}$$

Here, N is the problem size, p is the number of compute nodes, b is the compute node memory bandwidth, C is the average number of non-zero elements per matrix row; floating point numbers are assumed 8 bytes, integers are assumed 4 bytes, and the sparse matrix is stored in the CSR format. Following [3,7], it is expected the overhead to calculations due to communications for local non-blocking point-to-point communications in Eq. (3) is negligible and ignored in the matrix-vector execution time expression.

The Pipelined BiCGStab method requires 43 vector read/write operations, 2 matrix-vector multiplications and 2 global reductions, but allows to overlap these communications by the matrix-vector multiplications:

$$T^{PipeBiCGStab}(p) = 43T_{vec}(p) + 2\max\left(T_{SpMV}(p) + \gamma T_G(p), T_G(p)\right), \tag{5}$$

where γ is the overlapping overhead parameter, characterizing the efficiency of the asynchronous non-blocking global communications ($\gamma = 0$ indicates ideal overlap and no overhead when performing global communications; $\gamma = 1$ indicates no overlap of communications by computations; $\gamma > 1$ indicates the slowdown of communications due to progression of asynchronous non-blocking communications performed in background).

To conclude, the expressions for the local and global communication times must be provided. These functions are expressed in the form of polynomials, and the coefficients of the polynomials are fitted by results of the specific MPI benchmarks. The details of the benchmarks developed can be found in [3]. For the Lomonosov and HPC5 supercomputers used in the tests the following values have been obtained:

- Lomonosov:

$$T_G(p,l) = 3.5\cdot 10^{-6} + 1.7\cdot 10^{-6}\, l^{0.21}\, p^{0.54}, \tag{6}$$

$$T_L(l) = \begin{cases} 2.4\cdot 10^{-6} + 6.9\cdot 10^{-8}\, l^{0.56} & , l \le 2048 \text{ bytes}, \\ 3.2\cdot 10^{-6} + 2\cdot 10^{-9}\, l & , l > 2048 \text{ bytes}. \end{cases} \tag{7}$$

- HPC5:

$$T_G(p,l) = 5.6\cdot 10^{-6} + 9.7\cdot 10^{-7}\, l^{0.3}\, p^{0.63}, \tag{8}$$

$$T_L(l) = \begin{cases} 4.4\cdot 10^{-6} + 1.2\cdot 10^{-9}\, l & , l \le 2048 \text{ bytes}, \\ 7.1\cdot 10^{-6} + 6.4\cdot 10^{-10}\, l & , l > 2048 \text{ bytes}. \end{cases} \tag{9}$$

Table 1. Specifications of Lomonosov and HPC5 supercomputers.

Supercomputer	Lomonosov	HPC5
Processor model	Intel Xeon X5570	Intel Xeon E5-2650 v2
Processors	2	2
Cores	4	8
Instruction set	SSE4.2	AVX
LLC size, MB	8	20
RAM Bandwidth, GB/s	16	40
LLC Bandwidth, GB/s	46	170
Interconnect	Infiniband QDR	Infiniband FDR
MPI library	Intel MPI 2017	Intel MPI 2017
Compiler	Icc 17.0.1	Gcc 7.3.0

3. Test Cases and Testing Methodology

Both the analytical model predictions and calculation results presented below were performed with the test matrix of 10^6 unknowns obtained as a result of discretization of 2D Poisson equation on a regular grid of 1000^2 cells with the 5-point stencil. To measure the single iteration execution time for the BiCGStab methods, the constant number of iterations $N_{it} = 1000$ was calculated. The benchmarking was performed for two hardware platforms Lomonosov and HPC5; the key characteristics of these compute systems are summarized in Table 1. The estimates for the real random-access memory (RAM) and last level cache (LLC) bandwidths presented in the table were obtained with help of STREAM benchmark [8].

The implementations of BiCGStab and Pipelined BiCGStab methods in the newly developing library for solving large sparse systems of linear algebraic equations *XAMG* were used for the numerical validation. The current version of the *XAMG* library supports MPI parallelization. All the presented simulation results (up to 128 and 64 nodes for Lomonosov and HPC5 supercomputers respectively) were performed utilizing all available CPU cores per node with one rank per core mapping. In all the cases the lexicographical ordering of the matrix was used when distributing data across the compute processes.

4. Validation Results

The proposed simple execution time model allows to compare the efficiency of various methods and outline their range of applicability. The current section contains results of the validation of the proposed analytical model and detailed comparison of the efficiency of classical BiCGStab and Pipelined BiCGStab methods.

4.1. Analytical Model Validation

The first step of the validation and benchmarking procedure considers comparison of the analytical model predictions with the numerical calculation results. The corresponding comparison includes results for two iterative methods, classical BiCGStab and Pipelined BiCGStab, and two hardware platforms, Lomonosov and HPC5 supercomputers. For the

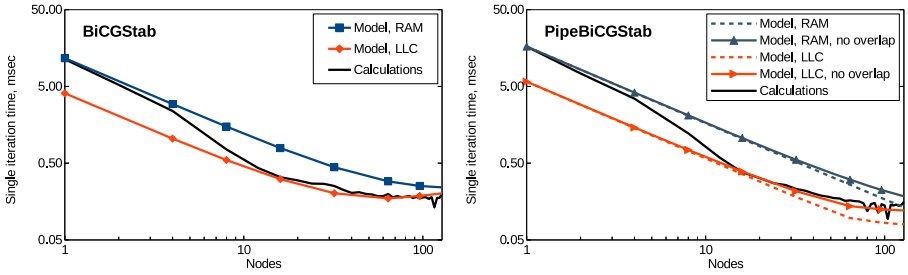


Figure 1. Analytical predictions and numerical simulation results for the Lomonosov supercomputer.

classical BiCGStab method the plots (Figures 1, 2) contain two theoretical curves corresponding to RAM and LLC bandwidth predictions; these curves limit the expected range of the execution times for the calculation results. For the Pipelined BiCGStab method the plots contain four theoretical curves. The proposed estimate Eq. (5) contains additional overlapping overhead parameter, γ, and the presented results show two limiting cases for this parameter ($\gamma = 0$ reflects ideal overlap of asynchronous non-blocking global communications by calculations and $\gamma = 1$ reflects the case with no overlap).

Calculation results obtained for the Lomonosov supercomputer (Figure 1) fit the range outlined by the analytical model predictions. At the scale of several compute nodes the calculation results comply with the RAM bandwidth predictions. Increasing the number of compute nodes the memory consumption per node decreases and the data starts to fit the cache. This is reflected in shifting of the calculation results towards the LLC bandwidth predictions.

Results for the Pipelined BiCGStab method presented in this section do not assume any special techniques to perform the asynchronous non-blocking global communications. One can see that the calculation results demonstrate better match with the analytical model predictions for the case with no overlap of global communications by calculations; this agrees with results shown in [3,7].

Results for the second hardware platform presented in Figure 2 reproduce all the tendencies indicated for the Lomonosov supercomputer. For the test matrix considered the calculation times even with the single compute node become lower than the RAM bandwidth predictions. This is a consequence of the 2.5 times higher LLC capacity of the processors installed in the compute nodes of the HPC5 system and rather small size of the test matrix (the test matrix considered needs about 60MB to store the data in the CSR format).

Results presented above allow to conclude that:

- the proposed analytical execution time model correctly predicts the corresponding methods execution times achieved in practice;
- implementations of the methods in the *XAMG* library do not contain any serious performance issues.

This makes possible to use the analytical model and the implementations of the methods for the systematic comparison of the efficiency of classical BiCGStab and Pipelined BiCGStab methods.

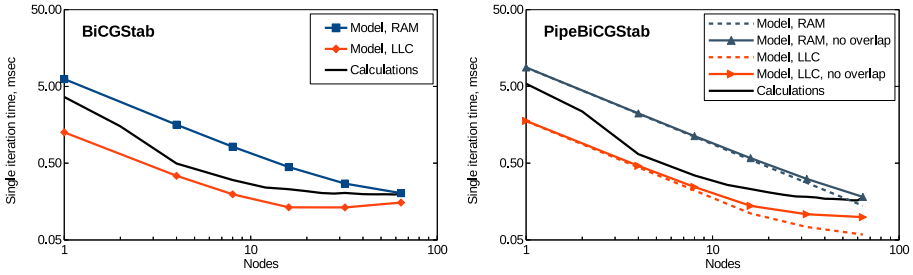


Figure 2. Analytical predictions and numerical simulation results for the HPC5 supercomputer.

4.2. Comparing Efficiency of BiCGStab Methods

Analytical expressions Eq. (1) and Eq. (5) for the execution times of the methods allow to estimate the range of applicability for the methods considered in the paper. Comparing these expressions, one can see that even in case of ideal overlap of global communications by the calculations without any overhead, $\gamma = 0$, the typical range of applicability for the Pipelined BiCGStab method starts with

$$N \lesssim \frac{1}{10} \frac{T_G b p}{8}. \tag{10}$$

This relation gives very narrow range of optimality for the Pipelined BiCGStab method: the matrix block per each compute node must be small, $N \lesssim 10^5$, or the amount of compute nodes must be large enough the global synchronization time to be comparable with milliseconds. Accounting the realistic values of the overlapping overhead when performing asynchronous non-blocking global reductions with short messages for the current compute platforms, the expected range of applicability would be even smaller. Otherwise, the classical BiCGStab will outperform the modified method.

It should be noted the conclusion formulated above contradicts with results presented in [2]. For the similar test matrix and compute system architecture as Lomonosov supercomputer (the test platform used in [2] has the same interconnect and the CPU generation, only differing in the number of cores) the authors have shown the advantage of Pipelined BiCGStab starting with 4 compute nodes. The plot in Figure 3 summarizes the calculation results obtained in this work and the ones from [2]. The present results demonstrate much better execution times and parallel efficiency for both compute systems used for benchmarking. The classical BiCGStab outperforms the modified method for all the points except the only one, 20 nodes for HPC5 compute system, where the performance for both formulations become equal. It is reasonable to expect that equalling of the performance for both methods is related with the decreased number of global reduction operations (modified method has two global reductions compared three for the original one), but not the effect of global communications overlap.

The authors in [2] used the progression threads implemented in the MPICH-3.1.3 library to perform the asynchronous execution of non-blocking collective communications. The same functionality is also available in Intel MPI 2017, used in the current test session. The same series of experiments has been repeated with activated software progression threads. The obtained results are summarized in Figure 4. These results, however, raise lots of questions. Activation of software progression threads for

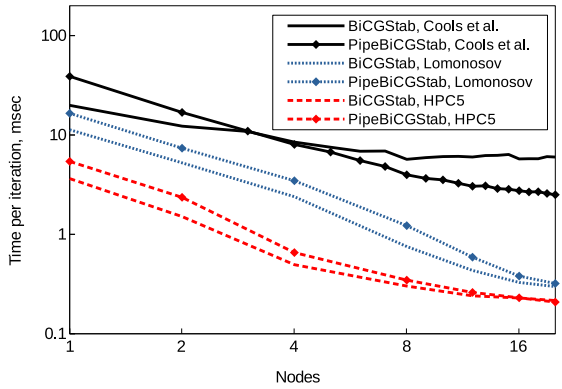


Figure 3. Comparison of the execution times for the single iteration of BiCGStab and Pipelined BiCGStab methods performed on two compute systems with results presented in [2].

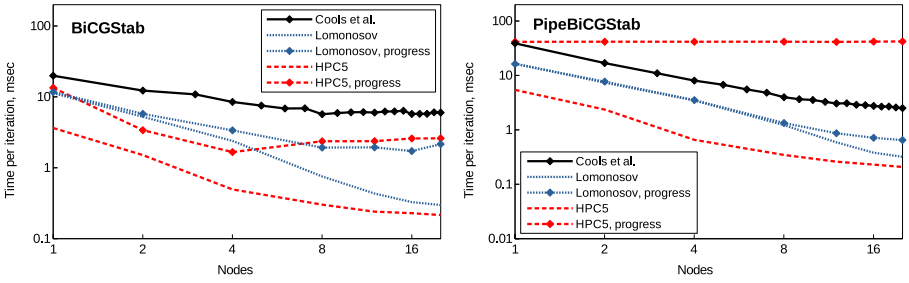


Figure 4. Effect of software progression threads on performance of the iterative methods.

the Lomonosov supercomputer leads to a smooth degradation of performance with increasing the number of compute nodes. This effect is more noticeable for the classical method formulation, while for the Pipelined BiCGStab it is, probably, compensated by the communications overlap.

Completely different situation is observed for the HPC5 compute platform. Activation of software threads leads to a performance degradation even for the calculations with the single compute node. While for the classical BiCGStab some speedup still can be achieved with increasing the number of compute nodes, execution times for the Pipelined BiCGStab method remain almost constant independently on the amount of compute nodes. Such a behaviour is expected strange and needs further investigation. Meanwhile, the obtained results clearly show that the software progression does not allow to obtain any simulation speedup for the algorithms with asynchronous non-blocking global reductions with short messages and lead to a performance degradation compared with synchronous global communications.

5. Conclusion

The current paper discusses the efficiency of various Krylov subspace iterative methods for solving systems of linear algebraic equations, and, specifically, focuses on the comparison of the BiCGStab method formulations. The analytical execution time model is

proposed to compare the performance of the methods and outline the range of applicability for each of them. The model is based on the volume of data transfers with the memory needed to perform the calculations. Correctness of the model is validated by results of numerical simulations. It is shown that the calculation results fit the range of the expected execution times predicted by the analytical model.

The range of optimality for the Pipelined BiCGStab method is highlighted. For the moderate numbers of compute nodes the size of the problem should not exceed 10^5 unknowns per compute node for the typical compute system.

The obtained analytical model predictions and calculation results are compared with results of other authors. It is shown that the current implementation of the methods significantly outperforms results published in [2] and suggests different conclusions about the preference of the Pipelined BiCGStab method. The range of optimality for the modified method formulation is expected at much higher scales of compute nodes than indicated in [2].

The efficiency of software progression threads to reduce the time spent on asynchronous global communications is investigated. The obtained results demonstrate significant slowdown for the runs with activated software progression. This clearly indicates that software progression functionality is inapplicable for the algorithms actively performing global reductions with short messages.

6. Acknowledgments

The presented work is supported by the RSF grant No. 18-71-10075. The research is carried out using the equipment of the shared research facilities of HPC computing resources at Lomonosov Moscow State University and computing resources of the federal collective usage center Complex for Simulation and Data Processing for Mega-science Facilities at NRC “Kurchatov Institute”.

References

[1] L. Yang, R. Brent, The improved BiCGStab method for large and sparse unsymmetric linear systems on parallel distributed memory architectures, *Fifth International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'02)* (2002), 324–328.

[2] S. Cools, W. Vanroose, The communication-hiding pipelined BiCGStab method for the parallel solution of large unsymmetric linear systems, *Parallel Computing* **65** (2017), 1–20.

[3] B. Krasnopolsky, Revisiting performance of BiCGStab methods for solving systems with multiple right-hand sides, *arXiv:1907.12874*, 2019.

[4] E. de Sturler, H.A. van der Vorst, Communication cost reduction for Krylov methods on parallel computers, *High-Performance Computing and Networking*, (1994), 190–195.

[5] S.-X. Zhu, T.-X. Gu, X.-P. Liu, Minimizing synchronizations in sparse iterative solvers for distributed supercomputers, *Computers & Mathematics with Applications* **67** (2014), 199–209.

[6] S. Williams, A. Waterman, D. Patterson, Roofline: an insightful visual performance model for multicore architectures, *Communications of the ACM* **52** (2009), 65–76.

[7] A. Medvedev, Towards benchmarking the asynchronous progress of non-blocking MPI point-to-point and collective operations, *https://github.com/a-v-medvedev/mpi-benchmarks/blob/master/doc/progression-article_v2.pdf* (to appear in ParCo-2019 proceedings).

[8] J.D. McCalpin, Memory bandwidth and machine balance in current high performance computers, *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, (1995), 19–25.

Parallel Applications

This page intentionally left blank

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200043

Gadget3 on GPUs with OpenACC

Antonio RAGAGNIN [a,b,c,d], Klaus DOLAG [c,e], Mathias WAGNER [f], Claudio GHELLER [g], Conradin ROFFLER [g], David GOZ [d], David HUBBER [b], Alexander ARTH [b]

[a] *Leibniz-Rechenzentrum, München, Germany* (`antonio.ragagnin@inaf.it`)
[b] *Excellence Cluster Universe, München, Germany*
[c] *University Observatory of Munich, München, Germany*
[d] *INAF - OATs, Trieste, Italy*
[e] *Max-Planck-Institut für Astrophysik, Garching, Germany*
[f] *NVIDIA GmbH, Würselen, Germany*
[g] *CSCS-ETH, Lugano, Switzerland*

Abstract. We present preliminary results of a GPU porting of all main Gadget3 modules (gravity computation, SPH density computation, SPH hydrodynamic force, and thermal conduction) using OpenACC directives. Here we assign one GPU to each MPI rank and exploit both the host and accellerator capabilities by overlapping computations on the CPUs and GPUs: while GPUs asynchronously compute interactions between particles within their MPI ranks, CPUs perform tree-walks and MPI communications of neighbouring particles. We profile various portions of the code to understand the origin of our speedup, where we find that a peak speedup is not achieved because of time-steps with few active particles. We run a hydrodynamic cosmological simulation from the Magneticum project, with $2 \cdot 10^7$ particles, where we find a final total speedup of ≈ 2. We also present the results of an encouraging scaling test of a preliminary gravity-only OpenACC porting, run in the context of the EuroHack17 event, where the prototype of the porting proved to keep a constant speedup up to 1024 GPUs.

Keywords. GPU, OpenACC, SPH, Barnes-Hut, Astrophysics

1. Introduction

The parallel N-body code Gadget3 [1,2] is nowadays one of the most used high-performing codes for large cosmological hydrodynamic simulations [3]. Gadget3 exploits hybrid MPI/OpenMP parallelism. Each MPI task owns a region of the domain composed by contiguous chunks of Hilbert-ordered particles, and, at each time-step communicates guest particles that interact with regions belonging to other MPI tasks. Dark matter, gas and stars are sampled by particles and interact through gravity using the Barnes-Hut [4] approximation for short-range interactions and Particle-Mesh for long range interactions. Hydrodynamics of gas particles is modelled using an improved version of Smoothed Particle Hydrodynamics (SPH) [5] by [6].

SPH is implemented in Gadget3 with two different modules: the first one computes particle densities by multiple iterations and the second one computes hydrodynamic forces. Additionally Gadget3 implements other physical processes as thermal conduction

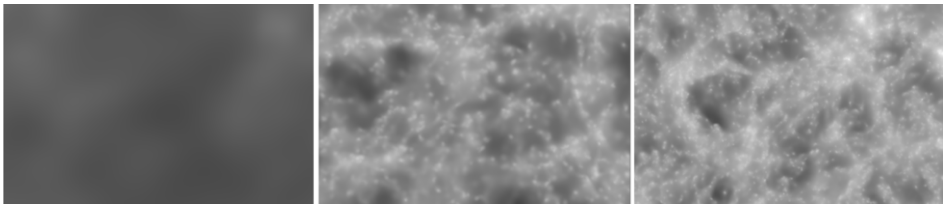


Figure 1. Gas projected density of a portion of the cosmological simulation Magneticum Box4/hr ($2 \cdot 10^7$ particles). Left panel shows the gas distribution of the initial conditions of the simulation; central panel shows particles in the middle of a simulation (where dark matter haloes start forming); right panel show particles at the end of the simulation. Color values are log-scaled.

and sub-resolution models for star formation and black hole evolution. Star particles interact only through gravity, however, they are created based on properties of gas particles as in [7].

Each of the above mentioned modules share the same pattern: they compute the force acting on a list of active particles by performing a tree walk over groups of one or more active particles and find all neighbouring particles within a given distance [8]. Being Gadget3 a parallel code, when searching for neighbours, the code will also identify regions of the tree that belongs to a different MPI rank. After this identification, MPI ranks will exchange neighbouring particles. In the second phase of each module, MPI ranks will compute forces acting on guest particles due to local contributions. When a MPI rank has computed the interactions over the received guest particles, it will send the results back to their original MPI rank, which will merge the received contributions with the one from local neighbours. Gadget3 uses a relatively small exchange buffer ($\approx 300MB$ per node) compared to the memory occupied by particles in a large simulation ($> 20GB$ per node), and for this reason, it is not possible to exchange boundary particles all at one time: first and second phases must be repeated until all active particles have been processed.

Figure 1 shows the projected gas density in three different phases of a simulation. While the initial conditions of a simulation (left panel) contains nearly homogeneous matter distribution, as simulation time increases (from left to right panels), dark matter forms haloes and filaments. Particles inside a clustered region are driven by much stronger accelerations than particles outside these regions, thus Gadget3 uses an adaptive time-stepping scheme where active particles are updated with a kick-drift-kick solver[2].

Since small time-steps have only few active particles, to improve the performance during the neighbour search, the code drifts only tree-nodes and non-active particles that are encountered during such neighbour search. The previously mentioned drift and filling of the export buffer are not thread-safe and are encapsulated inside OpenMP critical regions.

The most time consuming modules are Gravity ($\approx 15\%$, of the time), SPH ($\approx 30\%$ of the time) and thermal conduction ($\approx 14\%$. of the time). The remaining time is mostly taken by the domain decomposition ($\approx 16\%$ of the time) and the so called halo finder [9] ($\approx 8\%$ of the time).

In this work we present a porting of all main Gadget3 modules (gravity computation, SPH density computation, hydrodynamic force, and thermal conduction) on GPUs using OpenACC [10].

We decided to port these modules, because, besides being the most time consuming modules, they are the ones that spend most time in loops of kernel functions, and thus are suitable for GPUS.

Our approach overlaps computations between the host and the CPU. GPUs asynchronously compute physical interactions between particles within the same computing node while CPUs perform tree walks, fills the export buffer and communicates particles. Because of memory limitations, we offload a module per time to the GPU. We test our porting using the Magneticum[1] suite of simulations. In particular we use Box4/hr with $2 \cdot 10^7$ particles and Box3/hr with $3.8 \cdot 10^8$ particles. We also test our code with different architectures, as P100 and V100 GPUs with NVLink technology [11], with the PGI compiler with and without CUDA [12] Unified Memory [13].

In Section 2 we discuss the obstacles that prevent an easy porting of the Gadget3 code and our choices of GPU porting. In Section 3 we profile the code and show the speedup of our porting over different portions of a cosmological hydrodynamic simulation. In Section 4 we draw our conclusions and discuss future projects.

2. Challenges and Strategies in Accelerating Gadget3

Here below we list various limitations that prevent an easy porting of the whole code Gadget3 to the GPUs:

- The code do not benefit from vectorisation because it stores data in arrays of large data structures ($\approx 500B$ each) that do not fit modern architecture caches. Changing the data layout to a structure of arrays would require a massive refactoring effort and introduce additional memory movement (of packing and unpacking data) in the domain decomposition.
- The use of blocking MPI communications (to exchange neighbouring particles between MPI ranks) poses a limit in fully utilising GPUs and CPUs.
- Time-steps with too few active particles won't fully exploit GPU parallelism, thus preventing the code to speedup;
- There are thread-locking operations at each tree walk (drift of particles and fill of shared export buffer for communications).
- GPUs memories have less capacity than their host memories, thus simulations that keeps all data in GPUs will require more computing nodes than CPU only runs.
- Gadget3 has been built over a decennial effort of developers who implemented various flavours of gravity, SPH solvers, and sub-resolution models that have been extensively tested; rewriting these modules using CUDA/OpenCL languages would imply a massive rewrite of portions of such modules with associated risks of adding mistakes.

For these reasons, a directive-based approach that uses OpenACC [10] has been adopted. This reduces modifications of the ongoing development of Gadget3 and furthermore makes it possible to still run the code on CPU-only systems.

[1] http://www.magneticum.org

2.1. Memory Transfer

To minimize communication between CPUs and GPUs, one would ideally load the initial conditions of the simulation in the memory of the GPU and run the whole simulation on GPUs. This solution has two problems: first of all, time steps with few active particles won't perform on the GPUs, and since current GPUs typically have less memory than their hosts, one would need more nodes than a CPU-only run.

To clarify the last point, let's consider the case of a very large cosmological simulation that was run within the LRZ Extreme Scaling Workhop in 2015 [14]. Such simulation (Magneticum Box0/mr) had $1.2 \cdot 10^7$ particles per node, each node was allocating $4GB$ for the Barnes Hut tree, $22GB$ for the basic quantities used in gravity (e.g. position, mass, acceleration ecc..), and additional $14GB$ for the SPH-only part (that is split in density computation and hydro-force computation), $0.6GB$ for the metal evolution and an additional amount of $4GB$ for the active particle list and to store the Hilbert space-filling-curve keys, for a grand total of $40GB$ per node.

It is clear that a $16GB$ GPU system (as for instance, the ones in Piz Daint[2]) would not be able to store the same number of particles of its underlying host. On the other hand, it has enough memory to store the particle properties of each single Gadget3 module at a time.

To solve this issue and to be able to exploit the GPU memory at its best, we decided to only upload, for each Gadget module, the properties that are necessary for such module (or for other successive modules) in the current time step. With this technique we are able to upload more particles per timestep, but we can upload only the minimal set of properties required by each module at time. The drawback of this approach is that at each timestep we need to download the data back to the GPU, with its associated overhead.

To further minimise the data transfer of the particle properties, we send separately properties that are read-only (masses, positions, ecc..) and download only updated properties (e.g. acceleration).

Additionally, with this approach we minimise the amount of code we have to write/-modify: we use the same Gadget routines used to process guest neighbouring particles coming from a different MPI rank. We set up the code so GPUs use the already existing routines to exchange data, but in this case, particles are exchanged between host and GPUs.

2.2. Adaptive Timesteps

Large, high resolution, cosmological simulations have both void regions and clustered regions. Particles in void regions evolve with large timesteps because of the small force acting on them, compared to clustered regions where the stronger force requires very small timesteps.

After nearly half of the simulation time, it is very common to have timebins with only one or very few active particles. Since time-steps with such a low amount of active particles won't benefit from the single instruction multiple thread (SIMT) paradigm of GPUs, we decided to keep small timebins (with less than a given threshold N_{min} active particles) to run on the CPU only.

[2] https://www.cscs.ch/computers/piz-daint/

The reason behind this choice is twofold: (i) with a high number of active particles, the offload time is small compared to computations and (ii) it is possible to drift all particles and tree-nodes of the simulated volume at the beginning of these time-steps in the host, with OpenMP.

OpenACC turned out to be the best tool to implement this decision because it makes it possible to use the same code on both GPU and CPU with a small effort.

2.3. MPI Communication

One of the main advantages of our porting is that it overlaps CPU work with the GPU computation. We decided to overlap the CPU and the GPU computation in the following way: while the GPU loops over the active particles and computes local interactions, the CPU takes care of walking the tree for each active particle in order to perform all MPI send/receive of guest particles.

When the host receives a list of guest particles it decides to queue it to the GPU computation or to process it on the CPU, based on the facts that (i) the GPU did finish local interactions or not and (ii) the number of received particles is less than N_{min}.

2.4. Barnes-Hut, SPH and Thermal Conduction Differences

Although SPH, thermal conduction and Barnes Hut algorithms have many similarities, there are some main differences to take into account when porting Gadget on GPUs. First of all, in a Barnes-Hut solver, particles interact with distant tree nodes as they were point-like pseudo particles, in contrast with SPH and conduction solvers where there are only particle-particle interactions within a pre-defined distance. As a consequence, the implementation of Barnes-Hut algorithm embed the particle-particle interaction computation in the tree walk itself. On the other hand, in SPH and thermal conduction solvers, neighbours are collected in a list and processed in a separate step. Additionally, SPH and thermal conduction need to find a set of neighbours within a fixed distance, while Barnes-Hut operates with a so-called opening criteria, namely the angle between the target particle and the tree cells.

In our OpenACC porting, this implies that gravity acceleration computation will be inside a tree walk branch, which will limit the peak GPU performance. While in the SPH and conduction modules it is possible to disentangle the tree walk from the force computation. The drawback is that it is not well known a priori the amount of neighbours of a given SPH particle (especially in zoom-in simulations). The CPU implementation overcomes this problem by allocating a neighbour buffer for each thread of a size that is equal to the number of local particles. Since it is practically impossible to allocate such a long buffer on each GPU thread, our porting performs a tree walk and neighbour interactions in chunks of N_{chunk} neighbours.

3. Profiling

We tested our implementation over different setups and architectures, were we found the values of $N_{min} = 10^3$ and $N_{chunk} = 32$ to be optimal in always maximizing the speedup. Time steps with a number of active particles less than 10^3 tipically performs better in the CPU than in the GPU.

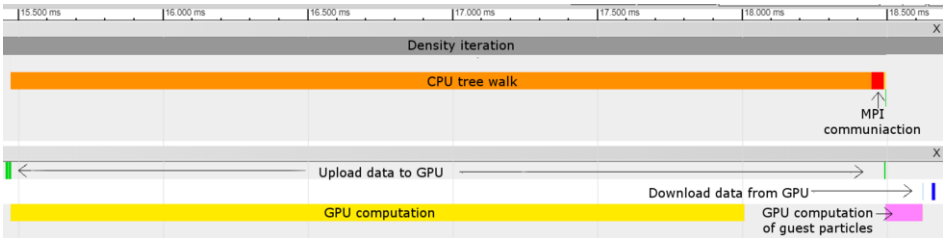


Figure 2. Timeline of the profiling obtained with the nvToolsExt library of the OpenACC code: upload and download to GPU (green and blue bars), GPU computation (yellow and purple bars), CPU tree walk (orange bar) and MPI communications (red bar). The full iteration takes 2.4*s*, while the CPU version of the code, run with the same setup, took 11.4*s*.

The value of N_{min} do not need to be extremely accurate. In fact the number of particles between one time bin and the other changes exponentially, from our experience, the number of particles between a small step and the next one goes from few hundreds to few thousands.

3.1. Tests of One Density Iteration

Figure 2 shows a time-line of the profiling (obtained with the nvToolsExt library[3]) of a SPH density iteration over all particles of the Magneticum/Box4/hr simulation ($2 \cdot 10^7$ particles) on a Power9 system with 2 MPI ranks per node, each with 20 OpenMP threads plus one Tesla V100 GPU. With this setup we used all cores of a node (and without using hyper-threading). Each socket has NVLink interconnection technology between CPUs and GPUs.

In this setup, upload and download timings (green and blue bars) sums up to 0.053*s* (for a total of 1.2*GB*) and are negligible compared to the GPU computation time (yellow and purple bars), that take up to 2.4*s*. CPU tree walks (orange bar) takes 2.9*s* and MPI communications (red bar) take 0.04*s* and overlaps the GPU computations.

The whole density iteration took 3.2*s*, while the same set-up, when run completely on CPUs, took 11.4*s*. Of which 11*s* spent in computation and the remaining 0.4*s* spent in MPI communications.

Thus, a SPH density iteration over all active particle have a speedup of 4.5. However, the speedup of the full simulation will be lower because a number of iterations have only very few active particles and do not perform well on GPUs.

We then briefly tested the possibility of using Unified Memory for our OpenACC porting. In particular, we run Magneticum Box4/hr simulations with 2 MPI rank, each using one V100 GPU connected with NVLink.

The iteration without Unified Memory took 1.9*s*, where 0.4*s* were spent in memory transfer, while the run with Unified Memory: 2.0*s*. They pratically takes the same time. The advantage of Unified Memory is that one does not have to manually write code to restrict the transfer of data to its minimum necessary amount.

[3] https://docs.nvidia.com/gameworks/content/developertools/desktop/nsight/nvtx_library.htm

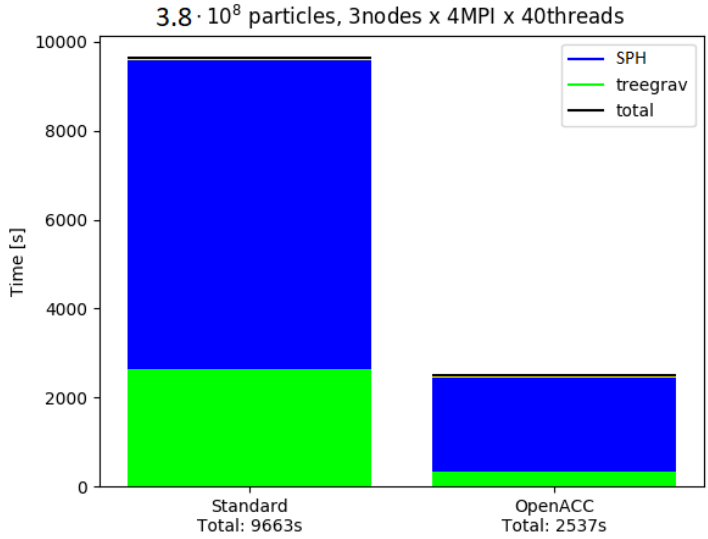


Figure 3. Time consumed by SPH and gravity for a simulation with $3.8 \cdot 10^8$ particles. Left panel shows the time consumed by the standard version, run over 3 nodes, each with 4 MPI ranks and each with 40 threads. Right panel shows the same configuration for the OpenACC porting where each MPI rank had one Tesla V100 GPU connected with NVLink.

3.2. Tests of One Timestep

Figure 3 shows the timing of a whole timestep of the various Gadget modules. In this test, to maximize the number of particles for a given node, we run Magneticum Box4/hr simulation, that has $3.8 \cdot 573^3 = 1.2 \cdot 10^8$ particles. We run this simulation on a Power9 system with 4 sockets per node, each with 10 cores and one V100 GPU. We used 3 nodes, each with 4 MPI ranks, and each MPI rank with 40 OpenMP threads (thus using Power architectures hyper threading).

Here we can see how, at least for the first time-step, most of the speedup is consistent over both the Barnes-Hut solver and the full SPH computation to a factor of 3 for SPH and ≈ 4 for the gravitaty computations.

In particular, all SPH density iterations within the timestep took $1600s$ for the GPU version and $5400s$ (with a speedup of 3.3) for the CPU version. While the SPH computation of hydrodynamic forces tooks $200s$ for the GPU version and $700s$ for the CPU version (with a speedup of 3.5).

The speedup of this test case is lower than the one obtained in the previous test case because in a whole timestep there are density iterations that have a very low number of particles.

3.3. Tests of Full Run

We then run a whole simulation with Barnes-Hut, SPH and thermal conduction ported with OpenACC. As described above, we offload these modules to the GPU only when the number of active particles is greater than the treeshold $N_{min} = 10^3$.

We run such simulation on the Piz Daint system. Here we used 8 MPI tasks in 1 node, 4 OpenMP threads and one Tesla P100 for each MPI task. In such system, GPUs are connected to the host with PCI Express technology. At the end of a simulation, the speed-up (compared to the same set-up of the Gadget3 standard version) are as follows:

- Barnes-Hut speedup: 1.8
- SPH speedup: 2.6
- Thermal conduction speedup: 3.0
- Total speedup: 2.1

Noteworthy, SPH speedup is lower than the speedup obtained for a single timestep as in the previous sub section (and for a single iteration of a density computation). The reason behind this slowdown is twofold: the number of density iterations that contains a small number of active particles increases as the simulation time evolves. For this reason we found a final total speedup to be lower than the one obtained for the first timestep.

After porting Gravity, SPH and thermal conduction to the GPU, one of the upcoming bottlenecks became the cooling and star formation module, taking $\approx 5\%$ of the computing time. Here we upload the cooling tables to the GPUs and keep them there as long as needed. We then run the whole cooling and star formation process in the GPU, which have a speedup of ≈ 1.6, when comparing a run with P100 GPUs and a run with 12 Haswell CPUs[4].

3.4. Scaling Test of Gravity Only

Figure 4 shows the results of a scaling obtained at the EuroHack17 at CSCS[5]. At that time the preliminary version of the code was able to run over one GPU per computing node, and we ported only the first phase of the Barnes-Hut gravity solver. In this test we run a gravity-only run with increasing particle sizes in order to occupy more and more computing nodes, and varied the number of *MPIranks* up to 1024. Where the data point with the largest number of CPUs (and GPUs) is simulation is Magneticum Box2/hr, that has with $2 \cdot 1584^3 = 7.9 \cdot 10^9$ particles. Both the OpenACC and the standard runs uses the same amount of CPUs.

4. Conclusions and Outlook

We presented a porting of all main Gadget3 modules (gravity computation, SPH density computation, hydrodynamic force, and thermal conduction) on GPUs using OpenACC.

We justified our choices of the porting as:

- the use OpenACC minimizes the rewriting of code and to let the community keep working on both CPU and GPU;
- OpenACC is also useful since we offload to the GPU only timesteps with a high number of active particles (as they won't perform well in a GPU);
- during a simulation, and at every timestep, we offload to the GPU only one module per time as to maximize the number of particle per each host;

[4] https://www.cscs.ch/publications/stories/2018/conradin-roffler-my-internship-at-cscs/
[5] https://github.com/fomics/EuroHack17/wiki/GadgetACC

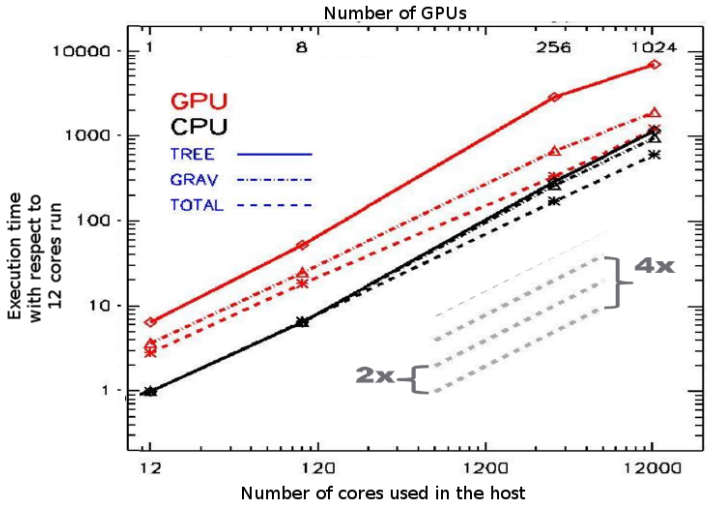


Figure 4. Scaling of a preliminary Gadget3 OpenACC porting of the gravity module, done at the EuroHack17 at CSCS. Y-axis show the speedup with respect to the CPU only version over 12 cores. Black (bottom) lines show the data by varying the number of MPI ranks (X axis) for a CPU-only run. Red (upper) lines show data for the OpenACC version, on the same number of CPUs and one additional GPU for each MPI rank. Continuous lines show the tree-walk speedup, dotted-dashed lines show the speedup for the gravity module, dashed lines show the speedup of the whole time step.

- by doing so, we exploit the host machine by overlapping GPU and CPU computations (as the CPU takes care of neighbour exchanges).

We used the same kind of porting paradigm on all Gadget modules, and showed how it keeps its speedup over different architectures (e.g. V100+NVLink or P100+PCI Express) and number of devices. This points to the direction that this kind of porting, which involves CPU/GPU computational overlap, is stable over different modules and architectures and may be useful for other multi-node N-body solvers.

Although we performed only one test that executes all modules up to the end of the simulation (Sec. 3.3), the various tests gave us the possibility to probe the performance on different configurations: with V100+NVLink technology (Sec. 3.1), with P100+PCI Express (Sec. 3.2), and over a large number of GPUs (Sec. 3.4). The EuroHack17 scaling in particular, showed how our approach (although it tests only one module, namely the gravity module) is capable of keeping its speedup up to a thousand of GPUs.

These tests were also useful to investigate the origin of the speedup by gradually increasing the profiled region of simulations, here we found that: (i) a single SPH density iteration, where we found a speedup of ≈ 4.5; (ii) a full time step, where we found a timestep of ≈ 3.5; (iii) to a full simulation and to a large number of GPUs where we found a total speedup of ≈ 2.

We briefly tested Unified Memory and found that, in our preliminary tests, this technology reaches the same performance of our explicit memory management. Unified Memory is a solution we will explore further because one does not have to manually set up the data transfer (which is not trivial in Gadget, since every timestep has only a subset of active particles).

Additionally, from that experience we found that the Domain Decomposition and the Tree Build are the new bottleneck of very large runs, once one speeds up the other modules with our OpenACC porting.

An initial step towards porting other modules of Gadget have been done, where we ported the cooling and star formation module. The other upcoming bottlenecks are the domain decomposition and the tree build algorithms, which by now are neither MPI parallel nor OpenMP parallel.

Acknowledgement

This work was carried out within the EuroEXA and ExaNeSt (FET-HPC) project (grant no. 754337 and no. 671553). We thank Emmanouil (Mano) Farsarakis from EPCC; Margarita Petkova and Rupam Bhattacharya from TUM, Milena Valentini from LMU; Luigi Iapichino, Nicolay Hammer, and Michele Martone from LRZ.

References

[1] V. Springel, N. Yoshida, and S. D. M. White, "GADGET: a code for collisionless and gasdynamical cosmological simulations," *New Astronomy*, vol. 6, pp. 79–117, Apr 2001.

[2] V. Springel, "The cosmological simulation code GADGET-2," *MNRAS*, vol. 364, pp. 1105–1134, Dec. 2005.

[3] M. Allalen, G. Bazin, C. Bernau, A. Bode, D. Brayford, M. Brehm, J. Diemand, K. Dolag, J. Engels, N. Hammer, H. Huber, F. Jamitzky, A. Kamakar, C. Kutzner, A. Marek, C. B. Navarrete, H. Satzger, W. Schmidt, and P. Trisjono, "Extreme scaling workshop at the LRZ," in *Parallel Computing: Accelerating Computational Science and Engineering (CSE), Proceedings of the International Conference on Parallel Computing, ParCo 2013, 10-13 September 2013, Garching (near Munich), Germany* (M. Bader, A. Bode, H. Bungartz, M. Gerndt, G. R. Joubert, and F. J. Peters, eds.), vol. 25 of *Advances in Parallel Computing*, pp. 691–697, IOS Press, 2013.

[4] J. Barnes and P. Hut, "A hierarchical O(N log N) force-calculation algorithm," *Nature*, vol. 324, pp. 446–449, Dec. 1986.

[5] R. A. Gingold and J. J. Monaghan, "Smoothed particle hydrodynamics: theory and application to non-spherical stars," *Monthly notices of the royal astronomical society*, vol. 181, no. 3, pp. 375–389, 1977.

[6] A. M. Beck, G. Murante, A. Arth, R.-S. Remus, A. F. Teklu, J. M. F. Donnert, S. Planelles, M. C. Beck, P. Förster, M. Imgrund, K. Dolag, and S. Borgani, "An improved SPH scheme for cosmological simulations," , vol. 455, pp. 2110–2130, Jan. 2016.

[7] L. Tornatore, S. Borgani, V. Springel, F. Matteucci, N. Menci, and G. Murante, "Cooling and heating the intracluster medium in hydrodynamical simulations," , vol. 342, pp. 1025–1040, Jul 2003.

[8] A. Ragagnin, N. Tchipev, M. Bader, K. Dolag, and N. J. Hammer, "Exploiting the Space Filling Curve Ordering of Particles in the Neighbour Search of Gadget3," in *Advances in Parallel Computing*, pp. 411–420, May 2016.

[9] K. Dolag, S. Borgani, G. Murante, and V. Springel, "Substructures in hydrodynamical cluster simulations," , vol. 399, pp. 497–514, Oct 2009.

[10] R. Farber, *Parallel programming with OpenACC*. Newnes, 2016.

[11] D. Foley and J. Danskin, "Ultra-performance pascal gpu and nvlink interconnect," *IEEE Micro*, vol. 37, no. 2, pp. 7–17, 2017.

[12] NVIDIA Corporation, "NVIDIA CUDA C programming guide," 2010.

[13] D. Negrut, R. Serban, A. Li, and A. Seidl, "Unified memory in cuda 6.0. a brief overview of related data access and transfer issues," *Tech. Rep. TR-2014–09, University of Wisconsin-Madison, 2014*, 2014.

[14] N. Hammer, F. Jamitzky, H. Satzger, M. Allalen, A. Block, A. Karmakar, M. Brehm, R. Bader, L. Iapichino, and A. Ragagnin, "Extreme Scale-out SuperMUC Phase 2 - lessons learned," *arXiv e-prints*, p. arXiv:1609.01507, Sep 2016.

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200044

Exploring High Bandwidth Memory for PET Image Reconstruction

Dai YANG, Tilman KÜSTNER, Rami AL-RIHAWI, and Martin SCHULZ
{*d.yang, tilman.kuestner, martin.w.j.schulz*}*@tum.de*
Chair of Computer Architecture and Parallel Systems
Technical University of Munich

Abstract. Memory bandwidth plays an essential role in high performance computing. Its impact on system performance is evident when running applications with a low arithmetic intensity. Therefore, high bandwidth memory is on the agenda of many vendors. However, depending on the memory architecture, other optimizations are required to exploit the performance gain from high bandwidth memory technology. In this paper, we present our optimizations for the Maximum Likelihood Expectation-Maximization (MLEM) algorithm, a method for positron emission tomography (PET) image reconstruction, with a sparse matrix-vector (SpMV) kernel. The results show significant improvement in performance when executing the code on an Intel Xeon Phi processor with MCDRAM when compared to multi-channel DRAM. We further identify that the latency of the MCDRAM becomes a new limiting factor, requiring further optimization. Ultimately, after implementing cache-blocking optimization, we achieved a total memory bandwidth of up to 180 GB/s for the SpMV operation.

Keywords. Intel Xeon Phi, MCDRAM, Sparse Matrix-Vector Multiplication, Maximum Likelihood Expectation-Maximization, Positron Emission Tomography

1. Introduction

The Intel Xeon Phi product family, based on the Intel Many-Integrated-Core (MIC) architecture, targets high-performance computing (HPC) applications [26]. Especially the Knights Landing (KNL) microarchitecture provides a large number of cores, achieving a high aggregated performance and a high performance per watt ratio [5]. The package also includes 3D-stacked DRAM, which provides a high memory bandwidth. The KNL may be a discontinued product, but the design of both utilizing many-core architecture and high bandwidth memory remains important for modern HPC systems. Other notable products featuring a high bandwidth memory architecture include both NVIDIA and AMD's graphics cards.

Positron Emission Tomography (PET) is a medical imaging modality with clinical value for the detection, staging, and monitoring of many diseases. It is a functional imaging technique, as it allows the observation of metabolic processes. A radioactive tracer is injected into the patient or subject. The tracer

undergoes beta decay, emitting positrons, which annihilated with electrons, creating two 511 keV gamma photons traveling in opposite directions. The scanner consists of a ring of detectors, with scintillator crystals and photodiodes. When two detectors each record a photon within a certain time window, an annihilation event is assumed somewhere along the line connecting the detectors, called the Line of Response (LOR). In reality, we have a Tube of Response (TOR), as two detectors can detect events not only from a line but from a larger, roughly polyhedral portion of the three-dimensional space inside the scanner tube, called Field of View (FOV). The number of detected events influences the quality of the measurement, while the covered area of the field of view by LORs affects the achievable resolution. The resolution is usually better at the center than at the edges of the field of view. For image reconstruction, the field of view is divided into a three-dimensional grid, where each grid cell is called a voxel.

In this paper we use the small animal PET scanner MADPET-II as an example (see Figure 1 (left)). The scanner has a unique design consisting of two concentric rings of detectors, which increases sensitivity while preserving spatial resolution [15]. To obtain an image from the scanner, the detector output – called list-mode *sinogram* – needs to be reconstructed using a system matrix.

There is a number of image reconstruction algorithms used in medical imaging. The algorithms used in our MADPET-II is Maximum Likelihood Expectation-Maximization (MLEM) [24]. A detailed mathematical description of the physical processes involved in tomography systems, such as the attenuation and scattering of photons in the body, is presented by Vazquaz et al. [29].

The main contributions of this paper are:

- We present an optimized MLEM implementation for modern many-core systems.
- We show the impact of increased memory bandwidth on Sparse Matrix-Vector (SpMV) operation performance in MLEM by benchmarking our implementation on an Intel Xeon Phi (KNL) based system.
- We identify the role of latency and propose future optimization recommendations for MLEM and SpMV codes in general.

2. Intel Xeon Phi Knight's Landing

Applications can be classified by the limitation on their performance into three categories, namely compute bound, (memory) latency bound, and (memory) bandwidth bound. To improve the performance of a compute bound application, the utilization of more cores is sufficient. To improve the performance of a memory (bandwidth) bound applications, two memory technologies with a higher memory bandwidth are developed: the *High Bandwidth Memory*(HBM) and the *Hybrid Memory Cube*(HMC). Both of the technologies are based on 3D-stacking of the classic DRAM dies. These memory modules are physically installed onto the processor package. However, stacking of the DRAM chips requires a significantly higher amount of wiring and controlling logic, which results in a higher latency of the memory. On the Intel Xeon Phi processor with Knight's Landing (KNL) ar-

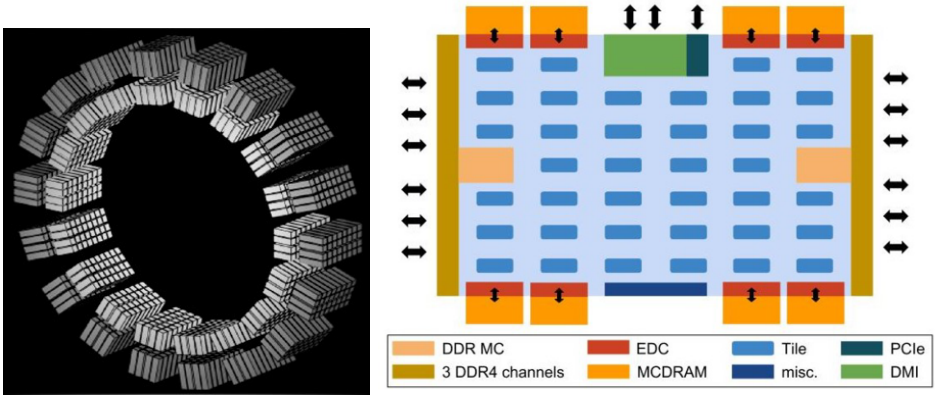


Figure 1. (Left): Illustration of the small animal PET scanner [14]. (Right): Diagram demonstrating an overview of the KNL Architecture.

chitecture, an HMC-based high bandwidth memory called Multi-Channel DRAM (MCDRAM) is embedded in the processor. Finally, to improve the performance of memory (latency) bound application, advanced optimization techniques such as cache-blocking and prefetching can be used.

The Xeon Phi Knights Landing (KNL) is part of the Xeon Phi family of processors, which is based on the Intel MIC architecture. The MIC architecture integrates many x86 processor cores with vectorization support to deliver massive parallelism. It is designed for high-performance computing applications [7].

In general, the KNL chip consists of tiles, MCDRAM, DRAM, and I/O, as shown in Figure 1. Each tile consists of two simple out-of-order cores that are derived from Intel Atom cores (based on the Silvermont microarchitecture). Each core supports up to 4 (hyper-)threads per core - running at 1.3 to 1.5 GHz. They are equipped with a 32KB L1 data cache and a 32KB L1 instruction cache. A 1MB L2 cache is shared among a single tile. The MCDRAM consists of 8x2GB blocks connected to different memory controllers located on different regions of the processor. The two DDR memory controllers support six channels with a bandwidth of up to 90 GB per second. The MCDRAM on KNL processors can be used in three different modes, namely Cache, Flat, and Hybrid. An overview is given below. In this paper, we used the flat mode to control MCDRAM usage.

- **Cache**: MCDRAM can be used as the Last Level Cache (LLC).
- **Flat**: The entire MCDRAM memory is added to the address space extending the space of the existing DDR4 Memory. In this mode, it is possible to allocate memory in the MCDRAM explicitly.
- **Hybrid**: It allows the MCDRAM to be partitioned into a part-cache and part-flat configuration by specifying a ratio between the two, either 75% - 25%, 50% - 50%, or 25% - 75%.

In addition to the MCDRAM configuration, there are several ways to configure the memory subsystem: All-to-all (A2A), Quadrant (quad), Hemisphere (HEMI), Sub-NUMA Clustering 4 (SNC4) and Sub-NUMA Clustering 2 (SNC2).

Detailed descriptions of these cluster modes are given by Jeffers et al. [9] and Sodani et al. [26]. A short overview of the differences between the cluster modes used in this paper is presented through the following scenario:

- All-to-All (A2A): In this cluster mode, memory addresses are uniformly distributed across all Tile Directories (TDs) plus the memory (MCDRAM and DDR) is set to UMA.
- Sub-NUMA Clustering 4 (SNC4): The memory subsystem divides the tiles into 4 clusters resulting in separate cache-coherent clusters.

3. The Maximum Likelihood Expectation-Maximization (MLEM) Algorithm

One widely used iterative reconstruction method for emission tomography is the Maximum Likelihood (ML) reconstruction using the Expectation-Maximization (EM) algorithm, which was proposed by Shepp et al. [24] in 1982. The algorithm uses the iteration scheme given in (1), where N is the number of voxels; M is the number of detector pairs; f is the 3D image that is reconstructed; A is the system matrix of size $M \times N$, which describes the geometrical and physical properties of the scanner; g is the measured list-mode sinogram of size M and q is the iteration number. The algorithm is based on the probability matrix $A = a_{ij}$, where each element represents the probability of a gamma photon discharge from a voxel j being recorded by a given pair of detectors i.

$$f_j^{(q+1)} = \frac{f_j^q}{\sum_{l=1}^{N} a_{lj}} \sum_{i=1}^{N} \left(a_{ij} \left(\frac{g_i}{\sum_{k=1}^{M} a_{ik} f_k^q}\right)\right) \tag{1}$$

The algorithm starts with an initial estimate, a grey image. Then, in each iteration, it executes the following steps:

- **Forward projection**: $h = Af$. Project the current approximation of the image into the detector space.
- **Correlation**: $c_i = \frac{g_i}{h_i}$. Correlate the projection to the actual measurement.
- **Backward projection**: $u = A^T c$. Project the correlation factor back into image space by multiplying with the transposed matrix.
- **Update image**: $f_j^{q+1} = \frac{f_j^q}{n_j} u_j$. Update the image with the back-projected correlation factor and apply a normalization n.

The runtime of the algorithm is dominated by the two sparse matrix-vector operations, forward and backward projection. Note that we do not need to create and store the transposed matrix A^T, as the backward projection can be computed as $u^T = c^T A$.

The system matrix describes the geometrical and physical properties of the scanner. For MADPET-II, the field of view is divided into a grid of $140 \times 140 \times 40$ voxel in x-, y- and z-dimension, respectively. The 1152 detectors result in 664,128 unique detector pairs or lines of response. The matrix was generated by the Detector Response Function (DRF) model [10,27]. The matrix is stored in Compressed Sparse Row (CSR) format using single-precision floating-point numbers. (For a list of commonly used formats see Barrett et al. [1]).

For parallelization, the matrix is partitioned into blocks of rows with approximately the same number of non-zero elements per block. This results in good, albeit not perfect load balancing. A more fine-grained approach, which cuts elements within one row, is possible, but would result in additional management or copying overhead.

4. First Optimization for KNL Architecture and MCDRAM

Our existing MLEM code uses both OpenMP and MPI. In order to achieve the best performance for KNL, we have optimized our MLEM implementation with the following steps:

- We make **all memory allocations on the MCDRAM** by using the `memkind` library.
- We rewrite the matrix loading part to support the special memory allocation in the MCDRAM. In particular, a set of OpenMP threads are created prior to memory allocation, and the matrix is directly copied into the corresponding memory by each thread during initialization. This ensures memory first touch for all threads. We further enforce thread reuse and thread pinning during kernel execution.
- We add `#pragma unroll` and `#pragma ivdep` into the kernel to assist **auto-vectorization** for SIMD execution using the AVX512 units of the processor.

To summarize, we improved the data loading process to support high bandwidth memory and ensure locality. In addition, we enabled and assisted the auto-vectorization to improve instruction-level data parallelization.

5. Evaluation

To show the influence of MCDRAM on the execution time of MLEM, we compile and run our code on CoolMUC-III, which is built of 148 compute nodes. Each node consists of one Intel Xeon Phi 7210F (Knight's Landing, KNL) processor with 64 cores, 256 threads, 96 GB of main memory and 16 GB of on-chip MCDRAM. The memory subsystem is configured for Sub-NUMA clustering (*SNC4*) mode and all-to-all (*A2A*) modes. The MCDRAM is configured to be *flat* addressable. According to Intel, maximum bandwidth of 490GB/s for the MCDRAM and 90GB/s for the DDR RAM can be achieved on this processor [19].

To investigate the effect of high bandwidth memory, we have run our MLEM code with three different memory configurations: *A2A, DDR-A2A, and SNC4*, where *DDR-A2A* represents the result of the native execution on the DDR4 main memory. The MCDRAM itself is also set to flat mode. As mentioned in Section 4, we use the *memkind* library and its `hbw_alloc` to explicitly allocate memory on the MCDRAM. For experiments on the DDR-4 memory, a standard `malloc` is used. We run setup runs ten times. During each run, the algorithm records the iteration time for the forward projections and backward projections, as well as the total iteration time. The first iteration is disregarded as we consider it

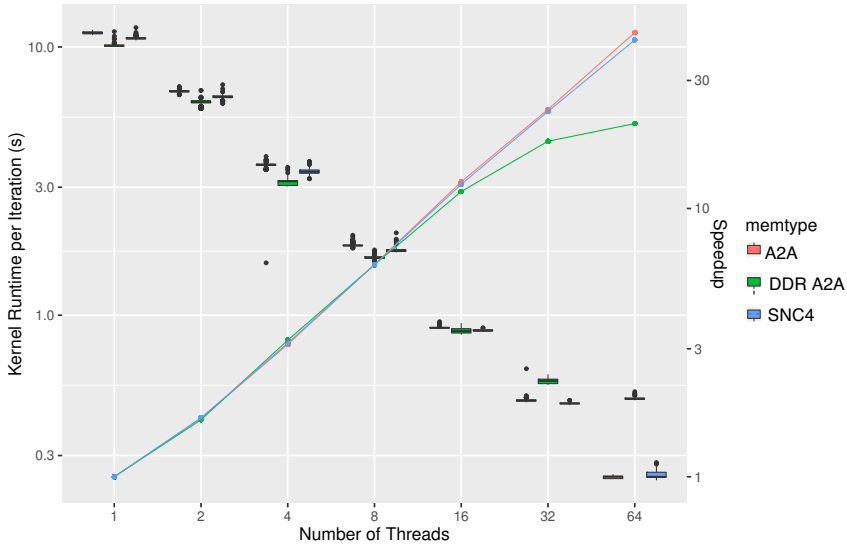


Figure 2. Runtime Comparison of MLEM on CoolMUC-III. The bars (from left to right in each cluster) represent *A2A*, *DDR-A2A*, and *SNC4*.

as warm-up time. The total kernel runtime per iteration, its speedup, and the memory bandwidth for the forward projection are assessed. In this paper, we analyze the forward projection for evaluation because the backward projection requires random access into memory space, which is significantly slower than streaming as required by the forward projection. The results of our experiments are summarized in Figure 2 and 3.

Figure 2 shows the comparison of runtime per iteration in seconds on CoolMUC-III using different memory configurations/modes. Best performance is achieved using the *A2A* setting on the MCDRAM. The *SNC4* setup follows with a slightly higher runtime. The fastest runtime with 64 threads on *A2A* mode is around 0.24s per iteration. We achieve a maximum speedup of about 50x, showing almost perfect scaling behavior. The runtime of the *DDR-A2A* version is significantly higher when using 32 or 64 OMP threads. In addition, the speedup curve indicates a typical saturating line that shows reducing speedup with increasing numbers of threads, indicating that the memory bandwidth limit is hit.

However, our speedup curve indicates a near linear increase of speedup in relation to the number of threads for executions on the MCDRAM. This shows that the code is capable of exploiting the higher memory bandwidth.

Figure 3 shows the corresponding bandwidth achieved for the forward projection, which contributes up to ˜50% to the runtime per iteration. The best memory bandwidth is achieved using the *A2A* mode on MCDRAM, which reflects the result from Figure 2. The slightly higher bandwidth on *A2A* mode over *SNC4* mode is also found in other research, such as stated by Ramos et al. [20]. However, the difference between A2A and *SNC4* is not significant. The maximum bandwidth at 76GB/s on the DDR-4 memory is close to the maximum of 90GB/s with stream benchmark. The bandwidth observed on the MCDRAM is double as high as on the DDR-4, with a maximum of appox. 150GB/s. Although we are

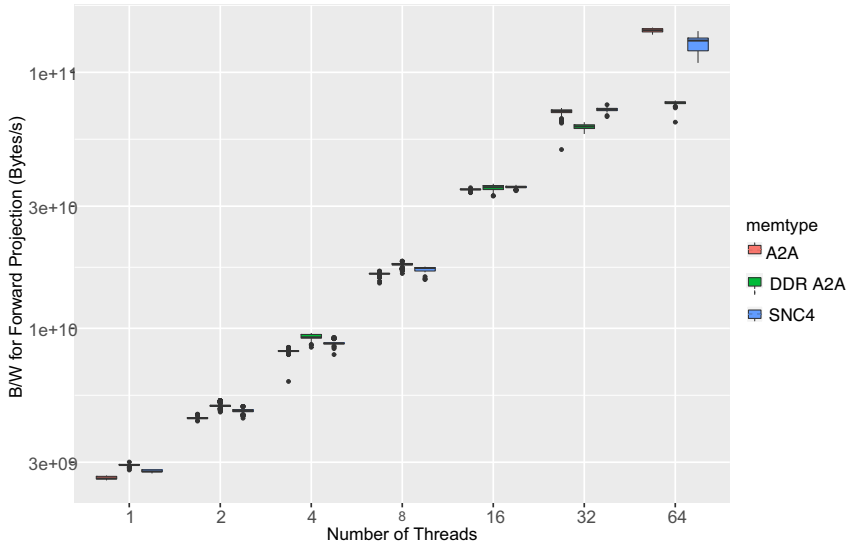


Figure 3. Memory Bandwidth Utilization Comparison of MLEM on CoolMUC-III. The bars (from left to right in each cluster) represent *A2A*, *DDR-A2A*, and *SNC4*.

able to exploit the higher bandwidth of the MCDRAM, we are not able to utilize the full memory bandwidth limit of 490GB/s on MCDRAM. We assume that the cause of this huge gap is the small L2 cache size, which makes it impossible to store the entire working vector within the cache, requiring more frequent load-/store operations. Combined with a high latency introduced by the MCDRAM, the kernels suffer from the frequent load/store operations. Similar results can also be found in related research by Saule et al. [23]. Anecdotal evidence collected by enforcing software prefetching by using `-qopt-prefetch` shows slightly increased performance, which reflects the memory latency boundary.

Further optimization featuring the implementation of a cache-blocking scheme for the working vector is implemented to reduce latency impact. This way, we are able to achieve a bandwidth of approx. 180GB/s on the MCDRAM with the *A2A* configuration.

6. Related Work

Speeding up iterative emission tomography image reconstruction, such as MLEM, has been an important research topic. Improvements have been made on the algorithmic side [6, 11, 21], as well as on the implementation side [3]. Work has also been done porting the MLEM algorithm to distributed GPU clusters [3, 16].

Lui et al. [12] investigated the performance of sparse matrix-vector multiplication (SpMV) on the Knights Corner (KNC), the predecessor of the Knights Landing architecture. They are able to reach 90% of the device's peak memory bandwidth by using a specialized data structure. Bell and Garland [2] show techniques on how to implement SpMV on GPUs (which typically include high band-

width memory nowadays), resulting in a good performance in several sparsity classes.

As KNL offers a number of configuration possibilities, it is especially important to review the work done in this field, with respect to cluster modes, memory modes, and thread affinity.

Rosales et al. [22] investigated the effect of cluster modes on the performance of HPC applications. The paper uses mini applications (MiniFE [4], MiniMD [4], and LBS3D [30]) to observe the performance differences in *A2A* and *QUAD* cluster modes running 1 to 256 threads. The results from MiniFE and MiniMD, using DRAM or MCDRAM, show that *A2A* mode scales comparable to or better than the quad mode. On the other hand, the results from LBS3D when using MCDRAM show varying performance behavior. When using DRAM with *A2A* mode, the code performs better or similar to *quad* mode. Moreover, *A2A* mode scales slightly better than *QUAD* mode when using MCDRAM and significantly better when using DRAM. Ultimately, the effects of the cluster modes seem to be dependent on the application.

Smith et al. [25] compared the effect of the MCDRAM memory modes, *flat* and *cache*, on the performance of sparse tensor factorization, using several datasets. The results reveal that both *flat* and *cache* mode perform identically when the dataset fits into MCDRAM; otherwise *flat* mode performs better than *cache* mode. Peng et al. [18] thoroughly investigated the effects of memory modes across several applications and benchmarks. The paper shows the performance of XSBench [28] over a range of problem sizes, Graph500 [17] over a range of graph sizes, GUPS [13] over a range of table sizes, MiniFE [4] over a range of matrix sizes, and DGEMM [13] over a range of array sizes in flat and cache modes. The results of XSBench and DGEMM show similar performance, Graph500 and GPUs show varying performance, and MiniFE shows flat outperforms the cache mode.

Jabbie et al. [8] observed the performance of the classical elliptic test problem of the Poisson equation on KNL. The work tests two pinning techniques, scatter and balanced, over a range of processes and threads using a hybrid approach. The results show no observable difference in runtime behavior.

7. Conclusion and Future Work

In this paper, we present an implementation of a medical image reconstruction algorithm, the Maximum Likelihood Expectation-Maximization (MLEM), for Positron Emission Topography (PET), optimized for Intel's KNL architecture high memory bandwidth. We investigated the effects of the higher memory bandwidth on our MLEM and provide optimization considerations for SpMV-like code.

We show that SpMV kernels are able to exploit the higher memory bandwidth. However, the higher memory latency makes it hard to exploit the full potential of the MCDRAM on KNL. The massive parallelization combined with high memory bandwidth and latency hiding via cache-blocking provides a significant speedup of MLEM. Overall we achieve a maximum memory bandwidth of 180GB/s on the KNL processor with MCDRAM, which is a significant improve-

ment for our MLEM code.

The next steps are to optimize the matrix storage format and apply further latency hiding technologies to further speedup MLEM. We will also evaluate the efficiency of high bandwidth memory for MLEM on GPU architectures.

Acknowledgment This work is partially funded by German Federal Ministry for Education and Research under grant title 01|H16010D. Compute resource on CoolMUC-III is sponsored by Leibniz Supercomputer Centre under grant title `pr63qi`.

References

[1] Richard Barrett, Michael W Berry, Tony F Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk Van der Vorst. *Templates for the solution of linear systems: building blocks for iterative methods*, volume 43. Siam, 1994.

[2] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 18:1–18:11, New York, NY, USA, 2009. ACM.

[3] Jingyu Cui, Guillem Pratx, Bowen Meng, and Craig S Levin. Distributed MLEM: An iterative tomographic image reconstruction algorithm for distributed memory architectures. volume 32, pages 957–967. IEEE, 2013.

[4] Michael A Heroux, Douglas W Doerfler, Paul S Crozier, James M Willenbring, H Carter Edwards, Alan Williams, Mahesh Rajan, Eric R Keiter, Heidi K Thornquist, and Robert W Numrich. Improving performance via mini-applications. *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, 3, 2009.

[5] Yuta Hirokawa, Taisuke Boku, Shunsuke A Sato, and Kazuhiro Yabana. Performance evaluation of large scale electron dynamics simulation under many-core cluster based on knights landing. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, pages 183–191. ACM, 2018.

[6] H Malcolm Hudson and Richard S Larkin. Accelerated image reconstruction using ordered subsets of projection data. *IEEE transactions on medical imaging*, 13:601–609, 1994.

[7] Intel Corporation. Intel® Xeon Phi™ Processors. `https://www.intel.com/content/www/us/en/products/processors/xeon-phi/xeon-phi-processors.html`. Accessed: December 2018.

[8] Ishmail A Jabbie, George Owen, and Benjamin Whiteley. Performance comparison of Intel Xeon Phi Knights Landing. *SIAM Undergraduate Research Online (SIURO)*, 10, 2017.

[9] James Jeffers, James Reinders, and Avinash Sodani. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann, 2016.

[10] Tilman Küstner, Peter Pedron, Jasmine Schirmer, Melanie Hohberg, Josef Weidendorfer, and Sibylle I. Ziegler. Fast system matrix generation using the detector response function model on Fermi GPUs. In *2010 Nuclear Science Symposium and Medical Imaging Conference*, 2010.

[11] Robert M Lewitt and Samuel Matej. Overview of methods for image reconstruction from projections in emission computed tomography. volume 91, pages 1588–1611. IEEE, 2003.

[12] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 273–282, New York, NY, USA, 2013. ACM.

[13] Piotr Luszczek, Jack J Dongarra, David Koester, Rolf Rabenseifner, Bob Lucas, Jeremy Kepner, John McCalpin, David Bailey, and Daisuke Takahashi. Introduction to the HPC challenge benchmark suite. 2005.

[14] David P McElroy, Mark Van Hoose, Wendelin Pimpl, V. Spanoudaki, Torben Schüler, and Sibylle Ziegler. A true singles list-mode data acquisition system for a small animal PET scanner with independent crystal readout. *Physics in Medicine & Biology*, 50:3323, 2005.

[15] David P McElroy, Wendelin Pimpl, Marcis Djelassi, Bernd J Pichler, M Rafecas, T Schuler, and SI Ziegler. First results from MADPET-II: a novel detector and readout system for high resolution small animal PET. In *Nuclear Science Symposium Conference Record, 2003 IEEE*, volume 3, pages 2043–2047. IEEE, 2003.

[16] Debasis Mitra, Hui Pan, Fares Alhassen, and Youngho Seo. Parallelization of iterative reconstruction algorithms in multiple modalities. In *2014 Ieee Nuclear Science Symposium and Medical Imaging Conference (Nss/Mic)*, pages 1–5. IEEE, 2014.

[17] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the graph 500. *Cray User's Group (CUG)*, 19:45–74, 2010.

[18] Ivy Bo Peng, Roberto Gioiosa, Gokcen Kestor, Pietro Cicotti, Erwin Laure, and Stefano Markidis. Exploring the Performance Benefit of Hybrid Memory System on HPC Environments. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*, pages 683–692. IEEE, 2017.

[19] Karthik Raman. Optimizing memory bandwidth in knights landing on stream triad. `https://software.intel.com/en-us/articles/optimizing-memory-bandwidth-in-knights-landing-on-stream-triad`, 2016. accessed on 19. 07. 2019.

[20] Sabela Ramos and Torsten Hoefler. Capability models for manycore memory systems: A case-study with xeon phi knl. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 297–306, 2017.

[21] Andrew J Reader, Kjell Erlandsson, Maggie A Flower, and Robert J Ott. Fast accurate iterative reconstruction for low-statistics positron volume imaging. *Physics in Medicine & Biology*, 43:835, 1998.

[22] Carlos Rosales, John Cazes, Kent Milfeld, Antonio Gómez-Iglesias, Lars Koesterke, Lei Huang, and Jerome Vienne. A comparative study of application performance and scalability on the Intel Knights Landing processor. In *International Conference on High Performance Computing*, pages 307–318. Springer, 2016.

[23] Erik Saule, Kamer Kaya, and Ümit V Çatalyürek. Performance evaluation of sparse matrix multiplication kernels on intel xeon phi. In *International Conference on Parallel Processing and Applied Mathematics*, pages 559–570. Springer, 2013.

[24] Lawrence A Shepp and Yehuda Vardi. Maximum likelihood reconstruction for emission tomography. *IEEE transactions on medical imaging*, 1(2):113–122, 1982.

[25] Shaden Smith, Jongsoo Park, and George Karypis. Sparse tensor factorization on many-core processors with high-bandwidth memory. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, pages 1058–1067. IEEE, 2017.

[26] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. Knights landing: Second-generation intel xeon phi product. *Ieee micro*, 36:34–46, 2016.

[27] D Strul, RB Slates, M Dahlbom, Simon R Cherry, and Paul K Marsden. An improved analytical detector response function model for multilayer small-diameter pet scanners. *Physics in medicine & biology*, 48(8):979, 2003.

[28] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. XSBench-the development and verification of a performance abstraction for Monte Carlo reactor analysis. *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, 2014.

[29] C Vazquez, MJ Rodriguez-Alvarez, C Correcher, AJ González, F Sánchez, P Conde, and JM Benlloch. Parallelization of MLEM algorithm for PET reconstruction based on GPUs. In *Nuclear Science Symposium and Medical Imaging Conference (NSS/MIC), 2014 IEEE*, pages 1–4. IEEE, 2014.

[30] HW Zheng, Chang Shu, and Yong-Tian Chew. A lattice Boltzmann model for multiphase flows with large density ratio. *Journal of Computational Physics*, 218:353–371, 2006.

Parallel Architecture

This page intentionally left blank

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200046

The Architecture of Heterogeneous Petascale HPC RIVR

Miran ULBIN[a,1] and Zoran REN[a]
[a] *Faculty of Mechanical Engineering, University of Maribor, Maribor, Slovenia*

Abstract. The EU has launched EuroHPC Joint Undertaking initiative plan to build an exascale HPC by 2025. A petascale HPC will be built in Slovenia in a concerted effort by 2020. The aim is to establish a national HPC system by own design with low maintenance and power consumption costs of the system. The HPC architecture will be unique, built from the off-the-shelf state-of-the-art components, and will operate using the open-source system software. A small HPC prototype system of about 200 TFLOP/s computing capability will be built in the first phase to test various computing nodes and components, which will be later integrated into a full-scale supercomputer with approx. 2 PFLOP/s. The throughput of Infiniband and Ethernet interconnect solutions will be of particular interest. The presentation is first focused on the architecture of HPC prototype consisting of 82 heterogeneous nodes based on double AMD Epyc and Intel Xeon SCL processors in combination with GPU nodes, with the discussion of possible variations of interconnect configurations. The network configuration of full-scale HPC with 600 AMD Epyc nodes, GPU nodes and large hard drive storage with a connection to HPC prototype will be discussed next. Possibilities of open source software for operating, provisioning and maintaining system, as well as flexibility and security of several options for user access, will be given in conclusion.

Keywords. HPC architecture, petascale, heterogenous CPU-GPU, open-source

1. Introduction

Roadmaps are clear for building exascale HPC in China by 2020 [1] and the USA by 2021 [2], while the EU has the plan to build exascale HPC by 2025 [3]. Slovenia has joined to Declaration Cooperation framework on High-Performance Computing in 2017 with an obligation to build integrated high-performance computing infrastructure, which will enable a competitive level of research and industry.

There is some computer infrastructure in Slovenia, which could be classified as high-performance computing. Some systems had begun in the late 20th century as research projects building computer grids and HPC systems, but most systems were small scale. Largest HPC in Slovenia today is performing with speed about 43 TFLOPS. Therefore, it was decided to build an HPC in the scope of petascale. As this is considerably more than any other system in Slovenia, it will be established as a national HPC system.

[1] Corresponding Author, Miran Ulbin, Faculty of Mechanical Engineering, University of Maribor, Smetanova 17, 2000 Maribor, Slovenia; E-mail: miran.ulbin@um.si.

There are two ways to build an HPC system. The easiest way is to buy an existing system from vendors like IBM, HP, Cray, NVIDIA, etc. Beside of higher initial and maintenance cost, such a system is also more rigid than custom build system. Various researchers, from the very different research field, will use national HPC system. The system should also provide a large number of services for the research community. One major requirement is the provision of a national repository of open access publications and research data. HPC vendors usually compete with benchmark test, which will prove that the system efficiently solves customer problems. With a diversity of research areas from massive parallel simulations to artificial intelligence problems or big data analysis, it is impossible to identify a simple benchmark test.

Because of that, HPC-RIVR was designed to be custom build heterogeneous [4] and very flexible to cover various research areas. Besides that, it should include large storage for research data. Another aspect is the maintenance cost of HPC-RIVR system. Power consumption is the main cause of HPC running cost and one of the major requirement for equipment was power efficiency. Another aspect was compatibility with the majority of software and efficiency of various hardware solutions. After comparing different solutions using benchmarks [5], the initial design was drawn. Some unknown remains, so it was decided to build HPC prototype first, to test some hardware and software configurations.

The system software is also not vendor based and only open-source software will be used. This requires the development of custom-based procedures and scripts for provisioning and maintenance of system software. Beside standard batch submission of jobs, user-friendly interfaces for HPC usage are developed [6]. HPC prototype is dedicated to development and testing system software, user interfaces and special configurations, while HPC-RIVR is designed as a production system where tested changes will be applied when needed.

2. HPC prototype

HPC prototype was designed to test various configurations and setup. Size of HPC prototype is about 10% of the size of the complete HPC system. It is built with a flexible design with equipment installed in a container presented if figure 1. Container with a length of 6,5 m and width and height of 2,9 m is equipped with all support system needed for HPC computer. Several racks are installed in the container, as can be seen from figure 1. Four racks are dedicated to HPC servers and two racks include a power supply with UPS and supporting systems for fire alarm and remote surveillance. The redundant mechanical and electrical cooling system is installed in the container so that there is a warm zone on one side of racks and cool zone on the other side.

Network connection of HPC prototype is realized with optical connectors providing 10 Gb/s connections to the internet, which will be later replaced by 100 Gb/s connection to HPC-RIVR system. Interconnect is built with two Mellanox 3800 Ethernet 100Gb/s switches each with 64 ports. As every HPC component is at a short distance to the switch, DAC copper cables using QSFP28 connector are used for the connection. Switches also enable connection of 10GBase-LR SFP+ module using an adapter which converts 40 Gb/s speed to 10 Gb/s and enables connection to existing 10 Gb/s switches. Three additional Ethernet Quanta T148-LY4R 1 Gb/s switches are used for network management connection.

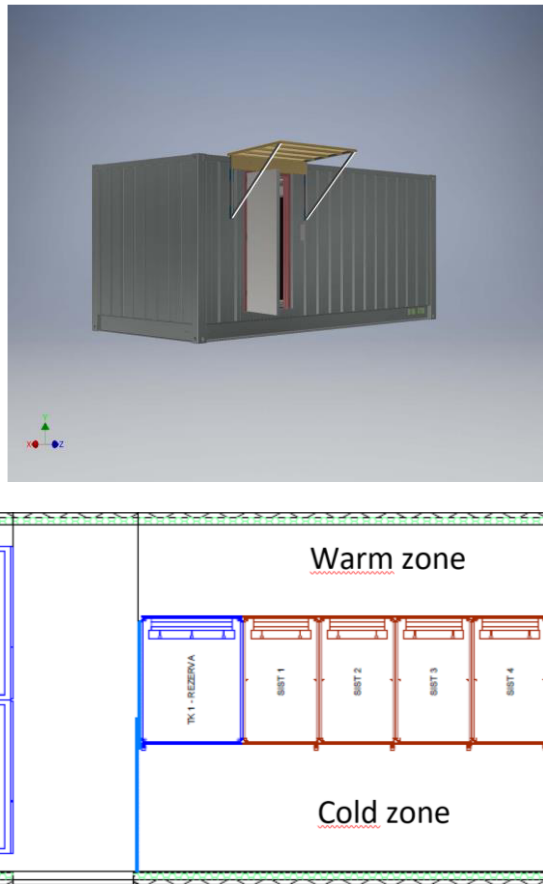


Figure 1. Container for HPC prototype.

Part of the HPC prototype is interconnected using Mellanox 8700 HDR100 100 Gb/s Infiniband switch. The first draft of the system considered EDR Infiniband switch with only 36 ports therefore, there were not enough ports for the whole system. With the new switch, it is possible to connect all nodes to Infiniband as each port can connect to two HDR100 interface cards. Also, previously planned interfaces in form of single port ConnectX-5 VPI network card were replaced by new ConnectX-6 interface cards which also have two modes of operation, one for 100 Gb/s Ethernet and one for HDR100 Infiniband. The distance of Ethernet connection between switch and component is rather small, so copper cables with QSFP56 HDR connectors were used.

The main purpose of the Infiniband switch is the comparison of performance between Ethernet and Infiniband interconnect. Although there are some results of Infiniband versus Ethernet comparisons [7], it is strongly dependent on packet size. In the report [7] the speed is almost identical for small packets while there is a huge difference for larger packets. Therefore, our purpose is to test network speed using typical applications utilizing either Infiniband or Ethernet interconnect. Results of comparison will influence the architecture of HPC-RIVR regarding interconnecting and it is possible that this will enable more cost-effective architecture of a network for our petascale HPC-RIVR.

HPC prototype is presented in figure 2, where it is shown that some nodes are connected only to Ethernet switch, while others are connected to Ethernet and Infiniband switch as discussed above. During tests, this configuration could change and SSD storage servers might be connected to Infiniband switch for testing purposes.

There are three computers of figure 2, which are designated with the name SuperMicro Server. These servers have the role of a head node and general-purpose servers with different services like Slurm, Web servers, etc. Each SuperMicro Server consists of two AMD Epyc 16C/32T 7301 2.2G 64M processors on board with 256 GB DDR4-2666 LRDIMM ECC, with two 480 GB SSD SATA drive configured in RAID 1 and with two 100 Gb/s Eth/IB Mellanox ConnectX-6 VPI network card. Each ConnectX-6 VPI network card could be connected to 100 Gb/s Ethernet switch or HDR Infiniband switch.

There are also three computers in figure 2, with the name SuperMicro Storage. These are storage servers with one AMD EPYC 24C/48T 7401P 2.0G 64M processor on board, with 256 GB DDR4-2666 LRDIMM ECC, with two 480 GB SSD SATA drive configured in RAID 1 and with two 100 Gb/s Eth/IB Mellanox ConnectX-6 VPI network card. Although there is a connection to Ethernet switch in figure 2, network cards can also connect to Infiniband switch. The better configuration will be considered for future use. Each storage servers contains 24 1,92 TB SSD drives beside two system drives. Storage servers are organized using CEPH [8], which also will provide testing for implementation on petascale HPC. The total size of SSD storage is 138 TB, which gives 69 TB of risky storage size and 46 TB of safe storage size [9].

Node SuperMicro GPU in figure 2 is a compute node with NVIDIA graphical accelerator boards. Each node consists of two Intel Gold SKL-SP 6128 6C/12T 3.4G processors, 256 GB DDR4-2666 LRDIMM ECC RAM, with two 480 GB SSD SATA drive configured in RAID 1, two 100 Gb/s Eth/IB Mellanox ConnectX-6 VPI network cards and four NVIDIA TESLA V100 32G PCI-E x16 boards. Each NVIDIA TESLA V100 board have 5120 cores enabling 7 double-precision TFLOPS. Usually, NVIDIA graphics boards are used with combination with INTEL processors and usage of AMD processors is not well proven with this combination. GPU nodes are connected to HDR100 Infiniband and to 100 Gb/s Ethernet and there is also plan to compare computational speed using one or another network.

Ethernet and Infiniband compute nodes in figure 2 are AMD processor-based. Because the first draft of HPC prototype was planned with one EDR Infiniband switch, there were only 36 ports available for connecting Infiniband nodes. Change to HDR Infiniband switch occurred at last moment, therefore only 28 nodes were equipped with two interface cards and other nodes will be equipped with additional interface cards in case it will be shown that HDR100 Infiniband offers considerable benefits over 100 Gb/s Ethernet.

Each compute node consist of two AMD EPYC 32C/64T 7501 2.0G 64M processors, 512 GB DDR4-2666 LRDIMM ECC RAM and two 960 GB SSD SATA drive configured in RAID 1. There are 28 nodes which have two 100 Gb/s Eth/IB Mellanox ConnectX-6 VPI network cards and are connected to HDR100 Infiniband and 100Gb/s Ethernet. Other 48 nodes have only one ConnectX-6 VPI network card and are connected only to 100 Gb/s Ethernet. Nodes are built into 2U chassis, which can contain four nodes. This enables compact build in half of the space required for 1U/1N chassis.

Figure 2. HPC prototype.

Power consumption is maximum 2,2 kW per chassis and total maximum power consumption for AMD compute nodes is 41.8 kW. GPU compute nodes maximum total power consumption is 12 kW, storage server consumes 6 kW, servers 1,5 kW and switches 3 kW. Maximum total power consumption without cooling is about 64,3 kW.

System software for HPC prototype is based on open source solutions. Head node operating system is CentOS [10] while compute nodes will use the most efficient UNIX flavour available. Provisioning is based on Foreman [11] with a combination of Puppet [12] customization. There will be several configurations tested on HPC prototype with either batch usage using Slurm [13] and Nordungrid ARC [14] and more direct access via web interfaces and virtualization. Most of the work will be based on running applications built-in containers like Singularity [15].

HPC prototype system was installed and is in the testing phase since July 2019. Container with cooling where cabinets with HPC servers and nodes were stacked is shown in figures 3 and 4.

Figure 3. HPC prototype container.

Figure 4. HPC cabinets with servers and nodes.

3. Petascale HPC-RIVR

While HPC prototype is assembled and different hardware and software solutions are tested, complete HPC-RIVR is still in design. Reason for that is partly in the fact that results of testing will influence the final design and partly because technology is currently changing. There are new processors with more cores presented and will be in production this year. Infiniband speed will double this year and products with new PCI 4.0 bus will emerge. Therefore, the design of HPC-RIVR is somehow outlined, but it is very flexible to adapt to new technology and findings of tests on HPC prototype. Therefore, the final version is still under consideration and might be influenced by the vendor's proposals.

Rough design of HPC-RIVR consist of Ethernet network and Infiniband interconnect of head nodes, servers, compute nodes with an x86 processor, compute node with GPUs, SSD storage servers and hard disk storage servers. Network scheme of the design is shown in figure 5.

Ethernet is designed with redundant switches which should have large buffers to allow high throughput as it is expected that a large amount of data will be transferred to and from HPC. The consequence of that is more latency, but traffic between compute nodes should flow on Infiniband with low latency. If testing on HPC prototype proves that 100 Gb/s Ethernet switches with low latency can compete with Infiniband, the design will change accordingly and the cost of interconnect will be much lower. Research results from literature shows, that latency of Infiniband is still much lower than Ethernet, hence, a dual network is required. Additionally, there is the third network for management, which is served by 1 Gb/s switches and is not shown in figure 5.

The speed of the Ethernet network with presented configuration can be much lower at compute nodes and hard disk storage servers. It is designed with speeds of 25 Gb/s resulting in a lower cost of Ethernet network. Hard disk storage servers are connected with redundant connections, which on one hand double the speed and on the other hand makes the network more reliable.

Infiniband interconnect is based on HDR 200 Gb/s, but 200 Gb/s speed will be used only between switches. Nodes will use HDR 100 Gb/s interfaces and for HPC-RIVR 24 HDR 200 Gb/s switches are required for level 1 and 10 HDR 200 Gb/s switches for level 2 switch with the 3-1 blocking scheme. The total number of nodes connected to Infiniband is about 700, which requires 700 cables or optical fibres, while the connection between level 1 and level 2 switch requires 240 cables.

There are 30 general-purpose servers, which will be used as head nodes or dedicated nodes for services like scheduling, maintenance or user interfaces. The configuration of such server is identical as SuperMicro servers in HPC prototype, which is described above.

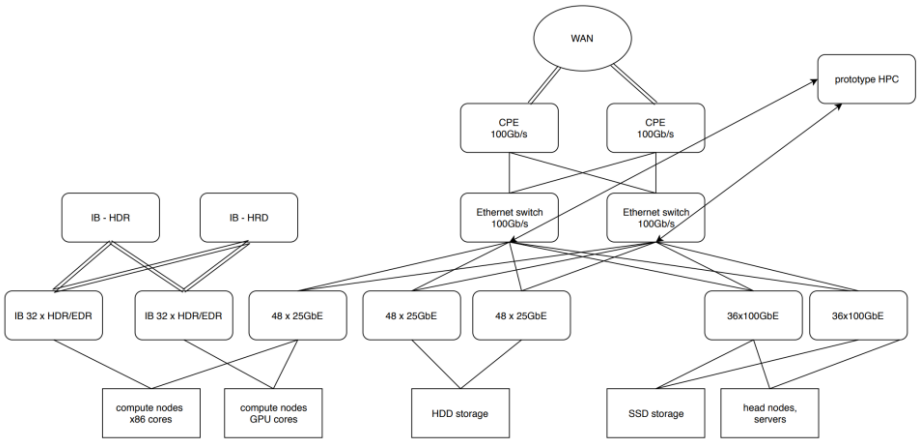


Figure 5. HPC-RIVR design.

In figure 5 HDD storage and SSD storage is presented in the bottom row. SSD storage is runtime storage for program and user data. SSD storage is designed with 22 servers each containing 24 SSD drives each with 1,92 TB space. The server is identical to SuperMicro storage in HPC prototype. Total SSD storage is therefore about 1 PB of raw space and will be configured using CEPH, which gives 483 TB of safe space. Similarly, HDD servers are designed with 20 servers where each is containing 90 hard disks each with 12 TB space. Storage servers will be configured using CEPH, with total raw space of 21.6 PB with 10.26 PB of safe space with two replicas.

GPU compute nodes are identical as in HPC prototype with four NVIDIA V100 boards. There is a total of 30 GPU servers in the design, with a total of 120 NVIDIA V100 boards. Compute nodes with x86 cores are same as Infiniband nodes in HPC prototype in the current design of HPC RIVR. As new processor with 64 cores emerges now, compute nodes might be different and there will be fewer nodes and required network connections.

The expected power consumption of HPC-RIVR is about 600 kW and an additional 200 kW for cooling. Power supplies and cooling equipment is currently under construction and dedicated space is prepared for installation in the next months. In figure 6 preparation for building HPC-RIVR is shown.

Figure 6. The equipment and dedicated space for HPC-RIVR.

4. Conclusions

Designing petascale HPC from scratch is a difficult task, comparing with the selection of the final product offered by established vendors. The final design has required an initial design of HPC prototype, where some results and solutions are expected. HPC prototype will also enable a more realistic estimate of our goal of achieving PFLOPS operation. Theoretically, designed hardware should result in value above 1 PFLOPS, but the final number will be calculated with the LINPACK [16] test.
As hardware is near the final design and system software will be finalized in HPC prototype, the majority of the work is still ahead. HPC-RIVR will be used by a variety of researchers with specific needs and requirements. Teams of researchers are currently building different applications for these needs and creating web portals and special user interfaces, which will enable user-friendly use of HPC. Findings during building HPC-RIVR and various software solutions could be used on the way to European exascale HPC.

Acknowledgements
The authors would like to thank the Ministry of Education, Science and Sport of the Republic of Slovenia and to the European Union – European Structural and Investment Fund for financial support.

References

[1] China Navigating The Homegrown Waters For Exascale (21.03.2019). Retrieved from: https://www.nextplatform.com/2018/11/15/china-navigating-the-homegrown-waters-for-exascale/ .
[2] The Roadmap Ahead For Exascale HPC In The US (21.03.2019). Retrieved from: https://www.nextplatform.com/2018/03/06/roadmap-ahead-exascale-hpc-us/ .
[3] R. Ammendola et al., 'Large Scale Low Power Computing System - Status of Network Design in ExaNeSt and EuroExa Projects', arXiv e-prints, p. arXiv:1804.03893, Apr. 2018.
[4] S. Lee, H. Seo, H. Kwon, and H. Yoon, 'Hybrid approach of parallel implementation on CPU–GPU for high-speed ECDSA verification', The Journal of Supercomputing, Jan. 2019.
[5] Kutzner C., Páll S., Fechner M., Esztermann A., L. de Groot B., Grubmüller H., ' More Bang for Your Buck: Improved use of GPU Nodes for GROMACS 2018', arXiv:1903.05918 [cs.DC], 2019.
[6] N. Fareghzadeh, M. A. Seyyedi, and M. Mohsenzadeh, 'Toward holistic performance management in clouds: taxonomy, challenges and opportunities', The Journal of Supercomputing, vol. 75, no. 1, pp. 272–313, Jan. 2019.
[7] Erickson, K., Kachelmeier, L., Van Wig, F.. 2016. Comparison of High Performance Network Options: EDR InfiniBand versus 100Gb RDMA Capable Ethernet; Poster; accepted Supercomputing Conference, SC'16 (21.03.2019). Retrieved from: https://permalink.lanl.gov/object/tr?what=info:lanl-repo/lareport/LA-UR-16-26126 .
[8] CEPH (21.03.2019). Retrieved from: https://ceph.com/ .
[9] CEPH storage calculator (21.03.2019). Retrieved from: http://florian.ca/ceph-calculator/ .
[10] CentOS (21.03.2019). Retrieved from: https://www.centos.org/ .
[11] Foreman (21.03.2019). Retrieved from: https://www.theforeman.org/ .
[12] Puppet (21.03.2019). Retrieved from: https://puppet.com/ .
[13] Slurm (21.03.2019). Retrieved from: https://slurm.schedmd.com/ .
[14] Nordugrid ARC (21.03.2019). Retrieved from: http://www.nordugrid.org/ .
[15] Singularity (21.03.2019). Retrieved from: https://www.sylabs.io/ .
[16] HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers (21.03.2019). Retrieved from: http://www.netlib.org/benchmark/hpl/ .

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200047

Design of an FPGA-Based Matrix Multiplier with Task Parallelism

Yiyu TAN[a,1], Toshiyuki IMAMURA[a], and Daichi MUKUNOKI[a]
[a]*RIKEN Center for Computational Science, Kobe, Hyogo, Japan*

Abstract. Matrix multiplication requires computer systems have huge computing capability and data throughputs as problem size is increased. In this research, an OpenCL-based matrix multiplier with task parallelism is designed and implemented by using the FPGA board DE5a-NET to improve computation throughput and energy efficiency. The matrix multiplier is based on the systolic array architecture with 10 × 16 processing elements (PEs), and all modules except the data loading modules are autorun to hide computation overhead. When data are single-precision floating-point, the proposed matrix multiplier averagely achieves about 785 GFLOPs in computation throughput and 66.75 GFLOPs/W in energy efficiency. Compared with the Intel's OpenCL example with data parallelism on FPGA, the SGEMM routines in the Intel MKL and OpenBLAS libraries executed on a desktop with 32 GB DDR4 RAMs and an Intel i7-6800K processor running at 3.4 GHz, the proposed matrix multiplier averagely outperforms by 3.2 times, 1.3 times, and 1.6 times in computation throughput, and by 2.9 times, 10.5 times, and 11.8 times in energy efficiency, respectively, even though the fabrication technology is 20 nm in the FPGA while it is 14 nm in the CPU. Although the proposed FPGA-based matrix multiplier only delivers 6.5% of the computation throughput of the SGEMM routine in the cuBLAS performed on the Nvidia TITAN V GPU, it outperforms by 1.2 times in energy efficiency even though the fabrication technology of the GPU is 12 nm.

Keywords. Matrix multiplication, FPGA, OpenCL

1. Introduction

Matrix multiplication is one of the fundamental building blocks of linear algebra, and has been widely applied in high performance computing (HPC) to solve scientific and engineering problems, such as deep learning, data analytics, and so on. In general, matrix multiplication requires computing systems to have huge computation capability and memory bandwidth as problem size grows. Nowadays, many methods and algorithms have already been developed to speed up computation through parallel programming techniques or improving the efficiency of memory hierarchy to reduce data access overhead in supercomputers, GPUs, multicores, many-cores, and cluster systems. On the other hand, power problems become more and more serious in HPC systems, and heterogeneous architectures are becoming the mainstream, in which GPUs or FPGAs are tightly integrated with multicore processors as accelerators to reduce power consumption, especially FPGAs. FPGAs deliver much higher energy efficiency through data parallelism and pipelining parallelism using a sea of individually PEs running at low

[1] Corresponding Author: Yiyu Tan, RIKEN Center for Computational Science, 7-1-26 Minatojima-minami-machi, Chuo-ku, Kobe, Hyogo, Japan; E-mail: tan.yiyu@riken.jp.

clock frequency. In recent years, an FPGA contains thousands of hardened floating-point arithmetic units and million-byte on-chip block RAMs (BRAMs). Furthermore, high-level synthesis tools let developers shift their focus from low-level HDL-based designs to C, C++, or OpenCL codes annotated with directives, which makes the system development much easier and development time being shortened. All these lead FPGAs to become attractive in HPC.

FPGA-based acceleration on matrix multiplication has received much attention in industry and academics. Zhuo et al. [1] and Dou et al. [2] proposed a matrix multiplier with PEs being one-dimensional linear array architecture. Zhuo et al. analyzed the design tradeoffs and optimized the design based on the hardware constraints. Dou et al. proposed a parallel block algorithm to improving performance by exploiting the data locality and reusability. Based on their work, Wu et al. [3] developed an I/O and memory optimized blocking algorithm to improve memory efficiency and reduce the required hardware resources. Kumar et al. [4] also presented a one-dimensional array architecture of PEs and applied the rank-1 update algorithm to schedule input data to PEs. Different with the architectures proposed by Dou and Zhou, this architecture distributes the same elements of the first matrix to all PEs through broadcasting. Pedram et al. [5][6] introduced a two-dimensional linear array architecture for matrix multiplication, in which data exchange between PEs was performed through row/column broadcasting buses. Such two-dimensional architecture provides benefits in scalability, addressing, and data movement over one-dimensional array architecture.

There are other FPGA-based accelerators on matrix multiplication for different purposes. Giefers et al. [7] presented an FPGA-based accelerator for matrix multiplication on a hybrid FPGA/CPU system to study energy efficiency. Jiang et al. [8] introduced a scalable macro-pipelined accelerator to perform matrix multiplication to exploit temporal parallelism and architectural scalability. Wang et al. [9] integrated multiple matrix accelerators with a master processor and built a universal matrix processor. Z. Jovanovic et al. [10] presented an accelerator to minimize resource utilization and maximum clock frequency by returning the computation results to the host processor as soon as they were computed. Andrade et al. [11] adopted high-level synthesis approach to generate two-dimensional embedded processor arrays for matrix algorithms. Holland [12] proposed high-level synthesis optimization strategies to maximize the utilization of the DSPs and BRAMs in blocked matrix multiplication. In this research, an FPGA-based matrix multiplier with task parallelism is presented for large-scale matrix multiplication. The major contributions of this work are as follows.

(1) Design and implementation of a matrix multiplier with task parallelism. The matrix multiplier is based on the systolic array architecture with 10 × 16 PEs, and data reuse and optimization techniques are applied to improve computing performance and energy efficiency.
(2) Task parallelism and data vectorization. The system is partitioned into different single-work-item kernels according to dataflow, and most of the kernels work at the autorun mode to reduce computation overhead. High-speed and high-bandwidth buffers are adopted to exchange data between PEs and kernels. Data vectorization is applied to compute the dot product of multiple data inside a PE to enhance the computation capability. The proposed system averagely achieves about 785 GFLOPs in computation throughput and 66.75 GFLOPs/W in energy efficiency in the case of data being single-precision floating-point.

The remainder of this paper is organized as follows. The system design and implementation are introduced in Section 2. In Section 3, performance evaluation results are presented, followed by conclusions drawn in Section 4.

2. System Design and Implementation

A matrix multiplication $C = A \times B$ is generally defined as follows:

$$C[i][j] = \sum_{k=0}^{P-1} A[i,k] \times B[k,j] \qquad (0 \le i < M,\ \ 0 \le j < N)$$

Where A, B, and C are $M \times P$, $P \times N$, and $M \times N$ matrices, respectively. The computations from the above formula can be described by the pseudocode shown in Figure 1. In Figure 1, a matrix multiplication consists of three loops, and their positions can be changed. The computations require $2 \times M \times P \times N$ floating-point operations and $M \times P + P \times N + M \times N$ memory accesses. To speed up the computations, firstly, the computation load in a clock cycle should be maximized, which means more floating-point multipliers and adders are involved into computation and work in parallel to get one or more of the scalar product C_{ij} at a clock cycle. This may be achieved by unrolling the inner loop (k) and outer loops (i and j) to get more parallel iterations at arithmetic level. At circuit level, it is achieved using deep pipelining of floating-point multiplication and addition units inside a PE, and many PEs are applied to calculate C_{ij}. Although the loops can be unrolled completely, they are limited by the number of DSP blocks inside an FPGA, which are applied to implement the floating-point arithmetic units. Secondly, parallel data stream is needed to feed the pipeline and shorten the overhead of data access. Thus, on-chip buffers are demanded to store the elements of matrices A and B in advance. In general, multiple-port and large-size buffers can read/write data in parallel and keep more data, but they are constrained by the size of BRAMs inside an FPGA. In addition, the overheads to writing data from external memory to on-chip buffers are affected by memory bandwidth. To address these, the blocked matrix multiplication algorithm is applied to partition the matrices into smaller blocks (sub-matrices), and parallelisms, such as data parallelism and task parallelism, are put on the sub-matrix multiplications. In this research, the matrix multipliers based on such two parallelisms are designed using OpenCL programming language and implemented using FPGA, respectively.

```
for i = 0; i < M; i++
  for j = 0; j < N; j++ {
    sum = 0.0;
    for k = 0; k < P; k++
      sum = sum + A[i][k] × B[k][j];
    C[i][j] = sum;}
```

Figure 1. Pseudocode of a matrix multiplication.

2.1 Matrix Multiplier with Data Parallelism

The matrix multiplier with data parallelism is referenced from the Intel's OpenCL design example [13], and the related code is shown in Figure 2. Both matrices A and B are stored by row-major in the host, and two local buffers, A_local and B_local, are defined to store the block data of matrices A and B, respectively (lines 4 and 5). Matrices A and B are written into the on-board DDR memory from the host machine before computation, and then a block of A and a block of B are read into A_local and B_local to perform the product of two blocks by the computation engine, which is defined by the N-dimensional index space (NDRange) in the OpenCL execution model through the attribute option

__attribute((reqd_work_group_size(BLOCK_SIZE,BLOCK_SIZE,1))). The calculated product C[i][j] is written into the on-board DDR memory (line 24), and finally transferred to the host machine after all computations are finished. Along with data reading from the external memory (lines 14 - 17), the block data of B are transposed before they are stored in the local buffer (line 17) to ensure consecutive data access during computation (line 21). Furthermore, the inner loop in Figure 1 is fully unrolled by adding the directive (line 19) to instruct the OpenCL compiler to implement parallelism by using more DSP blocks, and the outer loop in Figure 1 is parallelized and realized through introducing the two-dimensional computing engine defined by the NDRange. The main disadvantage of this matrix multiplier is that system will be stalled until computations in a block are completed (line 22) and new blocks are fed into the local buffers (line 18) to maintain data synchronization.

OpenCL code for blocked matrix multiplication

```
1:  __kernel void matrixmult ( __global float *restrict C, __global float *A,
2:                             __global float *B, int A_width, int B_width)  {
3:   //define local storage for a block of input matrices A and B
4:     __local float A_local[BLOCK_SIZE][BLOCK_SIZE];
5:     __local float B_local[BLOCK_SIZE][BLOCK_SIZE];
6:    int block_x = get_group_id(0);   //define block index: row
7:    int block_y = get_group_id(1);   //define block index: column
8:    int local_x = get_local_id(0);   //define local index: row
9:    int local_y = get_local_id(1);   //define local index: column
10:  int a_start = A_width*BLOCK_SIZE*block_y;   //loop start and stop points
11:  int a_end = a_start + A_width - 1;
12:  int b_start = BLOCK_SIZE * block_x;
13:  float sum = 0.0f;
14:  for (int aa=a_start, bb=b_start; aa<=a_end; aa+=BLOCK_SIZE, bb+= (BLOCK_SIZE*B_width)) {
15:    // load the matrices into local memory, and perform B<=B^T
16:    A_local[local_y][local_x]=A[aa+A_width*local_y+local_x];
17:    B_local[local_x][local_y]=B[bb+B_width*local_y+local_x];
18:    barrier(CLK_LOCAL_MEM_FENCE); // wait for the entire block to be loaded.
19:    #pragma unroll
20:    for (int k = 0; k< BLOCK_SIZE; ++k) {
21:        sum += A_local[local_y][k]*B_local[local_x][k]; }
22:    barrier(CLK_LOCAL_MEM_FENCE);  // wait for completion of computation.
23:   }
24    C[get_global_id(1)*get_global_size(0)+get_global_id(0)]=sum; // store result in matrix C
25:  }
```

Figure 2. OpenCL code for blocked matrix multiplication with data parallelism.

As shown in Figure 2, blocks of both matrices are read into the local buffers at each iteration, and multiplications are then carried out by the computation engine. The number of iterations is determined by the block size and matrix scale. The required hardware resources are affected by the complexity of the kernel shown in Figure 2, and not associated with the dimensions of matrices A and B, which only affect the number of iterations. In Figure 2, the consumed hardware resources are mainly determined by the size of local buffers (A_local and B_local), the unrolling of inner loop, the scale of the arithmetic array defined by the NDRange, and the kernel vectorization to specify the number of work items within a work group to execute in a single instruction multiple data (SIMD) fashion. Except for the kernel vectorization and the unrolling of inner loop being specified individually, the size of local buffers and the scale of the arithmetic array are determined by the block size. Consequently, the block size has great impacts on the required hardware resources and system performance.

2.2 Matrix Multiplier with Task Parallelism

In the matrix multiplier with task parallelism, the system is divided into different function modules in accordance to the data flow, and each function module is described as a kernel in OpenCL. As illustrated in Figure 3, the system consists of the computing kernel, data-feeding modules (feed_mat_A_kernel and feed_mat_B_kernel), matrix-loading modules (load_mat_A_kernel and load_mat_B_kernel), and data output module. Each kernel works with single work-item. Except for the matrix-loading modules, other kernels run at the autorun mode to reduce computation overhead. The function of each module is described as follows in detail.

- Matrix-loading modules. The matrix-loading modules read data from the on-board DDR memory into buffers according to the data vectorization, block size, block position, and data reuse.
- Data-feeding modules. The data-feeding modules read data from the buffer and feed the related data to the computing kernel. As shown in Figure 3, the number of data-feeding modules for matrices A and B equals to the number of rows and columns of the systolic array in computing kernel, respectively. Each data-feeding module will check and feed the corresponding data to the specific row or column of the computing kernel, and then forward the other data to the next neighbor data-feeding module.
- Computing kernel. The computing kernel is a systolic array with 10×16 PEs, and high-speed and high-bandwidth channels are applied to connect PEs and kernels. In the computing kernel, data of matrix A are shifted from the left to right while data of matrix B are moved from the top to bottom in the systolic array to reuse data.
- PE. The PE is the arithmetic unit to perform computation. The block diagram of a PE is presented in Figure 4, which consists of eight arithmetic units to compute the dot product of eight data at a clock cycle through pipelining, namely the data vectorization is eight. In Figure 4, each arithmetic unit is generated by the IP generator from the Quartus Prime Pro to perform different operations, and is implemented by using the hardened DSP blocks inside FPGA. In the current design, each arithmetic unit consumes one DSP block, which contains an adder and a multiplier with single-precision.
- Data output module. The data output module outputs the computation results to the on-board DDR memory. Similar as the data-feeding module for matrix B, several data output modules will be applied in accordance to the column of the systolic array in the computing kernel.

2.3 System Implementation

The matrix multipliers with data parallelism and task parallelism are compiled by using the Intel FPGA SDK for OpenCL 16.1 and implemented by using an FPGA board DE5a-NET from Terasic [16], which includes an Intel Arria 10 GX FPGA 10AX115N2F45E1SG and 8 GB on-board DDR3 memory. The FPGA contains 1518 hardened single-precision floating-point units, 427,200 ALMs, and 53 Mb M20K memory. The on-board memory is arranged at two independent channels with each being 4 GB. The matrices A and B are written into different banks of the on-board memory through PCIe bus, and they are accessed independently through different channels. The product, namely the matrix C, is firstly stored into the same memory bank as the matrix A, and finally written back to the host machine. In the matrix multiplier with data

parallelism, the block size is 128×128 and the kernel vectorization is four. In contrast, in the matrix multiplier with task parallelism, the data vectorization is eight, and the block size of matrix A is 320×256 while the block size of matrix B is 256×512.

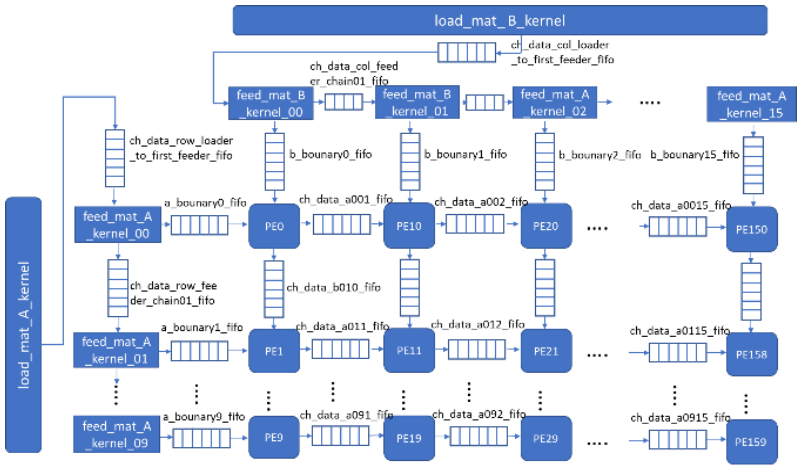


Figure 3. System block diagram.

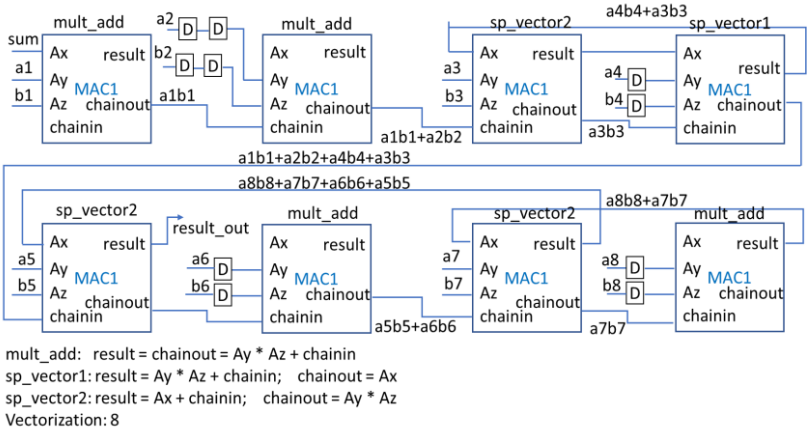


Figure 4. Block diagram of a PE.

The kernel code is compiled into the intermediate representations, applied necessary optimizations, converted to the Verilog files, and then performed synthesis, placement and routing to generate the final FPGA bitstream by the Intel FPGA SDK for OpenCL. Table 1 presents the hardware resource utilization in the two matrix multipliers in the case of data being single-precision floating-point. As shown in Table 1, the matrix multiplier with task parallelism utilizes more DSP blocks and gains much higher clock frequency over the matrix multiplier with data parallelism on FPGA. In both systems, the system performance is constrained by the size of on-chip BRAM blocks. Although the matrix multiplier with data parallelism consumes less DSP blocks (34%), if the block size is increase further to use more DSP blocks to improve system performance, such as 256×256, the on-chip BRAM blocks will be exhausted and the system cannot be synthesized because the block size affects the utilization of the BRAM blocks and DSP

blocks. On the other hand, in the matrix multiplier with task parallelism, the systolic array and data vectorization are determined by the available DSP blocks and BRAM blocks inside the FPGA. In the current design, the optimal systolic array contains 10 × 16 PEs, and the data vectorization is eight, which means each PE consumes eight DSP blocks. Therefore, the matrix multiplier requires 1280 (10×16×8) DSP blocks. Although the systolic array can be scaled up to 11 × 16 PEs according to the available hardware resources of the FPGA, the system is not synthesizable by the Intel FPGA SDK for OpenCL. In addition, the clock frequency of the proposed matrix multiplier with task parallelism is much higher than the matrix multiplier with data parallelism because of more optimized data path in accordance to the dataflow.

Table 1. Hardware resource utilization

Matrix multiplier	Hardware resource			
	Logic utilization	DSP blocks	RAM blocks	Clock frequency
Task parallelism	237128(56%)	1280(84%)	2529(93%)	305 MHz
Data parallelism	92002(22%)	520 (34%)	1774(65%)	235 MHz

3. Performance Evaluation

The proposed matrix multiplier is implemented by using the FPGA board DE5a-NET, and its performance is evaluated and compared with the matrix multiplier with data parallelism on FPGA [14-15], the SGEMM routine in the cuBLAS performed on the Nvidia TITAN V GPU [17], the SGEMM routines in the Intel MKL(version:2018.0.128) and OpenBLAS (version: 0.2.20) executed on a desktop machine with 32 GB DDR RAMs and an Intel i7-6800K processor running at 3.4 GHz. The operating system of the desktop machine is CentOS 7.0, and the compiler for the OpenBLAS is gcc 4.9.4. The compiler for the OpenCL is Intel FPGA SDK for OpenCL 16.1. In the GPU system, the host machine contains an Intel Xeon W-2123 processor with CentOS 7.4, CUDA 10.0, and the GPU driver version being 410.73. The fabrication technology in the FPGA Arria 10 is 20 nm while it is 14 nm and 12 nm in the Intel i7-6800K processor and TITAN V GPU, respectively. The execution time and power consumption are measured, and the computation throughput and energy efficiency are estimated. During estimation, the matrices are square and data are single-precision.

3.1 Computation throughput

Figure 5 shows the computation performance in the case of different matrix scales. The computation throughput is almost fixed on the FPGA system as the matrix scale is increased because the operations become computation-bound. However, it fluctuates in the SGEMM routines on the GPU with cuBLAS and the desktop machine with the MKL and OpenBLAS libraries. For example, when the matrix scale is increased from 7680 × 7680 to 20480 × 20480, the computation throughput in the FPGA-based matrix multiplier with data parallelism and task parallelism is about 240 GFLOPs and 780 GFLOPs, respectively. But the computation throughput of the SGEMM routine on the GPU is decreased from 13.47 TFLOPs to 11.28 TFLOPs. Similarly, the computation throughput is firstly increased from 586.23 GFLOPs to 631.03 GFLOPs, and then dropped to 532.43 GFLOPs in the SGEMM routine on the desktop machine with the MKL library. In Figure 5, the proposed matrix multiplier with task parallelism averagely

offers about 785 GFLOPs in computation throughput, which is about 3.2 times, 1.3 times, 1.6 times, and 6.5% in average over the matrix multiplier with data parallelism on FPGA, the SGEMM routines in the MKL and the OpenBLAS executed on the desktop machine, and the SGEMM routine in the cuBLAS performed on the TITAN V GPU, respectively. In addition, the block size of matrices A and B impacts on the computation throughput. If matrices A and B are 5120 × 4096 and 4096 × 8192, and the block sizes of A and B are 160 × 128 and 128 × 256, the computation throughput of the matrix multiplier with task parallelism is increased to about 868 GFLOPs.

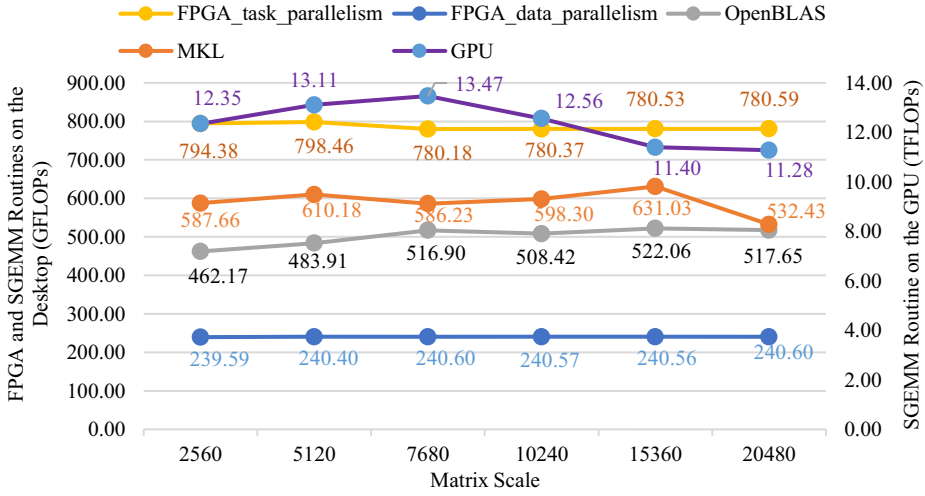


Figure 5. Computation throughput.

3.2 Energy Efficiency

To estimate the energy efficiency, the input current and voltage of the FPGA board and the desktop machine with the MKL and OpenBLAS libraries are measured every 200 ms by using a digital multimeter PC720M from the Sanwa when the FPGA system and the desktop machine are idle and active, respectively. The power consumption is calculated by multiplying the voltage and the current difference. The energy efficiency is computed by using equation 1.

$$P_{efficiency} = \frac{E_{computation\_throughput}}{V \times (I_{active} - I_{idle})} \tag{1}$$

where E is the computation throughput, V is the voltage, and I is the current. I_{active} is the current when the SGEMM routine is performed or the FPGA board is active, and I_{idle} is the system current without computation. The term $I_{active} - I_{idle}$ denotes the actual consumed current by computations in the FPGA and the desktop machine with the SGEMM routine. The I_{active} and I_{idle} are the average of the measured values. In the GPU, the energy consumption is obtained through the "nvmlDeviceGetTotalEnergyConsumption" function provided by the Nvidia management library.

Figure 6 shows the energy efficiency of the FPGA-based matrix multipliers, the SGEMM routines in the MKL and OpenBLAS executed on the desktop machine, and the SGEMM routine in the cuBLAS performed on the GPU. The energy efficiency of the proposed matrix multiplier with task parallelism ranges from 71.74 GFLOPs/W to 63.32 GFLOPs/W, and is about 66.75 GFLOPs/W in average, which is about 2.9 times, 1.2 times, 10.5 times, and 11.8 times, over the FPGA-based matrix multiplier with data parallelism, the SGEMM routine on the GPU, and the SGEMM routines on the desktop machine with the Intel MKL and the OpenBLAS, respectively, even though the fabrication technology of the FPGA (20 nm) is significantly lagged behind that of the CPU (14 nm) and the GPU (12 nm). In addition, the proposed system gains much higher energy efficiency in the case of small problem size because the computation throughput is almost fixed while the consumed current by the FPGA is increased as the problem size grows.

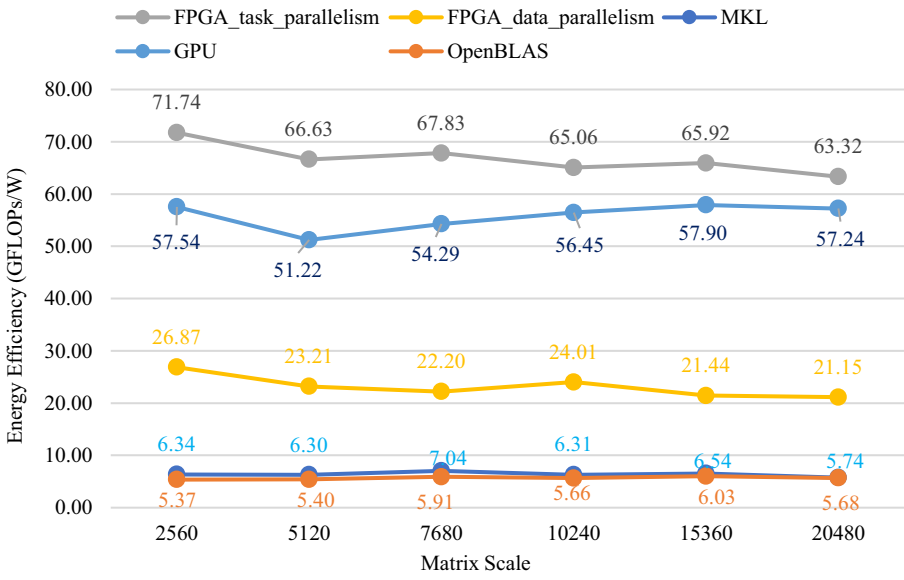


Figure 6. Energy efficiency.

4. Conclusions

Matrix multiplication is one of the basic building blocks of linear algebra, and widely applied in the HPC to solve scientific and engineering problems. Its performance significantly affects the whole system performance, especially when the problem size is large. In addition, power problem becomes more and more serious in HPC systems. In this research, an FPGA-based matrix multiplier with task parallelism is developed to improve system performance and energy efficiency for large-scale matrix multiplication, in which system is divided into different kernels in accordance to the data flow, a systolic array is adopted to carry out computations, and high-speed and high-bandwidth buffers are used to connect PEs in the systolic array and different kernels. It outperforms the FPGA-based matrix multiplier with data parallelism and the SGEMM routines in the highly optimized MKL and OpenBLAS libraries executed on a desktop machine in

computation throughput and energy efficiency. Compared with the SGEMM routine in the cuBLAS performed on the Nvidia TITAN V GPU, the proposed matrix multiplier is significantly defeated in computing performance, but it wins in energy efficiency. In future work, we will port and optimize the proposed design on other FPGA platforms with more hardware resources and multiple FPGAs to evaluate its performance, and compare the performance with the popular Xeon gold processor in HPC.

Acknowledgments

Thanks for Intel's donation of the FPGA board DE5a-NET and the related EDA tools through University Program. This work was partly supported by the Grant-in-Aid from Foundation for Computational Science (FOCUS). The performance on GPUs was obtained using the GPU cluster installed in Tokyo Woman's Christian University.

References

[1] L. Zhuo, and V. Prasanna, High-performance designs for linear algebra operations on reconfigurable hardware, *IEEE Transactions on Computers* 57(8) (2008), 1057-1072.
[2] Y. Dou, S. Vassiliadis, G. Kuzmanov, and G. Gaydadjiev, 64-bit floating-point FPGA matrix multiplication, *ACM/SIGDA 13th international symposium on Field Programmable Gate Arrays*, 2005, pp. 86-95.
[3] G. Wu, Y. Dou, and M. Wang, High performance and memory efficient implementation of matrix multiplication on FPGAs, *2010 International Conference on Field-Programmable Technology*, 2010, pp. 134-137.
[4] V. Kumar, S. Joshi, S. Patkar, and H. Narayanan, FPGA based high performance double-precision matrix multiplication, *International Journal of Parallel Programming* 38 (2010), 322–338.
[5] A. Pedram, A. Gerstlauer, and R. Geijn, Algorithm, architecture, and floating-point unit codesign of a matrix factorization accelerator, *IEEE Transactions on Computers* 63(8), 1854-1867 (2014).
[6] A. Pedram, and R. Geijn, Co-design tradeoffs for high-performance, low-power linear algebra architectures, *IEEE Transactions on Computers* 61(12) (2012), 1724-1736.
[7] H. Giefers, R. Polig, and C. Hagleitner, Analyzing the energy-efficiency of dense linear algebra kernels by power-profiling a hybrid CPU/FPGA system, *25th International Conference on Application-Specific Systems, Architectures and Processors*, 2014, pp. 92-99.
[8] J. Jiang, V. Mirian, K. Tang, P. Chow, and Z. Xing, Matrix multiplication based on scalable macro-pipelined FPGA accelerator architecture, *International Conference on Reconfigurable Computing and FPGAs*, 2009, pp. 48-53.
[9] W. Wang, K. Guo, M. Gu, Y. Ma, and Y. Wang, A universal FPGA-based floating-point matrix processor for mobile systems, *International Conference on Field-Programmable Technology*, 2014, pp. 139-146.
[10] Z. Jovanovic, and V. Milutinovic, FPGA accelerator for floating-point matrix multiplication, *IET Computers & Digital Techniques* 6(4) (2012), 249–256.
[11] R. Andrade, C. Huitzil, and R. Cumplido, Processor arrays generation for matrix algorithms used in embedded platforms implemented on FPGAs, *Microprocessors and Microsystems* 39 (2015), 576–588.
[12] E. H. D'Hollander, High-level synthesis optimization for blocked floating-point matrix multiplication, *ACM SIGARCH Computer Architecture News*, (2016) 74-79.
[13] https://www.intel.com/content/www/us/en/programmable/support/support-resources/design-examples/design-software/opencl/matrix-multiplication.html
[14] Y. Tan and T. Imamura, Performance evaluation and tuning of an OpenCL based matrix multiplier, *24th International Conference on Parallel and Distributed Processing Techniques and Applications*, 2018, pp. 107-113.
[15] Y. Tan and T. Imamura, An energy-efficient FPGA-based matrix multiplier, *24th IEEE International Conference on Electronics, Circuits and Systems*, 2017.
[16] https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=231&No=970
[17] https://www.nvidia.com/en-gb/titan/titan-v/

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200048

Application Performance of Physical System Simulations

Vladimir GETOV[a,1], Peter M. KOGGE[b] and Thomas M. CONTE[c]
[a]*University of Westminster, London, UK*
[b]*University of Notre Dame, Indiana, USA*
[c]*Georgia Institute of Technology, Atlanta, Georgia, USA*

Abstract. Various parallel computer benchmarking projects have been around since early 1990s but the adopted so far approaches for performance analysis require a significant revision in view of the recent developments of both the relevant application domains and the underlying computer technologies. This paper presents a novel performance evaluation methodology based on assessing the processing rate of two orthogonal use cases — dense and sparse physical systems — as well as the energy efficiency for both. Evaluation results with two popular codes — HPL and HPCG — validate our approach and demonstrate its use for analysis and interpretation in order to identify and confirm current technological challenges as well as to track and roadmap the future application performance of physical system simulations.

Keywords. Performance and energy efficiency analysis, Peta- and Exa-scale systems, Performance evaluation methodology

1. Introduction

Computer simulation of physical real-world phenomena emerged with the invention of electronic digital computing and has been increasingly adopted as one of the most successful modern methods for scientific discovery. Arguably, the main reasons for this success has been the rapid development of novel computer technologies that has led to the creation of powerful supercomputers, large distributed systems, high-performance computing frameworks with access to huge data sets, and high throughput communications. In addition, unique and sophisticated scientific instruments and facilities, such as giant electronic microscopes, nuclear physics accelerators, or sophisticated equipment for medical imaging are becoming integral parts of those complex computing infrastructures. Subsequently, the term 'e-science' was quickly embraced by the professional community to capture these new revolutionary methods for scientific discovery via computer simulations of physical systems [1].

Focusing on the application domain for physical system simulations, this paper explains in detail our performance evaluation methodology with the most-recent results, analysis and interpretation based on the relevant technical report [2] produced by the Applications Benchmarking (AB) International Focus Team (IFT) as part of the IEEE

[1] Corresponding Author, School of Computer Science and Engineering, University of Westminster, 115 New Cavendish Street, London W1W 6UW, United Kingdom; E-mail: V.S.Getov@westminster.ac.uk.

International Roadmap for Devices and Systems (IRDS) initiative[2]. Since 2015, IRDS is the successor of the International Technology Roadmap for Semiconductors (ITRS), which used to be provided by the Semiconductor Industry Association [3]. The mission of AB IFT is to identify key application areas, and to track and roadmap the performance of these applications for the next 15 years. Given a list of market drivers from the Systems and Architectures IFT, the AB IFT investigates and applies long-term analysis to identify the important or critical application areas for different user communities. Table 1 summarizes the ones that are under consideration at present.

Table 1. Application areas.

Application area	Description
Big data analytics	Data mining to identify nodes in a large graph that satisfy a given feature or features.
Feature recognition	Graphical dynamic moving image (movie) recognition of a class of targets (e.g. face, car). This can include neuromorphic / deep learning approaches such as deep neural networks.
Discrete event simulation	Large discrete event simulation of a discretized-time system. (e.g., large computer system simulation) Generally used to model engineered systems. Computation is integer-based.
Physical system simulation	Simulation of physical real-world phenomena. Typically, finite-element based. Examples include fluid flow, weather prediction, thermo-evolution. Computation is floating-point-based.
Optimization	Integer NP-hard optimization problems, often solved with near-optimal approximation techniques.
Graphics, augmented reality, virtual reality.	Large scale, real-time photorealistic rendering driven by physical world models. Examples include interactive gaming, augmented reality, virtual reality.

In order to track these areas, the AB IFT relies upon existing standard benchmarks where available. These benchmarks should fulfil two criteria:

- Benchmark Code Availability: There are several sets of benchmark codes available that cover each application area. However, many of these benchmarks either cover only a portion of an application area or cover more than one application area.
- Benchmark Results Availability: In order for benchmarks to be useful for projecting a trend in performance vs. time, there must be a sufficiently long history of benchmark scores. At a minimum, AB IFT believes that at least 4 years prior to the current day of results should be available.

The most important application codes for physical system simulations are typically based on finite-element algorithms — such as boundary element method, N-body problem, fast multipole method, hierarchical matrices, iterative stencil computations — while the computations constitute heavy workloads that conventionally are dominated by floating-point arithmetic. Example applications include areas such as climate modelling, plasma physics (fusion), medical imaging, fluid flow, and thermo-evolution. In addition, physical system simulation is critical to product design in the automobile and aerospace industries as well as for obtaining more accurate climate modelling and prediction. Our results confirm that:

- The area of physical system simulations requires innovative computer architectures because the data locality we have been expecting from our

[2] https://irds.ieee.org/

applications for three decades is disappearing. Novel solutions that can help addressing the "3rd Locality Wall" challenge [4] are urgently needed.

- Since the application area of physical system simulations is based predominantly on floating-point arithmetic, novel architecture proposals that address floating-point processing challenges are also expected to have substantial impact, particularly for dense system computations.
- Energy efficiency indicators need urgent improvements by at least an order of magnitude. This is equally valid for both homogeneous and heterogeneous architectures including accelerators and FPGAs.

The rest of this paper is organized as follows. Section 2 provides a review of previous work in the area. Section 3 introduces our novel approach and methodology while Section 4 presents experimental results with corresponding discussions. Section 5 outlines some of the important technological challenges. Finally, Section 6 concludes the paper.

2. Background

Taking the viewpoint of application programmers and end-users, this section outlines the major benchmarking efforts that have been part of the developments in this field over the years.

2.1. NAS Parallel Benchmarks

The NAS Parallel Benchmarks (NPB) include the descriptions of several (initially eight) "pencil and paper" algorithms [5]. Realistically, all of them are computational kernels although the authors claim that the suite includes three "simulated applications" but this claim is from the early 90s and it does not sound convincingly today. The NPB benchmarking methodology does not involve any hierarchy and each of the kernels is to be used individually for performance measurements. The codes cover only the Computational Fluid Dynamics (CFD) application domain which is of primary interest for NASA.

2.2. GENESIS Distributed-Memory Benchmarks

The GENESIS codes [6] were developed in a 3-layer hierarchy — low-level micro-benchmarks, kernels, and compact applications. This was intended to express the performance of higher-level codes via a composition of performance results produced by the kernels in the layer below. However, this proved to be a difficult task, particularly when including sufficiently broad set of computational science codes in the compact applications layer.

2.3. PARKBENCH Committee

The PARKBENCH Public International Benchmarks for Parallel Computers [7]. This was an ambitious international effort to glue together the most popular parallel benchmarks at that time — NPB, GENESIS, and several kernels including LINPACK

[8]. The PARKBENCH suite adopted the hierarchical approach from GENESIS, thus inheriting the same difficulties described above.

2.4. SPEC

All major machine vendors have participated in the development of SPEC HPG (High Performance Group), since achieving portability across all involved platforms has been an important concern in the development process [9]. The goal was to achieve both functional and performance portability. Functional portability ensured that the makefiles and run tools worked properly on all systems, and that the benchmarks ran and validated consistently. To achieve performance portability, SPEC accommodated several requests by individual participants to add small code modifications that took advantage of key features of their machines. There are many SPEC HPG benchmarking results available, but their main role is to confirm that new hardware products and platforms have been validated by the vendors.

2.5. Dwarfs — Computational Patterns

Another more recent "pencil and paper" parallel benchmark suite is the Dwarfs Mine based on the initial "Seven Dwarfs" proposal (2004) by Phillip Colella. The Dwarfs (computational patterns) are described as well-defined targets from algorithmic, software, and architecture standpoints. The number of Dwarfs (which are really kernels with some of them mapped to NPB) was then extended to 13 in the "View from Berkeley" Technical Report [10]. The report confirms "presence" of the 13 Dwarfs in 6 broad application domains — embedded computing, general-purpose computing, machine learning, graphics/games, databases and RMS (recognition/mining/synthesis) codes. Some recent studies suggest that more Dwarfs should be added for other application domains, while it is also not clear if the existing ones are sufficient for the domains described in the "View from Berkeley" Technical Report. The Dwarfs Mine description adopts a bottom-up hierarchical approach like GENESIS and then PARKBENCH. Although more systematic, it suffers from the same benchmarking hierarchy difficulties. Furthermore, the availability of benchmarking codes and results is very limited but even more importantly, the application domains are different from the ones selected by the AB IFT in the IEEE IRDS initiative.

3. Methodology

Over the years, the relevant benchmarking projects described in Section 2 above, have covered predominantly dense physical system simulations, in which high computational intensity carries over when parallel implementations are built to solve bigger problems faster. As long as emphasis was on dense problems, this approach resulted in systems with increasing computational performance and was the presumption behind the selection of the LINPACK benchmark [8] for the very popular semi-annual TOP500 rankings of supercomputers [11].

Many new applications with very high economic potential — such as big data analytics, machine learning, real-time feature recognition, recommendation systems, and even physical simulations - have been emerging in the last 10-15 years. However,

these codes typically feature irregular or dynamic solution grids and spend much more of their computation in non-floating-point operations such as address computations and comparisons, with addresses that are no longer regular or cache-friendly. The computational intensity of such programs is far less than for dense kernels, and the result is that for many real codes today, even those in traditional scientific cases, the efficiency of the floating-point units that have become the focal point of modern core architectures has dropped from the >90% to <5%. This emergence of applications with data-intensive characteristics — e.g. with execution times dominated by data access and data movement — has been recognized recently as the "3rd Locality Wall" for advances in computer architecture [4].

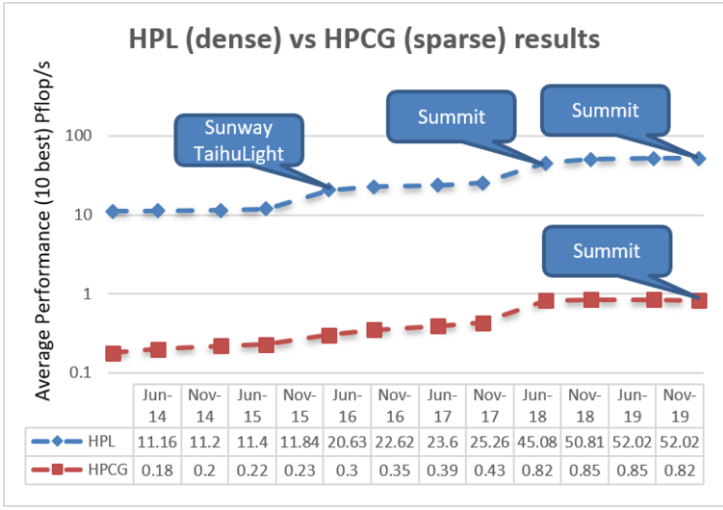
To highlight the inefficiencies described above, and to identify architectures which may be more efficient, a new evaluation code was introduced in 2014 called HPCG[3] (High Performance Conjugate Gradient) benchmark [12]. HPCG also solves Ax=b problems, but where A is a very sparse matrix — normally, with 27 non-zeros in rows that may be millions of elements in width. On current systems, floating point efficiency mirrors that seen in full scientific codes. For example, one of the fastest supercomputers in the world in terms of dense linear algebra is the Chinese TaihuLight, but that same supercomputer can achieve only 0.4% of its peak floating-point capability on the sparse HPCG benchmark. Detailed analysis lead to the conclusion that HPCG performance in terms of useful floating-point operations is dominated by memory bandwidth to the point that the number of cores and their floating-point capabilities are irrelevant [13]. There are of course application codes with highly irregular and latency-bound memory access that deliver significantly lower performance, but they are uncommon. While HPCG does not represent the worst-case scenario, it has been widely accepted as a typical performance yardstick for memory-bound applications.

Therefore, our selected benchmark codes that cover the "Physical System Simulation" application area of interest are the High-Performance LINPACK (HPL) and the HPCG. Both are very popular codes with very good regularity of results since June 2014. Another very important reason for selecting HPL and HPCG is that they represent different types of real-world phenomena — the HPL models dense physical systems while the HPCG models sparse physical systems. Therefore, the available benchmarking results provide excellent opportunities for comparisons and interpretation, as well as lay out a relatively well-balanced overall picture of the whole domain for physical system simulation applications. Our approach is to explore a 3-dimensional space — dense systems performance, sparse systems performance, and energy efficiency for both cases.

4. Performance Results

With HPL as the representative of dense system performance and HPCG as the representative for sparse systems, there are readily available performance and energy results published twice per year (June and November) with rankings of up to 500 systems for those two benchmarks since June 2014.

[3] http://www.hpcg-benchmark.org/



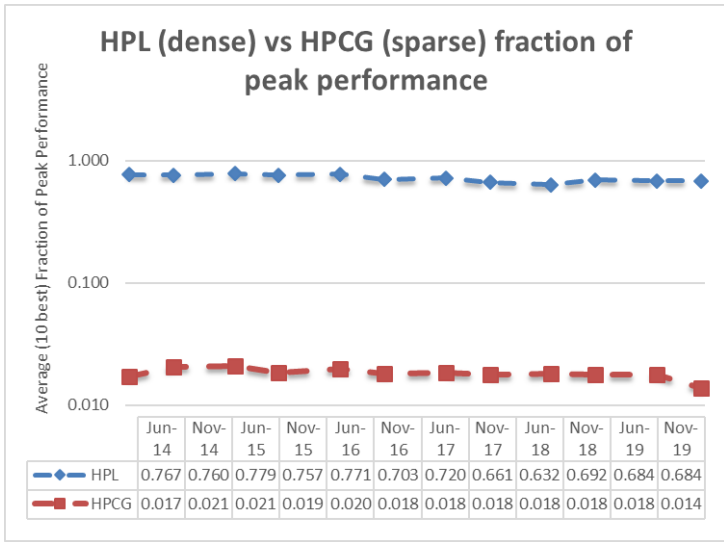
	Jun-14	Nov-14	Jun-15	Nov-15	Jun-16	Nov-16	Jun-17	Nov-17	Jun-18	Nov-18	Jun-19	Nov-19
HPL	11.16	11.2	11.4	11.84	20.63	22.62	23.6	25.26	45.08	50.81	52.02	52.02
HPCG	0.18	0.2	0.22	0.23	0.3	0.35	0.39	0.43	0.82	0.85	0.85	0.82

Figure 1. Average performance of HPL (dense systems) vs. HPCG (sparse systems).

We have further decided to use the average of the top 10 performance and energy results for each of these two benchmarks. This latter choice could be a point for further discussion and optimization of the benchmarking approach for this application domain. We have selected the 10 best only (rather than a larger number) because of the very limited HPCG results in the early years of publicly available HPCG measurements.

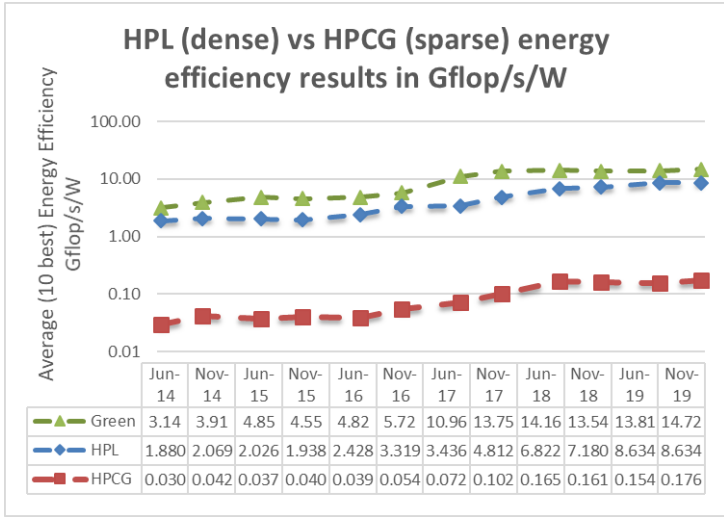
Figure 1 shows a significant performance gap of nearly 2 orders of magnitude between HPL and HPCG results in the last several years. The increase of the average HPL performance since June 2016 is because of the introduction of the Chinese Sunway TaihuLight system. The most recent increase of both HPL and HPCG performance is visible since June 2018 after the installation of the Summit supercomputer at ORNL. An optimistic expectation here would be to observe that the gap keeps closing and then assess the rate of this progress. Unfortunately, we do not have any evidence that the observed performance gap is in fact closing to any degree. Thus, we can draw the conclusion that one of the main challenges ahead will be to significantly increase sparse systems performance with any future computing systems designed for this application domain. While it is clear that reaching Eflop/s performance with HPL will happen soon, it is equally clear that this achievement will leave this significant gap between dense and sparse system performance unchanged.

Figure 2 complements the above analysis by showing a similar gap of approximately 2 orders of magnitude for the fraction of peak performance results between HPL and HPCG. This provides clear evidence of something we have known for years — our production codes, which usually implement sparse system simulations, are unable to deliver more than a few percent of the peak system performance that HPL results would seem to promise. The figure shows that this gap has not been reducing, and further points out the need to address sparse system performance in the next generation of computer architectures designed for this application domain.



	Jun-14	Nov-14	Jun-15	Nov-15	Jun-16	Nov-16	Jun-17	Nov-17	Jun-18	Nov-18	Jun-19	Nov-19
HPL	0.767	0.760	0.779	0.757	0.771	0.703	0.720	0.661	0.632	0.692	0.684	0.684
HPCG	0.017	0.021	0.021	0.019	0.020	0.018	0.018	0.018	0.018	0.018	0.018	0.014

Figure 2. Fraction of peak performance for HPL (dense systems) vs. HPCG (sparse systems).



	Jun-14	Nov-14	Jun-15	Nov-15	Jun-16	Nov-16	Jun-17	Nov-17	Jun-18	Nov-18	Jun-19	Nov-19
Green	3.14	3.91	4.85	4.55	4.82	5.72	10.96	13.75	14.16	13.54	13.81	14.72
HPL	1.880	2.069	2.026	1.938	2.428	3.319	3.436	4.812	6.822	7.180	8.634	8.634
HPCG	0.030	0.042	0.037	0.040	0.039	0.054	0.072	0.102	0.165	0.161	0.154	0.176

Figure 3. HPL (dense systems) vs. HPCG (sparse systems) vs. the most energy-efficient supercomputers on the Green 500 list.

The energy efficiency dimension of our evaluation is depicted in Figure 3. The current supercomputing designs appear to be able to scale up to 200 Pflop/s while remaining within the recommended 20 MW system power consumption envelope. An optimistic estimate based on this would require five times improvements in energy efficiency, and seven times improvements in the HPL performance currently delivered by the Summit supercomputer. However, such improvements are not realistic, since the best energy efficiency results and rankings are different from the HPL ranking (see comments above about the top 10 ranked results). Therefore, a more realistic projection

based on the current (end of 2019) Summit results is that one needs ten times energy efficiency improvement and ten times higher HPL performance to reach the Eflop/s barrier. Unfortunately, this would only fulfil the desired performance and energy efficiency for the computation of dense physical systems such as the HPL benchmark.

Similar performance versus energy efficiency analysis and projections for sparse systems based on the HPCG results look much more pessimistic. Here the two orders of magnitude lower performance delivered for sparse systems by the current supercomputing architectures strongly impact the energy efficiency.

5. Technological Challenges

Following the results and the discussion presented in the previous section, the main technological challenges that could help drive the future developments and improvements in the field of physical system simulation a summarised briefly below.

5.1. Reduced Data Movement

Since the late 1980s, reducing significantly the data movement has been one of the most important challenges towards achieving higher computer performance. Achieving higher bandwidth and lower latency for accessing and moving data — both locally (memory systems) and remotely (interconnection networks) — are key challenges towards building supercomputers at Eflop/s level and beyond. Breakthrough architecture solutions addressing those challenges could potentially enable up to two orders of magnitude higher performance particularly for sparse physical system simulations. More specifically, forthcoming designs of High Bandwidth Memory (HBM) such as HBM3+ and HBM4 expected to be released between 2022 and 2024, are likely to change substantially the application performance landscape for future supercomputers.

5.2. Efficient Floating-Point Arithmetic

Established in 1985, the IEEE 754 Standard for Floating-Point Arithmetic was renewed again in July 2019 [14]. However, the level of interest in this standard has been declining following critical comments about various important aspects of IEEE 754 including wasted cycles, energy inefficiencies, and accuracy. Unfortunately, the path forward is unclear at present and may involve keeping this standard as an option at least for backward compatibility while developing and implementing novel and more efficient solutions. Several efforts to address these problems follow two main approaches.

- Analysis of specific algorithms and re-writing of existing codes in order to improve the performance by using lower or mixed floating-point precision without compromising accuracy. This approach has been shown to work well but only for specific algorithms/codes, and with significant dedicated efforts for each case [15].
- More radical approaches proposing new solutions have been under development including the Posit Arithmetic proposal [16]. This work

introduces a new data type — posit — as a replacement for the traditional floating-point data type because of its advantages. For example, posits guarantee higher accuracy and bitwise identical results across different systems which have been recognized as the main weaknesses of the IEEE 754 Standard. In addition, they enable more economical design with high efficiency which lowers the cost and the consumed power while providing higher bandwidth and lower latency for memory access.

5.3. Low Consumed Power

During the last two decades, further developments of computer architecture and microprocessor hardware have been hitting the so-called "Energy Wall" because of their excessive demands for more energy. Subsequently, we have been ushering in a new era with electric power and temperature as the primary concerns for scalable computing. This is a very difficult and complex problem which requires revolutionary disruptive methods with a stronger integration among hardware features, system software and applications. Equally important are the capabilities for fine-grained spatial and temporal instrumentation, measurement and dynamic optimization, in order to facilitate energy-efficient computing across all layers of current and future computer systems. Moreover, the interplay between power, temperature and performance add another layer of complexity to this already difficult group of challenges.

Existing approaches for energy efficient computing rely heavily on power efficient hardware in isolation which is far from acceptable for the emerging challenges. Furthermore, hardware techniques, like dynamic voltage and frequency scaling, are often limited by their granularity (very coarse power management) or by their scope (a very limited system view). More specifically, recent developments of multi-core processors recognize energy monitoring and tuning as one of the main challenges towards achieving higher performance, given the growing power and temperature constraints. To address these challenges, one needs both suitable energy abstraction and corresponding instrumentation which are amongst the core topics of ongoing research and development work. Another approach is the use of application-specific accelerators to improve the application performance, while reducing the total consumed power which in turn minimises the overall thermal energy dissipation.

6. Conclusions

The application area of physical system simulations urgently needs novel and innovative architectures that provide solutions resolving the 3rd Locality Wall challenge. This includes both novel memory systems and interconnection networks offering much higher bandwidth and lower latency. Energy efficiency indicators also need urgent improvements by at least an order of magnitude. This requirement is equally valid for both homogeneous and heterogeneous architectures (including accelerators and FPGAs) that need further comparisons and analysis. Since the application area of physical system simulations is based predominantly on floating-point arithmetic, novel architecture proposals that address floating-point processing challenges can also be expected to have substantial impact, particularly for dense system computations.

7. Acknowledgements

We gratefully acknowledge the publicly available HPL and HPCG performance and power consumption results which enabled the presentation and analysis in Section 4 above. In addition, we would also like to thank Geoffrey Burr (IBM), Satoshi Matsuoka (RIKEN) and Takeshi Iwashita (Hokkaido University) for their useful comments as well as Linda Wilson (IEEE IRDS) for all her help and support.

References

[1] V. Getov, e-Science: The Added Value for Modern Discovery, *Computer* **41**(8), 30–31, IEEE Computer Society, 2008.

[2] G. Burr, T. Conte, V. Getov, P.M. Kogge, M. Levy, S. Matsuoka, D. Sunwoo, J. Torrellas, *Application Benchmarking*, The IEEE International Roadmap for Devices and Systems, IEEE, 2019.

[3] F.D. Wright, T.M. Conte, Standards: Roadmapping Computer Technology Trends Enlightens Industry, *Computer* **51**, (6), 100–103, IEEE Computer Society, 2018.

[4] P.M. Kogge, Data Intensive Computing, the 3rd Wall, and the Need for Innovation in Architecture, *Argonne Training Program on Extreme-Scale Computing*, August 2017, slides: http://extremecomputingtraining.anl.gov/files/2017/08/ATPESC_2017_Dinner_Talk_6_8-4_Kogge-Data_Intensive_Computing.pdf, video: https://youtu.be/ut9sBnwF6Kw.

[5] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, S.K. Weeratunga. The NAS Parallel Benchmarks, *Int. J. of Supercomputer Applications* **5**(3), 63–73, 1991, http://www.davidhbailey.com/dhbpapers/benijsa.pdf.

[6] C. Addison, V. Getov, A. Hey, R. Hockney, I. Wolton. The GENESIS Distributed-Memory Benchmarks. In: J. Dongarra and W. Gentzsch (Eds.) *Computer Benchmarks, Advances in Parallel Computing* **8**, 257–271, Elsevier Science Publishers, 1993.

[7] D.H. Bailey, M. Berry, J. Dongarra, V. Getov, T. Haupt, T. Hey, R.W. Hockney, D. Walker, "PARKBENCH Report-1: Public International Benchmarks for Parallel Computers", TR UT-CS-93-213, *Scientific Programming* **3**(2), 101–146, 1994, http://www.davidhbailey.com/dhbpapers/parkbench.pdf.

[8] J. Dongarra, P. Luszczek, A. Petitet, The LINPACK Benchmark: Past Present and Future, *Concurrency and Computation: Practice and Experience* **15**(9), 803–820, 2003.

[9] R. Henschel, S. Wienke, B. Wang, S. Chandrasekaran, G. Juckeland, J. Li, V.G. Vergara Larrea, Using the SPEC HPG Benchmarks for Better Analysis and Evaluation of Current and Future HPC Systems, Tutorial at ISC18, Frankfurt, Germany, June 2018, slides: http://pages.iu.edu/~henschel/ISC18/SPECBenchmarksTutorial-Presentation_ISC2018.pdf.

[10] K. Asanovic, R. Bodik, B.C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S. W. Williams, K. A. Yelick, The Landscape of Parallel Computing Research: A View from Berkeley, *TR UCB/EECS-2006-183*, University of California, Berkeley, Dec. 2006, http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf.

[11] E. Strohmaier, H.W. Meuer, J. Dongarra, H.D. Simon, The TOP500 List and Progress in High-Performance Computing, *Computer* **48**(11), 42–49, IEEE Computer Society, 2015.

[12] J. Dongarra, M.A. Heroux, P. Luszczek, High-Performance Conjugate-Gradient Benchmark: A New Metric for Ranking High Performance Computing Systems, *Int. J. of High-Performance Computing Applications* **30**(1), 3–10, 2016.

[13] V. Marjanović, J. Gracia, C. W. Glass, Performance modeling of the HPCG benchmark, *Proc. Int. Workshop on Performance Modeling, Benchmarking and Simulation of High-Performance Computer Systems*, pp. 172–192, Springer, 2014.

[14] *IEEE 754-2019 – IEEE Standard for Floating-Point Arithmetic*, IEEE xPlore, July 2019.

[15] A. Haidar, P. Wu, S. Tomov, J. Dongarra, Investigating Half Precision Arithmetic to Accelerate Dense Linear System Solvers, *Proc. ScalA Workshop at SC'17*, pp. 1–8, ACM, 2017.

[16] J.L. Gustafson, I.T. Yonemoto, Beating Floating Point at its Own Game: Posit Arithmetic, *Supercomputing Frontiers and Innovations* **4**(2), 71–86, 2017.

Parallel Methods

This page intentionally left blank

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200050

A Hybrid MPI+Threads Approach to Particle Group Finding Using Union-Find

James S. WILLIS [a,1], Matthieu SCHALLER [a,b], Pedro GONNET [c] and John C. HELLY [a]

[a] *Institute for Computational Cosmology (ICC), Department of Physics, Durham University, Durham DH1 3LE, UK*
[b] *Leiden Observatory, PO Box 9513, NL-2300 RA Leiden, The Netherlands*
[c] *Google AI Switzerland GmbH, 8002 Zürich, Switzerland*

Abstract. The Friends-of-Friends (FoF) algorithm is a standard technique used in cosmological *N*-body simulations to identify structures. Its goal is to find clusters of particles (called groups) that are separated by at most a cut-off radius. *N*-body simulations typically use most of the memory present on a node, leaving very little free for a FoF algorithm to run on-the-fly. We propose a new method that utilises the common Union-Find data structure and a hybrid MPI+threads approach. The algorithm can also be expressed elegantly in a task-based formalism if such a framework is used in the rest of the application. We have implemented our algorithm in the open-source cosmological code, SWIFT. Our implementation displays excellent strong- and weak-scaling behaviour on realistic problems and compares favourably (speed-up of 18x) over other methods commonly used in the *N*-body community.

Keywords. Friends-of-Friends; Union-Find; MPI; Threads; Efficiency

1. Introduction

Over the last four decades cosmological simulations have been the main tool used by physicists to confront their theoretical predictions to observations. By creating more-and-more realistic universes they have been able to revolutionise our understanding of the cosmos and establish the current cosmological model. These simulations typically involve the evolution of large numbers of particles or resolution elements under the laws of gravity and hydrodynamics. Given the large volumes simulated and the ever-growing need for more details, these simulations are often at the forefront of research in HPC and require ever-increasing computing capabilities. For instance, the current record holder, the *Euclid flagship simulation* [1], evolved 8×10^{12} particles from the Big Bang to the present day and generated peta-bytes of data.

Putting aside the question of running such simulations, analysing these large volumes of data poses huge computational challenges as even the most basic operations require sizeable facilities to simply host the data in memory. One of the most-widely used post-processing tool for such simulations is the Friends-of-Friends (FoF) method [2],

[1] Corresponding Author; E-mail: james_willis@hotmail.co.uk.

which is designed to identify *groups* of particles that are within a certain *linking-length*, l_x, of each other. If the linking-length is chosen to be small enough then the method will identify groups that correspond to structures of particles that have formed due to gravity and hence capture information about the evolution of the Universe. More specifically two particles are in the same *group* if they are at a distance smaller than l_x of each other. Particles can be linked to multiple other particles and all particles linked in this way are in the same group [2]. The size of a group is later defined as the number of particles that are linked to each other by this criterion. Particles without any neighbours within l_x form a group of size one. Since producing catalogs of particle groups in post-processing can be prohibitively expensive, in terms of i/o at least, it is common practice to apply the FoF method on-the-fly at fixed time intervals over the course of the N-body simulation. This also allows the production of FoF outputs at a higher frequency. Over the years many dedicated stand-alone FoF packages have been implemented, recent examples used in production runs include [3–5]. Nevertheless, the challenge of efficiently distributing the method over large numbers of nodes on-the-fly, i.e. *whilst reusing the pre-existing data structures put in place for the N-body solver*, still remains.

In this paper we present a FoF implementation that exploits the hybrid shared/distributed parallelism built into the SWIFT cosmological code [3] [6, 7] to achieve excellent efficiency whilst also being able to run at regular intervals over the course of large cosmological simulations.

2. FoF using the Union-Find algorithm

FoF is related to the more general problem of Euclidean minimum spanning trees (here in 3 dimensions), which is a very well-studied problem (e.g. [8, 9]) with algorithms that are near-linear in the worst case, but differs crucially in that:

- The maximum Euclidean distance considered is limited, thus limiting the range of neighbours for each node, and
- We are not interested in the exact structure of the resulting minimum spanning tree (or set of trees), but only in which nodes belong to the same trees.

The problem is therefore equivalent to the *disjoint-set union (or* union-find) problem [10, 11], and the FoF method we have implemented is based on the approaches used for its solution in shared/distributed-memory parallel settings [12–15].

A disjoint-set data structure is the basis for the algorithm, which maintains a collection of dynamic non-overlapping sets consisting of N distinct elements. Each set is identified by a representative element (the *root*). It is widely used in the calculation of minimum spanning trees in graphs and the computation of connected components.

The Union-Find algorithm is designed around two operations: *Union*, which merges a pair of sets and *Find*, which identifies the set a given element resides in. The data structure is typically implemented using a *forest*, where each *tree* represents a connected set and the root of each tree identifies the set. Initially each set contains one element which

[2] More mathematically, the problem can be expressed as determining the connected components of a graph $\mathcal{G}$, based on a set of points P, where $\mathcal{G}$ is defined as $\mathcal{G} = (P, E)$ with the set $E = \{\{u, v\} : dist(u, v) \leqslant l_x\}$ and $u, v \in P$.

[3] See also www.swiftsim.com.

is the sole member of its set and its set's representative. Two sets containing elements that are within the linking-length distance, l_x, are merged using the *Union* operation[4].

There are several standard ways to optimise the Union-Find algorithm. The *Union* operation for example, can be implemented using *Union-by-size* which links smaller sets to larger ones and *Union-by-rank* that links sets with shorter trees to sets which have taller trees. However, we will use *Union-by-root* and make the larger root always point to the smaller root, where the initial root of each set is assigned by its offset in the array. This allows us to bypass the issues with parallelism (see below) reported by [15].

Another common optimisation technique is *path compression*. Each tree vertex traversed in a Find operation is set to point to the root of the set. This means that subsequent Find operations are quicker as most vertices will point directly to the root; reducing the rank of each particle and hence lowering the (theoretical) loss of performance using a Union-by-root approach over a Union-by-rank.

The Union-Find algorithm has been extensively parallelised in the literature for both shared and distributed memory machines: [12–14]. The novelty of our paper is the introduction of a hybrid shared/distributed memory algorithm that uses a task-based framework, which can be run on-the-fly within our *N*-body code that imposes a spatial decomposition.

3. Implementation in the SWIFT code

3.1. Serial implementation

In practice the Union-Find data structure is implemented using an array of length, N, where N is the total number of particles and each element represents a particle. The array is initialised so that each particle exists in its own group, i.e each element is set to the offset of the particle in the array. A neighbour search is then performed over the particles using the linking-length, l_x, as the search criterion. The Find operation is used on all particles that are neighbours to return their roots. Two groups are then merged using the Union operation, where the smaller of the two roots is used as the group label henceforth. For example:

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    if(i == j) continue; // Avoid self
    r = particle_dist(parts[i], parts[j]);
    if (r < l_x)
      // Find operation
      int root_i = fof_find(i, group_index);
      int root_j = fof_find(j, group_index);
      // Union operation
      if(root_i < root_j) group_index[root_j] = root_i;
      else group_index[root_i] = root_j;
```

Code 1: Union-Find with a simple iteration over neighbours.

[4]In the context of the FoF method we use the following terminology: a *set* is referred to as a *group* and an *element* is an individual *particle*.

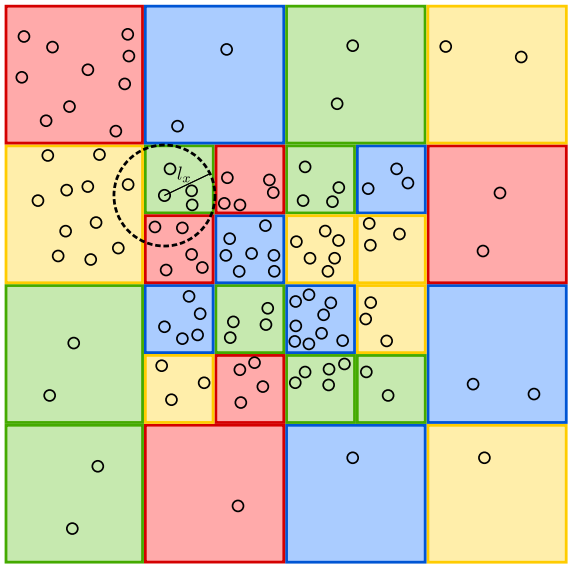


Figure 1. FoF Union-Find using task-based parallelism. Each coloured cell represents a single task. Self tasks are created for each cell and pair tasks are created between cells that lie within the cut-off radius, l_x, of each other. A self task performs a FoF search on particles in a single cell whereas a pair task carries out a FoF search between particles in neighbouring cells. Tasks are placed into a queue. A group of threads pick and execute tasks from the queue concurrently until there are none remaining.

where `parts` is the particle array and `group_index` is the array that represents the Union-Find data structure.

As in the case of minimum spanning tree problems, we make use of the octree (quadtree in 2D) present in SWIFT to significantly reduce the cost of the neighbour search, by only recursing on pairs of cells that are within the requested cut-off radius, l_x, of each other. We note, however, that the best performance is achieved when the size of the tree nodes matches the linking-length (see the technique of [4] or [5]), but that tailoring the octree node sizes would hinder the performance of the rest of the SWIFT code and is hence not an option. Once the tree has been setup, the problem becomes almost embarrassingly parallel and we split the workload evenly either between: (a) a group of threads, or equivalently (b) a set of tasks (see Fig. 1). We implement the latter in SWIFT using a variant of the QUICKSCHED tasking library [16].

3.2. Shared memory parallelism

In order to parallelise the algorithm a subtle issue needs to be taken care of, i.e. each thread must have a consistent view of the tree data. For example, consider two roots: r_i and r_j, we need to ensure that one thread does not find $r_i < r_j$ whilst another concludes $r_i > r_j$. One possibility would be to use locks when writing to the Union-Find data structure (`group_index`), but this would hinder scalability as more and more threads try to access the list. We instead solve this problem by checking that the value of r_i has not changed between being read and being found to be lower than r_j. If r_i has changed between these events the process is repeated until the value of r_i remains constant.

We implement the Union operation in a thread-safe manner by using the *Compare And Swap* (CAS) atomic, proposed by [12]:

```
1  int atomic_update_root(volatile size_t *address, size_t y) {
2
3    size_t *size_t_ptr = (size_t *)address;
4
5    size_t old_val = *address;
6    size_t test_val = old_val;
7    size_t new_val = y;
8
9    old_val = atomic_cas(size_t_ptr, test_val, new_val);
10
11   return (test_val == old_val);
12 }
13
14 void fof_union(size_t i, size_t j, size_t *group_index) {
15   int result = 0;
16   // Loop until the root can be set to a new value.
17   do {
18     size_t root_i = fof_find(i, group_index);
19     size_t root_j = fof_find(j, group_index);
20
21     if(root_j < root_i)
22       result = atomic_update_root(&group_index[root_i], root_j);
23     else
24       result = atomic_update_root(&group_index[root_j], root_i);
25
26   } while (!result);
27 }
```

Code 2: Using a CAS atomic operation to perform the *Union* of two groups in a thread-safe manner.

This ensures any update to `group_index` is lock-free, and hence avoids any performance penalties introduced by locks. A weakness of this method, however, is that the CAS operation can only update a single variable at a time[5]. Therefore, if a (formally more efficient) Union-by-size or Union-by-rank version of the algorithm were to be used, it would require a lock instead of an atomic to avoid data races. One solution to this problem is to adopt the approach by [15], where the Union is instead randomised. It avoids having to update two variables per Union as the size or rank of a group is not stored in addition to the root.

We also tested a version of our parallel algorithm using the randomisation technique proposed by [15]. This implementation showed similar times to solution compared to our basic approach. This is due to the fact that we only use the root of a group to perform a Union operation, and hence do not suffer from the weakness of the Anderson & Woll implementation [12]. Our specific workloads, where the *rank* of the elements added in the *Union* operations are typically small, are another reason why we did not see a noticeable increase in performance. For these reasons we chose to use our simpler solution and stick to the *Union-by-root* method.

[5]There has been an attempt by [17] to implement a multi-variable CAS operation, but their results show that in practice the performance of this approach is not superior to traditional locking techniques.

3.3. Distributed memory parallelism

For larger simulations, particles are distributed across multiple nodes (see Fig. 2). To address the problem of groups spanning multiple nodes, we follow the strategy outlined by [14] and improve upon it to handle the case where the number of groups is much larger than 10^2.

We first perform a multi-threaded local Union-Find on each node, as described in Section 3.2, followed by assigning unique group IDs across all nodes. This is done by computing an offset based upon the MPI rank of the node. Each rank, p, computes a sum of the total number of particles contained on every MPI rank lower than itself, $\sum_{i=0}^{i<p} N_i$, where N_i is the total number of particles present on rank i. The sum is then used to offset all group IDs on the local node. In practice this is done using `MPI_Scan`:

```
long long num_parts_cumulative;
long long num_parts_local = num_parts;
MPI_Scan(&num_parts_local, &num_parts_cumulative, 1, MPI_LONG_LONG,
         MPI_SUM, MPI_COMM_WORLD);
size_t node_offset = num_parts_cumulative - num_parts_local;
```

Code 3: Computing the node offset with `MPI_Scan`.

Next, we identify links between groups that span at least two node domains, only communicating information for groups that are within the linking-length, l_x, of the domain boundaries. This greatly reduces the amount of data replication. The final step performs a global gather communication (`MPI_Allgatherv`) on the list of group links so that every node has access to the global list of group links (`global_group_links`). Each node then applies the Union-Find to `global_group_links`, only updating the roots of groups which are local to them. This ensures that all spanning groups are merged and each node agrees upon group ownership.

In order to apply the Union-Find on the global list we map each group ID to a number between 0 and the total number of group IDs that span node domains. The same group ID may appear multiple times in the list, therefore we need to search for the first occurrence of it and use the index as input to the Find operation. This ensures that the result of the Find operation is correct, as the group ID could have previously been updated from a group merger earlier in the list. See Fig. 3.

Naively one may think that each rank need only run the Union-Find on the group links that it shares with its neighbouring ranks. However, Fig. 2 shows a particle distribution that forms a group on rank 0 that is indirectly linked to the group on rank 2 via the groups on ranks 3 and 4. This group linkage will be overlooked if each rank only searches for links with its direct neighbours.

If we use the same Union strategy as the local FoF, the distribution of roots of spanning groups will be skewed towards the lower MPI ranks. This can lead to a load imbalance between nodes when assigning new local roots during Step 4. To address this problem we use Union-by-size when merging groups across MPI domains. This creates an even work load between ranks as Union-by-size will assign roots more arbitrarily and will only be based upon the domain decomposition.

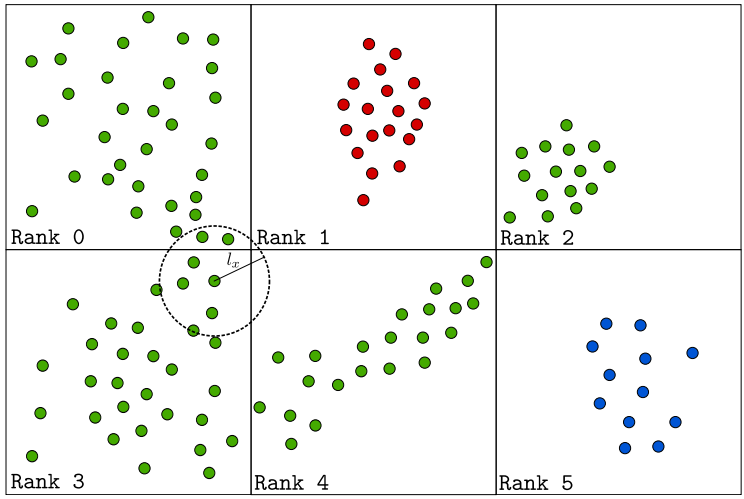


Figure 2. Distributed Union-Find over MPI. Particles are distributed across each MPI rank and the following steps are performed: 1) a local FoF is performed on each MPI rank; 2) relabel group IDs so that they are globally unique; 3) identify links between groups that span two MPI domains; 4) merge distributed groups and agree on ownership. The figure also illustrates an edge case that can occur. The group on rank 0 is indirectly linked to the group on rank 2 via the groups on ranks 3 and 4. If we were to only merge groups between MPI ranks that are direct neighbours in step 4), we would fail to take into account this subtlety and miss the indirect group links.

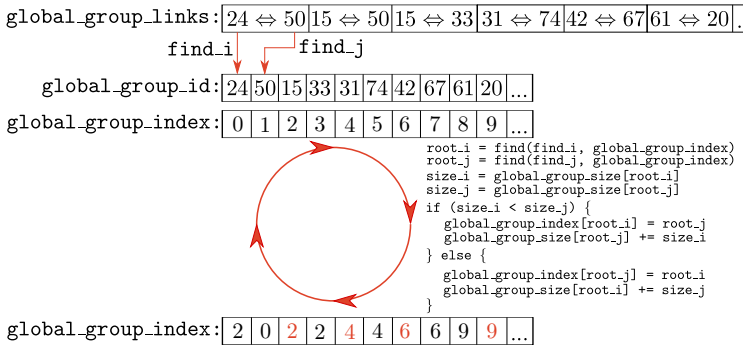


Figure 3. Step 4 in the distributed Union-Find method. Distributed groups are merged and each MPI rank agrees on group ownership. The `global_group_links` array stores all group links that span an MPI domain, each unique ID in the list is unpacked into `global_group_id` and mapped to a number between 0 and the total number of unique group IDs in the list (`global_group_index`). Find is applied to each pair of links in the list, where the group offset (`find_i` & `find_j`) into `global_group_id` is used as input. This ensures that Find returns the correct group ID in the case where it has been updated in an earlier group merger (Union). The pair of groups are then merged using the Union operation and `global_group_index` is updated.

3.4. Implementation details

3.4.1. Hash table

Performing the last step of the distributed FoF algorithm can become quite expensive, as the length of `global_group_links` scales with the node count. This is because searching for the index of a group ID into the list roughly takes $\mathcal{O}(N^2)$ operations. A

hash table on the other hand has constant look-up times, $\mathcal{O}(1)$. Therefore we construct a hash table of group IDs in the list and store their index into `global_group_links`.

We also make use of a hash table when calculating the group sizes in the local FoF. To find the group sizes in serial we loop through the `group_index` array and increment `group_size` indexed by the root of the group that each particle is in:

```
1    for (int i = 0; i < N; i++)
2      group_size[fof_find(i, group_index)]++;
```

Code 4: Group size calculation in serial.

In parallel we divide N by the number of threads and have each thread work on a section of `group_index`. We avoid race conditions between threads by protecting access to `group_size`. To do this we use a hash table to store the group sizes and root of each group. Once we have looped over `group_index`, we pull out each element of the hash table and write the intermediate group size to the global `group_size` array using an atomic addition (for instance GNU C's `__sync_fetch_and_add`).

3.4.2. Early elimination of small groups

The majority of groups in cosmological simulations are of lone particles. We were able to take advantage of this fact to lower the memory footprint significantly when calculating group sizes. When constructing the hash table only groups of size $\geqslant 2$ were stored. We achieved this by initialising each element of the `group_size` array to 1, which allowed us to exclude root particles in the hash table as their contribution to the group size was already accounted for.

3.4.3. Path compression optimisation

The Find operation is a tree traversal that retrieves the root of a group for a given particle. Hence, the execution time is dominated by the depth of the tree at each particle. To amortise the cost of this operation we have implemented path compression. But instead of compressing trees of all depths, we found it was quicker to only compress trees with a depth of at least 2.

4. Results

To test the performance of our FoF implementation we ran a number of different benchmarks. We measured the strong- and weak-scaling performance as well as the speed-up over another FoF application. All results were obtained on the COSMA-7 DiRAC 2.5x "Memory Intensive" System, located at the University of Durham [6]. The results are based

[6]The system consists of 452 nodes of 2 Intel Xeon Gold 5120 CPUs running at 2.2GHz (14 physical cores with AVX512 capability) with 512 GBytes of RAM. The nodes are connected using Mellanox EDR Infiniband in a 2:1 blocking configuration. The strong scaling results were obtained by running on the MAD02 machine at Durham with Turbo Boost disabled for the purposes of obtaining accurate measurements. It is a quad socket system each with an Intel Xeon Platinum 8180 CPU running at 2.5GHz (28 physical cores with AVX512 capability) with 1.5 TBytes of RAM. See https://dirac.ac.uk/resources/#MemoryIntensive for more details on each system.

on version 0.8.2 of SWIFT (`git` revision `f05bd301`), which implements the algorithm described in Section 3.

4.1. Measurement methodology

To get a realistic workload, all benchmarks were carried out using particle data from the flagship EAGLE simulations [18] at late times (redshift $z = 0.1$). The input data contains 4.25×10^8 particles split into $\sim 2 \times 10^5$ groups of length > 20. The workload is representative of an actual production run of SWIFT and nicely fits within a single node's memory. To create a weak-scaling test, we replicate the simulation volume periodically N times along each axis, creating problem sizes that are N^3 larger than the original volume.

We used the Intel compiler and MPI library v.18.0.2[7] as well as the GNU compiler v.9.1.0[8]. To obtain precise execution times we used the `RDTSC` cycle counter and converted the cycle counts to seconds using the clock-speed of the CPU. Each data point is the average time of 3 independent runs and the standard deviation is used to measure the uncertainty. For the weak-scaling tests, we use 4 MPI ranks per node (2 per NUMA region) and use the MPI version of the code even for the single-node data point in order to have the same MPI-related overheads throughout the test. The strong-scaling test does not use MPI and hence probes the efficiency of the shared memory algorithm.

4.2. Strong- and weak-scaling results

The strong scaling results are shown in the left hand panel of Fig. 4. We stress that these results were obtained starting from one core and keeping the problem size constant. Turbo Boost was also disabled on the node for the purposes of obtaining accurate measurements. We display very good strong scaling and maintain a high parallel efficiency, achieving 77% on 112 cores. Only dropping in efficiency when hyper-threads are used, but this can be explained by resource contention between competing threads. This is a result of our shared memory strategy: effective load balancing between threads using an octree and task-based parallelism; and a lock-less implementation of the parallel Union-Find algorithm.

The right-hand panel displays the weak-scaling performance, where we achieve good scaling up to 10,206 cores despite the overhead costs of MPI communication. The last data point corresponds to a simulation with 3×10^{11} particles. The jump from ½ a node to 4 nodes is a result of the MPI communication being performed over the network, as opposed to on a single node. Additionally, since that data point only uses half the available cores on the node, a better memory throughput is achieved and the cores are running at a slightly higher clock speed (2.9 vs. 2.6 GHz) thanks to Turbo Boost. We hence only consider the results starting from the next data point (4 nodes) where all the cores are busy on each node. From that point onwards, the gradual increase in runtime is a result of the network, as it has a greater effect at higher node counts and becomes the limiting factor. The loss in performance running on 10,206 with ICC is 34%. Starting from the second data point (where the nodes are now using all cores and do not suffer from the caveats mentioned above), we obtain a significant improvement only losing 15% going from 4 ($= 2^3$ the original problem size) to 364½ nodes ($= 9^3$ the original problem size).

[7] with the flags `-O3 -xCORE-AVX512`.
[8] with the flags `-O3 -ffast-math -march=skylake-avx512 -mavx512dq`.

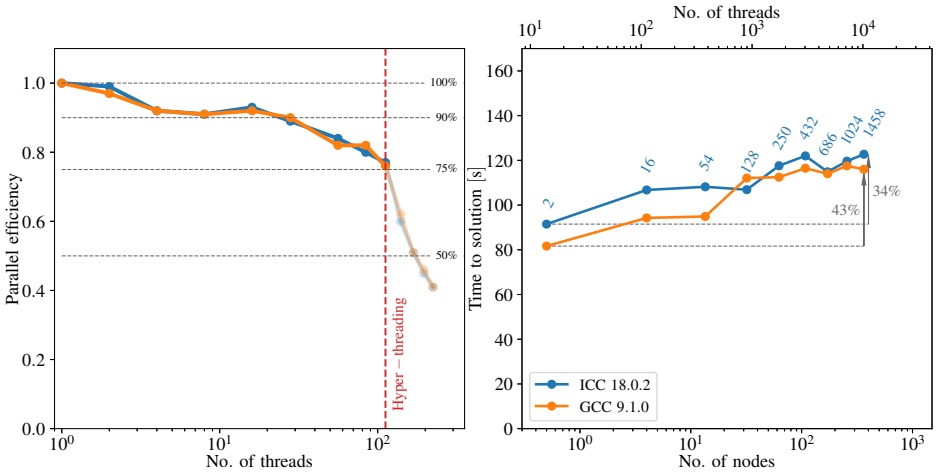


Figure 4. SWIFT FoF scaling results on a representative cosmological problem. The particle data is taken from the EAGLE simulations [18] from a snapshot at redshift $z = 0.1$, i.e. near the end of the calculation when the distribution of particles is far from uniform. (*Left*) Strong scaling results. The particle load was kept constant at 4.25×10^8 whilst the number of cores was increased. As the benchmark was performed on one node, the non-MPI version of the algorithm was used. We maintain very good strong scaling performance and obtain 77% parallel efficiency on 112 cores. The efficiency drops when running with hyper threads due to resource contention between threads. (*Right*) Weak-scaling results. The number of particles per core is kept constant at 3×10^7, as we increase the core count. We use 4 MPI ranks per node (2 ranks *per socket*). For convenience, the total number of MPI ranks used is indicated by the labels above the data points. The vertical arrow displays the percentage loss in performance running on 10,206 cores, which was 43% for GCC and 34% for ICC. We achieve good weak scaling from ½ a node to 364½ nodes (a factor of 729 increase in the number of particles and number of cores) despite the overhead costs of MPI communication. For both panels, the standard deviation of each measurement is smaller than the symbol size.

There is also a noticeable difference in runtime between the Intel and GNU compilers for the first three data points, with GNU showing a speed-up of ∼13% over Intel. A similar discrepancy is also seen in the strong scaling results.

This is a combination of a highly efficient parallel Union-Find algorithm within a single node and a scalable distributed memory strategy between nodes. The domain decomposition implemented in SWIFT also plays a role so as to keep the work load balanced between MPI ranks (see [6] and [19]).

4.3. Comparison to other software

As another performance test we compared our implementation against VELOCIRAPTOR [3], a FoF application commonly used in the literature. We used the same setup as in the strong-scaling test and ran on the MAD02 machine using the Intel compiler and MPI library v.18.0.2[9]. We ran the non-MPI version of our code and the MPI version of VELOCIRAPTOR with 1 rank per core. Our FoF took 13.2s to run to completion and VELOCIRAPTOR took 242s, leading to a net speed-up of 18.3x[10]. Both codes yield the same answer. Given the large difference in run time on one node and the good weak-

[9] with the flags `-O3 -xCORE-AVX512`

[10] Note that we used the MPI version of VELOCIRAPTOR as it was significantly faster than its shared-memory (OpenMP) version which took 1882s running with 112 threads on the same setup.

scaling displayed by our implementation, we decided not to compare our performance with VELOCIRAPTOR at scale.

5. Conclusions

We presented an efficient and scalable new implementation of the FoF method that is commonly used to identify structure in cosmological simulations. The Union-Find data structure was used to create a *forest* of particles, where each *tree* contains a set of particles that share the same group. A hybrid approach was adopted using threads and MPI, which allows it to optimally utilise both shared and distributed memory machines. We made use of atomics to update the list of particle groups which ensures our implementation remains lock-free. The neighbour search over particles was sped up using the octree present in the SWIFT code. A hash table was used in both the group size calculation and group merging across MPI domains to lower the memory footprint and improve the time to solution.

When implemented in the SWIFT code our FoF algorithm achieves good weak-scaling from 14 to 10,206 cores and displays good strong-scaling performance, maintaining 77% parallel efficiency running on 112 cores. We also compare favourably with the commonly used FoF application VELOCIRAPTOR, obtaining a speed-up of 18x over it. Together with the weak-scaling performance displayed up to 10^4 cores this speed-up should allow for an efficient run time when used on-the-fly in production simulations using $\gtrsim 10^5$ cores.

Acknowledgements

We thank the anonymous referees for their comments that greatly helped improve the paper. This work would not have been possible without Lydia Heck, Peter Draper, Richard Regan and Alastair Basden's help and expertise running on the cosma systems; as well as the SWIFT team for their help and input on this project. This work is supported by INTEL through establishment of the ICC as an INTEL parallel computing centre (IPCC). MS is additionally supported by the NWO VENI grant 639.041.749. This work used the DiRAC@Durham facility managed by the Institute for Computational Cosmology on behalf of the STFC DiRAC HPC Facility (www.dirac.ac.uk). The equipment was funded by BEIS capital funding via STFC capital grants ST/K00042X/1, ST/P002293/1, ST/R002371/1 and ST/S002502/1, Durham University and STFC operations grant ST/R000832/1. DiRAC is part of the National e-Infrastructure.

References

[1] D. Potter, J. Stadel, and R. Teyssier, "Pkdgrav3: beyond trillion particle cosmological simulations for the next era of galaxy surveys," *Computational Astrophysics and Cosmology*, vol. 4, p. 2, May 2017.

[2] M. Davis, G. Efstathiou, C. S. Frenk, and S. White, "The evolution of large-scale structure in a universe dominated by cold dark matter," *The Astrophysical Journal*, vol. 292, 06 1985.

[3] P. J. Elahi *et al.*, "Hunting for galaxies and halos in simulations with velociraptor," *Publications of the Astronomical Society of Australia*, vol. 36, p. e021, 2019.

[4] P. E. Creasey, "Tree-less 3d Friends-of-Friends using Spatial Hashing," *Astron. Comput.*, vol. 25, pp. 159–167, 2018.

[5] Y. Kwon *et al.*, "Scalable clustering algorithm for n-body simulations in a shared-nothing cluster," in *Scientific and Statistical Database Management*, (Berlin, Heidelberg), pp. 132–150, Springer Berlin Heidelberg, 2010.

[6] M. Schaller *et al.*, "Swift: Using task-based parallelism, fully asynchronous communication, and graph partition-based domain decomposition for strong scaling on more than 100,000 cores," in *Proceedings of the PASC Conference*, PASC 16, (New York, USA), pp. 2:1–2:10, ACM, 2016.

[7] P. Gonnet, "Efficient and scalable algorithms for smoothed particle hydrodynamics on hybrid shared/distributed-memory architectures," *SIAM Journal on Scientific Computing*, vol. 37, no. 1, pp. C95–C121, 2015.

[8] S. Arya and D. M. Mount, "A fast and simple algorithm for computing approximate euclidean minimum spanning trees," in *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pp. 1220–1233, 2016.

[9] M. Connor and P. Kumar, "Fast construction of k-nearest neighbor graphs for point clouds," *IEEE Transactions on Visualization and Computer Graphics*, vol. 16, pp. 599–608, July 2010.

[10] Z. Galil and G. F. Italiano, "Data structures and algorithms for disjoint set union problems," *ACM Computing Surveys (CSUR)*, vol. 23, no. 3, pp. 319–344, 1991.

[11] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *J. ACM*, vol. 22, pp. 215–225, Apr. 1975.

[12] R. J. Anderson and H. Woll, "Wait-free parallel algorithms for the union-find problem," in *In Proc. 23rd ACM Symposium on Theory of Computing*, pp. 370–380, 1994.

[13] F. Manne and M. M. A. Patwary, "A scalable parallel union-find algorithm for distributed memory computers," in *Parallel Processing and Applied Mathematics* (R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, eds.), (Berlin, Heidelberg), pp. 186–195, Springer Berlin Heidelberg, 2010.

[14] C. Harrison, H. Childs, and K. P. Gaither, "Data-parallel mesh connected components labeling and analysis," in *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization*, EGPGV '11, (Aire-la-Ville, Switzerland, Switzerland), pp. 131–140, Eurographics Association, 2011.

[15] S. V. Jayanti and R. E. Tarjan, "A randomized concurrent algorithm for disjoint set union," in *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, (New York, NY, USA), pp. 75–82, ACM, 2016.

[16] P. Gonnet, A. B. G. Chalk, and M. Schaller, "QuickSched: Task-based parallelism with dependencies and conflicts," *arXiv e-prints*, p. arXiv:1601.05384, Jan 2016.

[17] T. L. Harris, K. Fraser, and I. A. Pratt, "A practical multi-word compare-and-swap operation," in *Distributed Computing* (D. Malkhi, ed.), (Berlin, Heidelberg), pp. 265–279, Springer Berlin Heidelberg, 2002.

[18] J. Schaye *et al.*, "The EAGLE project: simulating the evolution and assembly of galaxies and their environments," *Monthly Notices of the Royal Astronomical Society*, vol. 446, pp. 521–554, 11 2014.

[19] J. Borrow *et al.*, "SWIFT: Maintaining weak-scalability with a dynamic range of 10^4 in time-step size to harness extreme adaptivity," in *Proceedings of the 13th SPHERIC International Workshop*, SPHERIC 13, pp. 44 – 51, Jun 2018.

Parallel Performance

This page intentionally left blank

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200052

Improving the Scalability of the ABCD Solver with a Combination of New Load Balancing and Communication Minimization Techniques[1]

Iain DUFF [a,b], Philippe LELEUX [a,2], Daniel RUIZ [c], and F. Sukru TORUN [d]
[a] *CERFACS, Toulouse, France*
[b] *Scientific Computing Dpt., Rutherford Appleton Laboratory, Oxon, England*
[c] *IRIT - Institut de recherche en informatique de Toulouse, Toulouse, France*
[d] *Ankara Yildirim Beyazit University, Ankara, Turkey*

Abstract The hybrid scheme block row-projection method implemented in the ABCD Solver is designed for solving large sparse unsymmetric systems of equations on distributed memory parallel computers. The method implements a block Cimmino iterative scheme, accelerated with a stabilized block conjugate gradient algorithm. An augmented pseudo-direct variant has also been developed to overcome convergence issues. Both methods are included in the ABCD solver with a hybrid parallelization scheme. The parallel performance of the ABCD Solver is improved in the first non-beta release, version 1.0, which we present in this paper. Novel algorithms for the distribution of partitions to processes are introduced to minimize communication as well as to balance the workload. Furthermore, the master-slave approach on each subsystem is also improved in order to achieve higher scalability through run-time placement of processes. We illustrate the improved parallel scalability of the ABCD Solver on a distributed memory architecture by solving several problems from the SuiteSparse Matrix Collection.

Keywords. Block Cimmino, hybrid solver, sparse matrix, distributed memory parallelism, iterative solver

1. The iterative and augmented block-Cimmino method

The Augmented Block Cimmino Distributed Solver (ABCD Solver) is a distributed hybrid scheme designed to solve large sparse unsymmetric linear systems of the form:

[1]This work was supported by the Energy oriented Centre of Excellence (EoCoE), grant agreement number 676629, funded within the Horizon2020 framework of the European Union. We acknowledge PRACE for awarding us access to MareNostrum at the Barcelona Supercomputing Center (BSC), Spain.
[2]Corresponding author: CERFACS, Toulouse, France; E-mail: leleux@cerfacs.fr

$$Ax = b, \tag{1}$$

where A is a full row rank $m \times n$ matrix, $m \leq n$, x is a vector of size n and b is a vector of size m. The approach is based on the *block Cimmino row projection method* (**BC**) [1]. BC is applied to the system which is partitioned in p row blocks where $p < m$. Starting from an arbitrary initial estimate $x^{(0)}$, a BC iteration improves the estimated solution by summing the projections of the current iterate on the subspaces spanned by the blocks of rows to converge to a solution. The convergence rate of BC is known to be slow [2]. When looking at the fixed point of the iterations, we obtain the following equivalent system:

$$Hx = k, \text{ where } \begin{cases} H = \sum_{i=1}^{p} \mathcal{P}_{\mathcal{R}(A_i^T)} = \sum_{i=1}^{p} A_i^+ A_i \\ k = \sum_{i=1}^{p} A_i^+ b_i. \end{cases} \tag{2}$$

As the row blocks A_i are assumed to have full row rank, H is symmetric and positive definite. To accelerate the convergence of the block Cimmino method, we solve instead this system using a block conjugate gradient algorithm (BCG) improved with stabilization of both residuals and directions [3]. The convergence of this method stays problem dependent and in some cases, convergence profiles with long plateaux can be observed. The eigenvalues of the matrix H are directly linked to the principal angles between subspaces spanned by the row partitions. If these angles are wider, the convergence becomes faster.

As an alternative, that we call **ABCD**, our solver also offers the possibility of constructing a larger system $\begin{bmatrix} A & C \\ B & S \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} b \\ f \end{bmatrix}$ where the numerical orthogonality between partitions is enforced. As a result, the block Cimmino method converges in exactly one iteration and x is the solution of the original system. This results in a pseudo-direct method [4] with the solution dependent on the projections as in BC, and on the direct solution of a system involving the matrix S. However, the efficiency of such an approach, compared to other sparse direct solvers, depends on the size or the density of the condensed system S which are problem dependent. Implementation of both ABCD and BC are available in the ABCD Solver[3] package.

2. Hybrid parallelism

In this section, we present the parallelization scheme of the ABCD Solver using MPI and OpenMP, and the need for an optimization of the load balancing and communication reduction. Both BC and ABCD methods perform the same preprocessing steps. Firstly, after scaling the system, we partition the matrix so that the principal angles between the subspaces given by the partitions are not too small, and the sizes of the partitions are balanced. There are many ways to construct these partitions. In the case of an iterative solution with BC, we will

[3] http://abcd.enseeiht.fr/

consider graph partitioners on the normal equations as they tend to reduce the number of iterations, as illustrated in [5]. In the case of the pseudo-direct solution with ABCD, we shall consider instead the multilevel hypergraph partitioner PaToH [6], which essentially decreases the size of the augmentation scheme, see [7]. Secondly, the basic idea is to distribute each partition to one process, called *master*, which builds an augmented system [8] used to compute the projection on the subspace spanned by the block of rows in the partition.

Thirdly, these augmented systems are solved using the sparse direct solver MUMPS[4] [9]. This direct solver uses the well-known multifrontal method and performs three steps: analysis (preprocessing, estimation of workload and memory), LDL^T factorization, and finally solve (forward elimination and backward substitution). Analysis and factorization must only be performed once, while one solve is needed to compute each projection at each iteration. These local projections are then summed through non-blocking point-to-point communications between masters. The amount of data communicated is equal to the number of shared columns, called *interconnections*. Note that additionally in ABCD, the matrix S is built in an embarrassingly parallel way by computing each column with a projection independently, then S is given in distributed form directly to MUMPS for a parallel solve on the global communicator (see [4] and [7] for the details of the construction and solution of S).

The ABCD Solver is a *hybrid scheme*, in the sense that the method is iterative but relies on a direct solver for each subproblem defined by the partition. The solver also implements a *hybrid parallelism* in the sense that several levels of parallelism are exploited at the same time:

1. the projections are independent and can be computed in parallel,
2. the MUMPS solver introduces two levels of parallelism: through the exploitation of its *elimination tree* and through the factorization of large frontal matrices using parallel linear algebra *dense kernels*.

Depending on the number of processes and the number of partitions, there are various possibilities for scheduling the computations. In the following sections of the article, we propose and study three different approaches for this. In the first approach, we consider an equal number of processes and partitions, in which case each master has exactly one partition. We experiment, in Section 3, to find the optimal number of processes per node to reduce the execution time.

In the second approach, the number of MPI processes is assumed to be less than the number of partitions. In such a case, the idea would then be to assign groups of partitions to the masters, which will construct one single block diagonal system, made with the various partitions. This block diagonal system can then be solved as before using MUMPS, and the goal is to balance the workload over all masters when distributing the partitions. In Section 4, we propose a new algorithm that aims to group partitions on each master so as to minimize the overhead of communication between masters, and at the same time equilibrate the load balance across masters.

In the third approach, we assume more MPI processes than partitions, in which case processes with no partitions can be associated with the masters, as

[4] http://mumps-solver.org/

slave processes, in order to contribute to the parallel computations in MUMPS. The target is to set master-slaves groups with balanced workloads, by taking into account the anticipated number of flops given by the MUMPS analysis of each partition. In Section 5, we first present a fast and optimal assignment of the slaves that balances the workload across subgroups of processes. Then we introduce a new method to assign the processes, masters and slaves, in the physical computing resources to decrease the communication overhead both within and between master-slaves groups depending on the method used, BC or ABCD.

In the ABCD Solver, we distinguish three types of communications [7]: the *inter-communication* between masters which occurs when summing the projections; the *intra-communication* inside master-slaves group which only occurs when computing a projection using MUMPS; finally in ABCD, *global communication* when solving the system based on S.

To illustrate the impact of our contributions, we run the ABCD Solver on three matrices from the SuiteSparse Matrix Collection [10]. Table 1 shows characteristics of the matrices. We conduct our experiments on MareNostrum4, a petascale supercomputer at the Barcelona Supercomputing Center[5]. It is a cluster with Intel Xeon Platinum processors. Each compute node is a 2-socket system where the 24 cores of each processor constitute a separate NUMA (non-uniform memory access) domain and nodes are interconnected with the Intel Omni-Path architecture. MareNostrum4 offers 96 GB RAM memory per NUMA domain, which means around 4 GB per core.

Table 1. Characteristics of the test matrices. n: the order of the matrix, nnz: the number of nonzero values in the matrix.

Matrix	n ($\times 10^6$)	nnz ($\times 10^6$)	nnz/n	kind
hamrle3	1.45	5.51	3.81	circuit simulation problem
cage15	5.15	99.20	19.24	directed weighted graph
memchip	2.70	13.00	4.93	circuit simulation problem

3. Optimal node configuration

When the number of partitions equals the number of processes, we determine the best distribution of MPI processes with respect to the execution time. With a fixed number of 128 MPI processes and an equal number of 128 partitions, we increase the number of processes per node from 2 to 64. Table 2 shows the execution times and we see that 2 MPI processes per node yields the minimum overall times. Although this results in more communication, because the linear algebra kernels used throughout the code and in MUMPS are memory-bound, they benefit from distributing the memory. Fewer processes per node implies less concurrent access to memory and faster computation. We will allocate 2 processes per node as our optimal configuration in the rest of the paper.
Since only a subset of nodes is used by MPI processes, when increasing the number of nodes we have the possibility of activating OpenMP parallelism but do not study this here where we focus on workload balancing and communication reduction.

[5]https://www.bsc.es/marenostrum/marenostrum

Table 2. Timings for the factorization of the augmented systems, for the BCG in BC, and for the pseudo-direct solution in ABCD. All runs were with 128 MPI processes spread with *ppn* processes per node and 128 partitions. Note that the memory required for ABCD was too large to solve the system cage15 on MareNostrum4.

Matrix	ppn	nodes	BC			ABCD	
			facto(s)	BCG(s)	it.	facto(s)	sol.(s)
Hamrle3	32	4	0.17	192	500	0.22	9.44
	16	8	0.19	138	”	0.21	9.48
	4	32	0.18	79	”	0.20	9.50
	2	64	0.18	77	”	0.22	10.10
cage15	32	4	1 550	65	17	-	-
	16	8	1 380	52	”	-	-
	4	32	1 230	40	”	-	-
	2	64	1 210	38	”	-	-
memchip	32	4	0.44	361	500	0.42	29.60
	16	8	0.31	269	”	0.31	29.70
	4	32	0.28	171	”	0.29	28.80
	2	64	0.27	168	”	0.29	28.10

4. Load balancing: distribution of partitions

In the case where the number of partitions is higher than the number of processes, a master process owns a group of partitions. In this section, the goal is to distribute the partitions to the masters with the right trade-off between balancing the weight of the local groups of partitions over all processes and minimizing the overhead in communication between masters.

4.1. Balancing the weight of the local partitions

We first consider only balancing the weights of the partitions. The weights should represent the future workload to compute projections. In the absence of more precise data at this point of the solver, we simply use the number of rows as a crude measure. Although this gives reasonable results here, it can result in bad load imbalance. In the next section, we will use accurately estimated workloads from a latter phase of the solver to distribute the slave processes. To balance the weights, we use the greedy algorithm introduced in [7]. The algorithm distributes partitions sorted in decreasing order of weights to masters. At each step, the master with current lowest accumulated weight receives a partition. This process results in an optimal distribution of the partitions over all masters in terms of balancing our criterion.

4.2. Minimize the overhead of communication

Globally, balancing the weights of local sets of partitions is not the only concern, one should also consider the overhead from inter-communication between masters resulting from the distributed sum of local projections and, in ABCD, from the parallel solution of the condensed system S. Therefore, the best distribution of the partitions should find the right trade-off between this communication, i.e. minimizing the number of interconnected columns between processors, and balancing the workload over processes in order to achieve minimum parallel execution time.

We propose a new algorithm which is based on this principle. The algorithm first creates a graph $\mathcal{G}$. The vertices of $\mathcal{G}$ are the partitions weighted by their respective size. There is an edge between two vertices if the corresponding partitions are interconnected, i.e. they share a nonzero column, and the cost of that edge equals the number of such columns. In the final step, we partition $\mathcal{G}$ using the multilevel graph partitioning tool METIS [11] to minimize the number of interconnections between the groups of partitions for each master, with a parameter μ that allows a certain imbalance in the accumulated weight over the groups of partitions.

4.3. Experimental results

The experiments are conducted on the three matrices with the greedy algorithm (*Greedy*) and the communication reducing algorithm where $\mu = 1\%$ (*Comm1*) and $\mu = 10\%$ (*Comm10*). Each matrix is partitioned into 1 024 blocks and is solved using 128 MPI processes spread over 64 distributed nodes with no multithreading. The numerically aware partitioning [5] is applied for BC, and the PaToH hypergraph partitioner is used for ABCD. Results are reported in Table 3. The column 'Com. col%' of Table 3 reports the total communication volume, equal to the number of interconnected columns, normalized with respect to the greedy method. The table also reports execution times for the factorization as well as the imbalance ratio between the slowest and average factorization times over all masters. Finally, the table gives the BCG execution time and iterations for BC, and the time to compute the pseudo-direct solution including the solution of the system S for ABCD.

As seen in the table, for BC, the proposed methods *Comm1* and *Comm10* achieve around 55% and 62% reduction in the total number of exchanged columns for the cases of Hamrle3 and memchip, respectively. This improvement in turn leads to faster parallel execution of BCG for Hamrle3 and memchip. Our experiments show that the larger ratio μ has a limited effect on the reduction of the total size of communication. On the other hand, for cage15, although there is considerable reduction in the communication values, the execution time increases slightly because the overhead of load imbalance absorbs the gain from the minimization of communication.

In the case of ABCD, there is only one iteration, thus each communication is only performed once. Compared to the gain of having balanced workloads over the MUMPS instances, the final communication overhead is low and thus the time only increases, slightly, with the proposed algorithm.

5. Placement of masters and slaves

In this section, we consider the case where there are more processes than partitions. We make use of the extra processes to act as slaves to help the master MPI processes to parallelize the computation further. We balance the workload over all masters by assigning more slaves to a master with a relatively higher workload.

Table 3. Impact of the distribution of partitions on the execution times. All runs were with 1024 partitions and 128 MPI processes on 64 nodes with no multithreading. (Com. col: Normalized column reduction values with respect to the Greedy algorithm. tot: Total time in seconds. it: Number of iterations required for convergence. imb: ratio of maximum over average factorization times. Sol. time: Total solution time in seconds)

Matrix	Algo.	BC					ABCD			
		Com.	Fact.		BCG		Com.	Fact.		Sol.
		col%	tot	imb	tot	it.	col%	tot	imb	time
Hamrle3	Greedy	100	0.22	1.62	714.25	4249	100	0.24	1.21	**8.17**
	Comm1	46	0.19	1.26	**700.66**	4249	42	0.25	1.35	9.12
	Comm10	45	0.20	1.36	713.69	4249	41	0.20	1.36	8.79
cage15	Greedy	100	20.41	1.97	**28.01**	18	-	-	-	-
	Comm1	44	42.61	3.94	34.62	18	-	-	-	-
	Comm10	44	48.82	3.75	35.00	18	-	-	-	-
memchip	Greedy	100	0.36	1.18	299.23	791	100	0.32	1.18	**5.64**
	Comm1	38	0.33	1.22	298.89	791	32	0.34	1.27	5.74
	Comm10	38	0.35	1.21	**292.05**	791	31	0.33	1.23	5.72

5.1. Assignment of the slaves

We consider w_k the accurate estimated workload of master $k \in \{1..nb\_masters\}$ given by MUMPS, i.e. the number of flops required for MUMPS factorization. We propose a new 2-step algorithm for the distribution of the slaves. Firstly, considering the number of slaves corresponding to the relative workload of each master k:
$s_k^{(theo)} = (w_k / \sum_{i=1}^{\#masters} w_i) \times \#slaves$, a number of slaves equal to the floor part of this amount is assigned to each master. Since most of the slaves are now associated with a master, the second step only has to allocate the remaining slaves. Secondly, we apply a greedy algorithm: at each step, one of the remaining slaves is assigned to the master-slave group with the currently highest average workload, until all slaves have been assigned. We obtain an optimal distribution of the slaves in terms of average workload and, thanks to the first step, the number of greedy searches performed is decreased.

5.2. Hierarchy of the computing architectures

The ABCD Solver is designed to solve large systems on distributed memory architectures where the computing resources are hierarchically structured, as is the case here with the supercomputer MareNostrum4.

When launching our distributed application, we specify a certain number of MPI processes per node which are allocated by the batch system. As a result, when the program starts, processes are already allocated and placed on the system architecture in a certain way. Depending on the situation at runtime, we need to decide which processes will be given the role of master or slave in order to minimize the total overhead of the communication between masters (inter-communication) on the one hand, and inside master-slaves groups (intra-communication) on the other hand. This process consists of three steps: firstly the placement of the masters, secondly the assignment of the number of slaves as in the last section using the estimation of the workload with MUMPS, and thirdly choosing the slaves for each master depending on its position in the architecture.

Two opposite approaches emerge in this situation. We can place masters close to each other to accelerate inter-communication, and we refer to this approach as *Compact*, or we place the master-slaves group together on a node to simultaneously improve intra-communication, and decrease concurrent access to memory by masters. We refer to this latter approach as *Scatter*.

5.3. Explicit placement of masters and slaves over nodes

The approach first implemented in the ABCD Solver, see [7], is Compact: the first ranks of MPI make the masters and the rest of the processes are assigned in a sequence to them as slaves depending on the rank. Although this approach minimizes the inter-communication, both the intra-communication as well as the sequential calls to dense kernels, known to be memory-bound, are slowed down due to concurrent memory access among masters.

Based on the results obtained in Section 3, mainly for BC, we have seen that spreading processes over the nodes is better because of more efficient memory access. Thus, we propose to implement the Scatter approach to improve the execution time of the ABCD Solver. Note that we currently use a "manual" implementation of this approach, but this could be replaced by architecture aware mechanisms in the future [12]. We define two algorithms for placement of the masters and the slaves.

The principle of these algorithms is simple:

- To place masters, we first gather information to know which node each process is on. We then assign one master per non-full node in a zig-zag fashion, starting from the biggest node to smallest then alternating.
- To place slaves, we first sort the masters in descending number of desired slaves. Then for each master, we place the slaves in the corresponding node and, if some are left, we group the remaining ones in other nodes as closely as possible.

In Figure 1, we illustrate the effect of the Compact and Scatter approach on a toy example. We partition a matrix in 3 partitions solved using block Cimmino with 12 processes. We define 3 masters each with 3 slaves and launch the solver on 3 nodes each with 4 processes.

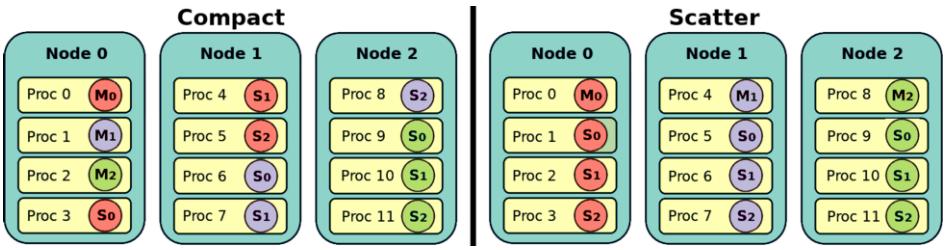


Figure 1. 3 nodes with 4 processes on each and we have 3 masters with 3 slaves each. M_i corresponds to the master i and the S_j of the same colour is its slave j. **(Left)** Compact scheme, **(Right)** Scatter scheme.

5.4. Experimental results

The results are presented in Table 4. Firstly, we observe that the execution times for factorization remain mostly unchanged for both algorithms for memchip and Hamrle3. In the case of cage15, which is dominated by this phase, the execution time of factorization is decreased in Scatter, benefiting from less concurrent access to memory. Concerning the BC method, the times for the sum of projections, which is included in the time for BCG, can increase in some cases with the Scatter approach, due to most master-slaves communicators being spread over the nodes. However, the overall BCG run-times always benefit from spreading the masters over the nodes, inducing less concurrency in memory access, and from grouping master-slaves groups, thanks to faster intra-communication. The effects of changing the algorithm are overall very small. In the end, we only have 2 processes per node so changing their placement does not change the global performance. We ran the experiment for the matrices Hamrle3 and memchip again, using 16 processes per node, thus 128 MPI processes on 8 nodes. The memory required for cage15 was too high for this configuration. Regarding the run-time of the BCG, Hamrle3 is solved in 203s with Compact and 161s with Scatter, while memchip is solved in 254s with Compact and 186s with Scatter. While the overall run-time with Scatter is higher than running with only 2 processes per node, the difference is only 4.5% for both matrices. Using the Compact algorithm however, the degradation is around 25%. This means that using the Scatter algorithm is more robust to having multiple active cores per node, which is a big step towards gaining scalability. However, in the case of ABCD the time to compute the pseudo-direct solution no longer benefits from spreading the masters with Scatter. In this approach, the computation is completely distributed, thus the overhead in communication absorbs the improvement from lower concurrent access to memory. Overall, the timings are not too different. Because of an implementation mixing together multiple layers of parallelism from MUMPS and the partitioning itself, the hybrid parallelism used is robust.

Table 4. Impact of the placement of masters and slaves on the execution times of ABCD Solver iterative method. All runs were with 32 partitions and 128 MPI on 64 nodes with no multithreading.

Matrix	Algo.	Block Cimmino				ABCD	
		facto(s)	BCG(s)	it.	proj. sum(s)	facto(s)	Sol.(s)
Hamrle3	Compact	0.41	159	500	76.4	0.36	43.2
	Scatter	0.40	154	500	77.4	0.33	45.1
cage15	Compact	567	22.7	15	14.6	-	-
	Scatter	560	22.3	15	14.7	-	-
memchip	Compact	0.40	184	365	89.1	0.46	24.8
	Scatter	0.43	178	365	85.8	0.45	28.8

6. Conclusion

We have shown the potential improvement that can be obtained in a master-slave scheme by considering the minimization of communication on an equal footing

with the balancing of workload. Firstly, we proposed a new distribution of partitions such that we decrease the communication between masters in the block Cimmino method, thus decreasing the total execution time in a context where many iterations are necessary with processes communicating for each iteration. Secondly, we propose a new way of attributing the roles of master or slave to processes depending on the run-time situation on the machine. We have identified two specific schemes : scattering the masters over the nodes is well adapted to the block Cimmino method, especially when the number of iterations is high, while compacting the masters in the same nodes is adapted for the augmented block Cimmino pseudo-direct method. Furthermore, the Scatter approach is more robust with respect to the number of processes per node, which is a big step towards scalability. Finally, we demonstrate the improved parallel scalability on a distributed memory architecture.

References

[1] Tommy Elfving. Block-iterative methods for consistent and inconsistent linear equations. *Numerische Mathematik*, 35(1):1–12, 1980.

[2] Randall Barry Bramley and Ahmed Sameh. Row projection methods for large nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 13(1):168–193, 1992.

[3] Daniel Ruiz. Solution of large sparse unsymmetric linear systems with a block iterative method in a multiprocessor environment. *CERFACS TH/PA/9*, 6, 1992.

[4] Iain S Duff, Ronan Guivarch, Daniel Ruiz, and Mohamed Zenadi. The augmented block cimmino distributed method. *SIAM Journal on Scientific Computing*, 37(3):A1248–A1269, 2015.

[5] F. Torun, M. Manguoglu, and C. Aykanat. A novel partitioning method for accelerating the block cimmino algorithm. *SIAM Journal on Scientific Computing*, 40(6):C827–C850, 2018.

[6] Umit V Catalyürek and Cevdet Aykanat. Patoh: a multilevel hypergraph partitioning tool, version 3.0. *Bilkent University, Department of Computer Engineering, Ankara*, 6533, 1999.

[7] Mohamed Zenadi. *The solution of large sparse linear systems on parallel computers using a hybrid implementation of the block Cimmino method*. PhD thesis, EDMITT, 2013.

[8] Åke Björck. Iterative refinement of linear least squares solutions i. *BIT Numerical Mathematics*, 7(4):257–278, 1967.

[9] Patrick R Amestoy, Iain S Duff, Jean-Yves L'Excellent, and Jacko Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[10] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[11] G Karypis and V Kumar. Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. *Department of Computer Science, University of Minnesota*, 1995.

[12] Emmanuel Jeannot, Guillaume Mercier, and François Tessier. Process placement in multicore clusters: Algorithmic issues and practical techniques. *IEEE Transactions on Parallel and Distributed Systems*, 25(4):993–1002, 2013.

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200053

Characterization of Power Usage and Performance in Data-Intensive Applications Using MapReduce over MPI

Joshua DAVIS [a], Tao GAO [a], Sunita CHANDRASEKARAN [a], Heike JAGODE [b], Anthony DANALIS [b], Jack DONGARRA [b], Pavan BALAJI [c], and Michela TAUFER [b,1]

[a] *University of Delaware*
[b] *University of Tenneesee Knoxville*
[c] *Argonne National Laboratory*

Abstract. This paper presents a quantitative evaluation of the power usage over time in data-intensive applications that use MapReduce over MPI. We leverage the PAPI powercap tool to identify ideal conditions for execution of our mini-applications in terms of (1) dataset characteristics (e.g., unique words in datasets); (2) system characteristics (e.g., KNL and KNM); and (3) implementation of the MapReduce programming model (e.g., impact of various optimizations). Results illustrate the high power utilization and runtime costs of data management on HPC architectures.

Keywords. Data management, KNL, KNM, PAPI, Combiner optimizations

1. Introduction

The contributions of this paper are in the growing high performance computing (HPC) field of data analytics and are at the cross-section of empirical collection of performance results and the rigorous, reproducible methodology for their collection. Our work expands traditional metrics such as execution times to include metrics such as power usage and energy usage associated with data analytics and data management. We move away from the traditional compute-intensive workflows towards data-intensive workloads with a focus on MapReduce programming models as they gain momentum in the HPC community.

Emerging HPC platforms are generally designed with data management and, in particular, the associated power usage in mind. Trends in HP exhibits the widening gap between flops and IO bandwidth peaks. The latter is capped to contain the power usage of the HPC systems. In other words, the need for power capping on these systems has created substantial constraint on the bandwidths with respect to moving data through the memory hierarchy. Overall, it is a common belief that data management (i.e., process-

[1]Corresponding Author: Address: Electrical Engineering and Computer Science Dept., The University of Tennessee, Knoxville, TN 37996-2250; E-mail: taufer@utk.edu.

ing data on the core and moving data through the memory hierarchy) is power-intensive. Still, little work is available in providing quantitative evaluations of these costs.

The paper tackles this problem by addressing the need to quantitatively measure the impacts of data management on performance in MapReduce-based applications when executed on HPC systems. Specifically, we present studies on the impact of power capping on performance metrics such as runtime and power usage over time for data-intensive application on top of a MapReduce over MPI framework. The key questions that we look to answer are: Can we determine what data characteristics are the most relevant to the trade-offs between power usage and performance? Can we identify the effects of power capping on the performance of data-intensive applications using the MapReduce programming model on HPC systems? Can we separate the impact of the application itself from the impact of the middleware and the monitoring system on power usage?

2. Testing Environment for Power Measurements

Multilayer Testing Environment: Our testing environment is composed of multiple layers . The middleware layer manages the data analytics. Because we are working on HPC systems, we use Mimir, an open source implementation of MapReduce over MPI, that we enhance with the Performance Application Programming Interface (PAPI) monitoring and capping performance tool. The hardware layer includes two many-core systems that fully support Mimir and PAPI. We target the **MapReduce (MR)** programming model [1] for the data management and analytics because these processes are broadly used and suitable for a wide variety of data applications. Implementations of MapReduce over MPI have gained the most traction in HPC because they provide C/C++ interfaces that are more convenient to integrate with existing scientific applications compared with Java, Scala, or Python interfaces. Moreover, they can use the high-speed interconnection network through MPI. More traditional frameworks such as Hadoop [2, 3] and Spark [4] or tuned versions of these popular MapReduce frameworks on HPC systems [5–8] do not support one or multiple of the following features: they do not provide on-node persistent storage; do not work well (or work at all) on many commodity-network-oriented protocols, such as TCP/IP or RDMA over Ethernet; are not tuned for system software stacks on HPC platforms, including the operating system and computational libraries; or are specialized for a given scientific computing, and hence lack generality. For example, supercomputers such as the IBM Blue Gene/Q [9] use specialized lightweight operating systems that do not provide the same capabilities as those that a traditional operating system such as Linux or Windows might.

We use **Mimir** [10–12] as our MapReduce over MPI framework. Mimir is among the few state-of-the-art, memory-efficient MapReduce over MPI implementations that are open source and support memory efficiency and scalability; reduced synchronizations; reduced data staging and improved memory management; improved load balancing; and capabilities to handle I/O variability. Mimir's in-memory workflow includes a pipeline combiner workflow to compress data before shuffling and reduce operation, using memory more efficiently and balancing memory usage; a dynamic repartition method that mitigates data skew on MapReduce applications without obviously increasing their peak memory usage; and a strategy for splitting single superkeys across processes and further mitigating the impact of data skew, by relaxing the MapReduce model constraints on key

partitioning. A MapReduce job traditionally involves three stages: *map*, *shuffle*, and *reduce*. The *map* stage processes the input data using a user-defined map callback function (map operation) and generates intermediate $\langle key, value \rangle$ (KV) pairs. The *shuffle* stage performs an all-to-all communication that distributes the intermediate KV pairs across all processes. In this stage KV pairs with the same key are also merged and stored in $\langle key, \langle value1, value2... \rangle \rangle$ (KMV) lists. The *reduce* stage processes the KMV lists with a user-defined reduce callback function and generates the final output. In Mimir, the shuffling stage consists of two decoupled phases: an aggregate phase that interleaves with the map operations and executes the MPI_AllToAllv communication calls and a convert phase that precedes the reduce operations by dynamically allocating space and assigning KV pairs to list the KVs. The map and reduce operations are implemented by using user callback functions. The aggregate and convert phases are implicit: the user does not explicitly start these phases. This design offers two advantages. First, it breaks global synchronizations between map and aggregate phases and between convert and reduce phases. Mimir determines when the intermediate data should be sent and merged. Mimir also pipelines the four phases to minimize unnecessary memory usage. We still retain the global synchronization between the map + aggregate and convert + reduce phases, which is required by the MapReduce programming model.

We plug **PAPI** [13] into Mimir to set power caps and collect power usage over time, using the powercap interface that is built into the Linux kernel. The purpose of this interface is to expose the Intel RAPL (Running Average Power Limit) [14] settings to userspace, as opposed to directly accessing the RAPL model-specific registers (MSRs), which would require elevated privileges. PAPI provides access to performance counters for monitoring and analysis. We use PAPI to measure the core power/energy consumption, power limit, power of memory accesses, and core frequency. PAPI's powercap component [15] gives access to Intel's RAPL interface, which employs the DVFS technique to apply power limits.

Data-Intensive Miniapplications: We implemented and use three data-intensive miniapplications extracted from the WordCount benchmark: (1) Map+Aggregate, which contains only the map and shuffle phases (without the convert and reduce phases); (2) GroupByKey, which adds the convert/reduce operations for a complete, traditional MapReduce execution; and (3) ReduceByKey, which locally combines KVs with matching keys immediately before shuffling and after shuffling (i.e., combiner optimizations in state-of-the-art MR frameworks).

Data Generation: We generate different datasets with a large number of words (4 billion) and a variable number of unique words. In the rest of the paper, each dataset is identified as *XBY*, where *X* is the number of billions of total words and *Y* is the number of unique words repeated in the dataset. Our dataset is larger than available datasets commonly used in benchmarking (e.g., the Wikipedia dataset from the PUMA dataset [16]). The controlled generation of data allows us to create a controlled testing environment that we can use to separate the different factors impacting power consumption in data management. It also enables reproducibility of our results. The Wikipedia dataset with its highly heterogeneous type and length of words would not allow us to handle the same level of controlled testing. The software used for the data generation is open source, as part of the Mimir software [17]. The total number of words defines the overall workload across the processes: the larger the number, the higher the workload per process. The number of unique words determines how the KMV are distributed across processes dur-

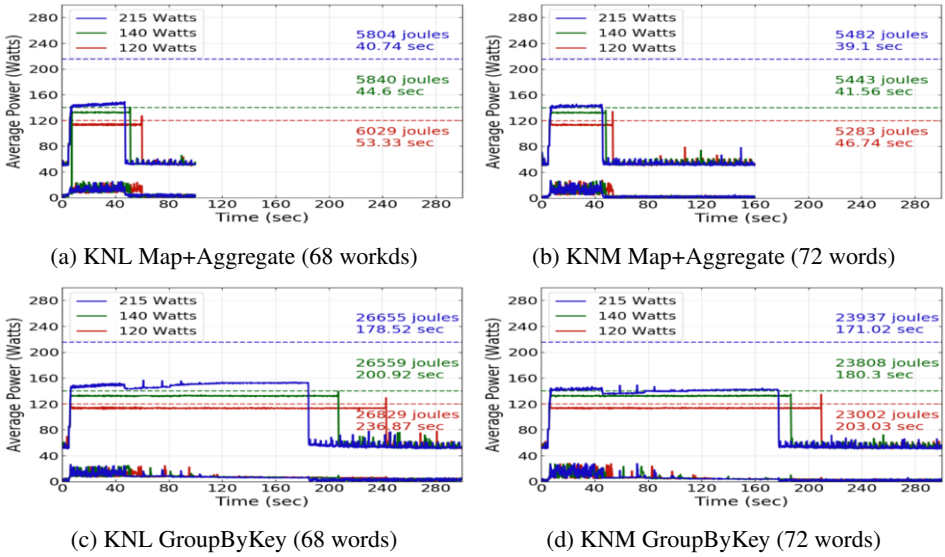
ing shuffle. The number of unique words also affects the percentage of the dataset that can be combined (i.e., unique words with the same key are combined locally to a single KV pair) before and after shuffling in ReduceByKey. In other words, a larger number of unique words means that fewer KV pairs can be combined (i.e., data can be reduced in size) before and after shuffling in ReduceByKey. For example, a dataset of 4 billion words with 72 unique words (4B72) can exhibit up to a 99% combinability rate before the shuffling, whereas a dataset of 4 billion words and 42 million unique words (4B42M) results in only a 25% combinability rate.

3. Performance Analysis

Environment Setting: We measure power usage over time, total energy, and runtime of our miniapplications using two generations of HPC architectures. The first system is a fat node with 68 cores and four hardware threads per core (272 total) the Intel Xeon Phi 7250 Knights Landing (KNL) with a thermal design point (TDP) of 215 watts. The second system is a fat node with 72 cores and four hardware threads per core (288 total) Intel Xeon Phi 7295 Knights Mill (KNM) architecture with the same TDP of 215 watts. We use PAPI to measure the impact of the data management across cores on power. PAPI's event *PACKAGE* allows us to monitor the energy consumption of the entire CPU socket. The PAPI event DRAM_ENERGY monitors the energy consumption of the off-package memory (DDR4). When not otherwise specified, we measure average energy on the core subsystem, including MCDRAM, memory controller, and 2-D mesh interconnect, and on the DDR4 memory at the 100 ms sample rate, a standard rate recommended by the PAPI developers. Finer-grained rates are also assessed. We generate synthetic data that fully fits into the Mimir's system memory, avoiding disk I/O during our tests. We repeat each test three times, with the first of each run shown on graphs to integrate cold-cache effects. We observed a modest variability (less than 2%) across the three runs.

Impact of MapReduce Stages and Architectures: To quantify the impact of the Map+Aggregate versus Convert+Reduce components of the MapReduce job and of different architectures, we measure the power metrics for (a) the combined map operations and aggregate phases of the shuffle stage and (b) GroupByKey (i.e., the complete Map+Shuffle+Reduce workflow) on KNL and KNM using datasets of 4 billion words and 68 unique words on KNL and 72 unique words on KNM, respectively. The number of unique words allows us to fully use the cores available on each HPC architecture. The words are distributed by cutting the dataset in chunks, with each chunk having interleaving sequences of the different words (e.g., at a smaller scale with four unique words "a," "b," "c," and "d," each chuck contains a sequence of "abcdabacdabcd...").

Figures 1a and 1b present power measurement for Map+Aggregate on KNL and KNM, respectively. Figures 1c and 1d present power measurement for GroupByKey on KNL and KNM, respectively. In each figure, we draw three pairs of curves for three power caps: 215, 140, and 120 watts. For each pair, the higher curve is the processor power, and the lower is the DRAM power. We observe that for all the tests, none of our runs exceed 160 watts during runtime (75 watts below system TDP). To study performance impacts, we intentionally impose caps of 140 and 120 watts, which are lower than the miniapplication's max power but higher than the minimum core power usage when idle. We also observe the same patterns for the power usage on both KNL and KNM



(a) KNL Map+Aggregate (68 workds)

(b) KNM Map+Aggregate (72 words)

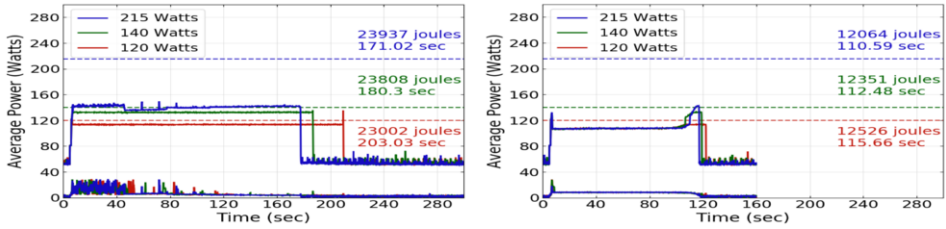
(c) KNL GroupByKey (68 words)

(d) KNM GroupByKey (72 words)

Figure 1. Average power for Map+Aggregate and GroupByKey on KNL and KNM.

architectures; however, tests on KNM have a lower runtime, especially at power limits that are below the normal power usage of the benchmark. This is due to the KNM's higher number of cores. Because of the similar patterns and fact that KNM is the more recent chip, we focus on KNM in the rest of the paper. More important, we observe additional 59–64% runtime associated with the convert phase and reduce operations. The DRAM power forms only a small portion of total power use, making up only between 4% and 15% of the combined processor and DRAM power. Memory power consumption increases during the map and aggregate stages of GroupByKey (and in the Map+Aggregate miniapplication). For Map+Aggregate and GroupByKey, when limiting power to 120 watts, we observe an increase in runtime ranging from 5% to 33%. Comparing Map+Aggregate with GroupByKey allows us to quantify the high runtime and energy costs of the reduce stage. The reduce stage adds 333%–355% more runtime and 335%–359% more energy to Map+Aggregate.

Impact of Combiner Optimizations: MapReduce frameworks have been substantially optimized by switching from the traditional Map+Shuffle+Reduce workflow in GroupByKey to the enhanced workflow based on combiner optimizations in ReduceByKey , for which before, both shuffling and reduce operations, unique words with the same key are combined locally. In general, combiner optimizations can be used for those MapReduce applications that are both associative and commutative (e.g., WordCount), supporting merging KVs with the same key and still providing the correct outcomes. In combiner phases, KV pairs with the same values are combined locally on each node that has just performed the map operation (preshuffling). Once the shuffling of KV pairs is completed and chunks of KV pairs are assigned to processes based on some MapReduce-specific hash function, KV pairs from different processes with the same keys are again combined locally before the reduce operation is performed (postshuffling). From an implementation point of view, in Mimir, the preshuffling and postshuffling processes are executed with combiner callbacks. A first combiner callback is applied before the MPI



(a) GroupByKey – without optimizations (b) ReduceByKey – with optimizations

Figure 2. Impact of combiner optimizations on power usage in a dataset with combinability rate of 99.99%.

Table 1. Datasets used to quantify the impact of different levels of combinability (combinability rate).

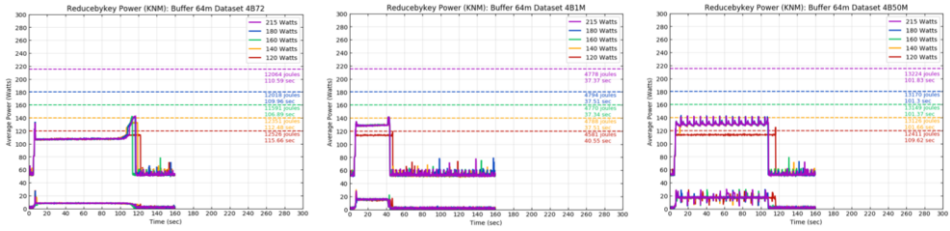
Dataset	Total Words	Unique Words	Combinability Rate
4B72	4B	72	>99%
4B1M	4B	1,111,104	98%
4B50M	4B	49,999,968	10%

communication stage, reducing the communication size, and a second combiner callback is applied after the MPI communication stage, reducing the memory size to store KVs. Figures 2a and 2b show results without combiner optimizations (i.e., GroupByKey) and with combiner optimizations (i.e., ReduceByKey) for a dataset that can exhibit a combinability rate of 99.99% (i.e., the number of KV pairs can be reduced by combining those with the same keys to 0.01% of the initial number because of the very high number of repeated key). Comparing GroupByKey with ReduceByKey allows us to quantify the cost of moving data between processes. Combining KVs before shuffling in ReduceByKey substantially reduces the runtime of the application (up to 46% less runtime) and the power usage over time (up to 11,800 joules saved, or a 50% reduction). One important observation that emerges from this comparison is the intrinsic power cap (without the need of PAPI's cap) that ReduceByKey exhibits for a highly combinable dataset (in Figure 2b). The power usage of the three executions (with 240, 140, and 120 power cap) are all substantially below the defined power caps. In the next two sections we further study the reasons for the implicit capping, by looking into the impacts of the data combinability rate and the MPI buffer size for the shuffling.

Impact of Data Combinability: A KV pair is combined with an existing KV if it is a duplicate of the other KV in the same map process (i.e., the combiner callback combines the new KV with the existing KV). Given a dataset and its number of unique words, the fraction of a dataset that is combinable (called combinability rate) can be calculated as follows:

$$Rate = \frac{N_w - (N_p * N_u)}{N_w}, \tag{1}$$

where N_w is the number of total words, N_u is the number of unique words, and N_p is the number of processors. Table 1 presents the datasets used in this paper with the total number of words (N_w), the number of unique words (N_u), and the combinability rate.



(a) 4B72 (b) 4B1M (c) 4B50M

Figure 3. Power usage with combiner optimizations for datasets with different combinability rates.

Figures 3a– 3c show the power and time metrics for selected datasets in Table 1, with increasing numbers of unique words and decreasing combinability rates. In the progression of the nine figures, we observe how, when zooming into the combiner performance for datasets with different numbers of unique words (and thus different combinability rates), both the power usage over time and runtime vary around a "sweet spot" region (i.e., a region of minimum runtime and energy). This sweet spot region is around 5,000 unique words for our 4 billion word dataset. For smaller numbers of unique words and high combinability rates, we encounter latency associated with the inability to fill MPI communication buffers and the intrinsic strategy of Mimir to less frequently initiate the shuffling. For larger numbers of unique words, the memory and processor power usage begin oscillating. This oscillatory pattern is due to (1) a lower capability of the combiner optimizations to reduce data chunks in size, (2) a larger number of KV pairs locally managed by each map process in its assigned chunk, and (3) the process's frequent memory accesses to swap parts of the larger chunk in and out of memory.

Impact of Communication Buffering: In the tests above, we used a default send buffer size in Mimir of 64 MB. Each map process uses an MPI send buffer to store the KV pairs in groups based on a receiving reduce process. The assignment of a group to a specific process is based on a hash function in Mimir. All the buffers are sent to the reduce processes with an MPI_AllToAllv call when one of the map processes has its send buffer full. In this section, we consider different communication buffer sizes in Mimir (i.e., different sizes triggering the exchange of data among processes in the aggregate phase of the MapReduce workflow) and different data features (i.e., number of unique words in our 4 billion word dataset) to study the impact of Mimir's setting on power usage over time.

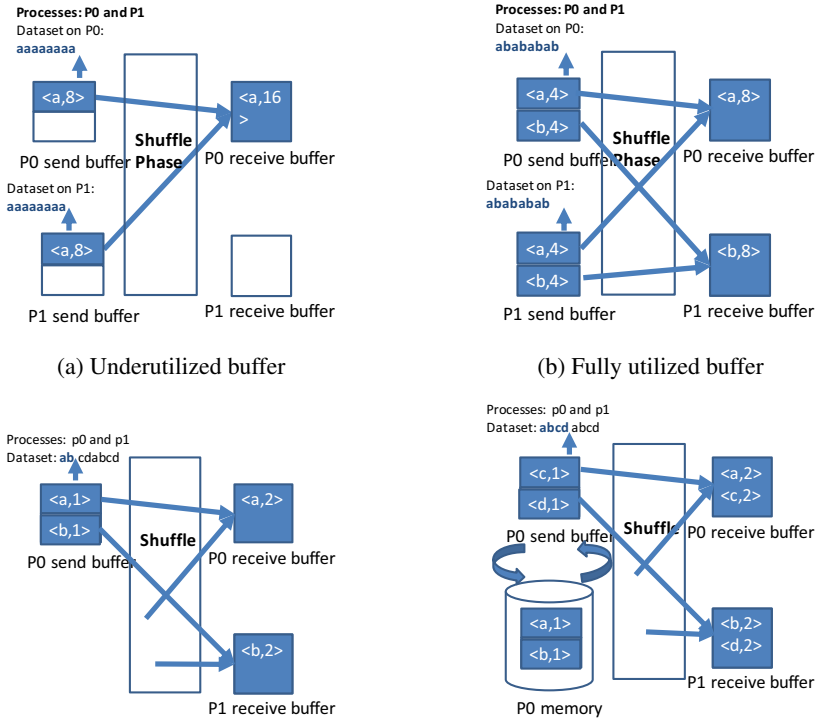
Three possible scenarios can be observed. In the first scenario, the send buffer size of each process is partially underutilized because the different unique KV pairs are not filling it. Note that we are using the combiner optimizations to reduce the use of the buffer. This is the case for our dataset of 4 billion words with only 72 unique words used to build the large dataset. Figure 4a shows an example for a small case study of two processes, each one with a data chunk of 8 "a" datasets. The send buffer is divided into two equal-sized partitions, where two is the number of processes executing our MapReduce application. Mimir's default setting temporarily suspends the computation stage and switches to the shuffling when a partition in the send buffer is full. The fact that there are not sufficient unique words to trigger the shuffling results in the execution of map operations and combiner callbacks in an almost-sequential execution. The implicit power cap is the ultimate consequence of a set of cores that are not pushed to their max workload poten-

tial. We observe the same phenomena with our datasets and 72 unique words. Figures 5a and 5d show the power usage over time when 72 unique words make up the entire 4 billion words dataset. Figures 5a refers to a scenario in which the send buffer is 32 MB; Figure 5d refers to a scenario in which the send buffer is 64 MB. The buffer that is twice as large results in a larger execution time (almost twice larger), indicating that the larger the buffer,the longer Mimir postpones triggering the aggregate phase in the shuffling. At the same time, the low power usage does not require any intervention with PAPI's power capping.

In the second scenario, the buffer size of each process is fully used: the different unique KV pairs are filling it while the map operations are performed. Figure 4b shows an example for a small case study of two processes, each one with a chunk "abababab" words (where "a" and "b" are the unique words). The send buffer fills regularly, and the aggregate phase is triggered and interleaves with the map operations, resulting in the performance sweet spot. Figures 5e and 5b show the power usage when 5K unique words make up the entire 4 billion word dataset but two different send buffer sizes (i.e., 32 MB and 128 MB). The sweet spot occurs for the same number of unique words, independently from the buffer size. Moreover, the power usage is no longer implicitly defined but is explicitly set up by the PAPI power cap. As with the previous tests, lower power cap results in larger runtime. The performance degradation is within tolerable range.

In the third scenario, the buffer size of each process is overutilized: the different unique KV pairs do not fit in the single partitions of the send buffers. Figure 4c shows an example for a small case study of two processes, each one with the chunk "abacdabcd" (where "a," "b," "c," and "d" are the unique words). At time t, the send buffer fills with the "a" and "b" words, Mimir does not trigger the shuffling but continues the map operations. The buffer has to be copied to the memory. When new map operations on "a" and "b" words are performed, segments of the buffer have to be retrieved, as show in Figure 4d for a hypothetic time $t + 1$. We observe the same phenomena with datasets with large number of unique words (e.g., 42 million in Figures 5f and 5c). These two figures show the oscillatory behavior of both CPU and memory power usage over time when moving the buffer through the memory hierarchy (i.e., cache to memory and back). The larger the buffer, the larger the waving pattern observed and, once again, the larger the runtime.

Overhead of Monitoring Frequency: The last aspect we study in this paper is the overhead associated with measuring power usage with PAPI. We consider three sample granularities to collect the power measurements: 100-millisecond sample rate (the default setting), 50-millisecond sample rate, and 5-millisecond sample rate. Values measured by PAPI are energy consumption over the interval of 100 ms, 50 ms, and 5 ms, respectively. The smaller the interval (and thus the more samples collected), the more the overhead. At the same time, the smaller the interval, the more accurate the power values (i.e., less biased by outliers within the sample interval). We measure the power usage for two critical cases: the case in which the system is intrinsically capping the power (e.g., with only 72 unique words in Figures 6a, 6b, and 6c) and the case in which the system forwards parts of the send buffer down the memory hierarchy (e.g., with 42 mission unique words in Figures 6d, 6e, and 6f). The figures show us that the overhead associated with PAPI is marginal and that the average values at 100 ms are sufficiently close to the



(a) Underutilized buffer

(b) Fully utilized buffer

(c) Overutilized buffer (time t)

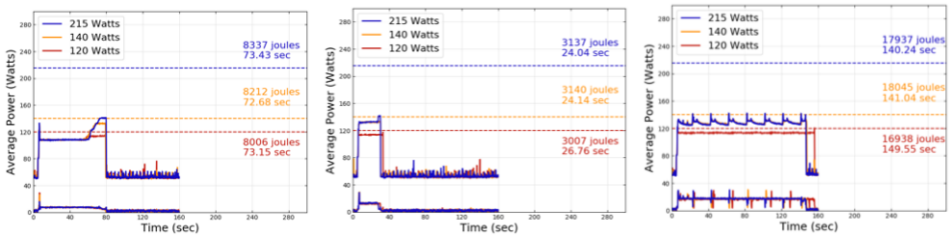
(d) Overutilized buffer (time t+1)

Figure 4. Small scale examples of underutilized, fully utilized, and overutilized send buffers in the map + aggregate phases of ReduceByKey.

values at 5 ms to conclude that power usage does not exhibit variability within the larger interval and there are no outliers within the 100 ms interval.

4. Lessons Learned

We have observed the following from our performance analysis. *First,* combiner optimizations offer significant power performance benefits for our miniapplications. We observe up to a 46% reduction in runtime and up to a 50% reduction in energy consumption when comparing results between GroupByKey and the combiner-optimized ReduceByKey on the same dataset. These results demonstrate the high power performance cost of data management on HPC systems, given the significant performance gain yielded by reducing data management via the combiner optimization. *Second,* he unoptimized reduce stage in the GroupByKey miniapplications has high runtime and energy costs. Comparing GroupByKey with the Map+Aggregate miniapplication, which stops before the reduce stage, shows costs of 333% to 355% more runtime and 335% to 359% more energy consumption when adding the reduce stage to the execution. *Third,* the Map+Aggregate and GroupByKey miniapplications suffer an increase of 5% to 33% more runtime when run under a 120-watt power cap. When running ReduceByKey, with



(a) 32 MB 4B72 (b) 128 MB 4B5K (c) 128 MB 4B42M

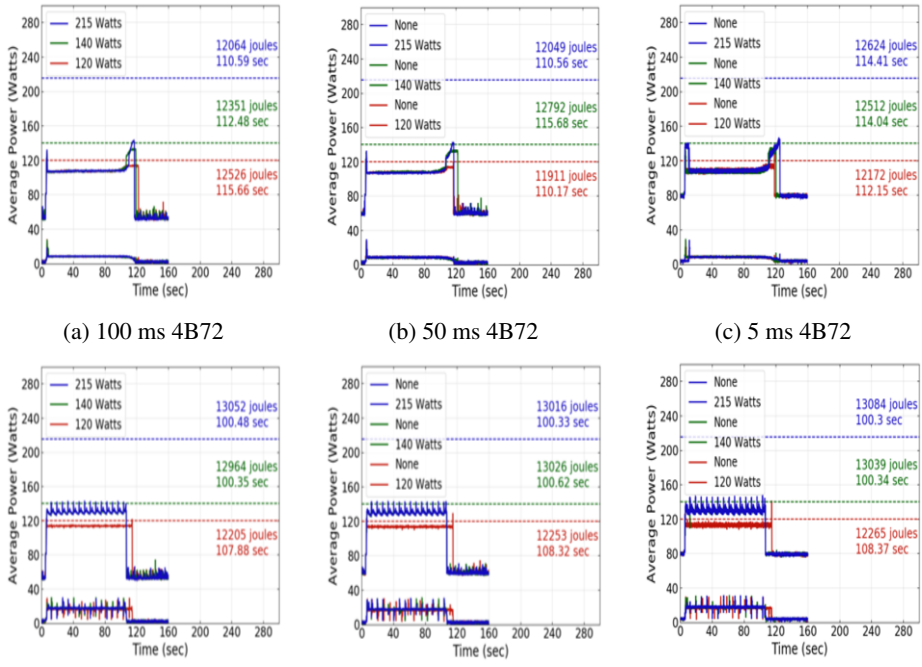
(d) 64 MB 4B72 (e) 32 MB 4B5K (f) 16 MB 4B42M

Figure 5. Power usage over time and runtime exhibiting underutilized buffers for 4 billion word datasets.

its combiner optimizations, over the same highly combinable dataset, we observe an implicit power cap, below 120 watts. This arises from the very high input dataset combinability and large communication buffer size. *Fourth,* the power performance of the combiner optimizations varies according to the combinability rate of the input dataset around a "sweet spot" region of minimum runtime and energy consumption. This sweet spot is at a combinability rate near 99%, or about 5,000 unique words for a dataset of 4 billion total words. At small numbers of unique words and high combinability rates, performance degrades because of the inability to fill the MPI communication buffers. At large numbers of unique words and low combinability rates, performance degrades because of the increased number of KVs that each process must manage. *Fifth,* the size of the Mimir communication buffer used is significant to the power performance of the combiner optimizations in the ReduceByKey miniapp. Performance degrades when the chosen buffer size is either over- or underutilized, depending on the combinability of the input data. *Last,* the overhead associated with PAPI measurement is marginal, as demonstrated by the lack of impact of decreasing the rate of sampling from 100 to 5 ms. Further, the 100 ms rate is sufficiently fine-grained, because decreasing the sample rate did not lead to greater variability in power usage over time. All these observations can serve as rules of thumb for effective data management using MapReduce over MPI programming languages on HPC architectures. Work in progress is integrating these lessons learned into Mimir, making the framework power-aware (i.e., able to leverage implicit power capping while still controlling power usage in the background, idle to the user).

5. Conclusion

In this paper, we quantitatively measure the impact of data management across cores on power usage for a set of MapReduce-based miniapps that are data intensive on two



(a) 100 ms 4B72 (b) 50 ms 4B72 (c) 5 ms 4B72

(d) 100 ms 4B42M (e) 50 ms 4B42M (f) 5 ms 4B42M

Figure 6. Power usage over time with different sample rates of 100 ms, 50 ms, and 5 ms, for a case study with intrinsically power capping (e.g., with only 72 unique words in Figures 6a, 6b, and 6c) and for a case study forwarding parts of the send buffer to memory during the map operations (e.g., with 42 million unique words in Figures 6d, 6e, and 6f).

many-core systems: KNL and KNM. Among our observations, we notice how combiner optimizations lead to up to a 46% reduction in runtime and a 50% reduction in energy usage, without the need for a power cap.

Our future work includes (1) understanding how far our observations are from a general principle relating power cap and performance; (2) studying ways of reducing data movement other than the combiner used in this paper; and (3) understanding how the settings of the underlying MapReduce framework can be tuned during runtime to extend the "sweet spot" regions (i.e., regions of minimum runtime and power usage).

6. Acknowledgments

This work was supported by NSF CCF 1841758.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] —, "Apahce Hadoop," `http://hadoop.apache.org/`.

[3] T. White, *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.

[4] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," *HotCloud*, vol. 10, p. 10, 2010.

[5] X. Lu, F. Liang, B. Wang, L. Zha, and Z. Xu, "DataMPI: Extending MPI to Hadoop-like big data computing," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS)*, 2014.

[6] N. Chaimov, A. Malony, S. Canon, C. Iancu, K. Z. Ibrahim, and J. Srinivasan, "Scaling Spark on HPC systems," in *25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2016, pp. 97–110.

[7] X. Yang, N. Liu, B. Feng, X.-H. Sun, and S. Zhou, "PortHadoop: Support direct HPC data processing in Hadoop," in *2015 IEEE International Conference on Big Data (Big Data)*, 2015, pp. 223–232.

[8] Y. Wang, R. Goldstone, W. Yu, and T. Wang, "Characterization and optimization of memory-resident MapReduce on HPC systems," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS)*, 2014, pp. 799–808.

[9] —, "IBM BG/Q Architecture," `https://www.alcf.anl.gov/files/IBM_BGQ_Architecture_0.pdf`.

[10] T. Gao, Y. Guo, B. Zhang, P. Cicotti, Y. Lu, P. Balaji, and M. Taufer, "Mimir: Memory-efficient and scalable MapReduce for large supercomputing systems," in *Proceedings of the 31th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017.

[11] T. Gao, Y. Guo, B. Zhang, P. Cicotti, Y. Lu, P. Balaji, and M. Taufer, "On the power of combiner optimizations in MapReduce over MPI workflows." in *Proceedings of the IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, 2018.

[12] T. Gao, Y. Guo, B. Zhang, P. Cicotti, Y. Lu, P. Balaji, and M. Taufer, "Memory-Efficient and Skew-Tolerant MapReduce over MPI for Supercomputing Systems" in *IEEE Transactions on Parallel and Distributed Systems (IEEE TPDS)*, 2019.

[13] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with PAPI-C," pp. 157–173, 2010.

[14] A. Haidar and et al., "Investigating power capping toward energy-efficient scientific applications," *Concurrency Computat Pract Exper.*, 2018.

[15] A. Haidar, H. Jagode, A. YarKhan, P. Vaccaro, S. Tomov, and J. Dongarra, "Power-aware computing: Measurement, control, and performance analysis for intel xeon phi," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep. 2017, pp. 1–7.

[16] F. Ahmad, S. Lee, M. Thottethodi, and T. N. Vijaykumar, "PUMA: Purdue MapReduce Benchmarks Suite," *Technical Report 437, Purdue University*, 2012.

[17] —, "Mimir: MapReduce over MPI," `https://github.com/TauferLab/Mimir-dev.git`.

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200054

Feedback-Driven Performance and Precision Tuning for Automatic Fixed Point Exploitation

Daniele CATTANEO Michele CHIARI Stefano CHERUBIN Antonio DI BELLO and Giovanni AGOSTA
Dipartimento di Elettronica, Informazione e Bioingegneria – Politecnico di Milano

Abstract. Precision tuning is an emerging class of techniques that leverage the trade-off between accuracy and performance in a wide range of numerical applications. We employ TAFFO, a compiler-based state-of-the-art framework that relies on fixed point representations to perform precision tuning. It converts floating-point computations into a fixed point version with comparable semantics, in order to obtain performance improvements. Usually, the process of fixed point type selection aims at the minimization of the round-off error introduced by the precision reduction. However, this approach introduces a large number of type cast operations, generating an overhead that may overcome the performance improvements of the conversion to fixed point formats. We propose a control loop architecture that exploits the static analyses provided by TAFFO to reduce the number of type cast operations while keeping the error under a given threshold. We evaluate our approach on three benchmarks of the AXBENCH suite, and we show that in all cases we are able to achieve performance improvements while keeping the introduced numerical error below the given tolerance threshold.

Keywords. precision tuning, fixed point, error estimation, performance estimation

1. Introduction

The scale of computer applications has been steadily increasing across all domains, from embedded systems to High Performance Computing (HPC). In the past, the Dennard scaling and Moore's Law have been the enabling factors for this growth, allowing application developers – especially in High Performance Computing – to reduce the effort spent in fine tuning the resource usage of applications [12]. However, their end has ushered in a new stage in application development, where careful allocation of computational resources is more rewarding than in the past.

In particular, in HPC application development it is common practice to oversize the data types with respect to the accuracy of the results needed by the application. In fact, tuning the size of data types is a time-consuming and error-prone task. In the context of resource-constrained embedded systems, it is customarily performed manually. However, in HPC application development such methods are not feasible due to the scale of the applications and their data sets. To relieve the programmer from this task, we introduced in our previous work [6,4] a compiler-based precision tuning assistant toolchain.

Subsequent evolutions lead us to the development of the state-of-the-art precision tuner based on the LLVM framework, TAFFO [7].

TAFFO performs precision tuning mainly by exploiting the fixed point numerical representation. Fixed point representations are an important resource in application development whenever the need to overcome computational resource limitations emerges. Such representations are predominantly employed in embedded applications. Additionally, they are also exploited as a mean to data size tuning for HPC tasks. TAFFO receives programmer hints as input and it later infers value range information on the data flows in a compilation unit. Then, it transforms the code to use the most appropriate fixed point data types at the intermediate representation level. It is robust enough to support automated conversion for complex C++ benchmarks without rewriting the computational kernels into less expressive languages, such as ANSI C. It is also able to operate both on parallel and serial code.

However, adopting fixed point data types requires fine tuning to achieve performance benefits. In fact, optimizing the allocation of data types to minimize precision loss will impact the execution time, because of the increased number of data type casts, i.e. conversions between different fixed point types. Additionally, we have to consider that some architectures are more suited to fixed point computations than others. To address the challenge of controlling the performance benefits of the floating point to fixed point conversion, we propose as the main contribution of this work a control-loop regulation approach to adjust the adverse effects of the precision tuning task. This control loop leverages a performance and accuracy estimation pass, tailored to the TAFFO toolchain. We call this new step *Feedback Estimator*. The Feedback Estimator is based on a combination of machine learning techniques, and traditional static control flow analyses. The control loop uses the data collected by the Feedback Estimator to improve the floating point to fixed point transformation, by making the chosen precision mix more homogeneous, thus minimizing the number of data type casts. After the aforementioned improvements, the compilation process is repeated, realizing a feedback process between the mixed-precision compiler transformation and the performance evaluation component.

We verify the effectiveness of the Feedback Estimator through a meaningful subset of AXBENCH [23], a well-known approximate computing benchmark suite. In all the benchmarks we considered, we were able to significantly reduce the amount of type cast operations, without significantly compromising the accuracy of the computation, which remains within a satisfactory threshold provided by the user. The reduction of the number of type casts results in a direct reduction of the number of instructions in the program, thus improving the performances of the converted code.

The rest of the article is organized as follows: in Section 2 we describe the main existing solutions concerning this problem, in Section 3 we describe the approach we propose in more detail, in Section 4 we show the results of the application of our technique to selected AXBENCH benchmarks, and we give our concluding remarks in Section 5.

2. Related Works

This work places itself in the prolific field of *reduced precision computation* and in particular, *static precision tuning*. Tools in the state-of-the-art are aimed at automatically producing an optimized version of a given numerical program, that sacrifices computa-

tion accuracy to obtain performance gains. Such tools either target the entire program [17,15,19,2], or just computational kernels identified by the user [16,22,20,9]. Performance gains are obtained by using smaller data types, by using fixed point in place of floating point computations, or both. In order to apply this transformation without excessively degrading their accuracy, the precision requirements on the numerical computations must be evaluated, either dynamically [8,19], or statically [9]. An instruction-wise estimation of such requirements allows a very tailored choice of the data types to be used, allowing to minimize data width while keeping a sufficient accuracy. However, this may result in a very heterogeneous precision mix, which requires a very frequent introduction of cast instructions (i.e. bit shifts, when using fixed point types), every time a type mismatch arises in the data flow graph. As a result, the performances of the optimized code degrade, possibly nullifying the gains caused by smaller type width. Also, a high variety of data types in the precision mix may decrease the vectorization opportunities for architectures supporting SIMD instructions.

Different approaches have been proposed in the literature to measure and to limit this overhead. *FRIDGE* [16], *Precimonious* [22], *CRAFT* [18], and *PetaBricks* [2] perform a dynamic estimation of the obtained performance gains by executing and profiling the optimized code on a representative input dataset. This solution may, however, not always be feasible, due to the time required to perform the profiling, or to the unavailability of a sufficiently representative input dataset. Alternatively, the overhead can be estimated a priori via heuristics, such as the number of cast instructions introduced by the code conversion. For example, *FPTuner* [8] exposes a user-defined threshold for the amount of type casts that the tool may insert into the code. This approach has the drawback of requiring a certain skill for the user to pick the threshold. *Autoscaler for C* [17] and other works [19] iteratively optimize the fixed point code by reordering instructions to remove the shift operations whenever possible. *Daisy* [9] estimates the profitability of type transformations by means of a cost function based on the number of cast instructions. Finally, *HiFPTuner* [15] minimizes the number of cast operations by building a data-dependency tree, and trying to assign the same data type to all values in the same cut of the tree.

An excessive reduction of precision mix heterogeneity may severely degrade computation accuracy. To pursue a reasonable trade-off between these two goals, precision tuning tools need to estimate the numerical error introduced by the transformation. *FRIDGE*, *Precimonious*, *CRAFT*, *Autoscaler for C*, and *HiFPTuner* decide whether the accuracy degradation is acceptable by performing an explorative run of the reduced precision version on a representative input dataset. The reliability of this approach depends on the extent to which the validation dataset covers the range of possible real inputs. Other tools, such as *Daisy*, perform a conservative static estimation of the error bounds by means of error propagation techniques.

3. Proposed Solution

We propose an extension of the TAFFO framework that implements a control loop regulation to adjust the effects of the precision tuning task. TAFFO is composed by five stages, namely code pre-processing, value range analysis, data type allocation, code conversion, and feedback estimation. As shown in Figure 1, our control loop design uses the feedback estimation stage to understand whether the proposed mixed precision version should be

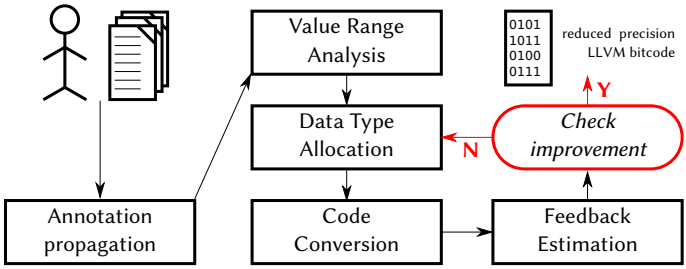


Figure 1. A flow-chart detailing the overall architecture of TAFFO. The extension with the control loop regulation over the latest components of the toolchain is highligthed in red.

improved or not. The control loop acts on the data type allocation stage. In particular, we expose a parameter q from that stage that represents the granularity of the bit partitioning for the fixed point data types.

The goal of this control loop regulation is to maximizing the performance improvement while keeping the error within an acceptable threshold. This task entails the minimization of the number of type cast operations in the final mixed precision code. We consider the number of type casts that are statically present in the program, as opposed to the number of type casts actually executed. The trivial solution of this minimization problem would be an uniform bit partitioning across the whole program. However, the uniform bit partitioning in the data type allocation stage would significantly impact on the error, which may exceed the given threshold. TAFFO already provides a fine-grained bit partitioning in the data type allocation stage. We aim at iteratively reducing the granularity of this allocation to limit the number of bit shift instructions. This new parameter q of the data type allocation can be interpreted as a similarity threshold. Whenever the distance between two fixed point bit partitioning p_1 and p_2 is lower than q, then p_1 and p_2 can be merged into a single bit partitioning p_{12}.

3.1. Similarity distance

Let p_1 and p_2 be two fixed point bit partitionings of the same total width, and let f_1 and f_2 be their respective number of fractional bits, defining the place of the decimal delimiter. The similarity distance between them is defined as $|f_1 - f_2|$. This definition of the distance between two types allows the data type allocator to remove type cast instructions while keeping a limit on the additional error introduced. The order of magnitude of the latter is directly proportional to q, the maximum distance between two types that allows them to be merged into a single one. The resulting type is the one among the two that has the highest number of integer bits (and so the minimum number of fractional bits), so that no potential overflows are introduced.

3.2. The Feedback Analyses

TAFFO implements two kinds of analyses in its feedback estimation step. The first one is a functional analysis of the mixed precision code. It is named *Error estimation* and it evaluates the impact of the round-off error due to the real number representation for

each intermediate and output value. It propagates rounding errors represented as Affine Forms [11,10], based on the variable ranges estimated by the value range analysis component. The second analysis classifies the mixed precision version on the basis of the expected speedup with respect to the original version. This *performance estimation* step predicts whether the mixed precision version is going to be much slower (speedup < 0.8), much faster (speedup ≥ 1.2), or almost the same of the original version. It is based on a *Gradient Boosting* classifier [14], provided by the machine learning framework `scikit-learn` [21].

Although the error estimation provides a conservative over-approximation of the round-off error, it captures the trend of the actual round-off error at runtime. Figure 2, Figure 3, and Figure 4 reflect this property. We want to save compilation time by avoiding the code generation and execution of mixed precision versions that are likely to be not profitable. Therefore, the minimization of the estimated error is a good proxy for the minimization of the actual error at runtime. On the contrary, the TAFFO performance estimation only provides a coarse-grained classification. The difference between the optimal solution and a solution that is close to the optimal is not likely to be captured by this classification. Thus, the TAFFO classification does not represent a metric which is sufficient to drive the regulator from the performance point of view.

We compute the number of type cast instruction that are removed by the merge of fixed point bit partitioning in the data type allocation stage by using the exposed parameter q. This metric is monotonous non-decreasing with respect to q. As this metric represents the number of instructions that were removed from the application, we design an heuristic regulation function that assumes a positive correlation between the number of removed type cast and the speedup.

3.3. Regulation Policy

The purpose of the regulation policy is to try to achieve a significant speedup, while maintaining the error within acceptable bounds. The user is required to provide a bound e_{max} for the maximum acceptable absolute error on the output values. Then, two different settings are available for the policy: it can be set to either maximize speedup, or minimize error. In the former case, it explores values of q starting from $q = 32$, and decreases q until the estimated error becomes lower than e_{max}. If the speedup is deemed negative at $q = 32$, or if it is still negative when the error reaches e_{max}, it means it is not possible to achieve a speedup while keeping the error acceptable. In this case, the program is not converted to fixed point format. The pseudocode in Algorithm 1 formalizes this description.

Algorithm 1.: Performance Maximization

```
q ← 32
error ← estimate_error(q)
speedup ← estimate_speedup(q)
while error > e_max and speedup == faster do
    q ← q − 1
    error ← estimate_error(q)
    speedup ← estimate_speedup(q)
end while
if speedup == faster and error ≤ e_max then
    return q
else
    return −1
end if
```

Algorithm 2.: Error Minimization

```
q ← 0
error ← estimate_error(q)
speedup ← estimate_speedup(q)
while error ≤ e_max and speedup ≠ faster do
    q ← q + 1
    error ← estimate_error(q)
    speedup ← estimate_speedup(q)
end while
if speedup == faster and error ≤ e_max then
    return q
else
    return −1
end if
```

The setting that minimizes error is essentially symmetric, since it starts from $q = 0$, and it increases it until the estimated speedup becomes greater than 1.2, while the error remains $\leq e_{max}$, as we can see in Algorithm 2.

4. Evaluation

We evaluated our feedback-driven approach on three benchmarks form AxBench [23], a popular approximate computing benchmark suite. The benchmarks we chose, which are the implementations of real-world numerical algorithms from different domains, are *Black-Scholes*, *FFT* and *K-means*. Below we describe the results we obtained for each benchmark, and the behavior of the regulation policy.

4.1. Black-Scholes

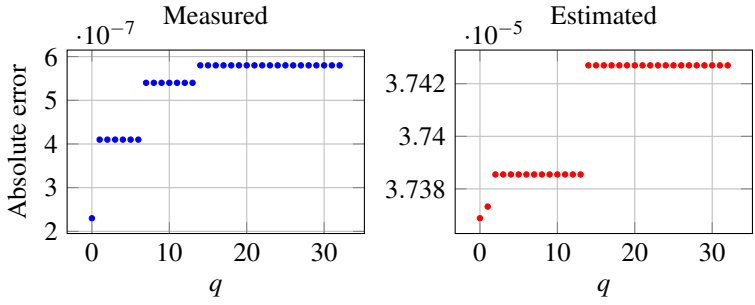


Figure 2. Measured and estimated error for the *Black-Scholes* benchmark.

Black-Scholes is a financial application that numerically computes the equation for the value of European call options according to the Black-Scholes model of a financial market. Its input dataset consists of 48,000 options. The accuracy of the optimized version is evaluated by computing the average absolute error of its output with respect to the floating point version.

Figure 2 shows the measured (left) and estimated (right) absolute errors with respect to parameter q. Note that the feedback analysis overestimates the absolute error by two orders of magnitude, which is in line with other results obtained with the technique we used [10]. Nevertheless, the estimated error consistently follows the shape of the measured one when varying parameter q. In the few cases it does not, the error bound is still conservative. Thus, it is possible to use it to tune parameter q, in order to improve performance. The performance estimator predicts a positive speedup for all values of q. If the regulation policy is set to maximize accuracy, the framework chooses $q = 0$ as the final parameter setting. If, on the contrary, it is set to maximize performance, it chooses $q = 32$, as the estimated error remains acceptable.

The number of removed casts, which is shown in Figure 5, increases with q, and its variation with respect to q is consistent with the absolute error. When $q = 32$, all casts are removed, which ensures that there is a performance improvement, due to the lower number of instructions involved in the computation. In all benchmarks, the maximum value of q is 32, because this is the width of all fixed point data types used.

Figure 6 shows the relation between the number of removed casts and the measured relative error on the output. Clearly, from the point of view of numerical accuracy Black-Scholes is not very sensitive to the removal of cast instructions, as its relative error remains well below 1%, even when removing all casts. This allows the optimized version of the benchmark to achieve the maximum performance improvement.

4.2. FFT

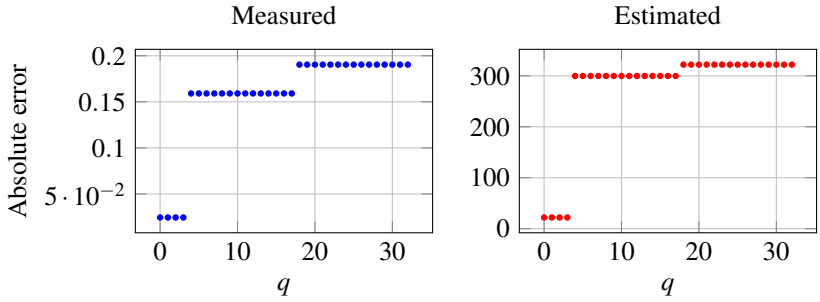


Figure 3. Measured and estimated error for the *FFT* benchmark.

FFT is an implementation of the Radix-2 Cooley-Tukey Fast Fourier Transform, an algorithm widely used in signal processing. It receives as an input signal a discrete rectangular wave of period K and duty cycle 1% in the time domain, and converts it into the frequency domain. Again, the output accuracy is measured by computing the absolute error.

The measured and estimated errors are reported in Figure 3. This time, the estimated error becomes extremely high for $q \geq 4$, exceeding the user-defined error threshold, which is $e_{max} = 50Hz$, corresponding to a relative error around 1%. The regulation policy chooses $q = 3$ when optimizing performance, thus removing around 19% of cast instructions. This is a rather significant improvement, even if the value of q remains low. Figure 6 shows how the measured relative error approaches and becomes greater than 1% as the amount of removed casts gets higher than around 19%.

Instead, $q = 0$ is chosen when optimizing error, since the speedup due to the sole conversion of floating point computations to fixed point types is still estimated as high.

Note that, according to Figure 5, even with $q = 32$, only 37.5% of the cast instructions are removed. This is due to the fact that the data type allocation stage always refrains from merging two types when this operation could potentially cause overflows during the execution, according to the value ranges estimated for each variable. This makes sure the accuracy reductions due to the optimization are gradual, and do not compromise the correctness of the program completely.

4.3. K-means

K-means uses a popular machine learning algorithm to classify pixels from an image into a user-specified number of clusters. As an input dataset for its evaluation, we use the one provided by AxBench, i.e. a set of RGB pictures. The error introduced by the fixed-point optimization is measured and estimated on the Euclidean distance between single

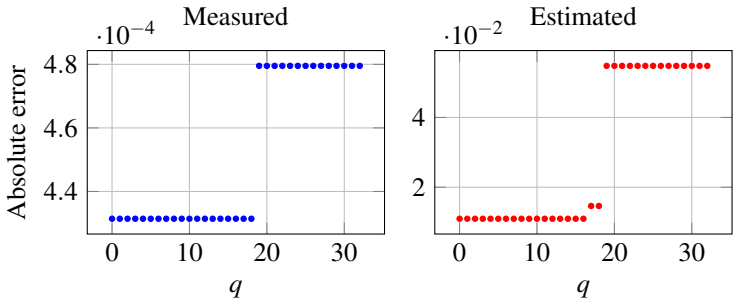


Figure 4. Measured and estimated error for the *K-means* benchmark.

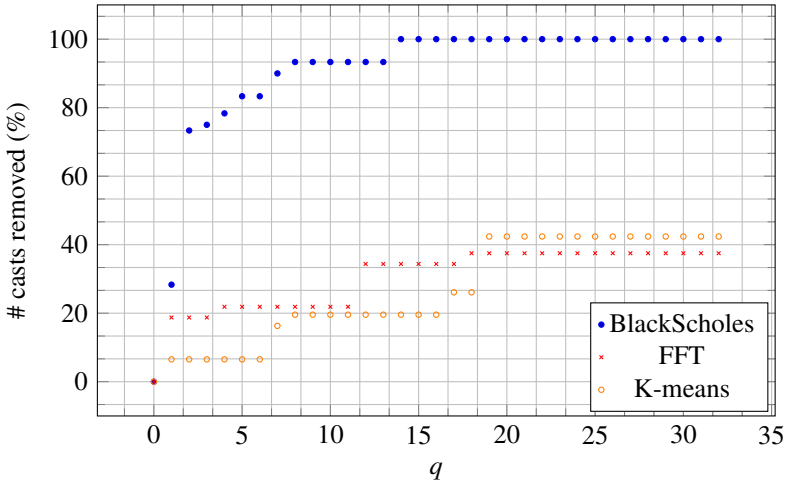


Figure 5. Percentage of type cast instructions removed for all valid values of q, with respect to the number of type casts when $q = 0$.

pixels and cluster centroids. The distance results from the main computational kernel of the application, and it determines the classification of each pixel in its cluster. Its error is thus significantly representative of the cluster misclassification rate introduced by the optimization.

In this case, the only significant change in accuracy occurs between $q = 18$ and $q = 19$, for both the measured and the estimated error. The chosen error threshold is $e_{max} = 2 \cdot 10^{-2}$, and the speedup is always estimated greater than 1.2. Therefore, the policy chooses $q = 0$ when optimizing for accuracy, and $q = 18$ when optimizing for speedup, thus removing 26% of the cast instructions. The number of removed cast instruction is sensible for this benchmark, too. Again, a number of cast instructions cannot be removed even with $q = 32$, due to overflow concerns.

5. Conclusion

In this paper, we presented a major extension of the TAFFO framework for precision tuning. In particular, we introduced a control loop for data type selection, which is governed by a feedback estimation component. The proposed modifications enable TAFFO to re-

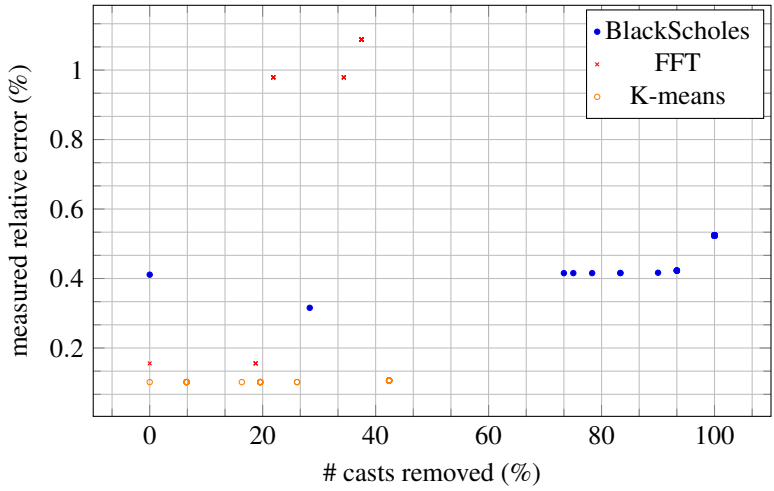


Figure 6. Measured relative error with respect to the amount of cast instructions removed.

duce the number of data type casts. This effect is achieved by merging similar fixed point configurations whenever the impact of such merge is zero or particularly small. This feature enables significant performance improvements. An experimental campaign carried out on the AXBENCH benchmark suite for approximate computing, restricted to the benchmarks that include significant floating point computations, shows the effectiveness of the proposed approach. Indeed, TAFFO is able to explore the approximation options, to correctly estimate the error introduced by different levels of optimization (corresponding to the aggressiveness of the data type casts removal), and to identify the best solution in performance at the requested accuracy level. As a result, when imposing an accuracy threshold of 1% numerical error, TAFFO produces an optimized version with a number of cast instructions between 19% and 100% lower than the baseline version with the default precision mix, resulting in a performance speedup, due to the reduced number of instructions.

As future development, we plan to extend the set of data types managed by TAFFO with the half-precision floating point `bfloat16` [1] and arbitrary precision data types [13,3]. This extension entails the porting of all the analyses and transformations to generic data types, but will expand usefulness of TAFFO to many HPC use cases, such as simulations of chaotic systems. An additional development line involves the coupling of the TAFFO framework with a dynamic partial re-compilation framework such as LIBVC [5] to implement dynamic precision tuning.

References

[1] BFLOAT16 – Hardware Numerics Definitions. Technical report, Intel Corporation, Nov 2018.

[2] J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms. In *International Symposium on Code Generation and Optimization (CGO 2011)*, pages 85–96, April 2011.

[3] D. H. Bailey. A thread-safe arbitrary precision computation package (full documentation). Technical report, Mar 2017.

[4] D. Cattaneo, A. Di Bello, S. Cherubin, F. Terraneo, and G. Agosta. Embedded operating system optimization through floating to fixed point compiler transformation. In *21st Euromicro Conference on Digital System Design (DSD)*, pages 172–176, Aug 2018.

[5] S. Cherubin and G. Agosta. libVersioningCompiler: An easy-to-use library for dynamic generation and invocation of multiple code versions. *SoftwareX*, 7:95 – 100, 2018.

[6] S. Cherubin, G. Agosta, I. Lasri, E. Rohou, and O. Sentieys. Implications of Reduced-Precision Computations in HPC: Performance, Energy and Error. In *International Conference on Parallel Computing (ParCo)*, Sep 2017.

[7] S. Cherubin, D. Cattaneo, M. Chiari, A. Di Bello, and G. Agosta. TAFFO: Tuning assistant for floating to fixed point optimization. *IEEE Embedded Systems Letters*, 2019.

[8] W.-F. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamarić. Rigorous floating-point mixed-precision tuning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 300–315, 2017.

[9] E. Darulova, E. Horn, and S. Sharma. Sound mixed-precision optimization with rewriting. In *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems*, ICCPS '18, pages 208–219, 2018.

[10] E. Darulova and V. Kuncak. Trustworthy numerical computation in scala. *SIGPLAN Not.*, 46(10):325–344, Oct. 2011.

[11] L. H. de Figueiredo and J. Stolfi. Affine arithmetic: Concepts and applications. *Numerical Algorithms*, 37(1):147–158, Dec 2004.

[12] M. Duranton, K. De Bosschere, B. Coppens, C. Gamrat, M. Gray, H. Munk, E. Ozer, T. Vardanega, and O. Zendra. *The HiPEAC Vision 2019*. HiPEAC CSA, Jan. 2019.

[13] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. Mpfr: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33(2), June 2007.

[14] J. H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[15] H. Guo and C. Rubio-González. Exploiting community structure for floating-point precision tuning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 333–343, New York, NY, USA, 2018. ACM.

[16] H. Keding, M. Willems, M. Coors, and H. Meyr. FRIDGE: A fixed-point design and simulation environment. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '98, pages 429–435, 1998.

[17] K.-I. Kum, J. Kang, and W. Sung. AUTOSCALER for C: an optimizing floating-point to integer C program converter for fixed-point digital signal processors. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 47(9):840–848, Sept 2000.

[18] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. Legendre. Automatically adapting programs for mixed-precision floating-point computation. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 369–378, 2013.

[19] D. Menard, D. Chillet, F. Charot, and O. Sentieys. Automatic floating-point to fixed-point conversion for dsp code generation. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '02, pages 270–276, 2002.

[20] R. Nobre, L. Reis, J. a. Bispo, T. Carvalho, J. a. M. P. Cardoso, S. Cherubin, and G. Agosta. Aspect-driven mixed-precision tuning targeting gpus. In *Proceedings of the 9th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and the 7th Workshop on Design Tools and Architectures For Multicore Embedded Computing Platforms*, PARMA-DITAM '18, pages 26–31, Jan 2018.

[21] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[22] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. Precimonious: Tuning assistant for floating-point precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 27:1–27:12, Nov 2013.

[23] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran. AxBench: A multiplatform benchmark suite for approximate computing. *IEEE Design Test*, 34(2):60–68, April 2017.

Parallel Programming

This page intentionally left blank

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200056

A GPU-CUDA Framework for Solving a Two-Dimensional Inverse Anomalous Diffusion Problem

P. De Luca [a], A. Galletti [b], H. R. Ghehsareh [c], L. Marcellino [b] and M. Raei [c]
[a] *University of Salerno, Department of Computer Science, Fisciano, Italy*
[b] *University of Naples Parthenope, Department of Science and Technology, Italy*
[c] *Malek Ashtar University of Technology, Department of Mathematics, Isfahan, Iran*

Abstract. This paper deals with the solution of an inverse time fractional diffusion equation described by a Caputo fractional derivative. Numerical simulations, involving large domains, give rise to a huge practical problem. Hence, by starting from an accurate meshless localized collocation method using radial basis functions (RBFs), here we propose a fast algorithm which exploits the GPU-CUDA capabilities. More in detail, we first developed a C code which uses the well-known numerical library LAPACK to perform basic linear algebra operations in order to implement an efficient sequential algorithm. Then we propose a GPU software based on ad hoc parallel CUDA-kernels and efficient usage of parallel numerical libraries available for GPUs. Performance analysis will show the reliability and the efficiency of the proposed parallel implementation.

Keywords. Fractional calculus; GPU computing; parallel algorithms; CUDA

1. Introduction

In recent decades, fractional calculus theory received high interest due to its application in several fields in science and engineering. Indeed, fractional models are beneficial and powerful mathematical tools to describe the inherent properties of processes in mechanics, chemistry, physics, and other sciences [1,2,3]. Main methods to solve fractional models include meshless and radial basis functions based methods which represent reliable and efficient techniques, specially suitable for high-dimensional and irregular computational domains [4,5]. This choice is because they allow to avoid the expensive task related to the mesh construction. However, as is well-known, the use of global RBFs in such cases gives rise to very ill-conditioned discrete problems. Moreover, practical problems with large domains increase the computational cost dramatically. Therefore, the use of fast algorithms and parallel computational kernels becomes unavoidable and necessary. So, in this work, by starting from an accurate meshless localized collocation method using local radial point interpolation (LRPI), we propose a fast algorithm which exploits the GPU-CUDA capabilities. More in detail, we first developed a C code based

on the numerical library LAPACK to perform all basic linear algebra operations. Then, we moved on a GPU-parallel software based on ad hoc parallel CUDA-kernels and efficient usage of parallel numerical libraries available on CUDA (Compute Unified Device Architecture) environment. To be specific, to improve performances, we introduce a parallel approach which implements parallel modules by using the CUDA computing platform for GPUs [6] (a massively parallel architecture, also know as Graphic Cards, for general purpose computing). Moreover, this GPU-parallel software involves the use of the cuSOLVER library [7], which in turn provides all the functions of LAPACK for the GPU environment, and it includes a easy-to-use interface making possible to vary both the main parameters of the problem and the blocks and threads configuration which are required by the parallel execution.

The rest of the paper is organized as follows: section 2 briefly describes the problem we deal with and the numerical procedure to discretize the related inverse time fractional diffusion equation; in section 3 the description of both sequential and parallel implementation details of the algorithms is given; in section 4 the experimental results highlight the performance gain, in terms of execution times and speed-up, compared to the sequential version; finally, conclusions close the paper in section 5.

2. Mathematical model and numerical procedure details

In the current work we deal with the solution of a two-dimensional inverse time fractional diffusion equation [8,9,10], defined as follows:

$$ {}_0^cD_t^\alpha v(\mathbf{x},t) = \kappa \Delta v(\mathbf{x},t) + f(\mathbf{x},t), \qquad \mathbf{x} = (x,y) \in \Omega \subseteq R^2, \quad t \in]0,T], \tag{1} $$

with the initial and Dirichlet boundary conditions:

$$ \begin{aligned} &v(\mathbf{x},0) = \varphi(\mathbf{x}), && \mathbf{x} \in \Omega, \\ &v(\mathbf{x},t) = \psi_1(\mathbf{x},t), && \mathbf{x} \in \Gamma_1, \quad t \in]0,T], \\ &v(\mathbf{x},t) = \psi_2(\mathbf{x})\rho(t), && \mathbf{x} \in \Gamma_2, \quad t \in]0,T], \end{aligned} \tag{2} $$

and the non-local boundary condition:

$$ \iint_\Omega v(\mathbf{x},t)d\mathbf{x} = h(t), \qquad t \in]0,T], \tag{3} $$

where $v(\mathbf{x},t)$ and $\rho(t)$ are unknown functions and ${}_0^cD_t^\alpha = \frac{\partial^\alpha}{\partial t^\alpha}$ denotes the Caputo fractional derivative of order $\alpha \in]0,1]$. The numerical approach to discretize the problem (1) is summarized as follows [8,9].

2.1 The time discretization approximation

Let us choose a time step $\tau > 0$ and set $t^n = n\tau$, for $n = 0,\ldots,T/\tau$ (assume T/τ be an integer). By substituting $t = t^{n+1}$ in the equation (1), the following relation is obtained:

$$ {}_0^cD_t^\alpha v(\mathbf{x},t^{n+1}) = \kappa \Delta v(\mathbf{x},t^{n+1}) + f(\mathbf{x},t^{n+1}), \quad (\mathbf{x},t^{n+1}) \in \Omega \times (0,T]. \tag{4} $$

Therefore, we exploit the following second-order time discretization for the Caputo derivative of $v(\mathbf{x},t)$ at point $t=t^{n+1}$ [11,12]:

$$\left[{}_0^cD_t^\alpha v(\mathbf{x},t)\right]_{t=t^{n+1}} = \sum_{j=0}^{n+1} \frac{\omega^\alpha(j)}{\tau^\alpha} v(\mathbf{x},t^{n+1-j}) - \frac{t^{-\alpha}}{\Gamma(1-\alpha)} v(\mathbf{x},0) + O(\tau^2), \tag{5}$$

where:

$$\omega^\alpha(j) = \begin{cases} \dfrac{\alpha+2}{2} p_0^\alpha, & j=0, \\ \dfrac{\alpha+2}{2} p_j^\alpha - \dfrac{\alpha}{2} p_{j-1}^\alpha, & j>0, \end{cases} \quad \text{and} \quad p_j^\alpha = \begin{cases} 1, & j=0, \\ \left(1-\dfrac{\alpha+1}{j}\right) p_{j-1}^\alpha, & j\geq 1. \end{cases}$$

By substituting the equation (5) in (4) the following relation is obtained:

$$\frac{\omega^\alpha(0)}{\tau^\alpha} v^{n+1} - \kappa \Delta v^{n+1} = -\sum_{j=1}^{n} \frac{\omega^\alpha(j)}{\tau^\alpha} v^{n+1-j} + \frac{t^{-\alpha}}{\Gamma(1-\alpha)} v^0 + f^{n+1}, \tag{6}$$

with $f^{n+1} = f(\mathbf{x},t^{n+1})$ and $v^{n+1-j} = v(\mathbf{x},t^{n+1-j})$ $(j=0,\ldots,n+1)$.

2.2 The meshless localized collocation method

This section describes the meshless localized collocation approach. This choice is because, in last decades, meshless methods have been employed successfully in several fields of science and engineering [4] and allow to avoid the expensive task related to the mesh construction. In the meshless localized collocation method, the global domain Ω is partitioned into local sub-domains Ω_i $(i=1,\ldots,N)$ corresponding to every point. These sub-domains ordinarily are circles or squares and cover the entire global domain Ω. Then the radial point interpolation shape functions, ϕ_i, are constructed locally over each Ω_i by combining radial basis functions and the monomial basis function [10] corresponding to each local field point $\mathbf{x}_i$. In the current work, it is used one of the most popular RBFs, i.e., the generalized multiquadric radial basis function (GMQ-RBF) $\phi(r) = (r^2+c^2)^q$ (q=2.5) where c is the shape parameter. The local radial point interpolation shape function generates the $N \times N$ sparse matrix Φ. Therefore v can be approximated by:

$$v(\mathbf{x}) = \sum_{i=1}^{N} \phi_i(\mathbf{x}) v_i \tag{7}$$

where $\phi_i(\mathbf{x}) = \phi(\|\mathbf{x}-\mathbf{x}_i\|_2)$ (the norm $\|\mathbf{x}-\mathbf{x}_i\|_2$ denotes the Euclidean distance between $\mathbf{x}$ and field point $\mathbf{x}_i$). Substituting formula (7) in equations (6), (2) and (3) yields:

$$\sum_{i=1}^{N} \left[\frac{\omega^\alpha(0)}{\tau^\alpha} \phi_i(\mathbf{x}_j) - \kappa \left[\frac{\partial^2 \phi_i}{\partial x^2} + \frac{\partial^2 \phi_i}{\partial y^2} \right] (\mathbf{x}_j) \right] v_i^{n+1} = -\sum_{j=1}^{n} \frac{\omega^\alpha(j)}{\tau^\alpha} \sum_{i=1}^{N} \phi_i(\mathbf{x}_j) v_i^{n+1-j}$$

$$+ \frac{t^{-\alpha}}{\Gamma(1-\alpha)} \sum_{i=1}^{N} \phi_i(\mathbf{x}_j) v^0 + f^{n+1}, \qquad j=1,\ldots,N_\Omega \tag{8}$$

$$\sum_{i=1}^{N} \phi_i(\mathbf{x}_j) v_i^{n+1} = \psi_1^{n+1}(\mathbf{x}_j), \qquad j=N_\Omega+1,\ldots,N_\Omega+N_{\Gamma_1}, \tag{9}$$

$$\sum_{i=1}^{N} \phi_i(\mathbf{x}_j) v_i^{n+1} = \psi_2^{n+1}(\mathbf{x}_j)\rho^{n+1}, \quad j = N_\Omega + N_{\Gamma_1} + 1, \ldots, N_\Omega + N_{\Gamma_1} + N_{\Gamma_2}, \tag{10}$$

$$\sum_{i=1}^{N} \left(\int_\Omega \phi_i(\mathbf{x}) d\Omega \right) v_i^{n+1} = h^{n+1}. \tag{11}$$

The collocation equations (8) are referred to the N_Ω interior points in Ω, while the N_{Γ_1} equations (9) and the N_{Γ_2} equations (10) (involving also the unknown $\rho^{n+1} = \rho(t^{n+1})$) arise from the initial and Dirichlet boundary conditions. Finally, a further equation is obtained by means of 2D Gaussian-Legendre quadrature rules of order 15. Therefore, the time discretization approximation and the local collocation strategy construct a linear system of $N+1$ linear equations with $N+1$ unknown coefficients ($N = N_\Omega + N_{\Gamma_1} + N_{\Gamma_2}$). The unknown coefficients $\mathbf{v}^{(n+1)} = (v_1^{n+1}, \ldots, v_N^{n+1}, \rho^{n+1})$ are obtained by solving the sparse linear system:

$$\mathscr{A}\mathbf{v}^{(n+1)} = \mathscr{B}^{(n+1)}, \tag{12}$$

where $\mathscr{A}$ is a $(N+1)\times(N+1)$ coefficient matrix and $\mathscr{B}^{(n+1)}$ is a $(N+1)$ vector. Let us notice that, unlike $\mathscr{B}^{(n+1)}$, the coefficient matrix $\mathscr{A}$ does not change its entries along time steps. Moreover, due to the local approach the coefficient matrix $\mathscr{A}$ is sparse.
Previous discussion can be summarized through the following scheme, Algorithm 1, which describes the main steps of the numerical procedure.

Algorithm 1 Pseudo-code for problem (1)

`Input:` $\kappa, \alpha, T, \tau, \varphi, \Psi_1, \Psi_2, h,$
$\{\mathbf{x}_i\}_{i=1}^{N_\Omega}$, % interior points
$\{\mathbf{x}_{i+N_\Omega}\}_{i=1}^{N_{\Gamma_1}}$ % Γ_1 boundary points
$\{\mathbf{x}_{i+N_\Omega+N_{\Gamma_1}}\}_{i=1}^{N_{\Gamma_2}}$ % Γ_2 boundary points

`Output:` $\left\{\{v_i^{n+1}\}_{i=1}^{N}\right\}_{n=0}^{T/\tau-1}$, $\{\rho^{n+1}\}_{n=0}^{T/\tau-1}$

1: `build` $\mathbf{A}$ *% by following (8,9,10,11)*
2: `for` $n = 0, 1, \ldots, T/\tau - 1$ *% loop on time slices*
3: `build` $\mathbf{B}^{(n+1)}$ *% by following (8,9,10,11)*
4: `compute` $\mathbf{v}^{(n+1)}$: *% solution of* $\mathscr{A}\mathbf{v}^{(n+1)} = \mathscr{B}^{(n+1)}$ *in (12)*
5: `endfor`

3. Sequential and parallel implementation

In order to implement the Algorithm 1, we need to define: a 2D regular grid, called `CenterPoints` and the sub-sets `center_I`, `center_b1` and `center_b2`, the interior points and the boundary points of the `CenterPoints`, respectively. Therefore, we find for each fixed interior point ($i = 1, \ldots,$size(`center_I`)) its local neighbors and, by eval-

uating the Laplacian of the local RBF interpolating function, we build the i-th row of `A_total` (i.e. $\mathbf{A}$). We highlight that this step requires to solve multiple linear systems of small size (number of neighbors) for each point in `center_I`. Thus we build next size(`center_b1`) + size(`center_b2`) rows of `A_total` (by using (9) and (10)) and we build last row of `A_total` by evaluating the integral in (3) by means of 2D Gaussian-Legendre quadrature rules of order 15. Finally, after, a discrete 1D time interval `tt`$= [0 : \tau : T]$, with step τ, generation, for each time in `tt`, we build the right-hand side vector `B` (i.e. $\mathbf{B^{(n+1)}}$) and we solve the sparse linear system `A_total` $\cdot$ `sol = B`, where `sol` is the computed value of $\mathbf{v^{(n+1)}}$.
Algorithm 2 illustrates the necessary steps in detail. Observe that, our sequential implementation provides the multiple linear systems solution, at lines 9, 10 and 11 by using the routine `dgesv` of the LAPACK library based on the LU factorization method, while to solve the sparse linear system, at line 16, a specific routine of the CSPARSE library is employed [14], i.e. the `cs_lusol` routine, typical for linear systems characterized by sparse coefficient matrices.

Algorithm 2 Sequential algorithm

```
1: STEP 0: input phase
2: generate CenterPoints
3: find Center_I        % interior points
4: find Center_b1       % boundary points
5: find Center_b2       % boundary points
6: STEP 1: construction of the coefficient matrix
7: for each point of CenterPoints
8:     find its local neighbors
9:     solve multiple linear systems     % one for each point of center_I
10:    solve multiple linear systems     % one for each point of center_b1
11:    solve multiple linear systems     % one for each point of center_b2
12: endfor
13: STEP 2: loop on time
14: for n = 0; n < T/τ; step = 1 do
15:    build B      % (by using results of lines 8,9,10,11)
16:    solve A_total · sol = B
17:    set sol_M[n+1]:=sol
18: end for
19: STEP 3: output phase and condition number evaluation
20: reshape matrix sol_M
```

Starting from some preliminary and interesting results obtained in [13], where a multicore strategy was used, here we propose a different parallel approach that exploits the powerful of modern GPU architectures. This new implementation comes from the idea of increasing the threads number (using all cores available on the GPU architecture) in order to observe the optimal gain obtained by using last generation parallel machines.

Firstly, to achieve a satisfying execution of our code (in terms of execution time but, above all, to ensure that the software reaches the highest performance) we car-

ried out an ad-hoc configuration of the CUDA environment, because for large input a high number of bytes are needed to be allocated with respect to the number of operations. So, for each thread, the local stack and heap size have increased by using the `cudaThreadSetLimit(op, size)` routine and by setting parameters:

– `op:=cudaLimitMallocHeapSize`
– *size:=1024*1024*1024*.

This routine has to be called before the CUDA environment starts. In this way, we tried to overcome, as possible, the well-known problem of the limited memory inherent to the GPU architectures. Moreover, with this arrangement it is possible to allocate memory dynamically on a GPU and, as explained later, to reduce the transfer of host-device data.
About the decomposition approach, we combine the classical *domain* decomposition with a more sophisticated *functional* decomposition, following this schema: each sub-domain of work is demanded to a set of threads, using one-dimensional blocks and grid, if the chosen number of threads is less or equal than 1024; otherwise the blocks and the grid are set as two-dimensional structures. Therefore, following the domain decomposition approach, each thread is linked at a single input point and it performs all the operations needed to build the final sparse matrix; while in accordance with the *functional* decomposition, a pool of threads, based on a fixed configuration, works on different tasks of the overall algorithm in a parallel way.
More in detail, we describe our GPU-parallel approach STEP by STEP referring to the serial version showed in Algorithm 2.

The input phase, in STEP 0, uses a domain decomposition-based parallelization strategy. The local neighbors are defined by considering a sub-set of the `CenterPoint` set. In particular, for each point the thread associated with it builds the sub-domains corresponding to the *inner* points and the *boundary* points. In this way, the construction of local structures becomes very simple and faster.
In STEP 1 the build of the sparse matrix coefficient is designed following the domain decomposition criterion and the functional decomposition approach: for each sub-domain of the `CenterPoint` set the local neighbors are found, as in the STEP 0, therefore, each thread (linked at one input point) builds the local corresponding multiple linear systems whose solutions will be computed by a pull of threads, asynchronously, and then collected in the global matrix `A_total`.

In order to avoid the copy overhead, a suitable workload distribution is performed by using Algorithm 3.

Algorithm 3 Compute global matrix for each thread

```
1: index = threadIdx.x + (blockDim.x * blockIdx.x) % thread index definition
2: g_Mats[], g_Terms[] % allocation on the host
3: FOR ALL thread
4:     g_Mats[index] % local matrix building
5:     g_Terms[index] % local note terms computation
6: ENDFOR ALL
```

Observe that, at lines 4, 5, 6 the values are computed and stored in the local memory of each thread in a one-dimensional array following the *row-major* order method.
Now, to solve the local multiple linear systems a specific technique has been implemented, because CUDA are not able to directly call the routines of the cuSOLVER library from the device. More precisely, firstly the local matrices are transferred on the host and collected in a global array whose the rows size is equal to the thread number. In this array each portion, related to a single thread, contains the local matrix. A similar approach is used to build and store the right-hand side vectors of each linear system. Therefore, from the host each local portion of the global matrices `g_Mats` and `g_Terms` are used as parameters for the routine `cusolverDnDgetrs` of the cuSOLVER-Dense library, called to solve each linear system, as shown in Algorithm 4.

Algorithm 4 Solving local Linear Systems - Host code

1: start cuSOLVER & cuBLAS environments
2: `index = threadIdx.x + (blockDim.x * blockIdx.x)` *% thread index definition*
3: FOR ALL thread-pull
4: call cuSOLVER routine with `g_Mats[index]` and `g_Terms[index]`
5: `__syncthreads()`
6: copy result in the `A_total` sparse matrix
7: ENDFOR ALL

The solutions, returned by the cuSOLVER routine are inserted in the `A_total` sparse matrix. During this phase we perform a synchronization by using the `__syncthreads()` routine, in order to avoid any memory contention. The final STEP 2 provides the sparse linear system solution. It is executed in a similar way to what described in the STEP 1, i.e. following the scheme showed in Algorithm 3 to manage the host-device data transfer. For this last STEP only the domain decomposition approach has been used. To be specific, the *loop for*, related to time discretization, runs in parallel on `T` threads (corresponding to the time interval size). However, before to compute this final STEP, a copy of the `A_total` sparse matrix is stored in the local memory of each thread using the CSR format, [16]. Everything described is shown in detail in Algorithm 5. More in details, each thread builds the local `B` vector by using the `cublasDgemv()` routine of the cuBLAS library [15], for solving a matrix-vector multiplication, at line 6. Hence, the main sparse linear system is solved by the host with the `cusolverSpDcsrlsvlu()` routine of the cuSOLVER library, at each time step.

4. Numerical tests

The GPU-parallel algorithm, described in the previous section, has been implemented on a computer machine with the following technical specifications:

- two CPU Intel Xeon with 6 cores, E5-2609v3, 1.9 Ghz, 32GB of RAM, 4 channels 51Gb/s memory bandwidth
- two NVIDIA GeForce GTX TITAN X, 3072 CUDA cores, 1 Ghz Core clock for core, 12 GB DDR5, 336 GBs as bandwidth

Algorithm 5 Solving the sparse linear system

```
start a CUDA pool of T threads
on the device:
index = threadIdx.x + (blockDim.x * blockIdx.x)  % compute index thread
allocation g_B[] % contains i-th B vector
FOR ALL thread
    build B
    g_B[i_th] = B
ENDFOR ALL
return to host:
for n = 0; n < T; step = 1/τ do
      solve A_total · sol = g_B[n]
      set sol_M[n+1]:=sol
end for
```

In the following we show some numerical tests in order to highlight the performance gain in terms of execution times and memory occupancy. In Table 1 some executions of our software are shown, by varying both the input size and the threads configuration. Time values are obtained by averaging ten test runs for all each considered case.

Table 1. Execution times in seconds (s) achieved, by varying the CUDA configuration: block $\times$ threads and the problem input size.

input size	serial times	1×256	1×512	1×1024	2×256	2×512	2×1024
3.6×10^3	1.6×10^3	5.48	1.40	2.30	3.66	4.32	7.50
8.1×10^3	1.2×10^4	7.60	1.86	3.45	5.42	6.72	13.22
1.0×10^4	3.19×10^4	13.30	3.60	4.56	7.88	8.24	18.77
1.69×10^4	1.66×10^5	17.25	5.20	6.28	10.62	12.32	23.00
1.98×10^4	2.46×10^5	19.89	8.12	9.10	13.4	16.20	27.82
2.25×10^4	3.9×10^5	26.80	8.12	14.22	17.65	21.20	32.10

As we can see, a significant gain with respect to the serial version is achieved. This is essentially due to a suitable choice for the memory setup and to the adaptive combination of the different parallel strategies considered. As illustrated in the previous section, these options allow us a good workload balance. More in details, the execution time of the sequential algorithm grows considerably as the input point number increases. However a noticeable speed up is observed in the GPU-parallel version. To be specific, the better execution is reached by using 1×512 threads. A probable simple reason which explains this result should be that for our graphic card, according the rules for a correct configuration of the CUDA environment (that depend on the relationship between the maximum size of the allocable constant memory and the number of CUDA cores per multiprocessor) the optimal number of threads per block is 512. Moreover, we highlight that, for all last executions where the input is very large, the execution time increases, because of the high bandwidth required during the computation. In fact, using a large number of threads execution times grow for the spurious threads given but not used for the computation. We just observe this phenomenon looking at the different ways of increasing the times by comparing the second last column and the last one.

In Table 2 the execution times, for each single kernel by varying the input size and fixing as CUDA configuration: 1×512, are shown. As expected the most expensive kernel, in terms of execution time, is the `cusolverDnDgetrs( )` routine, which is the kernel used to solve the multiple linear system needed to build the sparse matrix `A_total`. This task, despite the parallel decomposition/functional approach combination, remains the most expensive in the computation.

Table 2. Time execution analysis for each CUDA Kernel: the first lines corresponding to the execution times of computational kernels are expressed in milliseconds, while the execution times of transferred data (the last two lines) are reported in microseconds.

cudaKernel	3.6×10^3	8.1×10^3	1.0×10^4	1.69×10^4	1.98×10^4	2.25×10^4
firstKernel()	394.12	938.03	1095.78	1860.17	2167.66	2634.25
computeMatrix()	9.47	22.40	27.34	44.42	52.09	59.18
computeCenterb1b2	4.22	9.45	11.72	21.81	24.85	28.37
computeCenterI	3.04	6.85	9.53	14.77	17.82	19.02
cusolverDnDgetrs	962.32	1132.71	2306.38	2910.57	3729.67	5305.31
cublasDgemv()	1.89	4.95	5.25	8.87	11.42	13.81
cusolverSpDcsrlsvlu	205.81	464.50	571.69	966.14	1132.28	1328.66
CUDA memcpy HtoD	808.33	2117.74	2245.36	3814.86	4142.87	5082.12
CUDA memcpy DtoH	640.22	1667.41	1887.38	2809.85	3151.21	4002.37

Time values in previous table positively confirm the efficiency of the proposed software by showing the low weight of host-device-host communications with respect to the computational workload. Conclusive considerations on the performance of our algorithm can be made by analyzing Table 3.

Table 3. Memory usage in MBytes (MB) - configuration blocks/threads = 1×512.

N	**MB**
3.6×10^3	1165
8.1×10^3	1183
1.0×10^4	1215

N	**MB**
1.69×10^4	1251
1.98×10^4	1332
2.25×10^4	1491

From this table, we deduce that the GPU version allows us to obtain a very good memory occupation, typical of the use of CUDA architectures. In fact, the low use of global CUDA memory (for our graphic card maximum 12213 MBytes) is due to the high utilization of local threads memory, which in addition to reducing the time of device-host-device copy limits the use of the global memory, stemming all the work in the local thread work-space.

5. Conclusions

In this paper, we proposed a GPU-parallel algorithm to solve a two-dimensional inverse time fractional diffusion equation. The algorithm implements a numerical procedure based on the discretization of the Caputo fractional derivative and on the use of a meshless localized collocation method exploiting the radial basis functions properties.

The parallel approach, based our CUDA-kernels efficient implementation and on a reliable use of ad-hoc parallel numerical libraries available on CUDA, provides a significant performance gain in terms of execution times and memory occupancy.

Acknowledgement

This paper has been supported by project *Algoritmi innovativi per interpolazione, approssimazione e quadratura* (AIIAQ) and project *Algoritmi numerici e software per il trattamento di dati su larga scala in ambienti HPC* (LSDAHPC).

References

[1] Mohebbi, Akbar, Mostafa Abbaszadeh, and Mehdi Dehghan. "The use of a meshless technique based on collocation and radial basis functions for solving the time fractional nonlinear Schrödinger equation arising in quantum mechanics." Engineering Analysis with Boundary Elements 37.2 (2013): 475-485.

[2] Piret, CéCile, and Emmanuel Hanert. "A radial basis functions method for fractional diffusion equations." Journal of Computational Physics 238 (2013): 71-81.

[3] Aslefallah, Mohammad, and Elyas Shivanian. "Nonlinear fractional integro-differential reaction-diffusion equation via radial basis functions." The European Physical Journal Plus 130.3 (2015): 47.

[4] Cuomo, S., Galletti, A., Giunta, G., Marcellino, L., Reconstruction of implicit curves and surfaces via RBF interpolation (2017) Applied Numerical Mathematics, 116, pp. 157-171. DOI: 10.1016/j.apnum.2016.10.016

[5] Fasshauer, Gregory E. Meshfree approximation methods with MATLAB. Vol. 6. World Scientific, 2007.

[6] https://developer.nvidia.com/cuda-gpus

[7] https://docs.nvidia.com/cuda/cusolver/index.html

[8] L. Yan, F. Yang, The method of approximate particular solutions for the time-fractional diffusion equation with a non-local boundary condition, Computers and Mathematics with Applications, 70 (2015) 254-264.

[9] S. Abbasbandy, H. R. Ghehsareh, M. S. Alhuthali, H. H. Alsulami, Comparison of meshless local weak and strong forms based on particular solutions for a non-classical 2-D diffusion model, Engineering Analysis with Boundary Elements, 39 (2014) 121-128.

[10] Q. Liu, Y. T. Gu, P. Zhuang, F. Liu, Y. F. Nie, An implicit RBF meshless approach for time fractional diffusion equations, Comput Mech, 48 (2011) 1-12.

[11] W.Y. Tian, H. Zhou, W.H. Deng, A class of second order difference approximations for solving space fractional diffusion equations, Mathematics of Computation, 84 (2015) 1703-1727.

[12] A. Kilbas, M.H. Srivastava, J.J. Trujillo, Theory and Application of Fractional Differential Equations, North Holland Mathematics Studies, 204 (2006).

[13] De Luca, P., Galletti, A., Giunta G., Marcellino, L., Raei, M. Performance analysis of a multicore implementation for solving a two-dimensional inverse anomalous diffusion problem (2019) Lecture Notes in Computer Science - Proceedings of NUMTA2019, THE 3RD INTERNATIONAL CONFERENCE AND SUMMER SCHOOL

[14] http://faculty.cse.tamu.edu/davis/suitesparse.html

[15] https://docs.nvidia.com/cuda/cublas/index.html

[16] https://docs.nvidia.com/cuda/cusparse/index.html#csr-format

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200057

Parallelization Strategies for GPU-Based Ant Colony Optimization Applied to TSP

Breno Augusto DE MELO MENEZES [a,1], Luis Filipe DE ARAUJO PESSOA [a]
Herbert KUCHEN [a], and Fernando BUARQUE DE LIMA NETO [b]
[a] *University of Muenster, Germany*
[b] *University of Pernambuco, Brazil*

Abstract.
Graphics Processing Units (GPUs) have been widely used to speed up the execution of various meta-heuristics for solving hard optimization problems. In the case of Ant Colony Optimization (ACO), many implementations with very distinct parallelization strategies and speedups have been already proposed and evaluated on the Traveling Salesman Problem (TSP). On the one hand, a coarse-grained strategy applies the parallelization on the ant-level and is the most intuitive and common strategy found in the literature. On the other hand, a fine-grained strategy also parallelizes the internal work of each ant, creating a higher degree of parallelization. Although many parallel implementations of ACO exist, the influence of the algorithm parameters (e.g., the number of ants) and the problem configurations (e.g., the number of nodes in the graph) on the performance of coarse- and fine-grained parallelization strategies has not been investigated so far. Thus, this work performs a series of experiments and provides speedup analyses of two distinct ACO parallelization strategies compared to a sequential implementation for different TSP configurations and colony sizes. The results show that the considered factors can significantly impact the performance of parallelization strategies, particularly for larger problems. Furthermore, we provide a recommendation for the parallelization strategy and colony size to use for a given problem size and some insights for the development of other GPU-based meta-heuristics.

Keywords. Parallel Ant Colony Optimization, Graphic Processing Units, Parallelization Strategies

1. Introduction

Ant Colony Optimization (ACO) is a well-established algorithm to solve combinatorial optimization problems, such as the TSP, based on the behaviour of foraging ants [4]. Analogously to the ants behaviour in the natural environment, ants in ACO try to find the path (a combination of edges) with the lowest cost (e.g. shortest distance). At each iteration, each ant constructs a complete solution by a step-wise selection of the next edge to traverse based on its pheromone concentration: the higher the concentration, the higher is the probability of an edge of being chosen.

As the number of edges and vertices in the graph increases, the number of possible combinations explodes, thus requiring more calculations for each ant during the path cre-

[1]E-mail: breno.menezes@wiwi.uni-muenster.de

ation process. Since more ants are required to explore a large search space appropriately, the computation becomes even more costly. Hence, the parallelization of ACO is crucial for finding high-quality solutions to large problems in acceptable execution time.

The easy access to high-performance computing hardware, such as multi-core CPUs and GPUs, is promoting much effort towards the parallelization of metaheuristics by optimally splitting the workload among the several cores available, providing maximum occupancy as possible. The challenge is also to avoid overheads, mostly due to the communication between the different memory hierarchies [6], different processes, or different threads.

The different parallel formulations of ACO can be divided into groups according to the strategy applied and the degree of parallelization [8]. The first and most intuitive strategy is to parallelize the work of the different ants. The advantage is a simple, straightforward implementation, since the work of each ant just needs to be assigned to one of several concurrent threads. This approach already enables significant reductions of the execution time. It is often found in literature [2,3,5] and here referred as coarse-grained parallelization. The number of threads generated by the coarse-grained implementations is equal to the number of ants in the colony. This number is typically much lower than the value necessary to provide full occupation of a modern GPU with thousands of cores.

To overcome those drawbacks, fine-grained parallel strategies have been used in other works to explore better all computational power of a modern GPU [1,11,12]. When applied to ACO, this strategy parallelizes (in addition to the ants) the many probability calculations performed inside the ant's path creation process. In this approach, one ant is referred to as a block of threads and each probability calculation is performed in a single thread.

Considering both parallelization strategies mentioned so far, it has not been investigated how the problem configuration and the algorithm configuration affect the performance, and what are the benefits when compared to a sequential implementation. Therefore, in this work, both strategies are tested and evaluated in distinct scenarios. The goal is to state for which cases a fine-grained strategy is more beneficial than a coarse-grained parallelization.

This paper is organized as follow. Section 2 describes the ACO algorithm and the main features of the vanilla version. The parallelization strategies investigated in this work are described in Section 3. In Section 4, we describe our experimental comparison of the different parallelization approaches and the results obtained from them. Conclusions, insights and possible future work are presented in Section 5.

2. Ant Colony Optimization

Like other swarm-based metaheuristics, ACO is based on simple agents (i.e. the ants) that cooperate with each other to find improved solutions. The difference is that, in ACO, individuals do not represent a solution themselves. Instead, they are responsible for constructing solutions at every iteration of the algorithm over the influence of the environment (e.g. pheromones on each edge of the graph). Algorithm 1 presents the basic structure for ACO in its basic implementation for TSP.

The construction of a solution is performed step-by-step in a probabilistic manner. The solutions are built adding components to a partial solution, until a complete solution

Algorithm 1 Sequential ACO algorithm

```
Initialize graph
Initialize pheromone trails
while (Stop criterion is not met) do
  for each ant do
    Construction Solution
  end for
  Pheromone update
end while
```

is reached. In the case of the TSP problem, the probability of visiting a vertex is calculated considering the distance to this vertex and the amount of pheromone present on the edge leading to this vertex. It can be calculated as:

$$p_{i,j} = \frac{[\tau_{i,j}]^{\alpha}[\eta_{i,j}]^{\beta}}{\sum_{l \varepsilon N}[\tau_{i,l}]^{\alpha}[\eta_{i,l}]^{\beta}} \quad \forall j \varepsilon N \tag{1}$$

where $\tau_{i,j}$ is the amount of pheromone between vertices i and j, $\eta_{i,j} = 1/distance_{ij}$, α and β are parameters that determine, how much the pheromone quantity and the distance shall influence the probability, respectively. With all probabilities calculated, the ant is able to choose which vertex will be visited next. The process is repeated until a tour is complete.

The next phase is the pheromone update which can be divided into two parts, evaporation and deposit. Pheromone evaporation removes an equal fraction of pheromone of each edge in the graph (Equation 2). This process is important to decrease the relevance of edges that were visited in earlier stages of the search, thus favouring more exploration of the search space and avoiding getting trap in local optimum.

$$\tau_{i,j} := (1-\rho) * \tau_{i,j} \tag{2}$$

where, $\rho \in [0..1]$ defines the evaporation rate. Afterwards, the pheromone deposit take place. Now, each ant is responsible to deposit pheromone at each edge visited on the tour that was created during the latest tour construction phase. The amount of pheromone is proportional to the quality of the tour constructed by ant the ant that constructed the tour with the shortest path will deposit a higher amount of pheromone (Equation 3).

$$\tau_{i,j} := \tau_{i,j} + \Delta t \tag{3}$$

where Δt is calculated as $1/D_k$ and D_k is the length of the tour generated by ant k.

Over time, the amount of pheromone will increase on the edges that are often contributing to generate the shortest paths. The more pheromone they have, the more likely they will be chosen by ants in the subsequent tour construction phases. The ants will use these high-rated partial solutions to build new tours and possibly improved/shorter ones.

3. GPU-based Ant Colony Optimization

The conception of GPU-based algorithms can be a very complicated and error-prone task. The use of a framework, such as CUDA, becomes necessary to support the develop-

ment and execution of such algorithms. Supported by all Nvidia GPUs, CUDA provides several tools and a programming model for the development of highly parallel applications.

The computing power and aspects of a GPU and CUDA have notable differences compared to CPUs. A GPU is composed of a set of streaming multiprocessors (SM), each containing several cores. Cores inside an SM execute the same instruction simultaneously, following a SIMT schema (Single Instruction Multiple Threads).

GPUs also have different memory management. The main, largest and slowest memory space on the GPU is the global memory. Each SM has its own memory called shared memory. This memory is accessible to all threads running on this SM and it is not accessible for other SMs. Furthermore, each thread has a local memory which is only accessible by itself.

One of the advantages of using CUDA is the clear distinction of sequential and parallel parts of the code. The sequential parts are normally executed by the CPU, while the parallel parts, so-called CUDA kernels, are executed on the GPU. When launching a CUDA kernel, the programmer must specify how many CUDA blocks shall be used and how many threads each block will have. CUDA blocks are groups of threads that will be assigned to the same SM. Inside an SM, each block will be sub-divided into groups of e.g. 32 threads called warps. SMs execute one warp at a time until all threads have been processed.

Despite the bigger amount of cores present in GPUs, some limitations must be also taken into account. For example, when launching kernels, each block is assigned to an SM and blocks that are not allocated due to a lack of resources are queued until resources are available again. Limitations of an SM that determine this behavior are, for example, the maximum number of blocks, the maximum number of threads and the maximum size of data that can be loaded at once.

The implementations mentioned in this work consider all these factors. Algorithm 2 shows the basic template for the GPU-based ACO implementations used here, from the initialization and transfer of data to the GPU, to the parallel execution of the algorithm. The executions of the algorithms themselves run completely on the GPU side, avoiding overhead. [2]

Algorithm 2 Pseudo-code GPU-ACO

```
1: initialize ACO
2: initialize GPU
3: while stop criterion is not met do
4:    tour_construction_kernel <<< n_blocks, n_threads >>>;
5:    pheromone_update_kernel <<< 1, 1 >>>;
6: end while
7: copy results to host
8: clear GPU
```

[2] For simplicity, we here focus on the vanilla version of ACO as presented in Section 2 and do not take into account enhancements as in [9].

3.1. Coarse-grained ACO

The coarse-grained GPU-ACO has as its main idea the parallelization on the ant-level. To implement this strategy for the TSP, the CUDA kernel for the tour construction phase must execute the instructions regarding a single ant's work, as shown in Algorithm 3. Each ant is represented by a thread, which can be captured using CUDA's grid system (line 1). The *while* loop (line 2) is responsible for the tour construction, where in each step the next city to be visited is chosen. The calculation of the probabilities is executed sequentially, following the *for* loop in line 4. Afterwards, each ant decides on the next city using its probability calculations (line 7 and 8).[3]

Algorithm 3 Coarse-grained tour construction kernel

```
1:  ant_i = blockIdx.x * blockDim.x + threadIdx.x;
2:  while tour is not complete do
3:     city_i = get_current_city(ant_i)
4:     for each city_j do
5:        if city_j is a neighbor of city_i and city_j has not been visited yet then
6:           calc_probability(city_i, city_j);
7:        end if
8:     end for
9:     chose_next_city(ant_i);
10: end while
```

Typically, when launching a CUDA kernel, the number of threads per block is set to a multiple of 32, respecting the warp size as mentioned previously. In this implementation, it is not a problem to fit the colony size to a value that fits. Also, the number of blocks can be set to balance the threads among the SMs. For many typical colony sizes, this strategy makes use of a relatively small number of threads compared to what a modern GPU can handle. Thus, the coarse-grained parallelization is often not sufficient to exploit the computing power of a GPU and an additional parallelization approach is needed as presented in the next subsection.

Furthermore, the instructions executed inside the tour construction kernel make it not ideal for a GPU execution. The conditional statements used to check whether the cities are linked and whether the considered city has already been visited (line 5 of algorithm 3), create branches in the algorithm. As threads that belong to the same CUDA block must execute the same instruction at the same time (SIMT), branching makes some threads go idle while other threads execute a certain branch.

3.2. Fine-grained ACO

The fine-grained strategy for GPU-ACO aims at the *additional* parallelization of each ant's tour construction. In this strategy, each CUDA block represents an ant and the tour construction work is distributed among threads as shown in Algorithm 4. By doing so, the probability calculations performed at each step of the tour construction are now executed in parallel. Furthermore, control instructions are now assigned to a single thread (lines 9 and 14).

[3]The pheromone update could be executed in parallel as well. For simplicity, we refrain from doing so here.

Algorithm 4 fine grain parallel tour construction

```
1:  ant_i = blockIdx.x;
2:  city_j = threadIdx.x;
3:  while (tour_is_not_complete(ant_i)) do
4:     city_i = get_current_city(ant_i)
5:     d_eta = compute_inverted_distance(city_i, city_j);     // see line behind Eq. 1
6:     d_tau = compute_pheromone(city_i, city_j);             // see line behind Eq. 1
7:     Synchronize;
8:     if threadIdx.x == 0 then
9:        d_sum = compute_denominator(ant_i);                 // see denominator in Eq. 1
10:    end if
11:    Synchronize;
12:    calc_probability(city_i, city_j, d_eta, d_tau, d_sum); // see Eq. 1
13:    if threadIdx.x == 0 then
14:       chose_next_city(ant_i);
15:    end if
16:    Synchronize;
17: end while
```

This strategy benefits from the higher parallelization level. It naturally sets the quantity of CUDA blocks and threads in a way that better occupies the GPU, when compared to the coarse-grained parallelization, in particular for smaller colonies. A higher number of threads spread among several blocks are ideal for the GPU execution.

Furthermore, the instructions executed by each thread are simpler when compared to the coarse-grained implementation. In this case, only one thread per block is responsible for the instructions with conditional statements. In the coarse-grained implementation, all threads execute conditional statements, leading to the serialization of the execution.

The negative point in this strategy is the lack of control of the number of blocks and threads. As the number of threads per block is equal to the number of available edges in the graph, it is not necessarily a multiple of 32. Also, for bigger colony sizes, the number of blocks that can be allocated simultaneously is easily reached and the remaining blocks are then executed sequentially.

4. Experiments and Results

In order to run the different ACO approaches, an HPC cluster with GPUs was used. The sequential version of the algorithm used the normal partition of the cluster, in which the computing nodes are equipped with Intel Xeon Gold 6140 CPUs with 18 cores (36 threads) running at 2.3GHz (turbo boost frequency up to 3.7GHz) and 192GB RAM. Our GPU-ACO implementations used the partition that is equipped with K20 NVidia Tesla accelerators. Each of these GPUs has 2496 cores running at 706MHz, 5GB memory and they run CUDA 3.5.

The experiments performed in this work include several instances of the TSP problem taken from popular repositories [7,10]. The selected TSP instances have different number of vertices ($\rightarrow$ Table 1) so that the behavior of considered ACO implementations are evaluated on various problem sizes.

Table 1. TSPLIB

Instance	dj38	qa194	d198	a280	lin318	pcb442	rat783	lu980	pr1002
# vertices	38	194	198	280	318	442	783	980	1002

Each implementation was tested using different colony sizes (1024, 2048, 4096 and 8192 ants) for all TSP instances. It is important to remark that, in this work, the fitness achieved is not relevant and the focus of the analysis is rather on the execution time. Since all three implementations are based on the same algorithm and just vary on the parallelization strategy, they achieve similar fitnesses when using the same colony size.

The execution time of the sequential implementation of ACO is displayed in Table 2. Table 3 displays the speedup achieved with both parallelization strategies. It is important to mention that the speedup was calculated using the execution time for the whole algorithm. In the case of the GPU execution, it includes not only the tour construction process but also the initialization of the GPU, data transfers from host to device and other calculations.

Table 2. Execution time in seconds - sequential implementation

Problem\Ants	1024	2048	4096	8192
dj38	2.29	4.58	9.16	18.33
qa194	41.24	82.48	164.96	329.93
d198	43.47	86.94	173.88	347.76
a280	87.17	174.33	348.66	697.33
lin318	114.82	229.64	459.28	918.57
pcb442	227.28	454.56	909.13	1818.25
rat783	806.01	1612.03	3224.06	6448.14
lu980	2271.36	4542.71	9085.42	18170.85
pr1002	1529.82	3059.65	6119.30	12238.60

Both parallelization strategies used in this work presented significant speedups. Only for the smallest TSP instance ($dj38$), the coarse-grained ACO was slower than the sequential implementation, and the speedup achieved by the fine-grained implementation was also not so significant. In this case, the amount of work to be processed does not

Table 3. Speedup rates compared to sequential ACO

	Coarse-grained ACO				Fine-grained ACO			
Problem\Ants	1024	2048	4096	8192	1024	2048	4096	8192
dj38	0.47	0.62	0.72	0.71	0.75	0.78	0.79	0.72
qa194	0.96	1.37	1.71	1.93	2.11	2.11	2.03	2.00
d198	0.99	1.41	1.77	2.01	2.13	2.10	2.04	2.01
a280	1.09	1.64	2.23	2.66	2.60	2.65	2.67	2.69
lin318	1.13	1.73	2.41	2.95	2.85	2.93	2.98	3.01
pcb442	1.21	1.96	2.90	3.75	3.08	3.32	3.44	3.51
rat783	1.18	2.09	3.42	4.74	3.20	3.65	3.96	3.92
lu980	2.95	5.18	8.45	11.72	7.97	9.08	9.79	9.67
pr1002	1.22	2.24	3.88	5.81	4.15	4.67	4.99	4.99

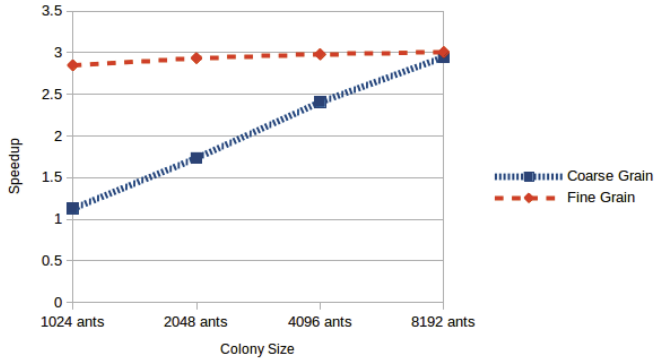


Figure 1. Speedup Comparison - lin318

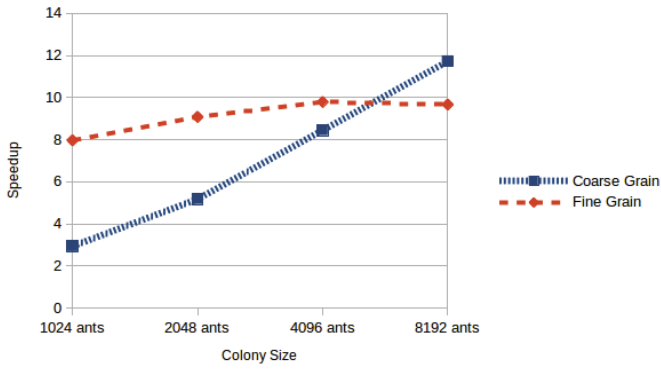


Figure 2. Speedup Comparison - lu980

justify the use of parallel strategies and the extra costs involved (i.e. starting the GPU, transferring data between devices).

Furthermore, in most of the cases, the fine-grained ACO enables a higher speedup when compared to the coarse-grained implementation, since the the later does not provide enough parallelism. The speedup comparison of the *lin*318 instance (Figure 1) shows that the fine-grained implementation achieved a higher speedup for all colony sizes, although the gap between the speedups decreases as the colony size increases.

For TSP instances with a higher number of cities (i.e. *rat*783, *lu*980, *pr*1002), the scenario changes and the coarse-grained implementation enables a higher speedup when using 8192 ants. For 8192 ants, there is now enough coarse-grained parallelism to fully exploit the hardware and, in contrast to the fine-grained implementation, there are no idle cores when combining the results obtained for different neighbor cities as in lines 9 and 14 of algorithm 4 (see Figure 2).

The results show that there is no universal best parallelization strategy between the ones evaluated in this work. The speedup correlates with the parameters of the problem, namely colony size and graph size. These parameters influence directly how the algorithm behaves on the GPU, how the data structures are created and how the multiprocessors are occupied by the program.

For the coarse-grained implementation, running ACO with a smaller colony size over problems with a low number of vertices is not enough to achieve a full occupation of the GPU. This fact explains why the speedup curve for the coarse-grained implementation is ascending with the colony size, while the fine-grained implementation is stable or decreasing. Using the coarse-grained strategy and independently of the TSP instance, a colony of 1024 ants, organized in 32 blocks of 32 threads, is mapped to the GPU cores at once. As the GPU is not fully occupied (the one used in our experiments has 2496 cores), all blocks can be loaded into the processors and execute simultaneously. The same happens for 2048 ants. For 4096 ants, where there are four times more threads, all blocks could still be resident on the streaming multiprocessors. Only with 8192 ants, the limit of resident blocks of the whole GPU is exceeded. In this case, 256 blocks are created, while the maximum number of resident blocks for the whole GPU is equal to 208. Hence not all blocks can be handled simultaneously.

Conversely, the fine-grained implementation makes better use of the high number of GPU cores even for the smallest colony sizes. It profits from creating a high number of blocks, one for each ant, and each block with several threads. Also, the threads processed during the tour construction have a smaller number of instructions and contain less conditional statements, which are serialized on the GPUs.

The fine-grained implementation is only penalized when applying a bigger colony size to a bigger graph. In this case, not only the number of blocks to be created is quite high, but also the GPU is not able to process the same quantity of blocks at the same time, since they need more resources. This scenario leads to an increase in the number of blocks that have to wait until resources are available on the GPU.

5. Conclusion

We have presented an analysis of two different parallelization strategies for the GPU-ACO algorithm applied to the TSP problem. One strategy parallelizes the ants work, while the other parallelizes the internal calculations. Both approaches were applied to distinct TSP instances in order to have a comparison of different scenarios and problem sizes. We also investigated how the colony size impacts the execution time of the algorithm according to the parallelization strategy chosen. The achieved speedup in relation to the sequential implementation was the measure used to compare both strategies.

Our experiments show that the coarse-grained implementation is already capable of generating a significant speedup when compared to the sequential version. The abstraction of parallelizing the work of each ant is quite intuitive and easy to implement, justifying its presence in many other works in literature. On the other hand, it is quite hard to generate enough parallelization (CUDA blocks and threads) to fully occupy a GPU using this coarse-grained strategy. Furthermore, the work performed inside each thread contains several conditional statements, creating branches in the execution of the algorithm, which impacts negatively the execution time.

With the fine-grained strategy, we were able to achieve a higher degree of parallelization, which presented a beneficial influence on the execution time of the algorithm. The additional parallelization of the internal work of each ant promoted a better distribution of processing among the GPU cores available. Furthermore, the work performed by each thread is much simpler than the threads generated by the coarse-grained strat-

egy. The fine-grained threads do not generate branching conditions, which is ideal for the GPU execution (SIMT). The speedup achieved by this strategy was higher than the coarse-grained strategy in most of the cases.

For experiments using a bigger instance of the TSP and a larger colony size, the fine-grained strategy generated so much parallelization that the GPU could not handle it in an optimal way. The number of blocks and threads was just bigger than the amount that could be handled simultaneously by the GPU, leading to a sequential execution of waiting blocks and threads. The overhead generated in these cases impacts the execution time negatively in a way that the coarse-grained was able to present shorter execution times. This fact magnifies the importance of knowing the hardware limitations and the characteristics of the problem being tackled in order to choose the proper parallelization strategy.

As a continuation of this work, we would like to extend the investigation towards other parallelization strategies and enhancements to the algorithm that can work in favour of parallelism. Also, other different parallel applications, such as other meta-heuristics, may benefit from our insights gained by the two-level parallelism. Furthermore, the knowledge acquired during this process will also be used to create a high-level parallelization framework that might be able to adapt itself aiming a better performance.

References

[1] José M. Cecilia, José M. García, Andy Nisbet, Martyn Amos, and Manuel Ujaldón. Enhancing data parallelism for ant colony optimization on GPUs. *Journal of Parallel and Distributed Computing*, 73(1):42–51, 2013.

[2] José M. Cecilia, Andy Nisbet, Martyn Amos, José M. García, and Manuel Ujaldón. Enhancing GPU parallelism in nature-inspired algorithms. *Journal of Supercomputing*, 63(3):773–789, 2013.

[3] Audrey Delévacq, Pierre Delisle, Marc Gravel, and Michaël Krajecki. Parallel ant colony optimization on graphics processing units. *Journal of Parallel and Distributed Computing*, 73(1):52–61, 2013.

[4] Marco Dorigo and Gianni Di Caro. Ant colony optimization: a new meta-heuristic, 1999.

[5] Huw Lloyd and Martyn Amos. Analysis of Independent Roule e Selection in Parallel Ant Colony Optimization. 2017.

[6] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008.

[7] University of Waterloo. National traveling salesman problems. `http://www.math.uwaterloo.ca/tsp/world/countries.html`. Accessed: 14.03.2018.

[8] Martín Pedemonte, Sergio Nesmachnow, and Héctor Cancela. A survey on parallel ant colony optimization. 11:5181–5197, 2011.

[9] Akihiro Uchida, Yasuaki Ito, and Koji Nakano. Accelerating ant colony optimisation for the travelling salesman problem on the GPU. *International Journal of Parallel, Emergent and Distributed Systems*, 29(4):401–420, 2014.

[10] Heidelberg University. Discrete and combinatorial optimization. `https://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/XML-TSPLIB/instances/`. Accessed: 14.03.2018.

[11] Fabian Wrede and Breno Augusto de Melo Menezes. High-level Parallel Implementation of Swarm Intelligence-based Optimization Algorithms with Algorithmic Skeletons. In *Parallel Computing is Everywhere, Proceedings of the International Conference on Parallel Computing, ParCo 2017*, number September, pages 573—-582, Bologna, Italy, 2017. {IOS} Press.

[12] Fabian Wrede, Breno Augusto de Melo Menezes, and Herbert Kuchen. Fish School Search with Algorithmic Skeletons. In *10th International Symposium on High-Level Parallel Programming and Applications*, pages 1–19, 2017.

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200058

DBCSR: A Blocked Sparse Tensor Algebra Library

Ilia SIVKOV [a,1], Patrick SEEWALD [a,2], Alfio LAZZARO [a,3], and Jürg HUTTER [a,4]
[a] *University of Zurich, Department of Chemistry, Switzerland*

Abstract. Advanced algorithms for large-scale electronic structure calculations are mostly based on processing multi-dimensional sparse data. Examples are sparse matrix-matrix multiplications in linear-scaling Kohn-Sham calculations or the efficient determination of the exact exchange energy. When going beyond mean field approaches, e.g. for Moller-Plesset perturbation theory, RPA and Coupled-Cluster methods, or the GW methods, it becomes necessary to manipulate higher-order sparse tensors. Very similar problems are also encountered in other domains, like signal processing, data mining, computer vision, and machine learning. With the idea that the most of the tensor operations can be mapped to matrices, we have implemented sparse tensor algebra functionalities in the frames of the sparse matrix linear algebra library DBCSR (Distributed Block Compressed Sparse Row). DBCSR has been specifically designed to efficiently perform blocked-sparse matrix operations, so it becomes natural to extend its functionality to include tensor operations. We describe the newly developed tensor interface and algorithms. In particular, we introduce the tensor contraction based on a fast rectangular sparse matrix multiplication algorithm.

Keywords. sparse matrix-matrix multiplications, sparse tensor algebra, multithreading, MPI parallelization, accelerators

1. Introduction

Most, if not all the modern scientific simulation packages utilize matrix algebra operations. Often, due to the nature of simulated systems, the structure of matrices and tensors is sparse with a low degree of nonzero elements ($< 10\%$). Applications exploiting the sparsity include the linear scaling density functional theory [1], cubic scaling RPA algorithm and a similar approach to fast, quadratic scaling Hartree-Fock exchange [2] in the quantum chemistry CP2K framework [3]. The first method works with sparse matrices, while the other two algorithms rely on contractions involving sparse 3-rank tensors. Due to the nature of the studied chemical systems, this naturally leads to a blocked sparsity pattern, with chemically motivated block sizes. Therefore, the implementation of such methods requires convenient and effective tools and libraries to work also with block-sparse matrices and tensors, with a range of occupancy between 0.01% up to dense.

[1]E-mail: ilia.sivkov@chem.uzh.ch
[2]E-mail: patrick.seewald@chem.uzh.ch
[3]E-mail: alazzaro@cray.com, now at Cray Switzerland GmbH, Switzerland
[4]E-mail: hutter@chem.uzh.ch

The highly optimized linear algebra library DBCSR (Distributed Block Compressed Sparse Row) has been specifically designed to efficiently perform block-sparse and dense matrix operations, covering a range of occupancy between 0.01% up to dense. It is parallelized using MPI and OpenMP, and can exploit GPU accelerators using CUDA. The more detailed description of these features can be found in the previous works [4,5,6]. Here we give an overview of the library in section 2.

Although DBCSR supports multiplications of rectangular matrices, the implemented algorithm was inefficient whenever the resulting matrix has a much smaller size than input matrix sizes (< 1000 smaller). This matrix multiplication can be used for the realization of tensor contraction since the tensor contraction can be mapped to matrix-matrix multiplications [7]. In section 3 we present an optimized implementation for such a case. Additionally, we have developed the tensor algebra operations as an extension of the DBCSR library. In section 4 we present an overview of the new functionalities. The main operation which is used in tensor algebra is a contraction between two tensors over a set of indices. In many of methods, the rank of tensors is no more than 4 and therefore the non-trivial contractions can be performed over 1-3 indices. Finally, section 5 reports the conclusion.

1.1. Related Work

Other implementations of tensor libraries are described in Ref. [8,9,10,11,12], while Ref. [13] presents an overview of tensor algebra applications. The proposed tensor library implementation in DBCSR differs from these implementations since it is specifically targeting block-sparse tensor contractions with a wide range of occupancy between $0.01 - 10\%$ by optimally exploiting block sparsity. Existing parallel sparse tensor libraries have limited parallel scalability [10], do not prove to be more efficient compared to the dense case [8], or have low sequential efficiency [9]. For matrix-matrix multiplications, the DBCSR library already provides an efficient and scalable solution without the above-mentioned shortcomings.

2. The DBCSR Library

DBCSR is written in Fortran and is freely available under the GPL license from `https://github.com/cp2k/dbcsr`. DBCSR matrices are stored in a blocked compressed sparse row (CSR) format distributed over a two-dimensional grid of P MPI processes. Matrix-matrix multiplication is implemented by means of the Cannon algorithm [14]. As part of this work, two novel implementations are specifically introduced for rectangular matrix multiplications similar to one iteration of CARMA algorithm [15] (see section 3) and for the tensor contraction algorithm (see section 4). The latter uses the same idea as for the rectangular matrix multiplication with a slightly different implementation.

In the Cannon algorithm, only the matrices A and B are communicated for the multiplication $C = C + A \times B$. The amount of communicated data by each process scales as $\mathcal{O}(1/\sqrt{P})$. These communications are implemented with asynchronous point-to-point MPI calls, using the MPI Funneled mode [6]. The local multiplication will start as soon as all the data has arrived at the destination process, and it is possible to overlap the local computation with the communication if the network allows that.

The local computation consists of pairwise multiplications of small dense matrix blocks, with dimensions $(m \times k)$ for A blocks and $(k \times n)$ for B blocks. It employs a cache-oblivious matrix traversal to fix the order in which matrix blocks need to be computed, in order to improve memory locality. First, the algorithm loops over A matrix row-blocks and then, for each row-block, over B matrix column-blocks. Then, the corresponding multiplications are organized in batches. Multiple batches can be computed in parallel on the CPU by means of OpenMP threads or alternatively executed on a GPU. A static assignment of batches with a given A matrix row-block to threads is employed in order to avoid race conditions. Processing the batches has to be highly efficient. For this reason, specific libraries were developed that outperform vendor BLAS libraries, namely `LIBCUSMM` for GPU and `LIBXSMM` for CPU/KNL systems [16,17].

For GPU execution, data is organized in such a way that the transfers between the host and the GPU are minimized. A double-buffering technique, based on CUDA streams and events, is used to maximize the occupancy of the GPU and to hide the data transfer latency [5]. When the GPU is fully loaded, the computation may be simultaneously done on the CPU. `LIBCUSMM` employs an auto-tuning framework in combination with a machine learning model to find optimal parameters and implementations for each given set of block dimensions. For a multiplication of given dimensions (m,n,k), `LIBCUSMM`'s CUDA kernels are parametrized over 7 parameters, affecting:

- algorithm (different matrix read/write strategies)
- amount of work and number of threads per CUDA block
- number of matrix element computed by one CUDA thread
- tiling sizes

yielding $\approx$ 30,000 - 150,000 possible parameter combinations for each of about $\approx$ 75,000 requestable (m,n,k)-kernels. These parameter combinations result in vastly different performances. We use machine learning to derive a performance model from a subset of tuning data that accurately predicts performance over the complete kernel set. The model uses regression trees and hand-engineered features derived from the matrix dimensions, kernel parameters, and GPU characteristics and constraints. To perform the multiplication the library uses Just-In-Time (JIT) generated kernels or dispatches the already generated code. In this way, the library can achieve a speedup in the range of 2–4x with respect to batched DGEMM in cuBLAS.

`DBCSR` operations include sum, dot product, and multiplication of matrices, and the most important operations on single matrices, such as transpose and trace. Additionally, the library includes some of the linear algebra methods, such as the sign matrix algorithm [1] and matrix inverse. These methods were ported from `CP2K` to `DBCSR`. The sign matrix algorithm is used in the linear scaling density functional theory in order to find a ground state of the quantum systems. As associated methods, we have ported the matrix-vector multiplication operation and an interface to some SCALAPACK operations.

3. Optimized Rectangular Matrix Multiplication Algorithm and Implementation

Despite the Cannon algorithm gives in general good performance for the sparse matrix multiplication of any size, it loses its efficiency in the case where the size of the resulting matrix C ($S_C = O_C MN$) is much smaller than the sizes of the input A ($S_A = O_A MK$)

and/or B ($S_B = O_B KN$) matrices, where M, N, K are the dimensions of the dense matrices and O_A, O_B, O_C their occupancy values. This is a direct consequence of the algorithm since it requires the communication of A and B data on a 2D grid of P processors, while C remains local to each processor. In particular, for the multiplication of two rectangular matrices the Cannon algorithm requires to communicate per each processor [5]

$$T_w = \frac{S_A + S_B}{\sqrt{P}} = \frac{K(O_A M + O_B N)}{\sqrt{P}}. \tag{1}$$

Therefore, the communication will be dominated by one of the dimension whenever is much larger than the other two. We can distinguish the two important cases:

1. $M \ll K$ and $N \ll K$, which corresponds to $S_C \ll \{S_A, S_B\}$
2. $K \ll M$ and $K \leq N$, which corresponds to $S_B \ll \{S_A, S_C\}$

According to Ref. [18], a communication-optimal algorithm for this case is obtained by dividing the original matrix multiplication into smaller tasks such that each task is local to a process subgroup. Inspired by this idea, we redistribute the matrices on a linear MPI grid (see Figure 1) and perform the multiplication locally. We describe the implementations for the two cases in the following subsections. We also report the results of some tests we performed for a variety of matrix, block sizes and occupancy values of our interests ($10\% - 50\%$ often present in `CP2K`). We used double precision matrices with sizes of the order $M, N = \mathscr{N}$, $K = \mathscr{N}^2$ and $\mathscr{N} = 10^3$. The calculations were performed using the Cray XC50 "Piz Daint" supercomputer at the Swiss National Supercomputing Centre (CSCS). Each node of the system is equipped by a CPU Intel Xeon E5-2690 v3 @ 2.60GHz (12 cores, 64GB DRAM) and a GPU NVIDIA Tesla P100 (16GB HBM). For the MPI configuration, we used 1 rank per node and 12 OpenMP threads per rank. Each multiplication was performed 100 times to exclude the fluctuations of performance due to hardware glitches.

3.1. $S_C \ll \{S_A, S_B\}$

Matrices A and B are redistributed on a linear MPI grid and the A matrix is transposed, such as the longest dimension K is now distributed over the P processors (see Figure 1a). Then a local multiplication is executed, which gives $\widetilde{C}_i = A_i^T \cdot B_i$, with $i = 1, ..., P$. Here $\widetilde{C}_i$ corresponds to a partial result of the full, undistributed, matrix C. Therefore we have to reduce all $\widetilde{C}_i$ and redistribute the result according to the original 2D grid distribution and sum to the input C matrix to get the final distributed C matrix result over the 2D grid (see Figure 2). This algorithm runs in P steps, where for each step we send and receive the proper C data and run the local reduction. It is implemented with MPI asynchronous communications, such as we do overlap the communication of the data with the local reduction. In the end, each processor requires S_C data. Including the initial redistribution of the A and B matrices, we get that the total amount of data communicated by each processor is:

$$T'_w = \overbrace{\left(\frac{S_A + S_B}{P}\right)}^{\text{2D} \rightarrow \text{1D grid}} + S_C. \tag{2}$$

[5] Here we assume a uniform distribution of the non-zero elements in the matrices without losing generality.

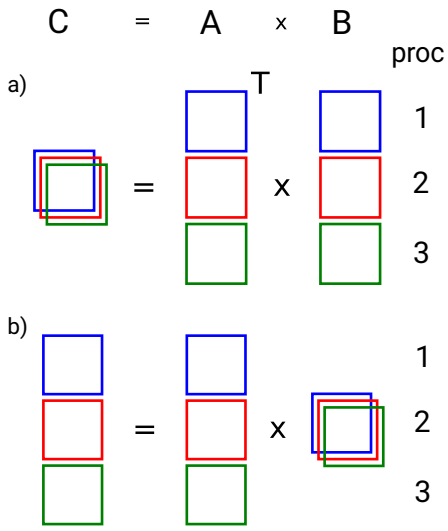


Figure 1. Communication-avoiding algorithm for the rectangular matrix-matrix multiplication. a) Middle dimension K is the largest (case 1), C is replicated and A and B are distributed on a linear grid. b) Outer dimension M is the largest (case 2), C and A are distributed on a linear grid and B is cloned or distributed.

We can now consider the ratio with the Cannon algorithm (Eq. 1), which leads to a reduction in communicated data $\sqrt{P}/(1+RP)$, where $R = S_C/(S_A+S_B)$. Therefore, the ratio scales as $\mathcal{O}(1/\sqrt{P})$. Finally, it is important to note that by multiplication of the sparse matrices even with high sparsity the result might be dense (so called Birthday Paradox [19]). We can evaluate an upper limit on the O_C by combining the Eq. 1 and Eq. 2 such that $T_w < T'_w$:

$$O_C < \frac{1}{MN}\left(T_w - \frac{S_A+S_B}{P}\right) \tag{3}$$

If we omit redistribution costs and assume that $O_A = O_B = O$ then we can write:

$$\frac{O_C}{O} < \frac{K(M+N)}{MN\sqrt{P}}. \tag{4}$$

As an example, for $M,N = \mathcal{N}$, $K = \mathcal{N}^2$ and $\mathcal{N} = 10^3$, $P = 100$ we get $O_C/O < 2\cdot 10^2$.

The results of the tests are presented in Figure 3a. Overall, the new implementation gives a speed-up up to 3x with respect to the Cannon algorithm for high occupancy ($> 10\%$), which becomes negligible when we reach the upper limit reported in the Eq. 3. As expected, the speed-up decreases with the number of processors.

3.2. $S_B \ll \{S_A, S_C\}$

Matrices A and B are redistributed in a linear MPI grid, such as the longest dimension K is now distributed over the P processors (see Figure 1b). A virtual column-grid is created for the A matrix to be compatible with the row-grid of the matrix B. Then the standard Cannon algorithm is executed over this virtual topology made of P steps. Virtual column-

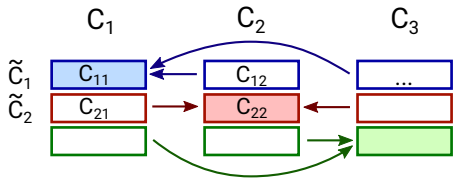


Figure 2. Reduce operation of the matrix C after the local multiplication when $S_C \ll \{S_A, S_B\}$.

grid does not require communication of the A data and therefore only the communication of the matrix B is required. Finally, C result is redistributed to the original 2D grid and accumulated to the input C matrix. The total amount of communicated data by each processor is:

$$T''_w = \overbrace{\left(\frac{S_A + S_B + S_C}{P}\right)}^{\text{2D} \to \text{1D} \to \text{2D grid}} + S_B. \tag{5}$$

Also in this case the ratio of the communicated data with respect to Cannon implementation scales as $\mathcal{O}(1/\sqrt{P})$.

The results of the tests are presented in Figure 3b. Overall, the new implementation gives a speed-up of up to 20% with respect to the Cannon algorithm for high occupancy ($> 10\%$) or up to 100% for the matrices close to dense ($\sim 50\%$). The time for the redistribution and the overhead introduced by the virtual grid creation limits the benefit of the new implementation. For the same reasons, the benefit of the new algorithm is negligible or even worse for low occupancy.

4. Sparse Tensor Algebra Implementation

DBCSR was originally developed to enable linear scaling electronic structure methods that are mainly based on the multiplication of sparse square matrices. Similar strategies employing sparse data can also be employed for methods beyond density functional theory that provide better accuracy at significantly higher computational costs than Kohn-Sham density functional theory. In the case of the electron correlation methods MP2 and RPA, the canonical implementation scales at least quartic with system sizes, thereby preventing the study of large systems (hundreds to thousands of atoms). An initial DBCSR-based cubic scaling implementation of RPA was reported in Ref. [2], enabling calculations of thousands of atoms. Here we report strategies to optimize and generalize this initial implementation by extending the DBCSR library to multi-dimensional tensors. A generalized implementation of tensor operations in DBCSR instead of specialized implementations in the application code is desirable to manage code complexity and to easily extend the current implementation to other methods such as Hartree-Fock exchange or GW. The formalism of our RPA implementation was already described in Ref. [2] and here we emphasize the general characteristics of the tensor operations appearing in this and similar methods. We describe the requirements we pose for a tensor framework that should provide all relevant tensor operations in a general API.

As in DBCSR, the sparsity of the tensors is based on the representation of molecular orbitals in terms of a localized atom-centered basis. A blocked sparsity pattern is

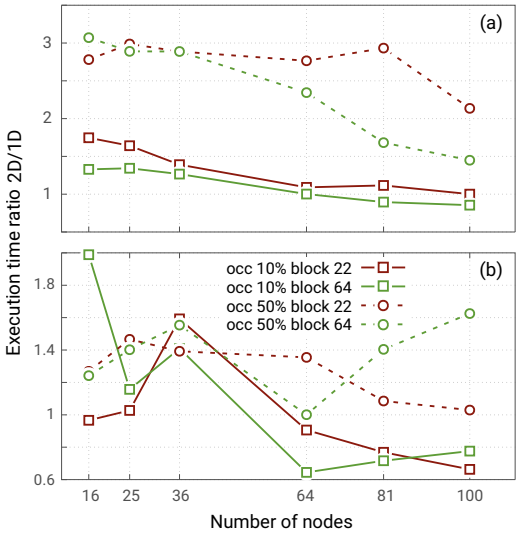


Figure 3. Execution time ratios for two types of considered rectangular matrix multiplications on a 1D grid in comparison with the regular 2D Cannon algorithm. (a) The first type, $S_C \ll \{S_A, S_B\}$, shows significant improvement for higher occupancy (50%) and less pronounced for the lower one (10%). (b) The second type, $S_B \ll \{S_A, S_C\}$, shows moderate to high speedup for higher occupancy (50%) and poor to moderate behavior for the lower occupancy (10%). In both cases, the benefit of the implementation reduces with the number of nodes, as expected.

equally important for the tensor implementation to efficiently incorporate sparse data while keeping the significant overhead for the handling of sparse indices low. Tensors can have arbitrary ranks, most relevant are tensors with ranks between 2 and 4. The main operation is tensor contraction where a sum over one or more indices of two tensors is performed. Our implementation is based on the property that tensor contractions are isomorphic to matrix-matrix multiplications [7]. This allows us to implement tensor contraction by mapping tensors to matrices – the contraction is then internally performed by a call to the existing implementation of sparse matrix-matrix multiplication.

Recasting tensor contraction in terms of matrix-matrix multiplication imposes some requirements on the distribution and the matrix representation of the tensors, most importantly that one matrix dimension represents the indices to sum over and the other matrix dimension represents all other indices. If these requirements are not met, conversion steps are required before and after matrix-matrix multiplication which involves the redistribution of all tensor data. In order to avoid these relatively expensive redistribution steps, the tensor API gives the caller tight control over the distribution and the matrix representation of tensors, such that tensors can be created in a compatible layout and the redistribution step can be skipped in a tensor contraction. Data redistribution is then only strictly needed if a tensor appears in multiple contraction expressions involving sums over different indices.

While this approach of mapping tensors to matrices allows for an implementation of tensor operations as thin layers around an existing matrix library, the resulting matrices are problematic since one dimension is much larger than the other dimension. For the example of a 3 rank tensor with size $N \times N \times N$, two tensor dimensions are mapped to one matrix dimension such that one matrix dimension grows quadratically with the size

of the other dimension. The DBCSR library must thus be extended in a way that it can efficiently store and multiply tall-and-skinny matrices contrary to the previous target of square matrices.

One limitation of the DBCSR matrix format is the index data replicated on all MPI ranks which contain information about block sizes and the distribution of blocks along each of the matrix dimensions. If the size of the matrix index corresponds to the number of atoms N in a system, this limits the scalability of DBCSR to a few tens of millions of particles [1]. For 3-rank tensors where the largest matrix dimension grows as N^2, this limit is already hit at a few thousand atoms, representing a much bigger issue in practice. Thus an extension to the DBCSR matrix format must be provided to store large tensors without exhausting memory due to replicated index data. Another challenge is to multiply tall-and-skinny matrices communication-efficiently, where the algorithm described in section 3 comes into play.

Our requirements for memory-efficient storage and communication-efficient multiplication can both be met by dividing the largest matrix dimension, resulting in smaller and approximately square submatrices that can be handled by DBCSR. The storing of the full matrix index and the multiplication acting on submatrices are managed by an in-between tall-and-skinny matrix layer on top of DBCSR that serves as a basis for the tensor implementation. The tall-and-skinny matrix layer is designed in a way that the index data is not explicitly stored but provided by externally defined function objects, to avoid the above-mentioned limitation of the DBCSR format. The matrix index is thus handled in the tensor layer and is calculated on the fly from the tensor index. Due to the fully distributed sparse data layout, the matrix index calculation happens only when accessing a locally present non-zero block and does not add any overhead.

The main difference between the implementation of tall-and-skinny matrix multiplication and the one implemented in DBCSR internally (see section 3) is that instead of relying on a linear process grid, the grid may have arbitrary dimensions. The submatrices are obtained on MPI subgroups by dividing any of the two grid dimensions by an arbitrary factor. This ensures that an optimal split factor can always be chosen, independently of the total number of processes, for any grid dimensions. Thus n-rank tensors can be represented on an arbitrary n-dimensional process grids where the grid dimension should be chosen as balanced as possible for a load-balanced distribution of data. The multiplication algorithm for contraction can then be run directly without additional costly redistribution steps (for tall-and-skinny matrices, the bandwidth cost of redistributing data exceeds the bandwidth cost for the multiplication [15]).

5. Conclusion

We have presented a new implementation for the rectangular matrix-matrix multiplication algorithm in the DBCSR library that is able to speed-up the execution up to 3x for matrix sizes and occupancy values of $10\% - 50\%$ which are often present in CP2K calculations. We have described the newly developed tensor operations that generalize the DBCSR library to multidimensional tensor contraction for low-scaling electronic structure methods beyond density functional theory. These functionalities are the basic building block for the CP2K quantum chemistry and solid-state physics software package.

Acknowledgments

We thank Shoshana Jakobovits (CSCS Swiss National Supercomputing Centre) for her work on the LIBCUSMM code. This work was supported by grants from the Swiss National Supercomputing Centre (CSCS) under projects S238 and UZHP and received funding from the Swiss University Conference through the Platform for Advanced Scientific Computing (PASC).

References

[1] Joost VandeVondele, Urban Borstnik, and Juerg Hutter. Linear scaling self-consistent field calculations for millions of atoms in the condensed phase. *The Journal of Chemical Theory and Computation*, 8(10):3565–3573, 2012.

[2] Jan Wilhelm, Patrick Seewald, Mauro Del Ben, and Juerg Hutter. Large-scale cubic-scaling RPA correlation energy calculations using a Gaussian basis. *Journal of Chemical Theory and Computation*, 2016. http://dx.doi.org/10.1021/acs.jctc.6b00840.

[3] Juerg Hutter, Marcella Iannuzzi, Florian Schiffmann, and Joost VandeVondele. CP2K: Atomistic Simulations of Condensed Matter Systems. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 4(1):15–25, 2014.

[4] Urban Borstnik, Joost VandeVondele, Valery Weber, and Juerg Hutter. Sparse Matrix Multiplication: The Distributed Block-Compressed Sparse Row Library. *Parallel Computing*, 40(5-6):47–58, 2014.

[5] Ole Schütt, Peter Messmer, Juerg Hutter, and Joost VandeVondele. GPU Accelerated Sparse Matrix Matrix Multiplication for Linear Scaling Density Functional Theory. In *Electronic Structure Calculations on Graphics Processing Units*. John Wiley and Sons, 2015.

[6] Alfio Lazzaro, Joost VandeVondele, Jürg Hutter, and Ole Schütt. Increasing the Efficiency of Sparse Matrix-Matrix Multiplication with a 2.5D Algorithm and One-Sided MPI. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '17, pages 3:1–3:9, New York, NY, USA, 2017. ACM.

[7] Edgar Solomonik, James Demmel, and Torsten Hoefler. Communication lower bounds for tensor contraction algorithms. Technical report, ETH-Zürich, 2015.

[8] Cannada A Lewis, Justus A Calvin, and Edward F Valeev. Clustered Low-Rank Tensor Format: Introduction and Application to Fast Construction of Hartree-Fock Exchange. *arXiv preprint arXiv:1510.01156*, 2015.

[9] Edgar Solomonik and Torsten Hoefler. Sparse Tensor Algebra as a Parallel Programming Model. *arXiv preprint arXiv:1512.00066*, 2015.

[10] Evgeny Epifanovsky, Michael Wormit, Tomasz Ku, Arie Landau, Dmitry Zuev, Kirill Khistyaev, Prashant Manohar, Ilya Kaliman, Andreas Dreuw, and Anna I. Krylov. New implementation of high-level correlated methods using a general block tensor library for high-performance electronic structure calculations. *Journal of Computational Chemistry*, 34(26):2293–2309, 2013.

[11] Samyam Rajbhandari, Akshay Nikam, Pai-Wei Lai, Kevin Stock, Sriram Krishnamoorthy, and P Sadayappan. Framework for distributed contractions of tensors with symmetry. *Preprint, Ohio State University*, 2013.

[12] Walter Landry. Implementing a High Performance Tensor Library. *Scientific Programming*, 11(4):273–290, December 2003.

[13] Tamara G. Kolda and Brett W. Bader. Tensor Decompositions and Applications. *SIAM Rev.*, 51(3):455–500, August 2009.

[14] Lynn Elliot Cannon. *A cellular computer to implement the Kalman Filter Algorithm*. PhD thesis, Montana State University, 1969.

[15] James Demmel, David Eliahu, Armando Fox, Shoaib Kamil, Benjamin Lipshitz, Oded Schwartz, and Omer Spillinger. Communication-optimal parallel recursive rectangular matrix multiplication. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, pages 261–272, Washington, DC, USA, 2013. IEEE Computer Society.

[16] Alexander Heinecke, Greg Henry, Maxwell Hutchinson, and Hans Pabst. LIBXSMM: Accelerating Small Matrix Multiplications by Runtime Code Generation. In *Proceedings of the International Confer-*

ence for High Performance Computing, Networking, Storage and Analysis, SC '16, pages 84:1–84:11, Piscataway, NJ, USA, 2016. IEEE Press.

[17] Iain Bethune, Andreas Glöss, Jürg Hutter, Alfio Lazzaro, Hans Pabst, and Fiona Reid. Porting of the DBCSR Library for Sparse Matrix-Matrix Multiplications to Intel Xeon Phi Systems. In *Advances in Parallel Computing, Proceedings of the International Conference on Parallel Computing, ParCo 2017, 12-15 September 2017, Bologna, Italy*, pages 47–56, 2017.

[18] James Demmel, David Eliahu, Armando Fox, Shoaib Kamil, Benjamin Lipshitz, Oded Schwartz, and Omer Spillinger. Communication-optimal parallel recursive rectangular matrix multiplication. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 261–272. IEEE, 2013.

[19] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200059

Acceleration of Hydro Poro-Elastic Damage Simulation in a Shared-Memory Environment [1]

Harel Levin [a,b], Gal Oren [a,c], Eyal Shalev [d] and Vladimir Lyakhovsky [d]
[a] *Department of Physics, Nuclear Research Center-Negev, P.O.B. 9001, Beer-Sheva, Israel*
[b] *Department of Mathematics and Computer Science, The Open University of Israel, P.O.B. 808, Raanana, Israel*
[c] *Department of Computer Science, Ben-Gurion University of the Negev, P.O.B. 653, Beer Sheva, Israel*
[d] *Geological Survey of Israel, 32 Yeshayahu Leibowitz, Jerusalem, Israel*

Abstract. Hydro-PED [1] is a numerical simulation software which models nucleation and propagation of damage zones and seismicity patterns induced by wellbore fluid injection. While most of the studies in geo-physical simulation acceleration and parallelization usually focus on exascale scenarios which are translated into vast meshes, encouraging a distributed fashion of parallelization, the nature of the current simulations of Hydro-PED dictates amount of data that can conveniently fit on a single compute node - NUMA and accelerator memory alike. Thus shared-memory parallelization (such as OpenMP) can be fully implemented. In order to utilize this insight, Hydro-PED was interfaced with Trilinos [2] linear algebra solvers package, which enabled an evolution to iterative methods such as CG and GMRES. Additionally, several code sectors were parallelized and offloaded to an accelerator using OpenMP in a fine grained manner. The changes implemented in Hydro-PED gained a total speedup of x5-x12, which will enable Hydro-PED to calculate long-term simulation scenarios of hundreds of years in a feasible time - a few weeks rather than a year.

Keywords. Geological Simulation, Accelerators, Numerical Linear Algebra, Shared Memory, Trilinos

1. Introduction

On the past few decades, the increasing compute power encourages the development and usage of scientific computer simulations as a preliminary step before traditional experiments and industrial development. The geophysical research area is no exceptional as applications like FLAC [4], GEOS [5] and others provide numerical simulations for a

[1]This work was supported by the Lynn and William Frankel Center for Computer Science. Computational support was provided by the NegevHPC project [3].
Special thanks goes to A. Tarabay, M. Homel and J. White, our colleagues in Lawrence Livermore National Laboratory for their priceless support and guidance during this work.
Source code is accessible at `https://github.com/harellevin/Hydro-PED`.

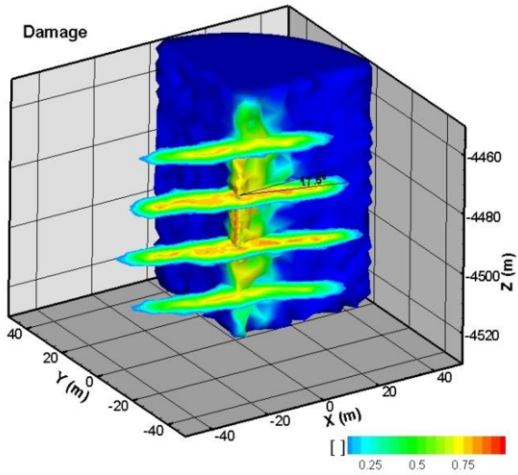


Figure 1. [1] Hydro-PED simulation of damage to granite wellbore after 80 hours of fluid injection.

wide range of geophysical scenarios [6] [7]. *Hydro-PED* [1] was developed in 2013 in order to simulate hydro fracturing [8].

Hydro-PED was first introduced as a serial program written in Fortran 90, which models nucleation and propagation of damage zones and seismicity patterns induced by wellbore fluid injection. The model formulation of Hydro-PED accounts for the following general aspects of brittle rock deformation: (1) Nonlinear elasticity that connects the effective elastic moduli to a damage variable and loading conditions; (2) Evolution of the damage variable as a function of the ongoing deformation and gradual conversion of elastic strain to permanent inelastic deformation during material degradation; (3) Macroscopic brittle instability at a critical level of damage and related rapid conversion of elastic strain to permanent inelastic strain; (4) Coupling between deformation and porous fluid flow through poro-elastic constitutive relationships incorporating damage rheology with Biot's poroelasticity. Figure 1 presents an output from Hydro-PED simulation of damage to granite wellbore after 80 hours of fluid injection, originally presented on [1].

Most of the studies in geo-physical simulation software acceleration and parallelization usually focus on exascale scenarios which simulate mesh simulation while focusing on either wide areas or high resolution. As a result, most of the simulations result in a vast and dense mesh of cells, encouraging software developers to adopt a distributed fashion of parallelization in order to divide the computational load between as many computational cores as possible. However, this is not the case in current Hydro-PED simulated scenarios as even a relatively coarse grained mesh with static pre-defined fine area near the wellbore edges, provides valuable insights about the simulated scenario. Therefore, the nature of the current simulations of Hydro-PED dictates the amount of data that can conveniently fit on a single compute node - NUMA and accelerator memory alike. thus shared-memory parallelization (such as OpenMP) can be fully implemented. Nevertheless, while Hydro-PED also supports simulations of a short-range of time, the *long-term* simulations are of special interest in the context of damage estimation. However, the time dimension is not subject to parallelization since each timestep depends on its precestor timestep. When it came to simulation of hundred of years in high definition, the run-

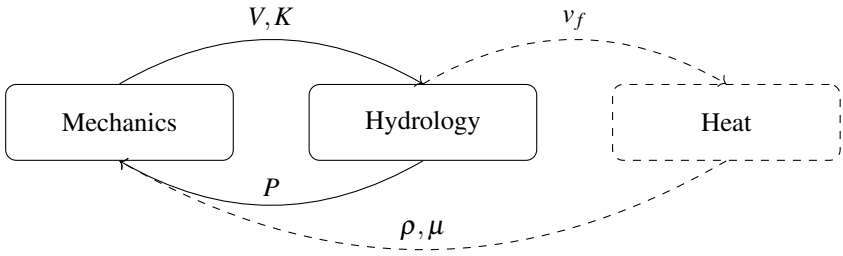


Figure 2. Schematic overview of the relations between Hydro-PED modules. The Thremodynamics module is currently under development.

time of Hydro-PED became unfeasible. Consequently, it was crucial to inspect various approaches in order to accelerate each simultaion timestep on a single node.

2. Hydro-PED - Hydro Poro-elastic Damage simulation

Hydro-PED consists of two modules: (1) Mechanical-Damage module and (2) Hydrological module. The mechanical module iterates over all simulation cells and solves relevant geo-physical equations using Explicit Finite Difference Lagrangian Method (EFDLM), while the hydrological uses the Finite Element Method (FEM) to transform the diffusion equations into a linear equations system. Later, the module initiates a third party direct solver (HSL [9]). A tetrahedral mesh is used to describe the physical area throughout all simulation modules. Each tetrahedron is referred as *element* and each of its vertices is referred as *node*. Another module which couples heat equations is currently under development. Figure 2 provides a schematic overview of the relations between the modules. These relations will be explained on the following subsection.

2.1. EFDLM Mechanics Module

Explicit Finite Difference Langrangian Method (Hence EFDLM) is a fully explicit numerical method relies on a large-strain explicit Langragian formulation originally developed by Cundall [10].

The module solves the force balance equation for each node in the mesh. The forces over the body are induced from the underground stresses and a Frequency Independent Damping. Using the force balance, the velocity of each node is obtained. Later, the strain tensor of each element is being calculated, which is induced by the combination of elastic and plastic strain. The strain tensor enables the calculation of the new coordinates of each node. Using this data, the module produces the volume (V) and permeability (K) of each element which will be used in the hydrological module.

The mechanical module uses an adaptive timestep. This timestep is calculated at each iteration such that simulation cycles will be more frequent whenever a rapid changes occur, and will be rare when there are no notable changes on rock's form. The damage state of the rock is calculated at each step. Whenever a failure occurs, the simulation walks into a subroutine called *drop* which performs several healing actions.

Prior to the current work, some OpenMP directives were implemented in certain parts of the mechanical module of Hydro-PED. However, the acceleration achieved by these directives were insufficient.

2.2. FEM Hydrology Module

On each hydrological step, the corresponding module receives the current volume and permeability level of each element. The module solves a differential equation which describes the diffusion of fluid pressure through the rock. In order to do so, the module uses the well-known finite elements method (FEM) which yields a system of linear algebraic equations. For a given N elements, the equations are of type $A\vec{x} = \vec{b}$, where A is a $N \times N$ sparse symmetric positive matrix, b is a pre-calculated vector of size N representing the flux on each element, and x is the target solution vector of size N. The vector x represents the pressure (P) on each element. These values will be used later by the mechanical module to simulate the next timestep.

Since Hydrological changes in the wellbore tends to be less extensive, a hydrological step will occur after every couple of mechanical steps, excepts when the mechanical timestep is too long. Due to the adaptive timestep mechanism implemented in Hydro-PED, the geological changes between each two consecutive steps are relatively small. Consequently, the numerical errors generated by the simulation are minor and can be neglected.

2.3. FEM Heat Module (under development)

The heat module will be called at every hydrological step. It shall receive the fluids velocity (v_f) and calculate the diffusion and advection of the temperature over the fluids. Using the finite elements method, the heat equations can be translated into a system of linear algebraic equations, $A\vec{x} = \vec{b}$. However, due to the advection, matrix A will be asymmetric. This fact enforces a usage of slightly different solution methods. The solution of the linear system yields the density (ρ) and viscosity (μ) of the fluid.

2.4. Implications of the input file

The nature of the input file implicate the concrete execution in several ways. First, it dictates the size of the grid which strongly influences the actual size of the arrays. By Amdahl's law, bigger problem size derives greater parallelization potential. Hence, the input file implicates the speedup factor.

Second, the material type and the balance of forces defined in the input file influences the size of the timestep of both the mechanics and the hydrology modules. A fractional crystalized rock suffering from great pressure forces, tends to have rapid nucleation and propogation of fraction, which in turn increases the amount of mechanical timesteps. On the other hand, porous rocks will require additional hydrological steps. Hence, the speedup factors of each module will dictate total speedup of hydro-PED which will be different for each simulation scenario.

2.5. Bottlenecks and Challenges

Considering the common scenario, most of the computation time of Hydro-PED is spent by the solution process of the hydrological linear algebraic system. After the construction of the the system (i.e. constructing matrix A, and vector $\vec{b}$, and after allocating memory space for the solution vector $\vec{x}$, the system was sent to a third-party solver library called HSL [9]. This solver uses an algorithm which has time complexity of $O(n^3)$ (for matrix

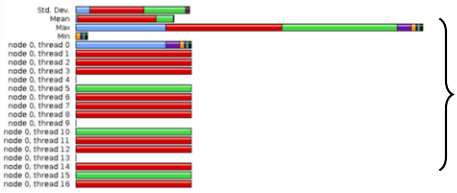


Figure 3. TAU profile results for the mechanical module. Red bars represents execution time of subroutine *derivation*.

Figure 4. TAU profile results for *move_grid*'s execution time per call. Most of the time was spent by the main thread.

of size $n \times n$), as will be explained on section 4. Hence, finding more efficient way to solve the linear system, given the modern heterogeneous system architectures, was one of the first necessary steps.

Another disadvantage of using HSL was its lack of support for asymmetric matrices. Keeping on mind that the next challenge will be the coupling of the heat module with the current modules, which will yield an asymmetric matrix, forced a search after new solvers.

Yet another aspect which was investigated is how to boost the performance of the mechanical module itself. As mentioned before, some parts of the module were parallelized using OpenMP directives in a fine-grained manner. However, we were curious whether the power of accelerators can be utilized to achieve another speedup, which, as mentioned before, was crucial in order to shorten the amount of time the run spent on each timestep.

3. Speedup Using Explicit Asynchronous Offload

In order to find how to exploit better performance from the mechanical module, a parallelization-driven profile of the code was conducted. The profile focused the search to a subroutine which calculated the displacement of mesh elements coordinates. While the execution-time per-call of the subroutine was relatively small, this subroutine was called over and over, resulting in a consumption of on not less than 41% of module's total runtime. By implementing both parallelization and time-sharing offload, extra speedup was achieved.

3.1. Profiling

TAU Performance System [11] is a commonly-used profiler which is aimed to the task of profiling runtime of parallel applications. TAU shows how much time was consumed by each subroutine on each MPI rank and by each thread. Profiling Hydro-PED using TAU provided the bar-chart provided on figure 3 (the bars representing threads #17-#31 were deleted as they showed just the same behavior as the other threads). It is clear from the chart, that most of the runtime is consumed by the subroutine indicated by dark-red color. This subroutine turned to be a subroutine called *derivation* which is initiated from the subroutine *move_grid*. The subroutine *move_grid* is used to calculate the displacement of elements' coordinates (based on the velocities induced by the pressure vectors). Closer inspection of the subroutine showed that each invocation of the subroutine is relatively fast (figure 4). However, this subroutine is used on frequent occasions which explains its vast time consumption.

```
1  subroutine move_grid(dt)
2    ...
3    ! calculate new coordinates based on velocities
4    do i=1,nodes_count
5        cord(i) = cord(i) + vel(i)*dt
6    end do
7
8    do i = 1,elements_count
9    ! derivation of basic functions
10     call derivation(i,cord,dr)
11     strain(i) = calculate_new_strain(strain(i), dr)
12     ! update fluid pressure
13     pf_el(i) = calculate_new_pressure(pf_el(i), dr, dt)
14   end do
15 return
16 end subroutine move_grid
```

Listing 1: The implementation of *move_grid* subroutine.

Listing 1 provides the implementation of subroutine *move_grid*. Note that parts of the code were encapsulated in subroutines (i.e. *calculate_new_pressure* and *calculate_new_strain*) for simplicity. After close inspection of the called subroutines, making sure that *move_grid* is a SIMD (Single Instruction Multiple Data) calculation, the entire content of the subroutine was wrapped in OpenMP's parallel-SIMD directive. In order to further improve the runtime, we considered offload to Intel® Xeon-Phi® 5110p co-processor (formerly known as Knights Corner - KNC) [12]. The performance on newer Xeon-Phis should outperforms our results. Offload to GPGPU accelerators using OpenMP 4.5 is not yet fully supported by most common Fortran compilers. As it will be implemented, we will be able to perform the same offload to NVIDIA® accelerators as well.

3.2. Asynchronous Accelerator Offload

In order to use the accelerator we consider a *host-target model* where the NUMA cores of the machine are considered as the *host* which executes the application. Whenever needed, the host can offload part of the computation to one of the accelerators attached to it. Hence, the accelerators are referred as the *targets*. The offload is done by special system calls who can be initiated either by run-time API or compiler directives (such as in OpenMP 4.5 [13]).

In order to perform a calculation on an accelerator, the input data should be offloaded to the accelerator first. Later, after the accelerator completes its calculations, the results should be sent back to the host. The time consumed by the communication between the host and the target is relatively large and is correlated linearly with the size of the data. Consequently, the effectiveness of of the offload is usually subject to the computational load of the calculation itself. Nevertheless, while *move_grid*'s computational load is pretty light, one can benefit from splitting the calculation between the host and the accelerator. The basic idea is to send the input arrays to the accelerator asynchronously as soon as possible, even before *move_grid* was called. Than, when the application initiates the subroutine, *cord* array will be calculated by both the host and the target simultaneously. Then, the host will calculate part of *strain* and *pf_el* array, while the target

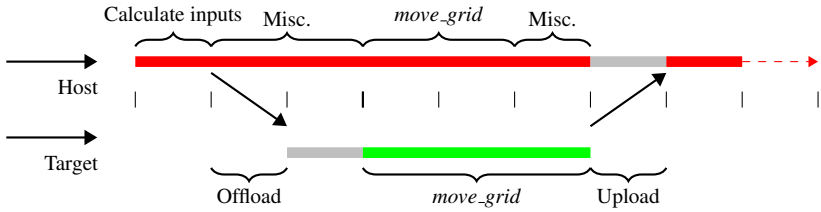


Figure 5. A timeline that demonstrates a common execution scenario where *move_grid* runs simultaneously on both host and target. The communication between the host and the target is asynchronous.

will calculate the rest of these arrays. Whenever the host finished calculating his part, he will continue the execution of the program to the point where the value of *strain* or *pf_el* are necessary. Whenever the target finishes his part of the computation he will send his output back to the host asynchronously.

This concept is demonstrated by the timeline presented on figure 5. The red line above represents the host, and the green line below represents the target. The timeline starts with the calculation of the arrays which are crucial for the calculations performed on *move_grid*. Whenever these arrays are ready, they will be offloaded to the accelerator, while the host performs another calculations. When the host steps into *move_grid*, the target will initiate the calculation on his part of the data. The host may continue with miscellenous calculations while the target still calculates his part. However, whenever the host reaches a part of the code where either *strain* or *pf_el* arrays are crucial, he will make sure the fresh data from the target was received (otherwise, he will wait).

```
1   subroutine move_grid(dt)
2     ...
3     !dir$ offload begin target(mic:0) wait(vel_signal) nocopy(cord : REUSE
       ↪  RETAIN) out(strain(1:phi_length): REUSE RETAIN) nocopy(vel : REUSE
       ↪  RETAIN) nocopy(m_biot : REUSE RETAIN) nocopy(nop : REUSE RETAIN)
       ↪  nocopy(de,dr,i,j,ii,nn,n) in(dt,nodes_count,phi_length)
       ↪  signal(strain_signal)
4     call move_grid_inline(1,phi_length)
5     !dir$ end offload
6
7     !dir$ offload begin target(mic:1) wait(vel_signal) nocopy(cord : REUSE
       ↪  RETAIN) out(strain(phi_length+1:2*phi_length) : REUSE RETAIN) nocopy(vel
       ↪  : REUSE RETAIN) nocopy(m_biot : REUSE RETAIN) nocopy(nop : REUSE RETAIN)
       ↪  nocopy(de,dr,i,j,ii,nn,n) in(dt,nodes_count,phi_length)
       ↪  signal(strain_signal)
8     call move_grid_inline(phi_length+1,2*phi_length)
9     !dir$ end offload
10
11    ! Host Part
12    call move_grid_inline(2*phi_length+1,ne)
13    return
14
15    contains
16      subroutine move_grid_inline(offset_begin,offset_end)
17  end subroutine move_grid
```

Listing 2: The implementation of the asynchronous offload in *move_grid* subroutine.

Listing 2 shows how the asynchronous offload was implemented using OpenMP. The main logic of *move_grid* subroutine was moved to another subroutine called *move_grid_inline* (listing 3) which gets two parameters which indicate the beginning and end indices of the strains array which should be handled on the current invocation. The subroutine *move_grid* calls *move_grid_inline* three time: one time for each MIC and one time for the host itself. The commands which call the MICs' *move_grid_inline* are wrapped in an offload directive which waits to a *vel_signal* signal indicating the calculation of the velocities array has already been done. The directive states the identity of the traget MIC, and dictates allocation and transport of the part of the strains array which is being handled. Whenever the MIC finishes calculating the strains vector, it sends a *strain_signal* signal.

```
1  subroutine move_grid_inline(offset_begin,offset_end)
2    !dir$ attributes forceinline :: move_grid_inline
3    !dir$ attributes offload:mic :: move_grid_inline
4    integer :: offset_begin,offset_end
5
6    !$OMP PARALLEL
7    !$OMP DO SIMD
8    do i=1,nodes_count
9      cord(i) = cord(i) + vel(i)*dt
10   end do
11   !$OMP DO SIMD PRIVATE(dr,ii,de,j,nn)
12   do i = (offset_begin),(offset_end)
13     call derivation(i,cord,dr)
14     strain(i) = calculate_new_strain(strain(i), dr)
15   end do
16   !$OMP END PARALLEL
17 end subroutine
```

Listing 3: The implementation of *move_grid_inline* subroutine.

3.3. Results

We implemented an asynchronous offload of *move_grid* subroutine. The results were tested on a machine with two sockets Intel® Xeon® CPU E5-2660 v2 processors. Each of them has 10 cores with clock rate of 2.2GHz. The machine contains two Intel® Xeon-Phi® 5110p co-processors. The total runtime of *move_grid* was measured on three settings: (1) Host only mode. That is, all the calculations were performed on the host itself. (2) Offload of about two-thirds of the array to one accelerator. (3) Offload of $5/11$ of the array to each of the two accelerators. Figure 6 shows the measured runtime for each of the settings. The offload to two coprocessors gained a speedup of about factor two compared to the host-only setting. This results suggests that asynchronous offloading is beneficial. As offload latency of accelerators is getting smaller ans smaller in modern architectures, asynchronous offload and host-target shared computations may be implemented on computations to enhance their performance.

We should note that the NUMA nature of the node is undifferentiated by the current implementation of the EFDLM module. That is, the memory is allocated on the master thread's socket memory. This implementations derives some overhead when remote access is done from the second socket. Obviously, when all the threads are allocated on a

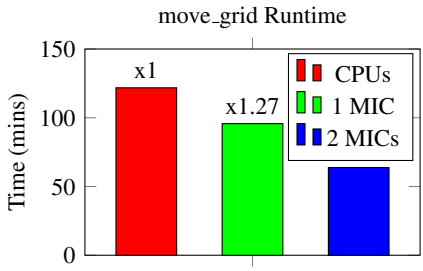


Figure 6. The runtime (in minutes) of subroutine *move_grid* using: (1) only CPUs, (2) CPUs and one Xeon-Phi accelerator, (3) CPUs and two Xeon-Phi accelerators.

single UMA (such as if a compact affinity was defined), this overhead should not occur. However, when the grid defining the scenario is sufficiently large, the speedup gained by distributing the problem to more than one socket's amount of threads overcomes the time overhead.

4. Speedup Using Advanced Linear Algebra Solvers

The task of solving linear algebraic systems of equations is one of the fundamental problems in scientific computing. During the past decades a lot of methods were devised in order to tie the best solution method to each problem given any hardware architecture. While initiation of third-party libraries of solvers for this task is usually the preferable step, it should be taken into account that not all the solvers were created equal. Each solver may use different methods which may applicable for different type of matrices [14]. Furthermore, not all the solvers implemented in a way that exploits the features and characteristics of the underlying hardware. The last fact is even more true when it comes to heterogeneous hardware and accelerators [15]. Consequently, when we looked for a way to speedup the hydrology module, the linear solver was the usual suspect.

4.1. Methods to Solve Linear Algebraic Equations

Roughly speaking, all the algorithms to solve linear systems can be divided to two types: (1) Linear methods, which translate the original system into equivalent more simple system, and then solve the equivalent system. (2) Iterative methods, which starts with initial guess ($\vec{x_0}$) for the solution vector $\vec{x}$. Later, the iterative algorithm calculates the residual which is the "distance" between the guess and the correct solution, i.e. $|A\vec{x_0} - A\vec{x}|$. The algorithm will try to refine the initial guess until the residual will be lower than a predefined threshold. Each linear algorithm is distinct by the permutations and factorizations it adopts to achieve the equivalent system. Each iterative algorithm is distinct by the way it performs the refinements.

In order to achieve the simple equivalent system in linear methods, there are several techniques to manipulate the original system. These techniques usually evolves all the vector and matrix cells in the entire system. Therefore, the time complexity of the equivalent system calculation step is usually larger than $O(n^2)$ (considering a system with matrix A of size $n \times n$). The time complexity of the solution step is usually $O(n^2)$. One way to achieve a simple equivalent system is to perform a *LU-factorization* [16], where $A = LU$ such that L (respectively U) is a matrix which have non-zero values only on

its lower (respectively upper) triangle. Prior to these work, Hydro-PED used the linear solver HSL_MA87 [17] which uses Cholesky factorization where $A = LL^T$ such that L is a matrix which have non-zero values only on its lower triangle, and L^T is transposed L. Both LU-factorization and LL^T-factorization have time complexity of $O(n^3)$.

While the calculation of the equivalent system in linear methods evolves the entire vector and matrix cells in the system, the calculation of the residual in iterative method can be done by using only the non-zero cells solely. Consequently, using iterative methods on sparse matrices (i.e. $\left|\{(i,j)|A_{i,j} \neq 0\}\right| = O(n)$), the time complexity of each iteration will be $O(n)$ plus the time complexity of the solution refinement step which is usually $O(n)$ too. In conclusion, the time complexity of iterative method which is applied on matrix A of size $n \times n$ and which is converged after k iterations will be $O(nk)$.

4.2. Trilinos - Solver for Heterogeneous Systems

Trilinos [2] is a collection of open source libraries which are used as building blocks. Following a recent work about usage of second-generation Trilinos [18], we used several libraries which we interfaced with Hydro-PED:

- Techos - Provides wrappers for BLAS and LAPACK, smart pointers and parameter lists [19].
- Kokkos - Implements a programming model in C++ for writing performance portable applications targeting all major HPC platforms. It supports MPI, OpenMP, Pthreads and CUDA [20].
- Tpetra - Implements linear algebra objects which are built on Kokkos [21].
- Belos - Implements most of the common iterative solution methods of linear systems [22].

The combination of these building blocks provides a strong and versatile framework to solve linear systems. The main additive value of Trilinos is the usage of Kokkos as an encapsulated framework which enables the programmer to exploit different types of HPC architectures and technologies without any major changes in the code. This fundamental feature makes Trilinos optimal for heterogenous systems which contains traditional CPUs along with GPGPUs and Xeon-Phis. Moreover, Trilinos uses blocking (i.e. tiling) methods in order to achieve better performance by dwelling each block in a single UMA. This NUMA-aware approach gains greater speedups as shown in [23].

We used *Belos* package to implement the well known iterative algorithm *Conjugate-Gradients* (CG) [24] which shows relatively rapid convergence for symmetric matrices (as the matrix yielded by the diffusion equations of the hydrology module). We may use the *Generalized minimal residual* (GMRES) method [25] in the future when we will deal with asymmetric matrices on the heat module.

4.3. Reuslts

In Hydro-PED's hydrology model, each timestep yields a new matrix which is slightly different from the previous matrix. Therefore, we would not be able to exploit the advantage of one-time factorization in the linear method. Furthermore, for a matrix of size $n \times n$ yielded by Hydro-PED, there will be about $12n$ non-zero values (due to geographical and geometrical considerations [1]). Moreover, the differences in the solution vector

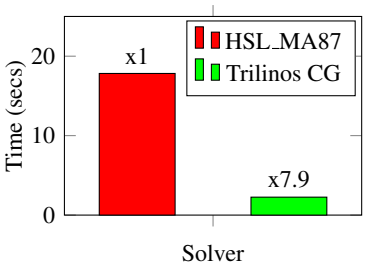


Figure 7. Solvers overall runtime with 11e4 nodes using HSL_MA87 and Trilinos.

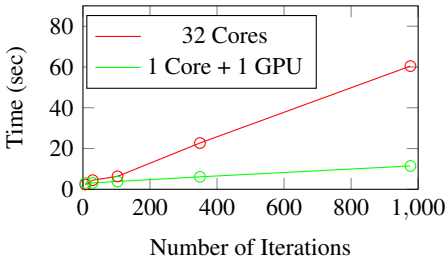


Figure 8. Trilinos overall runtime as a function of iterations amount, on multi-core CPUs and on GPU.

$\vec{x}$ between each two consecutive steps is relatively small such that the solution of previous timestep may be used as a good initial guess for the following timestep. As a result of these consideration, HSL's linear solver was replaced with the implementation of CG given by Trilinos' Belos package.

We tested both HSL and Trilinos on a common scenario of a mesh with 110,000 nodes. Both tests ran on 32 cores provided by machine with two sockets of Intel® Xeon® Gold 6130. Figure 7 shows the overall runtime of each solver (in seconds). The average number of iterations needed until Trilinos solver converged was 160. Trilinos showed speedup of almost x8 comparing to HSL_MA87.

We investigated the influence of GPU accelerators on the runtime of Trilinos. We used a relatively big (but still applicable) scenario of a mesh with 8.5 million nodes which took about 2GB of GPU memory capacity. We ran several simulations with this mesh, each takes different number of iterations to converge. The simulations ran on a machine with two sockets of Intel® Xeon® Gold 6130 and one NVIDIA® Tesla® V100 GPGPU. Figure 8 shows the overall runtime of Trilinos solver using different amount of iterations. We can learn from the trends of the chart that while the initialization of the solver using the GPU took about 3 times more than the initialization without accelerator (probably due to memory offload), the runtime of each iteration on the GPU was 15 times faster than on the CPUs. Consequently, the usage of GPU started to pay-off starting from 20 iterations.

5. Conclusion

In this paper we show several useful techniques to profile, analyze, and enhance the performance of scientific applications on a shared-memory environment, using geophysical application as a test-case. Hydro-PED's code was built in a heterogeneous and modular fashion which dictated different kinds of treatment. In the mechanics module, the explicit manner of calculations directed us to profile the runtime and find bottlenecks. In the hydrology module, which was based on FEM and linear systems, we used an of-the-shelf solution. However, choosing this solution should be done carefully, as we demonstrated.

Both on the hydrology module and the mechanics module, the usage of accelerators gained an extra speedup which are shown in figure 9. These speedups were gained either by synchronous or asynchronous offload. These techniques can be implemented in many other places, both in Hydro-PED and in another applications.

As both the balance between hydrological and mechanical steps, and the amount of iterations until convergence of Trilinos changes between different scenarios, the total

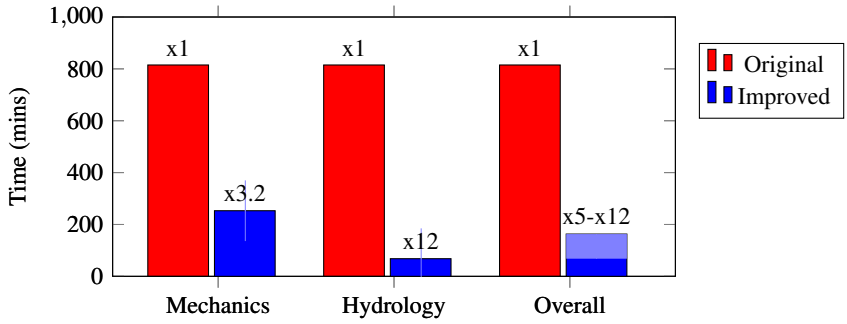


Figure 9. The speedup gained by: (1) asynchronous offload in the mechanics module, (2) using Trilinos in the hydrology module, and (3) overall. Note that the overall speedup of Hydro-PED is dictated by the balance between these modules (see 2.4).

speedup of our work is expected to be between x5-x12 comparing to the original execution. Using this speedup, a large-scale simulation which would have taken a year, will finish on a few weeks.

References

[1] Shalev, E., Lyakhovsky, V.: Modeling reservoir stimulation induced by wellbore fluid injection. In: Thirty Eighth Workshop on Geothermal Reservoir Engineering, Stanford University Stanford, California (2013)
[2] Heroux, M., Bartlett, R., Hoekstra, V.H.R., Hu, J., Kolda, T., Lehoucq, R., Long, K., Pawlowski, R., Phipps, E., Salinger, A., Thornquist, H., Tuminaro, R., Willenbring, J., Williams, A.: An overview of trilinos. Technical Report SAND2003-2927, Sandia National Laboratories (2003)
[3] NegevHPC homepage. http://www.negevhpc.com Accessed: 2019-02-19.
[4] FLAC homepage. https://www.itascacg.com/software/flac Accessed: 2019-06-24.
[5] Settgast, R., Johnson, S., Fu, P., Walsh, S., Annavarapu, C., Hao, Y., White, J., Ryerson, F.: Geos: a framework for massively parallel multi-physics simulations. theory and implementation. (2014)
[6] Homel, M., Herbold, E.: Fracture and frictional contact in the material point method using damage-field gradients for velocity-field partitioning. (2015)
[7] Annavarapu, C., Settgast, R.R., Vitali, E., Morris, J.P.: A local crack-tracking strategy to model three-dimensional crack propagation with embedded methods. Computer Methods in Applied Mechanics and Engineering **311** (2016) 815–837
[8] Shalev, E., Lyakhovsky, V.: The processes controlling damage zone propagation induced by wellbore fluid injection. Geophysical Journal International **193**(1) (2013) 209–219
[9] HSL: collection of fortran codes for large-scale scientific computation. See http://www.hsl.rl.ac.uk (2007)
[10] Cundall, P.A.: Numerical experiments on localization in frictional materials. Ingenieur-archiv **59**(2) (1989) 148–159
[11] Shende, S.S., Malony, A.D.: The tau parallel performance system. The International Journal of High Performance Computing Applications **20**(2) (2006) 287–311
[12] Chrysos, G.: Intel® xeon phi coprocessor (codename knights corner). In: 2012 IEEE Hot Chips 24 Symposium (HCS), IEEE (2012) 1–31
[13] OpenMP application program interface version 4.5. https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf
[14] Sarkar, T., Siarkiewicz, K., Stratton, R.: Survey of numerical methods for solution of large systems of linear equations for electromagnetic field problems. IEEE Transactions on Antennas and Propagation **29**(6) (1981) 847–856
[15] Li, R., Saad, Y.: Gpu-accelerated preconditioned iterative linear solvers. The Journal of Supercomputing **63**(2) (2013) 443–466

[16] Cryer, C.W.: The lu-factorization of totally positive matrices. Linear Algebra and its Applications **7**(1) (1973) 83–92
[17] Hogg, J.D., Reid, J.K., Scott, J.A.: Design of a multicore sparse cholesky factorization using dags. SIAM Journal on Scientific Computing **32**(6) (2010) 3627–3649
[18] Lin, P., Bettencourt, M., Domino, S., Fisher, T., Hoemmen, M., Hu, J., Phipps, E., Prokopenko, A., Rajamanickam, S., Siefert, C., et al.: Towards extreme-scale simulations for low mach fluids with second-generation trilinos. Parallel processing letters **24**(04) (2014) 1442005
[19] Bartlett, R.A.: Teuchos c++ memory management classes, idioms, and related topics, the complete reference: a comprehensive strategy for safe and efficient memory management in c++ for high performance computing. Technical report, Sandia National Laboratories (2010)
[20] Edwards, H.C., Trott, C.R.: Kokkos: Enabling performance portability across manycore architectures. In: 2013 Extreme Scaling Workshop (xsw 2013), IEEE (2013) 18–24
[21] Baker, C.G., Heroux, M.A.: Tpetra, and the use of generic programming in scientific computing. Scientific Programming **20**(2) (2012) 115–128
[22] Bavier, E., Hoemmen, M., Rajamanickam, S., Thornquist, H.: Amesos2 and belos: Direct and iterative solvers for large sparse linear systems. Scientific Programming **20**(3) (2012) 241–255
[23] Rajamanickam, S., Boman, E.G., Heroux, M.A.: Shylu: A hybrid-hybrid solver for multicore platforms. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium, IEEE (2012) 631–643
[24] Hestenes, M.R., Stiefel, E.: Methods of conjugate gradients for solving linear systems. Volume 49. NBS Washington, DC (1952)
[25] Saad, Y., Schultz, M.H.: Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. SIAM Journal on scientific and statistical computing **7**(3) (1986) 856–869

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200060

BERTHA and PyBERTHA: State of the Art for Full Four-Component Dirac-Kohn-Sham Calculations

Loriano Storchi [a,b] Matteo De Santis [b,c] Leonardo Belpassi [b]

[a] *Dipartimento di Farmacia, Universitá degli Studi 'G. D'Annunzio', Via dei Vestini 31, 66100 Chieti, Italy*

[b] *Istituto di Scienze e Tecnologie Molecolari, Consiglio Nazionale delle Ricerche c/o Dipartimento di Chimica, Biologia e Biotecnologie, Universitá degli Studi di Perugia, Via Elce di Sotto 8, 06123 Perugia, Italy*

[c] *Dipartimento di Chimica, Biologia e Biotecnologie, Universitá degli Studi di Perugia, Via Elce di Sotto 8, 06123 Perugia, Italy*

Abstract.

While since mid-seventies it has been clearly shown that relativistic effects play a crucial role for the complete understanding of the chemistry of molecules especially when heavy elements are involved, still nowadays in most of the case the relativistic effects are introduced via approximate methods. The main motivation behind the introduction of such approximation, respect to the natural and most rigorous component (4c) formalism derived from the Dirac equation, is the computational burden. In the present paper we are proposing a review of the BERTHA code that, together with the recently introduced Python bindings, represents the state of the art for full 4c calculations both in term of performances as well as in terms of code usability.

Keywords. four-component Dirac-Kohn-Sham

1. Introduction

Relativistic effects, arising by the fast moving of the core electrons and propagating into the valence region, it has been largely shown to become very important for the proper understanding of the chemical properties of molecules [1,2]. Indeed, expecially when heavy or Super-Heavy atom are involved the inclusion of relativistic effects is fudamental also in the deep understanding of the chemical bond [3,4]

More recently the developments of new Free Electron Lasers (FEL) provide a range of opportunities to achieve significant advances that extend the boundaries of our knowledge in the field of atomic and molecular science and promise to obtain direct information on the basis processes of energy relaxation/ transfer in molecules (how charge, spin, orbital and eventually nuclear degrees of freedom interact to redistribute the energy). The development of accurate theoretical and computational methods, based on first principles, for the accurate characterization of electronic dynamics including spin-orbit cou-

pling in molecular systems containing heavy atoms is now one of the most important challenges of theoretical chemistry and computational science [5].

Because of the computational cost of the rigorous way to include relativity (including spin-orbit effect) in the modelling of molecular systems, a disparate set of approximate methods have been derived from the strictly relativistic 4-component (4c) formalism derived from the Dirac equation by Bertha Swirles [6]. Among these the so called 2-components approximation, deriving from the decoupling the "large" and "small" components of the Dirac spinors [7], is the most used. Maybe the Douglas-Kroll-Hess [8] and Zero Order Regular Approximation hamiltonians [9] are, amoung others, the most popular two-component schemes. Both of them have found a wide range of applications with implementations in several modern commercial codes.

Clearly the introduction of the cited approximation scheme is motivated by the intrisic computational difficulty realted to a proper full 4c approach. The BERTHA code, we will describe here, is basically built around a smart and efficient algorithm for the analytical evaluation of relativistic electronic repulsion integrals, developed by Quiney and Grant in Oxford more than a decade ago [10], representing the relativistic generalization of the well-known McMurchie-Davidson algorithm [11]. As we will show in the following we have extended the applicability range of all-electron DKS calculations, exploiting density fitting techniques and parallelization strategies, to large clusters of heavy metals [12].

Aside what previously stated more recently we introduced a maior improvement in the BERTHA code usability introducing a set of Python bindings [13]. In the present paper, after a review of the parallelization strategies adopted, we will describe in details the PyBERTHA charecteristics and implementation. Finally we will conclude with a test case of the code involving the interection of a flerovium atom [14] with some gold atoms cluster.

The achievements, we will describe in the following, represent the state of the art for full 4c calculations and give us the ideal starting point for the necessary further development of methods of relativistic theory.

2. Computational details

In the following subsections we will give all the details about the fundamental features of the BERTHA code. Specifically we will summarizing the basic strategy behind the parallelization of the code, and ,maybe more importantly, we will give all the details about the newly developed Python interface of BERTHA.

2.1. Parallelization strategy

Historically the main motivation for the use of approximate methods to include relativity (e.g. including spin-orbit effect) in the modelling of molecular systems is the assumption that full 4c approach is computationally too demanding [15,16]. While this is in principle an obvious assumption, we have shown that one can drastically reduce the computational cost of a Dirac-Kohn-Sham (DKS) calculation, by implementing various parallelization and memory distribution [17,18,12] schemes and by introducing new algorithms, such as those based on the method "density fitting" [19]. We already shown that is now possible

to carry out DFT calculations at full relativistic 4c level in an efficient way, and thus exploit the full power behind the DKS equations, in a wide range of molecular systems [4, 20,3].

The main goal we prosecuted has been to almost nullify any serial portion of the code, being inspired by the basic idea behind Amdahl's law [21]. On the other hand, considering the amount of memory required to perform a DFT calculations at full relativistic 4c level, we designed an ad-hoc method to simulate shared memory for distributed memory computers following a path already designed by other quantum chemistry softwares (e.g., ADF [22] uses GlobalArray [23], GAMESS-US [24] uses the Distributed Data Interface [25] library). The ad-hoc method we designed as been specifically shaped on the data distribution induced by the problem, more specifically dictated by the grouping of G-spinor basis functions in sets characterized by common origin and angular momentum.

In order to recall the main aspects of the implemented parallelization strategy, it is important to point out the fundamental steps of each BERTHA run. Once the molecular system geometry has been specified, together with the basis and fitting set to be used, provided an initial guess density (i.e. cast as a superposition of atomic densities), the **density fitting** is carried out. Soon after the software builds the **Coulomb plus exchange-correlation matrix**, and at this stage, only during the first iteration, also the **one-electron** and **overlap** matrices are computed and stored in memory as they do not vary from cycle to cycle. Finally the **DKS matrix** is assembled and the **eigenvalue problem** is solved.

While the full description of the implemented parallelization strategy has been reported in our previous works[17,18,12], for the sake of completeness here we are reporting only a quick overview starting from the results presented in Figure 1. As the reported results indicate we are able to achieve good results both in term of speed-up as well in terms of memory distribution.

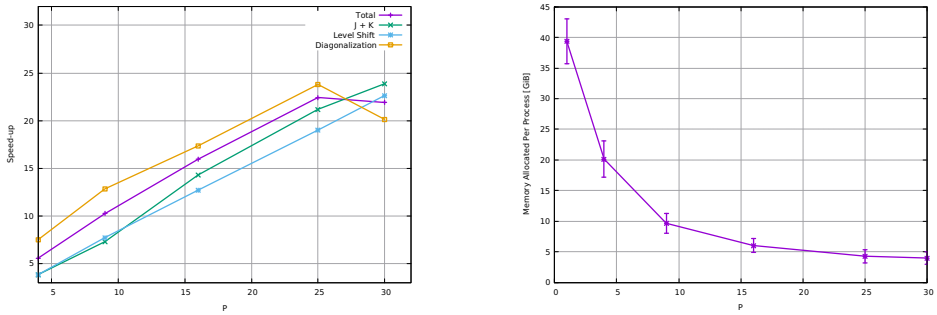


Figure 1. On the left panel we are reporting the speed-up obtained for the $Au_{20} - Fl$ complex with the current implementation of BERTHA. On the right panel we are instead reporting the memory allocated per process, for the same molecular system, when running the code using an increasing number of processes P. The results are obtained compiling the code with Intel Parallel Studio XE 2019 and Intel Math Kernel Library , and running on a two nodes cluster quipped with Intel(R) Xeon(R) CPU E5-2650 v2 at 2.60GHz and InfiniBand interconnection.

Whenever a linear algebra operation is needed we took advantage of the widely available ScaLAPACK [26] library. We recall here that the P processes of a generic parallel execution are, in ScaLAPACK, mapped onto a $P_r xP_c$ two-dimensional process grid of P_r process rows and P_c process columns. Almost consequently any dense matrix is then decomposed into blocks of suitable size, which are uniformly distributed along

each dimension of the process grid according to a specific the so-called Block Cyclic Distribution (BCD) pattern.

We implemented some utility functions (see [17,18,12] for details) that are used to efficiently map the main matrices as they are computed into the BCD distribution schema. It is indeed important to underline as the memory consumption per process, thanks to the cited approach, is always well under control as well reported in the right panel of Figure 1. We even observe cases of superlinear performance when the small size of the local arrays permits improved cache reuse to prevail over other factors. Also considering that the **PZHEGVX** function we are using to carry out the complex DKS matrix diagonalization is able to converge on a given subset of eigenvectors. That is, in our case, we are converging only in the occupied spinors subset and this gives us an evident advantage respect to the serial code, where instead we are always converging on the full set of spinors.

The grid shape affects appreciably the performance of the linear algebra routines and different routines may be differently influenced depending on the block size, number of processes, and size of molecular systems. We already reported a spread in performance of up to 50% when rectangular processes grids are used, this appear to be unfavorable mainly for the diagonalization step compared to square grids [18]. Clearly this explain the decrease on performances observed when 30 process are used. Indeed we are using a rectangular processes grid, that is a 5x6 or equivalently 6x5 one, thus a rectangular one.

2.2. PyBERTHA: a Python API for the BERTHA code

Undoubtedly the Python programming language is emerging as one of the most important and used HLL [27,13] also in the field of scientific computing. Python HLL, besides providing an extensive range of modules to be used to solve comprehensive set of computational problems, enables for a quick prototyping, being so a natural choice in the BERTHA project.

The very first step toward a Python binding started reworking the original "monolithic" FORTRAN [28] BERTHA code so that it becomes a set of SO libraries: **libbertha** containing all the basic kernel functions, **libberthaserial** to perform the serial run, and **libberthaparalleshm** to execute the parallel computation. Once this very first step has been completed the computational kernel of the DKS calculation it is driven by a FORTRAN module named **bertha_wrapper**. The FORTRAN module contains a set of methods to access to all the basic quantities, such as energy, density and DKS matrices and more. That same FORTRAN module is used to access all the basic functionality such as: **bertha_init** to perform all the memory allocations, **bertha_main** to run the main SCF iterations, clearly **bertha_finalize** to basically free all the memory, and more.

Finally the main PyBERTHA module, named **berthamod**, has been developed using the **ctypes** Python module. The cited module provides C compatible data types, and allows calling functions in shared libraries. In order to simplify the direct interlanguage communication between Python and FORTRAN, we developed a C layer called **c_wrapper**, as well summarized in Figure 2.

In the actual version of the code the input geometry, basis and fitting set are specified via a file and the related **set_fnameinput** and **set_fittfname** methods. Additionally the **pybertha** class is populated with all the basic functionality need to easily implements basic procedure as a single-point energy calculation (i.e. using the **run** and **get_etotal**

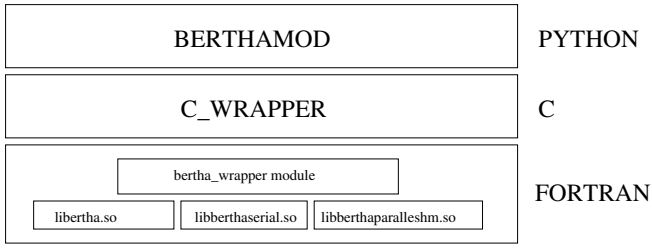


Figure 2. An overview of the software and HLL layers.

methods), or geometry optimization and also much more complex procedures as for example a real-time TDDFT [5] (i.e. using **realtime_init**, **get_realtime_dipolematrix** and **get_realtime_fock** to obtain the DKS matrix given as an input a density matrix).

All data produced at the FORTRAN layer and retrieved at the Python level can be easily handled in case of scalars. Instead when we need to transfer matrices, and in our case matrices of complex numbers, we used the FORTRANC interoperability **iso_c_binding** module and two Python functions: **doublevct_to_complexmat** to convert array of double into Python complex **numpy.array**[29], and **complexmat_to_doublevct** to perform the opposite operation. For the sake of completeness we are reporting in the following the code of the two cited functions:

```
def complexmat_to_doublevct (inm):

    if len(inm.shape) != 2:
        return None

    if inm.shape[0] != inm.shape[1]:
        return None

    dim = inm.shape[0]

    cbuffer = numpy.zeros((2*dim*dim), dtype=numpy.double)
    cbuffer = numpy.ascontiguousarray(cbuffer, dtype=numpy.double)

    cbuffer[0::2] = inm.flatten().real
    cbuffer[1::2] = inm.flatten().imag

    return cbuffer

def doublevct_to_complexmat (invector, dim):

    if (invector.size != (2*dim*dim)):
        return None

    outm = numpy.zeros((dim,dim), dtype=numpy.complex128)

    inmtxreal = numpy.reshape(invector[0::2], (dim,dim))
    inmtximag = numpy.reshape(invector[1::2], (dim,dim))
    outm[:,:] = inmtxreal[:,:] + 1j * inmtximag[:,:]

    return outm
```

The approach to move data from the FORTRAN layer up to the Python one we just described it is not necessarily the most efficient, indeed one can maybe use a direct memory mapping between the FORTRAN array and the numpy ones. Nevertheless, given the results of the Python overhead we will shortly illustrate, we believe that the way we used it is the less error-prone and the best compromise in term of code portability and efficiency.

Indeed adopting the described technique we performed some test to exactly estimate the overhead related to the Python binding. All the results, reported in Table 1, have

been obtained compiling the code with the Intel(R) FORTRAN and Python compilers (version: 2018.3.222), but similar results, in term of percentage of the Python binding overhead, have been obtained using the GNU compilers.

Table 1. We are reporting the impact of the Python binding in the total execution time using 10 SCF iterations. The code has been executed on a Intel(R) Xeon(R) CPU E3-1220 compiling the code with the Intel(R) compiler version: 2018.3.222

System	Matrix Dimension	Wall-time 10 SCF iterations with Python (s)	Wall-time 10 SCF iterations without Python (s)	Python overhead 10 SCF iterations
H_2O	140	3.910	3.906	0.09 %
Au_2	1560	104.458	104.354	0.99 %
Au_4	3152	613.912	613.483	0.07 %
Au_8	6304	3965.911	3964.078	0.05 %

Looking at Table 1 it is evident that the impact of the Python binding is almost always lower then 0.1 % and thus it is clearly negligible. The only system where the overhead is higher then 0.1 % is the Au_2 gold cluster. We may only speculate that in such a case there is some effect related to the cache memory size, indeed the most demanding part of the Python binding is essentially related to a memory-to-memory copy.

As we already pointed-out the Python binding overhead is almost solely related to the arrays copying process that, in the case of a single point ENERGY calculation, is executed just once at the end. Thus clearly, in the case of a standard single point energy calculation, the Python binding has no impact on the serial execution time of BERTHA at all.

3. Bond Analysis of $Au_{20} - Fl$ complex with NOCV/CD method

In the last decade, experiments were carried out to compare the chemical behaviour of SHEs (Super Heavy Elements) with that of their lighter homologues of the 6th period. For example by gas-phase thermochromatography studies of volatility through adsorption on gold surfaces [30].

Here we will try shed some light on the $Au_{20} - Fl$ interaction using the NOCV/CD analysis scheme [4]. In a previous work we presented the formalism used to decompose the CD function in terms of NOCVs in the context of the relativistic four-component framework where spin-orbit coupling is included variationally [4].

The core idea of the approach is the decomposition, via natural orbitals for chemical valence (NOCV), of the so-called charge-displacement (CD) function into additive chemically meaningful components.

Firstly we start recalling the CD function that is defined as a partial integration along a suitable z axis of the difference $\Delta\rho(x,y,z')$ between the electron density of the adduct and that of its non-interacting fragments (that is the Au_{20} gold cluster and Fl atom in our specific case) placed at the same equilibrium position they occupy in the adduct:

$$\Delta q(z) = \int_{-\infty}^{z} dz' \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \Delta\rho(x,y,z')dxdy \tag{1}$$

In the previous equation the integration axis is obviously chosen following some physical criteria, for instance the bond axis between the fragments appears to be the most convenient choice in our case. Any CD function features positive values along z when, upon formation of the bond, charge is transferred from right to left across a plane perpendicular to the bond axis through z. On the contrary negative values of the CD function identify charge flow in the opposite direction.

Finally, it is worth noting that the electron density difference can be further partitioned when both the adduct and its constituting fragments belong to the same symmetry group, $\Delta\rho$ can be expressed in terms of additive symmetry components. More generally, a different scheme can been applied to provide the decomposition of the $\Delta\rho$ in terms of contributions arising from the molecular spinors most involved in the bonding. Natural orbitals for chemical valence (NOCV) were introduced by Mitoraj and Michalak [31] as descriptors of chemical bond. The cited formalism allows a very compact description of the bonding phenomenon, indeed the electron density difference $\Delta\rho$ can be brought into diagonal contributions in terms of NOCVs (i.e. additive chemically meaningful components).

The cited NOVC/CD analysis has been applied to the $Au_{20} - Fl$ complex, as reported in Figure 3, using a molecular geometry as reported in a previous work [20]. It is somehow important to underline the fact that, while the geometry optimization of the gold cluster has been performed using the zero-order regular approximation (ZORA)[32], both the Flerovium gold cluster distance and clearly all the calculation needed to produce the final NOCV/CD results, have been performed using BERTHA.

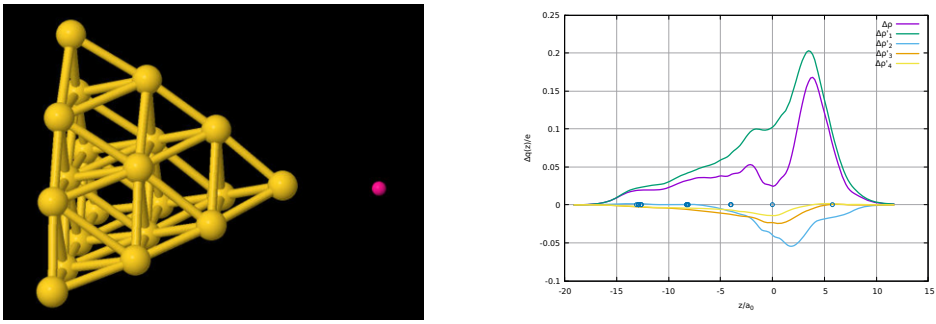


Figure 3. On the left panel we are reporting $Au_{20} - Fl$ complex. On the right panel instead the CD analysis for the $Au_{20} - Fl$ bond is reported, where the dots on the axis mark the z coordinate of the atoms. We are reporting the contribution to deformation density, $\Delta\rho$, of the four most significant NOCV-pairs ($\Delta\rho'_1$, $\Delta\rho'_2$, $\Delta\rho'_3$ and $\Delta\rho'_4$,).

In Figure 3 we are reporting the complex geometry, on the left panel, and the CD curves on the right. The fundamental feature of the $\Delta\rho$ CD curve is that $\Delta q(z)$ is appreciably positive everywhere in the cluster region. This means that there is a shift of charge from the Fl atom towards the gold cluster. More interestingly, the shift of charge does not stop at the nearest Au layer but extends appreciably down to the fourth layer. Another interesting feature of the $\Delta\rho$ CD curve is showing are the two peaks, one corresponding to the zone between the first and second Au layers, and the other corresponding to the gold-Fl binding region.

Finally looking at Figure 3, it is undoubtedly interesting to see how we are able to split the total CD curve into several additive chemically meaningful components. In

the figure we are reporting only the first four NOCVs, that are quantitatively the most relevant. It is clear how the total curve, that it is highlighting an shift of charge from the Fl atom towards the gold cluster, is instead made of two different contributions. The first, $\Delta\rho'_1$, is indeed representing a shift of charge from the Flerovium toward the gold cluster, but the other three curves are instead displaying and an opposite charge flow, from the Au_{20} cluster toward the Fl atom.

The reported results give a clear view of the power the NOVC/CD analysis, that it is able to give clear insights on the nature of a chemical bond in a simple and visual way. It is somehow important here to remark the fact that these results have been made possible only by the previously reported effort in terms of code optimization and code parallelization.

4. Conclusions and perspectives

As already stated mainly because of the computational cost of the rigorous way to include relativity in the modelling of molecular systems, a disparate set of approximate methods have been derived from the strictly relativistic 4-component (4c) formalism derived from the Dirac equation by Bertha Swirles [6].

In the present work we described the BERTHA code, that as a result of our effort, can be considered the state of the art for full 4c calculations. Indeed, thanks to the the parallelization strategies and density fitting techniques adopted, we have been able to extend the applicability range of all-electron DKS calculations to extremely big molecular systems.

In addition, we introduced also a set of Python bindings (so called PyBERTHA), that we demonstrated to have almost a negligible impact in terms of time consumption. Instead the introduction of such software layer improved enormously the code usability, especially respect to the original FORTRAN version.

Specifically, due to the actual enormous diffusion of the Python programming language, we hope to spread the use of full 4c calculations to a larger scientific population. Indeed, thanks to the introduced Python API, it is now simple and quick to implement and test new approaches based on the basic building blocks provided by the current version of PyBETHA.

Clearly, given the described abstraction layer it is now easy to extend the API as needed. In a future coming version we are planning to add all the fundamental functions to specify the input geometry and basis set in a more user-friendly (i.e. pythonic) way.

References

[1] L. Belpassi, L. Storchi, H. M. Quiney, and F. Tarantelli, "Recent advances and perspectives in four-component Dirac-Kohn-Sham calculations," vol. 13, pp. 12368–12394, 2011.

[2] P. Pyykkö and J. P. Desclaux, "Relativity and the periodic system of elements," *Accounts of Chemical Research*, vol. 12, no. 8, pp. 276–281, 1979.

[3] M. De Santis, L. Belapassi, F. Tarantelli, and L. Storchi, "Relativistic quantum chemistry involving heavy atoms," *Rendiconti Lincei. Scienze Fisiche e Naturali*, vol. 29, no. 2, pp. 209–217, 2018.

[4] M. De Santis, S. Rampino, H. M. Quiney, L. Belpassi, and L. Storchi, "Charge-displacement analysis via natural orbitals for chemical valence in the four-component relativistic framework," *Journal of chemical theory and computation*, vol. 14, no. 3, pp. 1286–1296, 2018.

[5] M. Repisky, L. Konecny, M. Kadek, S. Komorovsky, O. L. Malkin, V. G. Malkin, and K. Ruud, "Excitation energies from real-time propagation of the four-component dirac–kohn–sham equation," *Journal of chemical theory and computation*, vol. 11, no. 3, pp. 980–991, 2015.

[6] B. Swirles, "The relativistic self-consistent field," *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, vol. 152, no. 877, pp. 625–649, 1935.

[7] M. Reiher and A. Wolf, *Relativistic quantum chemistry: the fundamental theory of molecular science*. John Wiley & Sons, 2014.

[8] M. Reiher and A. Wolf, "Exact decoupling of the dirac hamiltonian. ii. the generalized douglas–kroll–hess transformation up to arbitrary order," *The Journal of chemical physics*, vol. 121, no. 22, pp. 10945–10956, 2004.

[9] E. v. van Lenthe, J. Snijders, and E. Baerends, "The zero-order regular approximation for relativistic effects: The effect of spin–orbit coupling in closed shell molecules," *The Journal of chemical physics*, vol. 105, no. 15, pp. 6505–6516, 1996.

[10] I. P. Grant, *Relativistic quantum theory of atoms and molecules: theory and computation*, vol. 40. Springer Science & Business Media, 2007.

[11] L. E. McMurchie and E. R. Davidson, "One-and two-electron integrals over cartesian gaussian functions," *Journal of Computational Physics*, vol. 26, no. 2, pp. 218–231, 1978.

[12] S. Rampino, L. Belpassi, F. Tarantelli, and L. Storchi, "Full parallel implementation of an all-electron four-component Dirac-kohn-Sham program," vol. 10, no. 9, pp. 3766–3776, 2014.

[13] M. F. Sanner *et al.*, "Python: a programming language for software integration and development," *J Mol Graph Model*, vol. 17, no. 1, pp. 57–61, 1999.

[14] P. Schwerdtfeger, "One flerovium atom at a time," *Nature chemistry*, vol. 5, no. 7, p. 636, 2013.

[15] H. M. Quiney, H. Skaane, and I. P. Grant, "Relativistic calculation of electromagnetic interactions in molecules," vol. 30, no. 23, p. L829, 1997.

[16] H. M. Quiney, H. Skaane, and I. P. Grant, "*Ab initio* relativistic quantum chemistry: four-components good, two-components bad!," vol. 32 of *Advances in Quantum Chemistry*, pp. 1–49, Academic Press, 1998.

[17] L. Storchi, L. Belpassi, F. Tarantelli, A. Sgamellotti, and H. M. Quiney, "An efficient parallel all-electron four-component Dirac-Kohn-Sham program using a distributed matrix approach," vol. 6, no. 2, pp. 384–394, 2010.

[18] L. Storchi, S. Rampino, L. Belpassi, F. Tarantelli, and H. M. Quiney, "Efficient parallel all-electron four-component Dirac-Kohn-Sham program using a distributed matrix approach II," vol. 9, no. 12, pp. 5356–5364, 2013.

[19] L. Belpassi, F. Tarantelli, A. Sgamellotti, and H. M. Quiney, "Poisson-transformed density fitting in relativistic four-component Dirac-Kohn-Sham theory," vol. 128, no. 12, p. 124108 (11), 2008.

[20] S. Rampino, L. Storchi, and L. Belpassi, "Gold–superheavy-element interaction in diatomics and cluster adducts: A combined four-component dirac-kohn-sham/charge-displacement study," *The Journal of chemical physics*, vol. 143, no. 2, p. 024307, 2015.

[21] J. L. Gustafson, "Reevaluating amdahl's law," *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, 1988.

[22] G. t. Te Velde, F. M. Bickelhaupt, E. J. Baerends, C. Fonseca Guerra, S. J. van Gisbergen, J. G. Snijders, and T. Ziegler, "Chemistry with adf," *Journal of Computational Chemistry*, vol. 22, no. 9, pp. 931–967, 2001.

[23] J. Nieplocha and R. Harrison, "Shared memory programming in metacomputing environments: The global array approach," *The Journal of Supercomputing*, vol. 11, no. 2, pp. 119–136, 1997.

[24] M. W. Schmidt, K. K. Baldridge, J. A. Boatz, S. T. Elbert, M. S. Gordon, J. H. Jensen, S. Koseki, N. Matsunaga, K. A. Nguyen, S. Su, *et al.*, "General atomic and molecular electronic structure system," *Journal of computational chemistry*, vol. 14, no. 11, pp. 1347–1363, 1993.

[25] G. D. Fletcher, M. W. Schmidt, B. M. Bode, and M. S. Gordon, "The distributed data interface in gamess," *Computer Physics Communications*, vol. 128, no. 1-2, pp. 190–200, 2000.

[26] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker, "Scalapack: A scalable linear algebra library for distributed memory concurrent computers," in *[Proceedings 1992] The Fourth Symposium on the Frontiers of Massively Parallel Computation*, pp. 120–127, IEEE, 1992.

[27] J. M. Perkel, "Programming: pick up python," *Nature News*, vol. 518, no. 7537, p. 125, 2015.

[28] S. J. Chapman and S. J. Chapman, *Fortran 95/2003 for scientists and engineers*. McGraw-Hill, 2008.

[29] S. Van Der Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: a structure for efficient numerical

computation," *Computing in Science & Engineering*, vol. 13, no. 2, p. 22, 2011.

[30] I. Zvára, *The Inorganic Radiochemistry of Heavy Elements: Methods for Studying Gaseous Compounds*. Springer Science & Business Media, 2008.

[31] M. A. Mitoraj M., "Natural orbitals for chemical valence as descriptors of chemical bonding in transition metal complexes," *J. Mol. Model.*, vol. 13, pp. 347–355, 2007.

[32] E. van Lenthe, A. Ehlers, and E.-J. Baerends, "Geometry optimizations in the zero order regular approximation for relativistic effects," *The Journal of chemical physics*, vol. 110, no. 18, pp. 8943–8953, 1999.

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200061

Prediction-Based Partitions Evaluation Algorithm for Resource Allocation

Anna PUPYKINA [a,1], Giovanni AGOSTA [a]
[a] *Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy*

Abstract. Resource allocation is a well-known problem, with a large number of research contributions towards efficient utilisation of the massive hardware parallelism using various exact and heuristic approaches. We address the problem of optimising resources usage on deeply heterogeneous platforms in the context of HPC systems running multiple applications with different quality of service levels. Our approach manages the partitioning within a single heterogeneous node aiming at serving as many critical applications as possible while leaving to the upper levels of runtime resource management the decision to preempt resources or to launch the critical application on a different node. We investigate predictive allocation algorithms, allowing to serve up to 20% more high priority requests when using a moving average or machine learning prediction model vs baseline without prediction.

Keywords. NUMA Shared Memory, Resource management, Prediction, Memory management, High Performance Computing

1. Introduction

The push towards Exascale supercomputers is leading to increasingly heterogeneous High Performance Computing (HPC) architectures, characterised by the coupling of accelerators to the more traditional HPC cores. Such future HPC architectures integrating different kinds of hardware accelerators, such as general-purpose graphics processing units (GPGPUs) and reconfigurable computing resources (e.g. Field Programmable Gate Arrays, FPGA), can be classified as *deeply heterogeneous architectures* [1]. At the same time, the new classes of applications, such as real-time high-performance applications, are emerging that require Quality of Service (QoS) guarantees. The typical practice of reserving a subset of the supercomputer to a single application becomes less attractive, leading to the exploration of cloud technologies in the context [2]. So that, a viable scenario is that of multiple applications, with different QoS levels, coexisting on the same deeply heterogeneous HPC infrastructure and sharing accelerators. For this scenario to be successful in practice, resources need to be allocated with a vision that includes both the application requirements and the current and future state of the overall system.

Thus, resource utilisation prediction can be employed to forecast the future state of the cluster based on statistical information on past behaviour. Then the prediction can be used to guide the response to the resource allocation request in the best way to ful-

[1] Corresponding Author: Anna Pupykina, Politecnico di Milano, Via Ponzio 34/5, 20133 Milan, Italy; E-mail: anna.pupykina@polimi.it.

fil the QoS requirements of the applications while optimising the use of resources (primarily, processing elements and memory) in a system-wide perspective. Recent studies in resource utilisation prediction are directed to resource optimisation in cloud architectures [3], on the other hand, research related to the prediction in HPC is mostly focused on the predicting execution time and queue waiting time. More specific prediction for HPC application is presented in [4]. A grammar-based approach for modelling and predicting the I/O behaviour of HPC applications allows to recognise when future I/O operations will occur (i.e., predict the interarrival time between I/O requests), as well as where and how much data will be accessed. [5] gives an in-depth survey of the most recent state-of-art memory management techniques for HPC and Cloud Computing which are used on the different layers of hardware/software stack.

In this work, we focus on the *memory-centric prediction-based partitions evaluation* algorithm for resource allocation on deeply heterogeneous Non-Uniform Memory Access (NUMA) architectures. Here, multiple accelerators coexist within a single node and can cooperate for a single application composed of multiple kernels, or they can be partitioned among different applications. For efficient resource allocation, resource and memory management solutions should be closely interrelated to take into account data dependencies between tasks. The proposed approach regards to the hierarchical resource management strategy that includes the following levels of resource management [1]: **The Global Resource Manager** (GRM) runs on a general-purpose node (GN), and it is in charge of workload balancing and thermal control of the entire system; **The Local Resource Manager** (LRM) runs on the heterogeneous nodes (HN) and on the slave GN, and it is in charge of the allocation of intra-node resources, allowing multiple applications to share resources. Our approach manages resource allocation across different applications within a single *heterogeneous node* as a part of LRM in a way that maximises resource usage while preserving the predictable execution time of critical applications.

The contribution can be summarised as follows: **concept and implementation:** a core concept of Prediction-based Partitions Evaluation Algorithm for resource allocation and several implementations of this concept were derived; **assessment:** a comprehensive assessment was provided, including measurements for a proof of concept implementation, showing the overall feasibility of the approach, as well as its scalability.

The rest of this paper is organised as follows. In Section 2 we briefly introduce the target heterogeneous architecture. In Section 3 we state our problem and describe our proposed solution, while in Section 4 we provide an experimental evaluation. Finally, in Section 5, we draw some conclusions and highlight future research.

2. Background: Runtime management in MANGO Project

The MANGO project aims at addressing the power, performance, and predictability space by dynamically using heterogeneous processing elements in a QoS sensitive computing scenario. The key feature of the MANGO resource management system is its tight integration with the programming model, which lightens the burden on the application developers, as they do not need to handle the mapping of kernels and buffers on suitable HN units [6]. A host-side low-level runtime API allows developers to indicate to the runtime which components (kernels, memory buffers, and synchronisation events) need to be shared within the heterogeneous node. These components are then connected into a

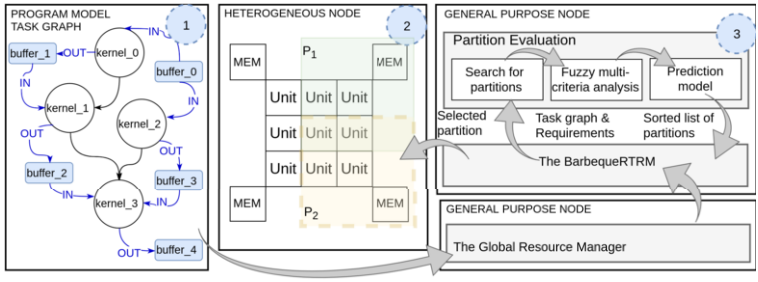


Figure 1. Overview of the target system: (1) task graph, (2) partition list, (3) partition evaluation system.

task-graph to provide the resource manager with the information needed to generate the best feasible resource allocation for the requested QoS. Figure 1 (1) shows an example of a task graph composed of several kernels and buffers. One or more processing units can perform each kernel. A buffer may be used as an output buffer for one kernel and as an input buffer for a different one. Therefore, the resource manager allocates resources based on knowledge of both the system hardware status and the application requirements and priorities.

2.1. Memory awareness

In our target architecture, all the memory modules in a given HN share a single physical address space that can be accessed by all the computational units (ARM-based nodes, GPU-like accelerators, and hardware accelerators) [1]. Despite the shared physical address space, we differentiate the following types of memory buffers: **shared memory buffer** is a memory buffer that may be simultaneously accessed by multiple kernels, such as `buffer_0` on Figure 1 (1); **private memory buffer** is a memory buffer that may be accessed by only one kernel, such as `buffer_4` on Figure 1 (1).

To achieve the optimal unit-memory connection characteristics, we proposed to use a fuzzy multi-criteria analysis with pairwise comparison [7]. The choice between memory units is based on a prediction of the future resource state to maximise the ability to allocate future high priority requests.

We adopt the following criteria for private buffer allocation: c_0^{pr} - distance between processing and memory units (in hops); c_1^{pr} - bandwidth; c_2^{pr} - jitter. Criteria $c_i^{sh}, i = 0..5$ are employed for shared buffer allocation and consist of the mean and standard deviation of the criteria c^{pr}. Fuzzy sets $\widetilde{C}$ are defined on the universal sets of P with the membership functions $m^l(p_i)$ that show the degree of membership of an element $p_i \in P$ to a fuzzy set $\widetilde{c_l}$ for each criterion in C ($l_{pr} = 0..2$ and $l_{sh} = 0..5$) on the basis of pairwise comparisons of elements of P with the relative importance coefficients w_l^{pr} and w_l^{sh}, $\sum w = 1$ for applying the concentration or dilation to the fuzzy sets.

3. Partition Evaluation Algorithm

3.1. Problem Statement

We are given a HN topology, $H = \{U_f, U_b, M_f, M_b\}$, where U_f and U_b indicates the set of free and busy units, respectively, M_f indicates the memory units with all the space

available for allocation, M_b indicates the memory units with fully or partially allocated space and a set of task-graphs $Tg = \{tg_0, \ldots, tg_n\}$, $tg =< B, K >$ where B is a set of requested memory buffers and K is a set of kernels. Considering a particular tg, for each buffer $b \in B$ of size $S(b)$ there is a kernel or set of kernels $K(b)$ which uses b (e.g., kernel read buffer $k \xrightarrow{r} b$ and/or write to buffer $k \xrightarrow{w} b$) and a partition or set of partitions $P = \{< M, U >_0, ..., < M, U >_m\}$ appropriate to allocate b on H, where M is a memory unit of size $S(M)$, $S(M) \geq S(B)$, $U \in U_f$ and $\forall k_i \in K \; \exists u_j \in U$ that able to execute k_i. For each kernel $k \in K$ there is a set of preferred target processing architectures $Arch_{prefs}(k) =< Arch^0, ..., Arch^l >$ that is noted by developer. Each application has a specific priority level $appl = \{appl_h, appl_l\}$, where the high priority application $appl_h$ needed to be allocated with the requested QoS on the current HN, and the low priority application $appl_l$ could be rescheduled on the another HN. Figure 1 (2) shows a graphical representation of an example of partitions, which could be produced by the resource allocation algorithm.

All combinations of the preferred processing architectures among with all the possible mappings of the kernel to the specific unit and memory allocations can be found by brute-force exploration. However, this approach can be time consuming other the leading to find a redundant set of mapping solutions. The overall number of mappings can be estimated by the following formula:

$$N_{map} = \prod_{i=1}^{N_K} \frac{((\sum_{a=1}^{arch(i)} N_U^a) - k_i)!}{((\sum_{a=1}^{arch(i)} N_U^a) - (1 + k_i))!} \times (N_M)^{N_B} \tag{1}$$

where $k_i = \sum_{kernel=1}^{i} [\exists kernel = j | j \in 1..i \wedge arch(i) = arch(j)]$ e.g. k_i is the number of kernels for which at least one preferred architecture is the same, N_{map} is the number of mappings, N_K and N_B are respectively the number of kernels and buffers in tg, N_U^a is the number of units with the specified architecture a, N_M is the number of memory units.

Given the size of the solution space, the time needed to find suitable solutions can often be too long for considering the execution at runtime. The heuristic goal is to limit the number of resource partitions to consider, based on the exploitation of historical data about the previous application executions. Without additional constraints, buffers from different task-graphs could be allocated on the same memory unit. This allocation can cause unpredictable interference of concurrent applications on shared memory and routing bandwidth. The easiest way to ensure that the bus access requests are served immediately is to separate resources for concurrent applications. This approach does not solve the interference problem between concurrent tasks of a single application but mitigates the stochastic influence of independent applications.

We aim to find the best $P_s =< M_s, U_s >, P_s \in P$ for each sequentially arriving $tg_i \in Tg$ by the criteria $C = \{C^{pr}, C^{sh}\}$ with the following conditions: **size:** $\sum S(B) \leq \sum S(M_s)$; **isolation:** $M_s \in M_f$ and $\forall m_i, u_j$ such that $m_i \in M_s, u_j \in U_s \; \exists < u_j, m_i >$ and $\forall m_i, u_j$ such that $m_i \in M_s, u_j \in U_b \; !\exists < u_j, m_i >$, where the tuple $< u_j, m_i >$ defines the permitted networking connection between u_j and m_i; **multi-criteria analysis:** $P_s = max_i \left\{ \frac{min_l [m^l(p_i)]^w}{p_i} \right\}$; **prediction model:** the usage of M_f for $appl_l$ is avoided in the case of predicting the use of M_f by $appl_h$.

In the case of heterogeneous systems, significant on-chip constraints, such as limited memory and route bandwidth, need to take into account. At this stage, we only consider

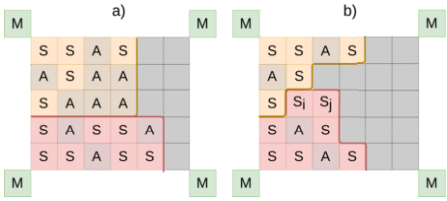


Figure 2. An example of partitions a) with rectangular isolated areas b) with irregular isolated areas (S - selected unit, A - additional unit, M - memory unit).

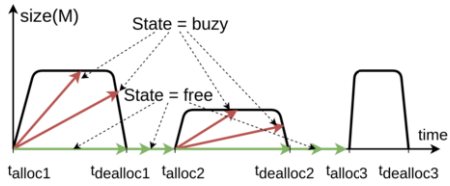


Figure 3. Input vectors and output classes for the prediction models vased on SVM algorithm.

the subproblem of resources allocation to available tiles without taking into account the consequences for the communication. Heuristics have to be used to find a solution with a reasonable quality within an acceptable time. Accordingly, we investigated the partition evaluation method based on memory usage prediction and memory characteristics comparison. A graphical overview of the proposed approach is presented in Figure 1 (3).

3.2. Isolated partition

The processing units and the memory units are connected through a 2D-mesh NoC. Each processing unit has four routing ports with an XY routing algorithm implemented to allow network connection between units. In order to limit bandwidth utilisation, processing units have to be allocated close to the memory unit. We proposed to select the processing units in *the nearest von Neumann neighbourhood* to the memory unit so as to ensure the smallest average distance from each processing unit to the memory unit. We consider two ways of forming the isolated area presented in Figure 2: **rectangular:** selected units are supplemented by adjacent ones to form a rectangular area; **irregular:** selected units are supplemented by adjacent ones to provide access to the memory units.

The available routing bandwidth on the boundary is set to 0 so as to avoid bandwidth resources usage from others partition. The rectangular area guarantees a networking connection from each processing unit to the memory unit in the specific partition. However, this isolation method occupies a large number of free processing units. On the other hand, the second approach makes it possible to use processing resources more economically. However, this type of partitioning requires moving from the simple XY routing algorithm to the adaptive XY routing algorithm, since memory unit becomes unreachable for some processing units (as an example units S_i and S_j in Figure 2b).

3.3. Prediction model

In general, a resource management system is based on an algorithm that takes runtime decisions on the basis of continuously updated information about the state of the resources. By predicting the future state of resources we can improve the quality of the management decisions [8]. We are suggesting the models that can, at runtime, predict the future use of resources so that management decisions aimed at increasing the service of high priority requests can be made.

Predictions are based on statistical information. These can be seen as statistical series, that is ordered collections of data $Y = \{Y_{i-n-1}, ..., Y_i\}$ beginning at moment t_{i-n-1}

and covering events up to the final moment t_i, where Y_j is a pair $Y_j = (t_j, v_j)$. The first element t_j defines the moment in time and the second element v_j defines the value of one variable of interest (in our case, it is an allocated size of the memory unit). We considered statistical series based on: **time:** time sampling is carried out so that the time between allocations is taken into account; **events:** resources allocation is considered as the occurrence of an event for which only the order of allocation and not the time matters.

Two linear prediction models that can be applied to runtime contexts were implemented: **the moving average method:** $\widetilde{Y_{i+1}} = m_{i-1} + \frac{1}{n}(Y_i - Y_{i-n})$, where m_{i-n} - moving average for n periods before the forecast; **the exponential weighted average method:** $\widetilde{Y_{i+1}} = \alpha Y_i + (1-\alpha)\widetilde{Y_i}$, where α - smoothing constant.

To take into account the adequacy of the prediction model Theil's coefficient of inequality $U = \sum_{i=1}^{n}(Y_i - F_i)^2 / (\sum_{i=1}^{n} F_i^2 + \sum_{i=1}^{n} Y_i^2)$ was chosen. Theil's coefficient takes the value equal to zero when the prediction model is accurate, and the value equal to one when the forecast is inadequate. The proposed evaluation algorithm does not consider prediction in cases coefficient is greater than 0.5.

To assess the possibility of using machine learning in runtime contexts, we used dlib C++ library's [9] implementation of the **pegasos algorithm for online training of SVM**. The prediction model was redefined to a simple binary classification problem in the following way: **input:** model is described as a two-dimensional input vector by the size and time derived from the last allocation; **output:** prediction is represented by two possible states of the units (busy or free) as output classes; **kernel:** radial basis function kernel defines the allocation-deallocation trend. A graphical representation of the input vectors and output classes is shown in Figure 3. In addition to online training, the **trainer for a C-SVM using the SMO algorithm** for solving the same binary classification problems was used.

3.4. Partitions evaluation

The overview of partitions evaluation algorithm is presented in [10] with buffer analysis described in detail in Algorithm 1. The input of the Algorithm 1 receives the quantitative interaction characteristics between the memory unit m onto which the analysed buffer b is mapped and the processing units u_i onto which the kernels that read and/or write to the buffer are mapped, such that $prop_r = (u_i \xrightarrow{r} m).Properties, i = 0..n_r$ and $prop_w = (u_i \xrightarrow{w} m).Properties, i = 0..n_w$. At the first step, it looks through the memory-units characteristics of each analysed partition (line 1). Some of these characteristics change at runtime (e.g., available bandwidth) or are constant (e.g., distance in hops). If the current buffer is used by one kernel privately (line 2), then the value of each criterion $val(c_{pr})$ is saved as-is (lines 3-4). Otherwise, the value of each criterion $val(c_{sh})$ is accumulated by calculating the mean and standard deviation (lines 6-9). In the next part, the matrices of pairwise comparisons M^c are filled. The total number of matrices coincides with the number of criteria. Since the matrix of pairwise comparisons is diagonal, symmetric and transitive, at line 11 all diagonal elements are set to 1, at lines 15-16 and 17-18 elements are calculated as a ratio of the specific criterion values of the two compared partitions i and j depending on the optimisation goal, that is, $c \to max$, as for bandwidth, or $c \to min$, as for distance. The degree of membership $dm^c[i]$ is calculated for each partition i (line 20) and for each criterion c (line 21) as one over the sum of the elements in the corresponding column of the matrix M^c (lines 21-23). At line 24, the

ALGORITHM 1: Buffer analysis

Data: Memory-units read-write characteristics $< m, prop_r, prop_w >^0, ..., < m, prop_r, prop_w >^n$
Result: Partition scores $s = \{s_0, ..., s_n\}$ for the buffer b

1 **for** $i = 0$ *to* n **do**
2 **if** $prop^i_r.size() + prop^i_w.size() = 1 \vee (prop^i_{r,w}.size() = 1 \wedge prop^i_r = prop^i_w)$ **then**
3 **foreach** $c_{pr} \in C$ **do**
4 $val(c_{pr}) \Leftarrow (prop^i_r \vee prop^i_w).GetProperty(c_{pr})$;
5 **else**
6 **foreach** $c_{sh} \in C$ **do**
7 **foreach** $prop^i_r$ *and* $prop^i_w$ **do**
8 $val(c_{sh}) \Leftarrow prop^i_r.GetProperty(c_{sh})$;
9 $val(c_{sh}) \Leftarrow prop^i_w.GetProperty(c_{sh})$;
10 **for** $i = 0$ *to* n **do**
11 $M^c[i][i] \Leftarrow 1$ `/* matrix of pairwise comparisons */`
12 **for** $j = i + 1$ *to* n **do**
13 **foreach** $c \in C$ **do**
14 **if** $c \rightarrow max$ **then**
15 $M^c[i][j] \Leftarrow val(c)[i]/val(c)[j]$;
16 $M^c[j][i] \Leftarrow val(c)[j]/val(c)[i]$;
17 **else**
18 $M^c[i][j] \Leftarrow val(c)[j]/val(c)[i]$;
19 $M^c[j][i] \Leftarrow val(c)[i]/val(c)[j]$;
20 **for** $i = 0$ *to* n **do**
21 **foreach** $c \in C$ **do**
22 $dm^c[i] \Leftarrow \sum_{j=0}^{n} M^c[i][j]$;
23 $dm^c[i] \Leftarrow (1.0 \div dm^c[i])^w$ `/* degree of membership */`
24 $s[i] \Leftarrow \min_c(dm[i])$ `/* intersection */`
25 **for** $i = 0$ *to* n **do**
26 **if** $GetPrediction(m_i) = Buzy$ **then**
27 $s[i] \Leftarrow s[i] \times -1.0$;

score of evaluated partition is set equal to the minimal value of the corresponding degree of membership. Finally, at lines 25-27, the overall score for each partition is updated according to the predicted memory state.

In the case of prediction errors, the proposed approach acts in the following way. If the appearance of the high priority applications is underpredicted, it allocates a higher number of the low priority applications and a fewer number of the high priority applications. In the case of overprediction of the high priority applications, the number of low priority applications is fewer than it could be allocated.

4. Experimental Evaluation

As the deeply heterogeneous architecture targeted by our work is currently under development, we investigated the proposed partition evaluation algorithm on the singe application execution in [10]. Overall, the proposed approach succeeded in evaluating the best and worst resource mappings. The memory status prediction among with partitions isolation is evaluated through a simulation-based approach.

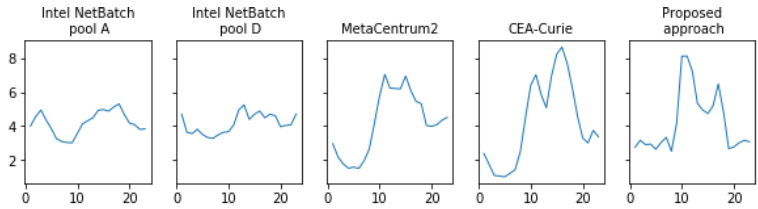


Figure 4. Arrival rates

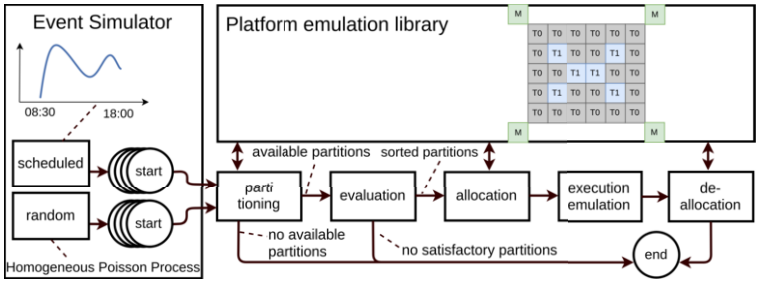


Figure 5. General scheme of the simulation

4.1. Experimental Setup

Event simulator mimics the job submission of users on a time-driven basis. The application pool was composed of workflows with a specific priority. According to our scenario where applications often perform the same tasks multiple times, but additional requests must also be handled, the arrival process was modelled including a mix of scheduled recurring applications targeting the same workflow with the same resources requests and non-recurring applications targeting various workflows. A polynomial function proposed in [11] defined the scheduled arrival process. As shown in Figure 4, accounting records [12] from the national grid of the Czech republic and Curie supercomputer operated by CEA have the arrival process similar to the mentioned above. To model events occurred completely at random at intermittent times, the Poisson process was used. The test tasks flow is consists of 30 days, where weekends are simulated by only non-recurring applications targeting various workflows occurred at random.

The target platform emulation library implements hardware-dependent API and allows performing resource allocation in a simulation mode. The three major resources, including units (processors/accelerators), memory buffers located in DDR memories, and bandwidth, are under control of the local resource manager of the emulator. All these resources are kept as internal configuration and offered for reservation based on their availability and platform restrictions. The selected configuration of the HN includes 30 processing units of two types of architecture placed in five rows and six columns grid and four memory units. The general scheme of the emulation is shown in Figure 5. Here, the experiments employed the same configuration and sequences of emulated applications.

Seven task-graphs presented in Figure 6 were created relying on the synthetic workflow [13]. The task-graphs have a various number of kernels (from 4 up to 13) and input/output buffers for each kernel. Runtime, types of kernels, memory requests and the

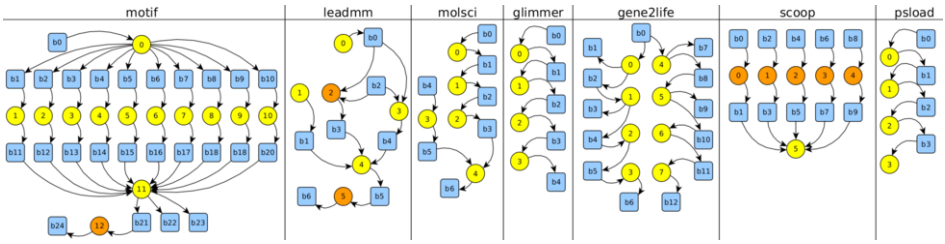


Figure 6. Task-graphs of the synthetic workflows

Table 1. Basic statistics for program simulation

	motif	leadmm	molsci	glimmer	gene2life	scoop	psload
runtime	9090	4990	1020	901	540	98	50
kernels T0/T1	12/1	4/2	5 /0	4/0	8/0	1/5	4/0
memory (MB)	1696	3535	55	2185	2.6	22	379
n. partitions	8.75E+30	1.25E+11	8.36E+10	2.61E+08	1.99E+18	1.81E+10	6.53E+07

maximum number of possible mappings on the experimental architecture according to the equation (1) are presented in Table 1.

In this paper, we consider the design-time mapping policies called recipe [10]. This file is used to specify both the per-task requirements and, optionally, a set of resource mapping solutions that the resource manager should consider at runtime. In order to investigate the time spent by evaluation and allocation algorithms, we have limited the number of mappings in the recipe to 50 and 700.

We analysed the proposed approach with the following prediction models and ways to form statistical series: **Base:** baseline approach without prediction model(PM); **MAonEvent:** approach with PM based on moving average method with event-based statistical series; **MAonTime:** approach with PM based on moving average method with time-based statistical series; **EXPonEvent:** approach with PM based on exponential weighted average method with event-based statistical series; **EXPonTime:** approach with PM based on exponential weighted average method with event-based statistical series; **SVM:** approach with PM based on pegasos algorithm for online training of SVM. **SVMtrain:** approach with PM based on C-SVM training on the 30 days arrival flow simulated in addition to the one used for experiments.

4.2. Experimental Results

First, we investigated the proposed approach with the number of mappings in the recipe limited to 50. As we expected, the resource allocation algorithm without isolation and without prediction models gives the high degree of successful allocations. As shown in Figure 7, the inclusion of the prediction model based on the moving average method in the algorithm without isolation increases by 17% the density of the high priority requests with a decrease by 12% of the overall number of the hosted application. As shown in the Figure 7 (row 3), it is caused by a high rate of rejected low priority application.Other prediction models have no significant impact on the amount of hosted applications (approx 5%). The isolation decreases the number of applications that can be allocated on the HN simultaneously. For instance, rectangular area isolation reduces the number of hosted ap-

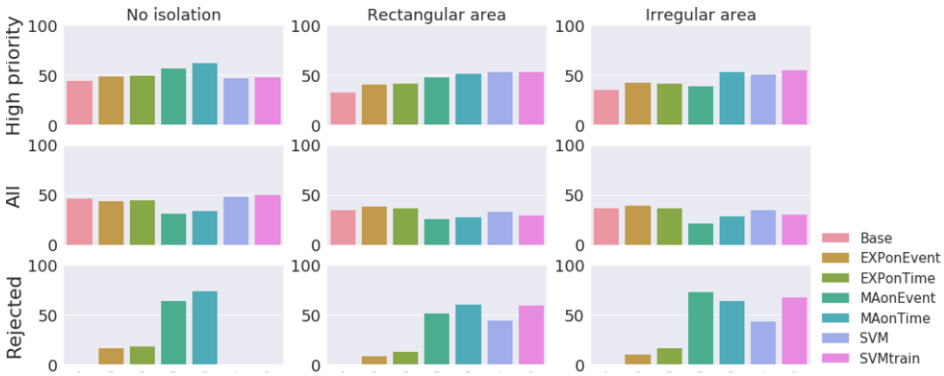


Figure 7. Percentage of the hosted high priority applications (row 1) all successful allocations (row 2) and rejected low priority applications by prediction (row 3) using resource allocation algorithm based on the rectangular and irregular area isolation and without isolation (max 50 mappings per application).

plications by 10%. Thus, for the resource allocation algorithms based on isolation, the prediction model increases the percentage of successful high priority applications. Let us consider hosted high priority applications which use resources that are allocated on the basis of rectangular area isolation using exponential weighted average prediction model. As shown in Figure 7, the percentage of these applications increases by 10% regarding the applications which use resources without prediction model and decreases by less than 5% regarding the applications which do not use isolation. In addition to this, the total amount of hosted applications increases slightly. The prediction model based on the moving average method in the algorithm with isolation shows a significant decrease in the overall number of the hosted application due to a large number of rejects due to the memory state prediction and therefore does not allocate applications with low priority.

The algorithm based on the SVM online training gives the approx. 20% advantage in hosted high priority applications regarding the baseline for the algorithm with rectangular area isolation and approx 8% increase regarding the baseline for the algorithm without isolation. Also, in the case of isolation, the prediction model based on the moving average method gives a large number of rejects due to the memory state prediction and therefore does not allocate applications with low priority. The trained SVM algorithm both for rectangular and irregular area isolations provides approx. the same number of high priority applications as the algorithm with prediction based on the moving average method. At the same time, for irregular area isolation, the amount of high priority application allocation increases by 20% regarding the baseline without prediction model, and increases approx. by 10% regarding the algorithm without isolation and prediction.

Generally, the proposed approach, with the number of mappings in the recipe limited to 50, gives adequate time for evaluation and allocation for considering the execution of the policy at runtime. As shown on Figure 8, algorithms with the prediction model based on the SVM algorithms, with both online and pre-trained learning, give approx. four times higher evaluation time than the algorithms with linear statistical prediction models. Nevertheless, the allocation time for the SVM algorithm with online training is approx. ten times higher than those of the statistical methods.

A more significant number of mappings in the recipe increase both evaluation and allocation time. Figure 9 shows the dramatical increase in evaluation time for algorithms with machine learning prediction models, especially for resource allocation without re-

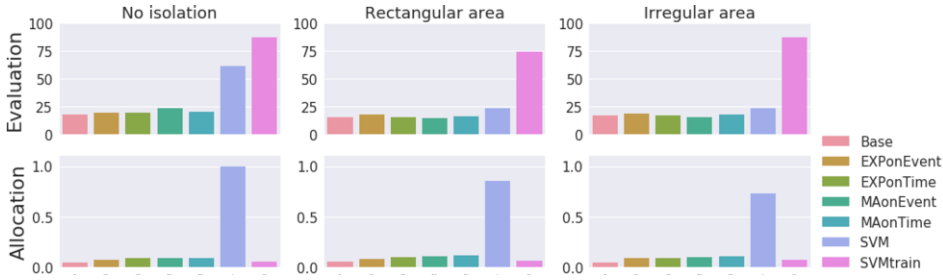


Figure 8. Evaluation (row 1) and allocation (row 2) time (in *ms*) using resource allocation algorithm based on the rectangular and irregular area isolation and without isolation (max 50 mappings per application).

Figure 9. Evaluation (row 1) and allocation (row 2) time (in *ms*) using resource allocation algorithm based on the rectangular and irregular area isolation and without isolation.

source isolation. The reason is in the fewer number of possible mappings with additional conditions on the simultaneous use of memory tile. The allocation time increases rapidly for approach with the prediction model based on the SVM algorithm with online training. It is caused by the time spent on training SVM following each new allocation.

5. Conclusions & Future Developments

In this paper, we have introduced a predictive method for partition evaluation within deeply heterogeneous architectures with NUMA shared memory. The target platform deals with workloads with different priorities for resource allocation requests, classified as a high priority and best effort. Through the use of predictive algorithms, we were able to serve up to 53% of the high priority requests vs a baseline of 32% without prediction on the isolated area. The effort on evaluation and allocation time by statistical prediction model is inessential and is about 15 and 0.1 ms respectively. It is worth noticing that prediction model based on machine learning algorithm with online training gives higher evaluation and allocation time (20 and 0.8 ms respectively) and pre-trained algorithm gives higher evaluation time (up to 100 ms) while allocation time is in the same range as for baseline algorithm. Nevertheless, partition evaluation using the prediction model based on machine learning is still under consideration due to relatively short time per allocation and slightly better results also for allocation without isolation. The proposed approach does not assume the only correct prediction model. We aim using the prediction model and the mapping isolation, depending on the global state of the system. The Global

Resource Manager taking into account the current workload and thermal state of the entire system is in charge of using the specific isolation and prediction policy. In future works, we will provide additional analysis by considering the influence of the knowledge of the possible plan of the application execution on the resources prediction models.

6. Acknowledgments

This research was partially funded by the H2020 EU projects "MANGO" (grant no. 671668) and "RECIPE" (grant no. 801137 [14]).

References

[1] J. Flich, G. Agosta, P. Ampletzer, D. A. Alonso, C. Brandolese, E. Cappe *et al.*, "Exploring manycore architectures for next-generation hpc systems through the mango approach," *Microprocessors and Microsystems*, 2018. [Online]. Available: https://doi.org/10.1016/j.micpro.2018.05.011

[2] B. Koller and M. Gienger, "Enhancing high performance computing with cloud concepts and technologies," in *Sustained Simulation Performance 2014*. Springer, 2015, pp. 47–56.

[3] D. Mishra and P. Kulkarni, "A survey of memory management techniques in virtualized systems," *Computer Science Review*, vol. 29, pp. 56–73, aug 2018.

[4] M. Dorier, S. Ibrahim, G. Antoniu, and R. Ross, "Omnisc'io: A grammar-based approach to spatial and temporal i/o patterns prediction," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2014, pp. 623–634.

[5] A. Pupykina and G. Agosta, "Survey of memory management techniques for hpc and cloud computing," *IEEE Access*, pp. 1–23, 2019. [Online]. Available: https://doi.org/10.1109/ACCESS.2019.2954169

[6] G. Agosta, W. Fornaciari, G. Massari, A. Pupykina, F. Reghenzani, and M. Zanella, "Managing heterogeneous resources in hpc systems," in *Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, ser. PARMA-DITAM '18. New York, NY, USA: ACM, 2018, pp. 7–12. [Online]. Available: http://doi.acm.org/10.1145/3183767.3183769

[7] A. Pupykina and G. Agosta, "Optimizing memory management in deeply heterogeneous hpc accelerators," in *2017 46th International Conference on Parallel Processing Workshops (ICPPW)*, Aug 2017, pp. 291–300. [Online]. Available: https://doi.org/10.1109/ICPPW.2017.49

[8] C. Ababei and M. Ghorbani Moghaddam, "A survey of prediction and classification techniques in multicore processor systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, pp. 1–1, 10 2018. [Online]. Available: https://doi.org/10.1109/TPDS.2018.2878699

[9] D. E. King, "Dlib-ml: A machine learning toolkit," *Journal of Machine Learning Research*, vol. 10, pp. 1755–1758, 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1577069.1755843

[10] G. Massari, A. Pupykina, G. Agosta, and W. Fornaciari, "Predictive resource management for next-generation high-performance computing heterogeneous platforms," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, D. N. Pnevmatikatos, M. Pelcat, and M. Jung, Eds. Cham: Springer International Publishing, 2019, pp. 470–483.

[11] M. Calzarossa and G. Serazzi, "A characterization of the variation in time of workload arrival patterns," *IEEE Transactions on Computers*, vol. C-34, no. 2, pp. 156–162, Feb 1985.

[12] D. Feitelson. (2015) Logs of real parallel workloads from production systems. [Online]. Available: http://www.cs.huji.ac.il/labs/parallel/workload/logs.html

[13] L. Ramakrishnan and D. Gannon, "A survey of distributed workflow characteristics and resource requirements," Department of Computer Science, School of Informatics Indiana University, Tech. Rep., 2008. [Online]. Available: http://www.cs.indiana.edu/l/www/ftp/techreports/TR671.pdf

[14] W. Fornaciari, G. Agosta, D. Atienza, C. Brandolese, L. Cammoun *et al.*, "Reliable power and time-constraints-aware predictive management of heterogeneous exascale systems," in *Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, ser. SAMOS '18. New York, NY, USA: ACM, 2018, pp. 187–194. [Online]. Available: http://doi.acm.org/10.1145/3229631.3239368

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200062

Unified Generation of DG-Kernels for Different HPC Frameworks

Jan HÖNIG [a] Marcel KOCH [b] Ulrich RÜDE [a] Christian ENGWER [b]
Harald KÖSTLER [a]
[a] *Friedrich-Alexander University Erlangen-Nürnberg*
[b] *University of Münster*

Abstract. Code generation specified by a DSL is a popular method to manage maintenance effort and introduce an abstraction layer for higher reusability. In the case of Galerkin methods, the Unified Form Language is a DSL for the weak formulation of a differential equation. In this paper, we present the framework-specific code generation for DUNE and ExaStencils from a problem formulated in the UFL. Moreover, we present optimization strategies, which are applied during the generation process.

Keywords. code generation, dune, exastencils, DG, UFL

1. Introduction

The Finite Element Method (FEM) is widely used for the numerical solution of partial differential equations. Variants like the discontinuous Galerkin (dG) method have drawn much attention within the last two decades. Applications range from hyperbolic problems like shallow water or Maxwell's equations to non-linear degenerated parabolic problems like multi-phase flow in porous media.

To solve these equations, the discretized problems at hand are usually manually expressed in a programming language. This usually involves using some FEM library or framework. Instead of this tedious work, the mathematical problem can be expressed at a higher level, and the code which obtains the numerical solution is generated. The code generation allows for a more flexible interface while remaining the performance compared to the manually written code. To define a given mathematical problem, we use the Unified Form Language (UFL) [1]. UFL is a domain-specific language (DSL) designed for specifying finite element discretizations in variational form and was initiated by the FEniCS Project [2].

We aim at providing code transformation components to support UFL in two different frameworks, DUNE and ExaStencils. DUNE [3,4] is a flexible framework for Multi-Physics- and Multi-Domain-Simulations, and the first option we consider as a backend for our code generation pipeline. ExaStencils [5] is a whole-program code generation framework working on block-structured grids that we use as a second backend for the generation of a dG-kernel. In this case, only certain types of elements, grids, and dG discretizations are supported.

In this paper, we introduce the targeted frameworks, explain the details of the generation pipeline, and compare the two approaches of generating dG-kernels in terms of flexibility and possible optimizations. To showcase the flexibility of our code generation, we consider a simple model problem, the linear transport equation. Within a single code transformation tool, we can handle transformations for different mesh types, two different frameworks, and mesh specific mathematical optimizations. The aim of our code generator is the generation of back end optimized target code, based on a simple problem description in UFL.

1.1. Related Work

UFL is used by both FEniCS [2] and Firedrake [6] frameworks for decleration of FEM of variational form. Recently the Firedrake project made efforts to incorporate loopy's intermediate representation (IR) into their generation framework [7]. The authors use loopy's IR to vectorize computations across multiple elements with promising performance results. The `dune-pdelab` specific code generation of our toolchain with a focus on optimizing dG kernels was first described in [8]. For the ExaStencils framework, a similar approach for quadrature-free shallow water equations was presented in [9].

2. Targeted Frameworks

Our code generation approach targets two different simulation frameworks, DUNE and ExaStencils. Although both aim to provide easy-to-use frameworks for solving partial differential equations (PDEs), the taken approaches differ significantly. DUNE can run its kernels on any mesh, whereas the specific nature of ExaStencils is limited only to regular and cartesian grids. To highlight these different needs and requirements for the code generator, we briefly describe these two frameworks.

2.1. DUNE

DUNE [3,4,10] is a C++ simulation framework for solving PDEs. DUNE relies heavily on generic programming. This allows the compiler to remove most interface-related runtime overhead. Instead of a monolithic codebase, DUNE is composed of several core modules, with a clear separation of concerns.

The `dune-common` module supplies basic dense linear algebra, MPI communications, a build system infrastructure and further basic functionality. `dune-localfunctions` supplies a wide range of finite element basis functions, e.g. (discontinuous) Lagrange functions, Raviart-Thomas basis functions or orthonormal basis functions. Reference element implementations for different geometries and quadrature rules defined on those elements are provided by `dune-geometry` and `dune-istl` offers iterative solvers and preconditioners for sparse matrices with blocking.

The `dune-grid` module defines a hierarchical grid interface, which is implemented by multiple grid managers. The interface is general enough to support grids with a wide range of features, e.g., structured and unstructured grids, conforming and nonconforming refinement, or support for multiple element types. This means that switching from one grid implementation to another usually only requires changing the type of

the grid, and further adjustment of user code is not necessary. Additionally, the interface supports parallelization using MPI.

Discretizations modules, such as `dune-fem` [11] or `dune-pdelab` [12], provide abstractions for finite volume or finite element methods. As an example, `dune-pdelab` introduces C++ classes equivalent to the mathematical notion of grid function spaces or grid operators, which apply element local kernels to every element in the specified grid. Using `dune-pdelab`, the finite volume or finite element assembly is automated to the point where users merely need to supply the element-local kernels and select the right solution scheme. Of course, users are still free to extend functionality by providing their implementations of the defined interfaces.

2.2. ExaStencils

ExaStencils is a whole-program generator [5,13], which provides a multi-layered domain specific language. Its primary focus lies on the generation of highly efficient geometric multigrid solvers for partial differential equations. ExaStencils itself is implemented in Scala. Scala, among other things, provides a powerful pattern matching mechanism, which makes the implementation of the compiler and generator software simpler in terms of development time and maintenance efforts.

The DSL, called ExaSlang [14] offers four different abstraction levels. The continuous specification of the whole simulated problem, i.e., equations, unknowns, boundary conditions, and the computational domain, is described in the first layer. The second layer states the discretized version of the problem, whereas the third layer describes a suitable solver. The combination of the second and third layers results in a complete program specification. Transitioning between these layers themselves is done in a semi-automatic manner under users' guidance. Users decide which layer is most suited for the description of the application. ExaStencils supports the transformation of ExaSlang into C++ and CUDA code, thus targeting different platforms. During this transformation, ExaStencils performs several optimization strategies, e.g., address precalculation, loop transformations including loop blocking, reordering and condition elimination, explicit vectorization, and loop carried common subexpression elimination.

In the case of C++ and a CPU target, ExaStencils can parallelize the generated code. Depending on the settings, it generates code with OpenMP, MPI, or both. For the MPI case, necessary ghost-layers or overlapping of fields can be automatically introduced as well. Given the required parallelization and the patches of fields, i.e., the splitting of the domain onto processes, ExaStencils provides the communication routines between the patches. Although ExaStencils was designed primarily for multigrid methods, it is perfectly capable of handling stencil-only applications as well.

3. Code Generation

Our code generation is based on the domain-specific language UFL [1,15]. Developed by the FEniCS [2] project, UFL describes the weak formulation of a PDE in Python. It is, therefore, best suited for finite element methods. Describing a PDE with UFL is closely related to the theoretical formulation of the PDE.

In the following example, we demonstrate the usage of UFL. Consider the Poisson Equation (1) with its weak formulation: Find $u \in H_0^1$ such that u solves Equation (2).

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega, \\ u &= 0 \quad \text{on } \partial\Omega. \end{aligned} \tag{1}$$

$$\begin{aligned} a(u,v) &= F(v) \quad \forall v \in H_0^1, \text{ where} \\ a(u,v) &= \int_\Omega \nabla u \cdot \nabla v \, \mathrm{dx} \\ F(v) &= \int_\Omega f v \, \mathrm{dx} \end{aligned} \tag{2}$$

A discretization of this problem can be constructed by choosing a triangulation for Ω and approximating H_0^1 by a space consisting of globally continuous and piecewise linear functions and the corresponding UFL formulation, without boundary treatment, now reads:

```
mesh = # triangulation of Omega
V = FunctionSpace(mesh, "CG", 1)
u = TrialFunction(V)
v = TestFunction(V)
f = # analytic definition of f
a = inner(grad(u), grad(v)) * dx
F = f * v * dx
```

Listing 1: Example UFL File

As can be seen in Listing 1, operator overloading and supplying appropriate named functions and constants allows for a straight forward translation of the weak formulation into UFL. UFL's representation is an abstract syntax tree (AST) of high-level mathematical objects as well as necessary linear algebra objects. However, representation on this level is insufficient for most optimizations or transformations towards high-performance computing. Therefore, the UFL-AST is transformed into an IR, which is suitable for the optimizations needed in the backend.

Our choice for this IR is loopy [16] together with pymbolic [17]. Pymbolic is a library for precise manipulation of symbolic expressions and is a perfect fit for representing expressions inside a code generation framework. We have chosen pymbolic over other computer algebra systems like sympy [18] because it does not change expressions implicitly and is easily extensible. Loopy's computational kernels are described by loop domains and instructions, and thus loopy is capable of handling statements, their dependencies, loops, and control flow. Additionally, loopy comes with a range of transformations based on the polyhedral model, e.g., loop tiling or loop fusion, which was shown using in a finite element method context [19]. Choosing loopy was evident since it fits perfectly as the IR.

Our code generator realizes the transformation from an UFL-AST into the IR by a tree traversal approach. As can be seen in Figure 1, this approach is used by both frameworks. The generation of the IR, which is post-processed by one of the backends afterward, also depends on the selection of the targeted framework. Thus the output expressed by the IR differs for both backends.

The traversal is realized by a visitor object with type-based function dispatch. We separate the UFL-AST node types into four different categories, geometry evaluations, basis evaluations, quadrature evaluation, and backend agnostic, which are mostly linear algebra nodes. For the node types in the first three categories, the transformation into the IR is backend-specific, for nodes from the last categories, it does not depend on the back

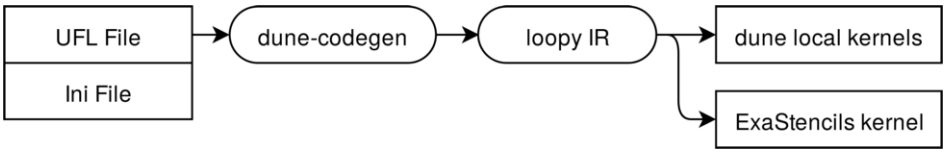


Figure 1. UFL Generation Pipeline

end. For each category, UFL-AST visitor classes are defined, which only handle the node types according to its category. The full visitor type is constructed using mixins with one class from each category. There is one additional back end specific mixin, keeping track of which equation in a system of PDEs is currently handled.

Together with the UFL file, the generation needs additional information, which is contained in the INI file. It may contain information about the selection of optimizations, spatial and temporal discretization, element type, and other settings.

3.1. DUNE Specific Generation

The `dune-pdelab` framework provides many components needed for finite element assembly, as mentioned in Section 2.1. Since these components have been thoroughly tested and used, even in high-performance settings, we rely on these parts and do not generate code replacing them. Instead, we generate local kernels, which compute element local or face local integrals. The local kernels are the most expensive part of the assembly process, except for trivial integrals. Thus, we expect the most performance gain from focusing on generating optimized code for these local kernels.

The `dune-pdelab` code generation is divided into three possible paths. The default path implements the generic `dune-pdelab` implementation of the tree visitor. The other two paths implement back end specific optimizations for high order dG or for low order continuous Galerkin (cG) discretizations. Each implementation has its optimizations for different grid types. We currently distinguish between equidistant, axis-parallel, multilinear, or generic grids. Currently, the optimizing paths of the generation require quadrilateral or hexahedral meshes, whereas the generic path also works with simplices.

In the case of tensor product finite element basis functions and reference elements, sum factorizing reduces the complexity of the local assembly process. This is especially rewarding for higher-order dG discretizations. The article [8] describes several vectorization strategies for sum-factorized kernels realized in our code generator, e.g., batching several sum-factorized sub kernels. The selection of the vectorization strategy can be defined manually or decided automatically either by a cost model or by auto-tuning.

Low order cG discretizations do not profit from sum factorization as much as dG discretizations. In these cases, locally structured meshes are a better approach. Using this optimization, a notable performance gain is achieved by operating on multiple elements in one local kernel. Additionally, this allows for cross element vectorization, which otherwise would be cumbersome to realize in `dune-pdelab`. Users have to request this optimization explicitly since using locally structured meshes increases the number of degrees of freedom similarly to uniform refinement. This needs to be addressed when creating a coarse grid.

Hardware-based optimizations, e.g., vectorization, loop tiling, or loop fusion, are possible in both optimized code generation paths. These kinds of optimizations are real-

ized as transformations of the loopy IR after the UFL form is transformed into the IR. In contrast, optimizations relying on the grid type or basis function type are handled during the transformation from UFL into the loopy IR, since these may induce algorithmic changes. Because of C's lack of standardized vectorization, loopy does not generate vectorized C code by default. We added a custom back end, which uses the wrappers defined in the vector class library [20] for generating vectorized instructions.

It is possible to generate local kernels for both matrix-based and matrix-free computations. Matrix-based computations can rely on a wide range of preconditioners for accelerating the solution of a linear system, but they gain only limited performance from recent hardware developments like vectorization. Matrix free computations, on the other hand, are FLOP bound and thus gain significant performance increases from vectorization, but the access to robust preconditioner is limited. Therefore, our optimizations are most effective for matrix-free computations.

3.2. ExaStencils Specific Generation

In contrast to DUNE, ExaStencils does not provide any components regarding the finite element method. The only data structure it provides is a Field. Because of this limitation, and because ExaStencils is a multigrid and stencil generation tool, we focus only on the particular case of a regular grid. More precisely, the focus lies on a cartesian grid, which has two triangles per square. With this limitation, the ExaStencils generation uses back end specific optimizations.

The generation itself consists of three steps. At first, the given UFL is preprocessed and traversed with an ExaStencils specific visitor, which translates the UFL description into an intermediate representation consisting of Loopy and Pymbolic expressions. During this step, we evaluate quadrature points, weights, basis functions, and the derivatives of basis functions. The evaluation is possible because of the limitations of the mesh and is the essential optimization step.

Secondly, the IR is expressed as ExaStencils code. This includes unrolling of loops, gathering additional information for the ExaStencils generator, and translate the IR as ExaStencils function. For the translation of the loopy IR, we introduced a new back-end for the ExaSlang language. Additional information given to the generation process contains the domain and array size, parallelization strategy.

The generation from the UFL formulation does not create any initialization of fields, visualization, time-stepping loop, nor a main-function. In those cases, we use the information from the INI file and use Jinja [21] templates for a flexible implementation of said accompanying program components.

In the last step of the whole generation process, the generated files are translated into C++ with ExaStencils. During this step, ExaStencils performs the optimization and parallelization, as described in the previous section.

The cartesian grid is visualized in Figure 2. Each cell has one lower and one upper triangle. The coefficients of the triangles are stored in two separate arrays and utilize the regularity of the grid, to create stencil kernels. The generated code consists of three kernels. One handles the integration of the volume of the triangle. The second one handles the integration on the faces and accesses its neighbors in a stencil pattern. The third kernel takes care of the boundaries, and triangles directly adjacent to boundaries.

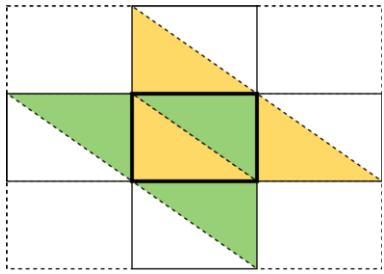


Figure 2. ExaStencil Grids

4. Numerical Evaluation

In this section, we verify our toolchain in terms of correctness and scaling using the linear transport equation,

$$\begin{aligned} \partial_t u + \beta \cdot \nabla u = 0 \quad & \text{in } \Omega \times (0,T) \\ u = 0 \quad & \text{on } \Gamma_D \\ u(\cdot,0) = u_0 \quad & \text{in } \Omega \end{aligned}$$

This equation can be used to model the transport of a concentration through a domain. We choose a discontinuous Galerkin approach with upwinding for the discretization of the problem, leading to the following UFL description in Listing 2.

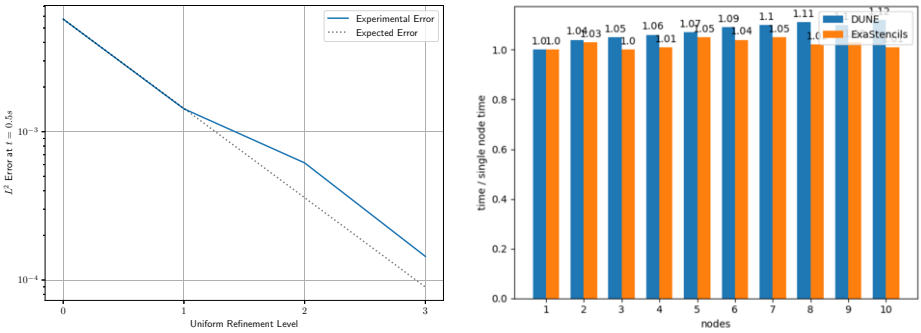
```
def upwinding_flux(normal, inside, outside):
    return (conditional(inner(beta, normal) > 0, inside, outside) *
            inner(beta, normal))
# definition of test and trial functions u and v, beta and initial value
n = FacetNormal(cell)('+')
# mass operator for temporal discretization
mass = u * v * dx
# residual operator r(u,v) = a(u,v) - F(v) for spatial discretization
r = (-1. * u * inner(beta, grad(v)) * dx +
     upwinding_flux(n, u('+'), u('-')) * jump(v) * dS)
```

Listing 2: Linear Transport

In terms of expressiveness, we can compare the lines of code (LOC) of the UFL and INI specification, and the generated code. The specification consists of 76 LOC. The code generation produces 339 LOC for DUNE and 1203 LOC for ExaStencils. In the following, we verify the correctness of our code generator by examining the convergence for a simple configuration. Additionally, we investigate the weak scaling of our generated code.

4.1. Convergence Test

For the convergence test, we use $\Omega = [0,1]^2$ and $T = 0.5$ with an constant advection $\beta = [1\ 1]^T$. The inital condition $u_0(x)$ has a bell shaped concentration of the form



(a) Convergence (b) Parallel Efficiency

Figure 3. Numerical Evaluation

$u_0(x) = cos(\pi||x - x_0||)$ for x in a radius of $r = 0.15$ around $x_0 = [0.25\ 0.25]^T$ and otherwise $u_0 = 0$. The exact solution at time $t = 0.5$ is the same as the initial value, except that the concentration is centered around $x_0 = [0.75\ 0.75]$. During this time frame, the homogenous Dirichlet condition is applicable, since the concentration does not reach the boundary.

Both backends use explicit time-stepping schemes for the temporal discretization, with a timestep size small enough to achieve a stable simulation. Currently, ExaStencils only supports the explicit Euler method, while `dune-pdelab` also supports higher-order Runge-Kutta methods, in this case, a third-order strong stability preserving scheme from [22].

ExaStencils uses a structured simplicial grid, while `dune-pdelab` uses an unstructured simplicial grid with the grid implementation from `dune-uggrid`. The coarse structured grid consists of 1600 elements, while the unstructured grid has 1700 elements on the coarsest level. In both cases, the grid is refined up to three times. Figure 3a shows the the L^2 error of the approximate solution at time $t = 0.5$ for each refinement level. As expected, a convergence order of 2 can be seen.

4.2. Weak Scaling

Next, we investigate the weak scaling of our generated codes. With weak scaling, we can show that the generated kernels still work with the respective framework at hand. Since only element local kernels are generated for the `dune-pdelab` backend, the following results are only influenced by the scalability of the used DUNE components. In [23, 24] the scaling capabilities of the `dune-istl` module are shown and in [25] scaling results using the `dune-pdelab` module can be found. In the case of ExaStencils, the kernels are generated for individual MPI processes, and its performance capabilities were demonstrated in [26]. The communication itself happens outside of the kernels and is entirely handled by the ExaStencils framework.

We consider the same test case as above, with 500 timesteps and a grid of the size 100×100 per core for DUNE and of the size 128×128 per core for ExaStencils. We simulate on the SuperMUC-NG system, which is located at Leibnitz Supercomputing Center (LRZ) in Munich.

From Figure 3b it can be seen that the `dune-pdelab` code achives over 90% parallel efficiency, which is consistent with earlier findings [23,24]. The generated code for ExaStencils achieves over 95% parallel efficiency. This demonstrates the proper usage of our frameworks.

5. Conclusion and Outlook

In this paper, we have shown that we can use one toolchain to generate UFL for different back ends, namely DUNE and ExaStencils. This approach provides us with a general and flexible description of a mathematical problem, which can be numerically solved with code generation to said back ends. In the particular case of the cartesian grid, we can utilize the excellent speed of ExaStencils, while still having the possibility to rely on DUNE's performance for any more general problem. This approach is extensible to other back ends as well, which is already done for FEniCS and Firedrake frameworks. However, each framework can still have a different intention, approach, and specific optimizations. FEniCS's primary focus is on usability and generality, at which it excels, while our work is primarily focused on performance. Firedrake is also geared towards performance, but during their code generations, they use different IRs for algorithmic and hardware optimizations. For future projects, a detailed comparison with Firedrake and its pipeline is inevitable.

The simple example of linear transport shows a promising possibility of generating fast code for a cartesian grid with ExaStencils. In the future, this approach should be expanded to regular grids together with completing all features of the UFL. This includes having a solver for systems of linear equations and condition-based fluxes. In future work, the `dune-pdelab` specific code generation will explore additional optimizations possible within the IR. Furthermore, the generation of an optimized preconditioner will be investigated.

6. Acknowledgments

This research has been funded by the Federal Ministry of Education and Research of Germany (BMBF) through the HPC2SE project. We are grateful to the Leibniz Rechenzentrum Garching for providing computational resources. Dominic Kempf has initialized the code generation project `dune-codegen` from the interdisciplinary center for scientific computing (IWR) at the Heidelberg University. He and René Hess (IWR) are the main contributors of the sum-factorization specific optimizations for the `dune-pdelab` backend. We are grateful to Sebastian Kuckuk, one of the leading developers of ExaStencils, for his support.

References

[1] M. S. Alnæs *et al.*, "Unified form language: A domain-specific language for weak formulations of partial differential equations," *CoRR*, vol. abs/1211.4047, 2012.

[2] M. S. Alnæs *et al.*, "The fenics project version 1.5," *Archive of Numerical Software*, vol. 3, no. 100, 2015.

[3] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander, "A generic grid interface for parallel and adaptive scientific computing. part i: abstract framework," *Computing*, vol. 82, pp. 103–119, Jul 2008.
[4] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, R. Kornhuber, M. Ohlberger, and O. Sander, "A generic grid interface for parallel and adaptive scientific computing. part ii: Implementation and tests in dune," *Computing*, vol. 82, pp. 121–138, 01 2008.
[5] C. Lengauer, S. Apel, M. Bolten, A. Größlinger, F. Hannig, H. Köstler, U. Rüde, J. Teich, A. Grebhahn, S. Kronawitter, S. Kuckuk, H. Rittich, and C. Schmitt, "Exastencils: Advanced stencil-code engineering," in *Euro-Par 2014: Parallel Processing Workshops*, (Cham), pp. 553–564, Springer International Publishing, 2014.
[6] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. McRae, G. Bercea, G. R. Markall, and P. H. J. Kelly, "Firedrake: automating the finite element method by composing abstractions," *CoRR*, vol. abs/1501.01809, 2015.
[7] T. Sun, L. Mitchell, K. Kulkarni, A. Klöckner, D. A. Ham, and P. H. J. Kelly, "A study of vectorization for matrix-free finite element methods," *CoRR*, vol. abs/1903.08243, 2019.
[8] D. Kempf, R. Heß, S. Müthing, and P. Bastian, "Automatic code generation for high-performance discontinuous galerkin methods on modern architectures," *arXiv preprint arXiv:1812.08075*, 2018.
[9] S. Faghih-Naini, S. Kuckuk, V. Aizinger, D. Zint, R. Grosso, and H. Köstler, "Towards whole program generation of quadrature-free discontinuous galerkin methods for the shallow water equations," *CoRR*, vol. abs/1904.08684, 2019.
[10] "Dune numerics."
[11] A. Dedner, R. Klöfkorn, M. Nolte, and M. Ohlberger, "A generic interface for parallel and adaptive discretization schemes: abstraction principles and the dune-fem module," *Computing*, vol. 90, pp. 165–196, Nov 2010.
[12] P. Bastian, F. Heimann, and S. Marnach, "Generic implementation of finite element methods in the distributed and unified numerics environment (dune)," *Kybernetika*, vol. 46, no. 2, pp. 294–315, 2010.
[13] "Advanced stencil-code engineering." `https://www.exastencils.fau.de`.
[14] C. Schmitt, S. Kuckuk, F. Hannig, J. Teich, H. Köstler, U. Rüde, and C. Lengauer, "Systems of partial differential equations in exaslang," in *Software for Exascale Computing - SPPEXA 2013-2015*, (Cham), pp. 47–67, Springer International Publishing, 2016.
[15] "UFL: Unified form language." `https://fenics.readthedocs.io/projects/ufl`.
[16] A. Klöckner, "Loo.py: transformation-based code generation for GPUs and CPUs," in *Proceedings of ARRAY '14: ACM SIGPLAN Workshop on Libraries, Languages, and Compilers for Array Programming*, (Edinburgh, Scotland.), Association for Computing Machinery, 2014.
[17] "Pymbolic." `https://documen.tician.de/pymbolic`.
[18] A. Meurer *et al.*, "Sympy: symbolic computing in python," *PeerJ Computer Science*, vol. 3, p. e103, Jan. 2017.
[19] A. Klöckner, L. C. Wilcox, and T. Warburton, "Array program transformation with loo.py by example: High-order finite elements," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2016, pp. 9–16, ACM, 2016.
[20] "C++ vector class library." `https://www.agner.org/optimize/#vectorclass`.
[21] "Jinja." `https://palletsprojects.com/p/jinja`.
[22] C.-W. Shu and S. Osher, "Efficient implementation of essentially non-oscillatory shock-capturing schemes," *Journal of Computational Physics*, vol. 77, no. 2, pp. 439 – 471, 1988.
[23] O. Ippisch and M. Blatt, "Scalability test of $\mu\phi$ and the parallel algebraic multigrid solver of dune-istl," in *Jülich Blue Gene/P Extreme Scaling Workshop, no. FZJ-JSC-IB-2011-02. Jülich Supercomputing Centre*, 2011.
[24] M. Blatt, O. Ippisch, and P. Bastian, "A massively parallel algebraic multigrid preconditioner based on aggregation for elliptic problems with heterogeneous coefficients," *arXiv preprint arXiv:1209.0960*, 2012.
[25] S. Müthing, M. Piatkowski, and P. Bastian, "High-performance implementation of matrix-free high-order discontinuous galerkin methods," *arXiv preprint arXiv:1711.10885*, 2017.
[26] S. Kuckuk and H. Köstler, "Whole program generation of massively parallel shallow water equation solvers," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 78–87, Sep. 2018.

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200063

Invasive Computing for Power Corridor Management

Jophin JOHN, Santiago NARVAEZ and Michael GERNDT
Technical University of Munich
Department of Informatics
85748 Garching, Germany
E-mail: {john, narvaez, gerndt}@in.tum.de

Abstract. This paper investigates the use of invasive computing to enforce the power budget in an HPC infrastructure. Invasive MPI along with the Invasive Resource Manager (IRM) provides an infrastructure for developing malleable/invasive applications. In IRM, a power model is used to predict the power consumption of each application. If a violation in power corridor is predicted, IRM reconfigures the node allocation among the applications to keep the whole system back into the power corridor. Since development of invasive applications is a complex task, a new programming model called Elastic Phase Oriented Programming (EPOP) is developed to simplify the invasive programming. This model is also capable of collecting and sharing power usage metrics as well as performance metrics to IRM.

Keywords. dynamic resource management, power corridor enforcement, MPI, High Performance Computing, Slurm Batch Scheduler

1. Introduction

Current contracts between energy companies and compute centers are written in accordance to the so called power corridor. Therefore, the power consumption must be bounded by certain upper and lower limits. If the compute center goes beyond those limits (i.e., if consumes less or more than what it is stipulated in the contract) some fines could be applied by the energy company. The compute center can act as well as a power stabilizer for the grid load [1]. This means that dynamic adaptions of the power corridor might be part of the electricity contract, and could be requested by the electricity company. The compute center will have economic incentives for doing so, decreasing the electricity costs. To enforce the upper limit it is possible to use well-known techniques such as power capping; nevertheless these cannot be used to enforce the lower limit (i.e., to increase the system power consumption). In this work we show how a new paradigm for parallel computing, namely invasive computing, can be used for such case.

Invasive computing is a paradigm introduced by Teich [2]. A program that follows this paradigm, called henceforth ”invasive program”, should be able to request, use and finally free processing, communication, and memory resources in the neighborhood of its computing environment.

An invasive program is by definition malleable. This in turn means that certain optimizations, which would be otherwise hindered, are now possible. A nice example that

comes to hand is MPI, which usually has a very static view of the application. All the tasks that are created at the beginning of the application run when the batch scheduler has distributed the resources. The resources are exclusively reserved for this job and the tasks continue running until the end of it. In contrast, the resources assigned to an invasive program could change at runtime.

Within the Transregio Special Research Centre Invasic (TRR89), TUM investigates invasive resource management for HPC systems. We developed an MPI extension called iMPI [3,4] which allows through three new MPI functions writing invasive MPI applications. In combination with an extension of the Slurm batch scheduler, nodes can be dynamically redistributed among running MPI applications. This enables reducing idle nodes by more flexible scheduling, increased energy efficiency by redistributing nodes according to application efficiency, and supporting novel, dynamic applications, such as a Tsunami simulation that can use the resources more efficiently.

Development of invasive applications presents certain challenges. For example, the developer must take care of defining where an adaptation is possible, handling the newly joining processes, redistribution of data, among others. This results in a complex control flow which makes the development of invasive application difficult.

This paper reports on our two contributions. First contribution is a high level programming model on top of iMPI called EPOP (Elastic Phase Oriented Programming model) that simplifies the programming of iMPI applications by providing explicit control flow between elastic and rigid program phases. Second contribution is a power corridor management infrastructure using extensions we made to iMPI, IRM and EPOP system to collect power measurements, compute a power model and use it to keep the system inside the power corridor by redistributing resources. The presented work is based on an early prototype developed in [5,6] which was consolidated and extended.

This paper is divided into 6 sections. Section 2 will give an overview about the related work. Section 3 acts as an introduction to create invasive applications using our infrastructure. Section 4 explains in detail how the power corridor mangement was implemented followed by section 5 which presents the evaluation of the infrastructure. Finally, section 6 presents the conclusions.

2. Related work

EPOP is a programming model that provides malleability to MPI applications using the invasive infrastructure provided by iMPI and IRM. Charm++ and Adaptive Message Passing Interface (AMPI) [7] also supports the malleability of jobs by checkpoint restart along with the task migration and dynamic load balancing. AMPI abstracts the MPI processes as migratable threads and the runtime system of Charm++ deals with the scheduling and migration of these threads. Standard MPI is extended to support the Charm++ runtime system. AMPI follows a message-driven execution model and there is oversubscription due to the threading. In contrast, EPOP is based on the invasive properties of the iMPI and uses the standard MPI execution model with no oversubscription. EPOP can also provide application specific profiling information like the node level power usage and mpi time to IRM.

There are several techniques employed by supercomputing centers to control the system-wide power consumption. One such notable technique is dynamically shutting

down the jobs when a power budget is reached [8]. There is an approach where scheduler decides the future job allocation based on an application's power efficiency in the past runs. Another technique is the usage of Intelligent energy-aware backfilling algorithms along with stoppage of a node to control the total power usage [9]. In some techniques, idle nodes are selectively powered down to meet the power requirements [10]. Another power management approach is to utilize the power capping mechanisms supported by the hardware as well as forcing a system to operate at specified frequencies. There are plenty of researches [11] focussing on power capping as well as dynamic frequency scaling techniques to bring down the power usage of the system. One such approach is using CPU and memory Dynamic Voltage and Frequency Scaling (DVFS) for system-wide power capping [12]. One important difference between our work and these techniques is that ours use invasive computing for dynamic power corridor management. Also, most of these systems are using a reactive approach, which means that they only act once the system is out of the power corridor. In contrast, we use a proactive approach where resource adaptations are performed based on the power usage predictions. Additionally, our system can also handle dynamic power budget requirements.

3. Programming Invasive MPI Applications

Invasive applications can be developed using an invasive infrastructure, which in this case is constituted by the Invasive Resource Manager (IRM) along with the Invasive MPI (iMPI). IRM provides dynamic resource management and iMPI provides routines to utilize this dynamism.

IRM is an extension of the Simple Linux Utility for Resource Management (Slurm) [13]. IRM decides to expand/shrink an application based on its performance. IRM informs iMPI of the decision. iMPI[4] is an extension to MPICH [14], where the following new operations have been added to bring dynamism:

MPI_Init_adapt(...) signals the resource manager that the application will be adaptive.

MPI_Probe_adapt(...) is used to check whether there are any resource changes.

MPI_Comm_adapt_begin(...) is called to begin the adaptation window.

MPI_Comm_adapt_commit() finalizes resource adaptation.

Pseudocode for creating an iMPI application is shown in Listing 1. In the beginning, `MPI_Init_adapt()` is used to signal IRM that the application is invasive. It is also used to distinguish whether a process was created as part of a resource change or was it created at the start of application (Listing 1, lines 4-7). This is essential since `MPI_Comm_adapt_begin()` should immediately be called by the newly joining process to start the adaptation(Listing 1, lines 8-11). This call will notify IRM that the newly created processes are ready and IRM then notifies the existing processes about resource redistribution.

```
1  ...
2  MPI_Init_adapt(..., mytype)
3  // Initialization block
4  if mytype = starting_process{
5   set phase_index = 0
6  }
7  else{// Newly joining processes
8   MPI_Comm_adapt_begin(...);
9     // Redistribute data
10  MPI_Comm_adapt_commit( );
11 }
12 // Begin elastic block 1
13 if(phase_index == 0){
14  while ( block_condition ){
15   MPI_Probe_adapt(...)
16   if resource_change {
17    MPI_Comm_adapt_begin(...)
18    // Redistribute data
19    MPI_Comm_adapt_commit( )
20   }
21   iteration_number++;
22 // Compute Intensive part
23 }
24  phase_index++;
25 }
26 // End elastic block 1
27 ...
28 // Begin elastic block n
29 if(phase_index == n){
30 ...
31 }
32 // End elastic block n
33 // Finalization block
34 ...
```

Listing 1: Pseudocode of a simple iMPI program

```
...                                          1
void init_block(...){                        2
// Code in initialization block              3
}                                            4
setInit(init_block);                         5
                                             6
                                             7
                                             8
void elastic_block_1(...){                   9
/* Compute intensive part of                 10
 elastic block 1*/                           11
}                                            12
bool block_condition (...){                  13
// Looping of elastic_block                  14
}                                            15
void resource_change(...){                   16
// Redistribute data                         17
}                                            18
setElastic(elastic_block_1,                  19
              block_condition,               20
                resource_change);            21
                                             22
                                             23
...                                          24
                                             25
setElastic(elastic_block_n,...);             26
                                             27
...                                          28
                                             29
void finalize_block(...){                    30
// Code in finalization block                31
}                                            32
setRigid(finalize_block,...);                33
...                                          34
```

Listing 2: Pseudocode in Listing 1 as an EPOP program

Meanwhile, the existing processes should frequently check for the resource change using `MPI_Probe_adapt()` during the computation. In case of a resource change, `MPI_Comm_adapt_begin()` is called in order to take part in the adaptation (Listing 1, lines 15-19).

Once all the processes are at the adaptation window, the entry point, required data, etc. can be distributed (Listing 1, line 9 and 18) among new processes. Entry point refers to the application region where the new processes can safely join the existing processes. In the lines 13 and 29 of Listing 1, `phase_index` is used to identify these phases/entry points. As seen in Listing 1, the application is logically divided into different elastic blocks (parts of code where resource redistribution is possible) for creating suitable entry points for the joining processes. `MPI_Comm_adapt_commit()` is then called to finalize

the adaptation. After this point, all the processes continue the computation.

One of the issues with such an invasive application is the multiple control flows. As seen from Listing 1, the pre-existing processes should identify the entry points, probe for resource changes, enter the adaptation window, redistribute the data and entry points, etc while the newly joining processes should immediately enter the adaptation window and wait for the entry points, data, etc. This complicates the invasive application development.

The Elastic Phase Oriented Programming model (EPOP) simplifies this application development process by providing the concept of "Phases" to mark different parts of an application. A simple invasive application can have three logical parts/phases: an initialization part, a compute intensive part that can be benefited from the resource adaptation and a finalization part to write the results. EPOP provides different "Phases" to represent these parts. They are:

Init phase : to represent initialization part of an application.

Elastic phase : to represent compute intensive part of the application that can benefit from resource adaptation.

Rigid phase : to represent parts of an application that does not need resource adaptation.

Branch phase : to switch between different phases.

A simple EPOP version of the application in Listing 1 is shown in Listing 2. The EPOP driver, which is in charge of control flow in EPOP applications, will call iMPI routines in the background (not shown in the listing) to make it invasive.

Elastic block in Listing 1 (lines 13-26) contains a looping construct (line 14) that determines how many times the main compute part (line 22) will be called. It also contains a resource change probing part (line 15), which checks for a resource change and does data distribution, and an entry point transfer in case of a resource change (lines 16-20). These parts can be represented as a collection of simple functions like in Listing 2 (lines 14-26) and can be marked as an elastic phase using `setElastic(...)`. EPOP will probe for resource change and will call the `resource_change` function corresponding to the elastic phase whenever there is a resource redistribution. Whenever the newly joining processes are available, EPOP will bring it into the `resource_change` function of the current elastic phase. As a result, `phase_index` used in lines 13 and 29 of Listing 1 is not needed in EPOP. The phases are executed in the same order as they are declared (In Listing 2, lines 5,19,25 and 33 will declare phases).

EPOP and iMPI only provide methods to simplify the addition/removal of processes into/from an application. In addition EPOP will also bring all the processes to a common entry point specified by the developer. During resource change, a developer is responsible for maintaining the topological properties of the application (for example; create a new topology with a new number of processes) as well as redistributing the data among existing and joining/leaving processes. This design decision was made because each application has its own data distribution, which might be based on number of processes, threads, and other things known by the developer. For iMPI/EPOP application, users can redistribute the data among all processes during the adaptation window and hence after adaptation every process has the required data to do the computation.

4. Power corridor management

4.1. Measurements

Since we were using proactive approach for power management, we needed to predict whether the system will go out of power corridor. This prediction is done using the time series analysis techniques described in Section 4.2 which in turn require previous power measurements. On Intel systems, such as the one used for the testing, energy consumption estimations are done through the Running Average Power Limit (RAPL) sensors [15], and these values can be accessed via the Model Specific Registers (MSR). The power can then be derived by dividing this value by the time between measurements. There are multiple libraries that can be used to access these registers. For this case, we chose to use LIKWID [16].

The infrastructure was deployed on a job allocation in SuperMUC [17]. There was no cluster level measurement infrastructure available to us, and thus using RAPL was the only possible way to obtain power measurements. This also meant that we had to focus on the power consumed on the nodes, omitting the cooling system, networking components, storage system, etc. Nevertheless, measuring only the power consumed by the nodes is still enough to demonstrate the effectiveness of using resource redistribution as a power corridor management technique. Additionally, if a cluster wide tool becomes available, then the input power values can be taken from it instead of RAPL.

IRM communicates to the EPOP driver the frequency and number of measurements to be taken. Next, one rank per node will create a thread in charge of taking power measurements. Once it has accumulated the required number, they are aggregated by the leading node, and then sent to the scheduler. At this point, IRM receives the measurements and stores them. Once it has enough data, the forecast module comes into play. The main purpose of this module is to predict the future maximum and minimum power consumption of the system. They represent the worst case scenarios, i.e., the cases where the system could go out of the power corridor.

4.2. Forecasting

Using time series analysis one can try to find an underlying structure of some data, such as power consumption values. Two things are required: 1. a valid time series to work with. 2. A specific method to analyse the data. For the latter, we have chosen to use three techniques, the AutoRegressive Integrated Moving Average (ARIMA), Seasonal ARIMA with exogenous regressors (SARIMAX) and the Holt-Winters method.

ARIMA is a ”classical model”, in the sense that it has been studied extensively. It is composed of an Integrated component, which is in charge of making data stationary, and an ARMA component, which models this stationary data. The latter can again be subdivided into an AutoRegressive component (AR), which captures the relation between the current value of the time series and some of its past values, and a Moving Average component (MA) that represents the influence of an often unexplained random shock. Using both of them, plus the Integrated component, one can derive the ARIMA model. SARIMAX is an extension of ARIMA that supports time series with a seasonal component. The third method used in this work, called Holt-Winters or Third Exponential Smoothing, assigns exponentially decreasing weights to past observations. It is capable

of model data with both trend and seasonality, distinguishing for the latter the case of additive and multiplicative seasonality.

4.3. Decision Making

Every time the controller predicts that the system might go outside the power corridor, it is necessary to redistribute the nodes to try to prevent it. This problem is expressed more formally in Equation 1, where we have K applications running on a system with N nodes ($K \leq N$). We assume that every idle node consumes p_{idle} power. The idea is to minimize the power consumed $f(k_{idle})$ by the idle nodes, such that both the upper U and lower L boundaries are fulfilled.

$$
\begin{aligned}
&\textit{MINIMIZE}\\
&\quad f(k_{idle}) = k_{idle} * p_{idle}\\
&\textit{SUBJECT TO}\\
&\quad l \leq \sum_{i=0}^{K-1} k_i * p_{min}^{(i)} + k_{idle} * p_{idle}\\
&\quad u \geq \sum_{i=0}^{K-1} k_i * p_{max}^{(i)} + k_{idle} * p_{idle}\\
&\quad 1 \leq k_i \leq N,\ k_i \in \mathbb{N} \setminus \{0\},\ i = 0, \cdots, K-1\\
&\quad 0 \leq k_{idle} < N,\ k_{idle} \in \mathbb{N}
\end{aligned}
\tag{1}
$$

The solution to the system is found via Pulp, a Python Integer Programming Solver module [18]. In turn, Pulp acts as an interface to several solvers. In this case we have used Coin-or Branch and Cut (CBC). We tested Pulp with systems with $K = 2, 4, 8$ and 16, and the solving time was always under 0.5 seconds. Considering that a decision has to be made from one schedule pass to the next, Pulp is fast enough. Once a valid node distribution is found, an adaptation occurs (see Section 3).

4.4. Guarantees

In an infrastructure with a size of N Nodes, where K applications are running:

Our system will enforce upper power corridor U, if and only if the power consumption of the system when each application runs in only one node is less than the power corridor upper bound U, as expressed in Equation 2.

$$
U \geq \sum_{i=1}^{K} p_{max}^{(i)} + (N-K) * p_{idle} \tag{2}
$$

Similarly, our system will enforce lower power corridor L, if and only if, the power consumption of the system is greater than the lower power corridor boundary when the

most power consuming application A is running on $N-(K-1)$ nodes. This is shown in Equation 3.

$$L \leq \sum_{i=1}^{K-1} p_{max}^{(i)} + (N-(K-1)) * k_A * p_A \tag{3}$$

5. Evaluation

The power corridor management infrastructure was run as a standard Load Leveler job in SuperMUC Phase 2 [17]. The infrastructure was run on 32 nodes (896 processes) for the forecasting and upper power corridor enforcement tests while the remaining tests were performed on 16 nodes (448 processes). Three EPOP applications (2D Jacobi heat simulation, LU decomposition and Pi calculation) were used to evaluate the infrastructure. This evaluation is a proof-of-concept that the dynamic resource management can be used for enforcing the power corridor on a system with varying power constraints.

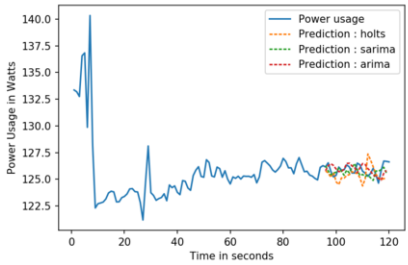
5.1. Forecasting

Power consumption predictions from three different models are shown in Figure 1a. These models were trained on-the-fly and the one with the highest accuracy was chosen by IRM for forecasting. As observed in Figure 1a, the SARIMA model produced more accurate predictions among the different models and was used by IRM to make the scheduling decisions. Accuracy of the model was determined using the Mean Absolute Percentage Error (MAPE).

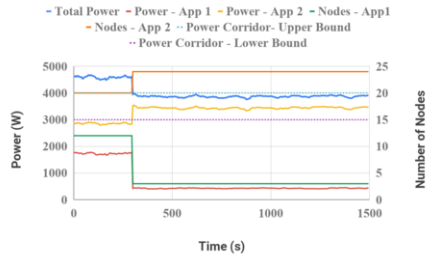
5.2. Upper and Lower Power Corridor Enforcement

Figure 1b and 1c shows the effect of dynamic resource adaptation on system wide power usage. Initially, two EPOP applications were started with 12 nodes for Application 1 and 20 for Application 2. The power corridor was set between 3000 and 4000 Watts. It can be seen from Figure 1b that the upper power bound has already been violated from the start of the applications. Therefore, during the first scheduler pass, the system redistributes the number of nodes. As a result, application 1 was reduced to 3 nodes and application 2 was expanded to 24 nodes. This lead to the reduction of power usage, bringing back the system to the power corridor after 300 seconds. During the next scheduler passes, the forecast module predicts no violation of the power corridor and as a result, the resource configuration remained same.

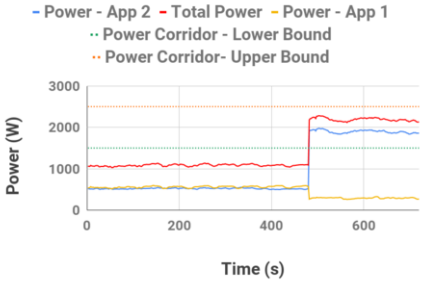
Similarly, the lower power corridor enforcement is shown in Figure 1c. The power corridor was set between 1500 and 2500 watts. Two applications (Jacobi heat simulation and LU decomposition) were started with 4 nodes each. It can be observed that the lower power corridor of the system was violated from the beginning. To enforce the power corridor, IRM shrunk Application 1 to 2 nodes and expanded Application 2 to 14 nodes during the first scheduler pass. As a result, the system was back in the power corridor.



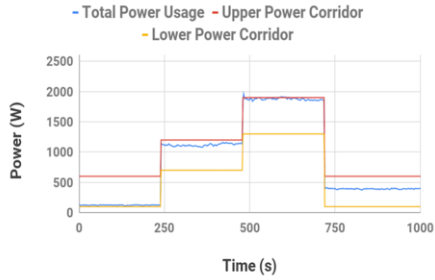
(a) Forecasting the power usage



(b) Upper power corridor enforcement



(c) Lower power corridor enforcement



(d) Dynamic power corridor enforcement

Figure 1. Power Corridor Management

5.3. Dynamic Power Corridor Enforcement

Dynamic power corridor enforcement is shown in Figure 1d. Initially, the power corridor was set between 100 and 600 watts. An invasive Pi calculation application was started on a single node. It can be observed that the system is in the power corridor. Then the power corridor was shifted to 700 and 1200 watts. IRM expanded the application to 9 nodes and brought back the system into the power corridor. The power corridor was then increased to 1300 and 1900 watts. We can observe from Figure 1d that IRM again redistributed the resources to bring the system back in the power corridor. This test simulates dynamically changing power constraints and how the system is responding to it.

6. Conclusion and Outlook

Power corridor management is crucial for supercomputing centers. As more and more renewable energy sources are used for power generation, HPC centers must be flexible in adapting to the energy requirements, since the supply of renewable energy will be varying due to external factors. Resource dynamism and flexible scheduling can be used to accommodate such dynamic scenarios. We have shown in this paper that invasive computing can be used as a mechanism to enforce the power corridor. We were able to regulate power consumption without taking drastic measures, such as killing power-hungry applications. One of the shortcomings of this invasive approach is that frequent

resource redistributions can be expensive. Our ongoing work, a hybrid system which can use DVFS along with invasive computing to manage power requirements, addresses this shortcoming.

Acknowledgements

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing (SFB/TR 89).

References

[1] H. Chen, M. C. Caramanis, and A. K. Coskun, "The data center as a grid load stabilizer," in *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2014, pp. 105–112.

[2] J. Teich, "Invasive algorithms and architectures invasive algorithmen und architekturen," *it-Information Technology*, vol. 50, no. 5, pp. 300–310, 2008.

[3] M. G. I. Compres Urena, "Towards elastic resource management," in *Proceedings of the 11th Parallel Tools Workshop, September 11-12, 2017*, 2017, to appear.

[4] I. A. Comprés Ureña, "Resource-elasticity support for distributed memory hpc applications," Dissertation, TU Munich, Munich, 2017. [Online]. Available: http://mediatum.ub.tum.de?id=1362721

[5] J. John, "The elastic phase oriented programming model for elastic hpc applications," Master Thesis, TU Munich, Munich, 2018. [Online]. Available: https://mediatum.ub.tum.de?id=1475008

[6] S. Narvaez, "Power model for resource-elastic applications," Master Thesis, TU Munich, Munich, 2018. [Online]. Available: http://mediatum.ub.tum.de?id=1475095

[7] L. V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on c++," in *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, 1993.

[8] M. Maiterth, G. Koenig, K. Pedretti, S. Jana, N. Bates, A. Borghesi, D. Montoya, A. Bartolini, and M. Puzovic, "Energy and power aware job scheduling and resource management: Global survey — initial analysis," 05 2018, pp. 685–693.

[9] P. Dutot, Y. Georgiou, D. Glesser, L. Lefevre, M. Poquet, and I. Rais, "Towards energy budget control in hpc," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, May 2017, pp. 381–390.

[10] J. Sun, C. Huang, and J. Dong, "Research on power-aware scheduling for high-performance computing system," in *2011 IEEE/ACM International Conference on Green Computing and Communications*, Aug 2011, pp. 75–78.

[11] M. Yadav, "A brief survey of current power limiting strategies," 03 2018.

[12] Y. Liu, G. Cox, Q. Deng, S. C. Draper, and R. Bianchini, "Fastcap: An efficient and fair algorithm for power capping in many-core systems," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2016, pp. 57–68.

[13] "Simple linux utility for resource management," http://slurm.schedmd.com/, accessed: 2019-07-09.

[14] MPICH, "Mpich," https://www.mpich.org/.

[15] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, and Z. Ou, "Rapl in action: Experiences in using rapl for power measurements," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, vol. 3, no. 2, p. 9, 2018.

[16] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *2010 39th International Conference on Parallel Processing Workshops*. IEEE, 2010, pp. 207–216.

[17] Leibniz-Rechenzentrum, "Supermuc petascale system," https://www.lrz.de/services/compute/supermuc/.

[18] S. Mitchell, M. OSullivan, and I. Dunning, "Pulp: a linear programming toolkit for python," *The University of Auckland, Auckland, New Zealand*, 2011.

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200064

Enforcing Reference Capability in FastFlow with Rust

Luca Rinaldi [a,1], Massimo Torquati [a] and Marco Danelutto [a]
[a] *Computer Science Department, University of Pisa, Italy*

Abstract. In this work, we investigate the performance impact of using the `Rust` programming language instead of the `C++` one to implement two basic parallel patterns as provided by the FastFlow parallel library. The rationale of using `Rust` is that it is a modern system-level language capable to statically guarantee that if a data reference is sent over a communication channel, the ownership of the reference is transferred from the producer to the consumer. Such reference-passing semantics is at the base of the FastFlow programming model. However, the FastFlow library does not enforce nor checks its correct usage leaving this burden to the programmer. The results obtained comparing the FastFlow/`C++`, and the `Rust` implementations of the same implementation schema of the Task-Farm and Pipeline patterns show that `Rust` is a valid alternative to `C++` for the FastFlow library with indubitable benefits in terms of programmability.

Keywords. Multi-cores, parallel programming, reference capability, Rust, C++

1. Introduction

Multi-core and many-core processors are today largely used both in professional and consumer settings. Multi-cores are tightly-coupled Multiple-Instruction Multiple-Data (MIMD) architectures. They are shared-memory multiprocessors systems integrated into a single chip, often referred to as Chip Multi-Processors (CMP). Many-core processors are CMP systems that are designed to employ a high degree of parallelism (currently up to a few hundred cores), by using a large number of simpler cores than those used in general-purpose multi-cores. The broad diffusion of CMP systems has had and still is having, an important effect on how software is developed.

In these systems, the physically shared memory is the primary means of cooperation among threads and processes running on different cores. Communications occur implicitly through loads and stores coordinated by synchronization protocols typically implemented using *locks*. Locks seriously limit concurrency, they are costly operations requiring the intervention from the OS to suspend the thread and restore it later. Moreover, locks might introduce deadlock situations into the application, and, therefore, increase the debugging and maintainability software phases.

A different approach is to use message passing semantics to coordinate the concurrent entities. A message induces an implicit synchronization between the sender and the receiver. This model may be used merely for synchronization purposes while data may

[1]Corresponding Author: Luca Rinaldi E-mail: luca.rinaldi@di.unipi.it

be shared exploiting the cache-coherent hardware capabilities of modern CMPs. Indeed, sharing mutable data in a producer-consumer fashion is generally more efficient than explicit copying, especially for large data structures. However data sharing is dangerous. Changes to a data reference might propagate producing unexpected data-races, i.e. two concurrent operations (where at least one is a write operation) to the same memory location without any synchronization.

The message passing model is used as a synchronization mechanism in some C++-based parallel library, such for example FastFlow [3] and GrPPI [5]. The `C++` programming language is commonly used for its large set of features and its performance. However, it does not provide strong guarantees for memory safety. Modern programming languages such as `Rust` [10] and Pony [4] have a *reference capability* system that statically checks access permissions to memory locations. In `Rust`, this feature is expressed with the concept of *ownership* [7]. The idea behind ownership is that, although multiple aliases to a resource may exist simultaneously, to perform specific actions on the resource (e.g., reading or writing a memory location) should require some unique capability *owned* by exactly one alias at any point in time during the execution of the program. This concept permits to enforce at compile time that every time a variable is sent over a communication channel, its ownership capability is also sent, so the sender cannot access the data anymore [11]. Such reference capability semantics is employed in the C++-based FastFlow parallel programming library. FastFlow is a library offering both high-level parallel patterns as well as composable parallel building blocks suitable for building run-time systems for new DSLs or for building new high-level parallel patterns. However, the reference capability semantics is not enforced by the FastFlow library, leaving the burden of respecting the semantics directly to the run-time system programmer.

In this work, we analyze the implications on the programming model and on the overall application performance of using the `Rust` language to implement the FastFlow parallel semantics. Specifically, we considered two simple synthetic benchmarks implemented by using two FastFlow parallel patterns: the Task-Farm pattern and the Pipeline pattern. These two patterns are particularly relevant because they are used in FastFlow as basic building blocks of other more complex parallel patterns (e.g., ParallelFor Divide&Conquer, and Macro Data-Flow). We aim to demonstrate that a system-level language such as `Rust` which provides strong statically checked features to the programmer can be a valid alternative to `C++` to write the parallel patterns offered by the FastFlow library. From the programming model standpoint, the `Rust` implementation has the additional advantage of statically enforcing the FastFlow producer-consumer semantics at the language level. To meet the objective, the FastFlow communication channel implemented as a Single-Producer Single-Consumer (SPSC) lock-free unbounded queue [2] has been adequately and safely wrapped to build a `Rust` library to be used for the implementation of inter-thread communication channels in `Rust`. The results obtained by running the synthetic benchmarks on a 24-core Intel multi-core platform, demonstrate that the `Rust` implementation of the benchmarks considered exhibits the same level of performance of the FastFlow `C++` implementation.

The remaining of this paper is organized as follows. The next section presents the background, specifically the FastFlow and the `Rust` language features. Then, Section 3 provides the motivations of this work. The experimental tests are described in Section 4. Finally, Section 5 briefly reviews similar works and summarizes our contributions.

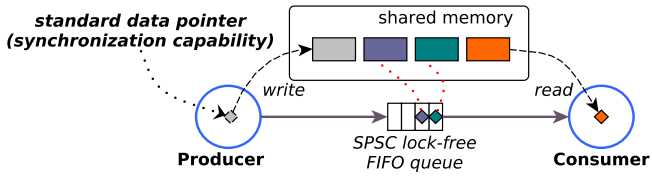


Figure 1. FastFlow library producer-consumer semantics: sending references to shared data over a SPSC lock-free FIFO channel.

2. Background

2.1. FastFlow

FastFlow is a C++ parallel programming library targeting multi/many-cores and offering a multi-level API to the parallel programmer [3,]. At the top level of the FastFlow software stack, there are some ready-to-use high-level parallel patterns such as *Pipeline Task-Farm*, *ParallelFor*, *Divide&Conquer*, *StencilReduce*, *Macro Data-Flow* and so on. At a lower level of abstraction, the library provides customizable sequential and parallel building blocks addressing the needs of the run-time system programmer. The idea is that new high-level patterns or new high-level libraries can be built by a proper assembly of the building blocks [1].

The library was conceived to support highly efficient stream parallel computations on heterogeneous multi-cores. The library is released open-source under the LGPLv3 licence [2].

The FastFlow library is realized as a modern C++ header-only template library that allows the programmer to simplify the development of parallel applications modeled as a structured directed data-flow graph (called *concurrency graph*) of processing *nodes*. A FastFlow *node* represents a basic unit of computation. Each *node* can have zero or more input channels and zero or more output channels. The graph of concurrent nodes is constructed by the assembly of sequential and parallel building blocks as well as higher-level parallel patterns. A generic node of the concurrency graph (being it either standalone or part of a more complex parallel pattern) performs a loop that: i) gets a data item (through a memory reference to a data structure) from one of its input channels; ii) executes a functional code (i.e. business logic) working on the data item and possibly on a state maintained by the node itself; iii) puts a memory reference to the result item into one or multiple output channels selected according to a predefined or user-defined policy. Input and output channels are implemented with a Single-Producer Single-Consumer (SPSC) FIFO queue. Operations on FastFlow queues (that can have either bounded or unbounded capacity) are based on non-blocking lock-free synchronizations enabling fast data processing in high-frequency streaming applications [2].

From the programming model standpoint, the FastFlow library follows the well-known Data-Flow parallel model where channels do not carry plain data but references to heap-allocated data. The semantics of sending data references over a communication channel is that of transferring the ownership of the data pointed by the reference from the sender node (producer) to the receiver node (consumer) (see also the schema in Figure 1). The data reference is *de facto* a *capability*, i.e. a logical token that grants access to a given

[2] FastFlow home: http://calvados.di.unipi.it/fastflow

data structure or to a portion of a data structure. On the basis of this *reference-passing* semantics, the receiver is expected to have exclusive access to the data value received from one of the input channels, while the producer is expected to not use the reference anymore. This semantics is not directly enforced by the library itself with any static or run-time checks.

2.2. `Rust`

`Rust` [10], is a modern system-level programming language that focuses on memory safety and performance.

The principal novelty of `Rust` is in the management of memory. Languages like C/C++ provide the user with total control on memory allocation and deallocation. Programmers can create, destroy and manipulate the memory space without any limitation. This is a very attractive feature for expert programmers, but it can also lead to very subtle bugs and vulnerabilities (e.g., buffer overflow). Other popular languages such as Java, rely on a Garbage Collector (GC) to safely manage memory without the explicit intervention of the user. The increased security comes along with some performance degradation due to the GC service running in the background trying to reclaim unused memory. Instead, the `Rust` language deals with memory management through the *ownership* concept [8]. The compiler statically checks a set of rules to control the memory allocation/deallocation and memory accesses. Therefore, the compiler guarantees a certain level of memory safety at the price of a more complex and longer compilation process but without any additional overheads at running time.

Concerning the *owenship* feature, once a variable is bound with a value, it gains exclusive ownership of it. Therefore, only the owner can access that memory location until it transfers the exclusive ownership to another variable. The *ownership rule* states three simple concepts [8]: 1) each value has a variable that is called *owner*; 2) there can be only one owner at a time; 3) when the owner goes out of scope, the value will be dropped.

Values stored in the heap maintain the same rules and when the owner variable goes out of scope the memory is automatically released. In this way the user does not have to directly deal with allocation and deallocation instructions avoiding the risk of double frees or memory leaks.

To improve the flexibility of the language, `Rust` also implements the *borrowing* concept through memory references. It is possible to create an immutable reference by using `&` and a mutable reference by using `&mut`. Both of them borrow the value from the original owner. The compiler imposes the following rules: 1) at any given point in time, only one mutable reference or any number of immutable references may exist; 2) the borrowed value cannot be accessed by the original owner; 3) when the reference goes out of scope the ownership goes back to the original owner.

`Rust` has also the *lifetimes* concept to avoid dangling references. A lifetime is the scope in which a reference is valid and the compiler enforces that it must be smaller of the scope of the value referenced. Lifetimes are usually inferred by the compiler. However, there are cases in witch the user has to annotate functions with life time parameters.

Finally, `Rust` provides native threads support, synchronization mechanisms such as mutex and atomic variables as well as Multi-Producer Single-Consumer (MPSC) communication channels for connecting threads. Indeed, the compiler guarantees that either

multiple threads have only read access to a memory location or only one thread has read and write access to it. To manage the mutability of variables and to guarantee memory safety in multi-thread applications, `Rust` defines the `Send` and `Sync` traits. The `Send` marker trait indicates that ownership of the type implementing `Send` can be transferred between threads. Almost every type in `Rust` implements `Send` and types composed entirely of types that are `Send`-able are themselves `Send`-able. The `Sync` trait indicates that it is safe for the type implementing `Sync` to be referenced from multiple threads.

3. Motivations

Many mainstream parallel programming libraries are written in C/`C++` primarily for performance reasons. Often, the burden of maintaining non-interference among threads implementing the application is in charge of the parallel programmer which has to correctly use either locks or (hopefully) suitable high-level parallel abstractions (e.g., parallel patterns). From the one hand, the usage of low-level synchronization mechanisms allows the programmer to have great flexibility and to tweak the code applying specific optimizations, but on the other hand, it exposes to potential unexpected behaviors and subtle data-races.

The so-called "modern C++" (i.e. C++11 and above) introduced *move semantics* and *smart pointers* features which greatly help the programmers to avoid errors related to pointer arithmetic without affecting (in the majority of cases) the overall performance. However, the responsibility to correctly use such new features is still in charge of the programmer that might not be a parallel programming expert. Moreover, in some situations, `C++` move semantics may produce additional data copies, for example in the one-to-many communication pattern implementing a data scattering operation.

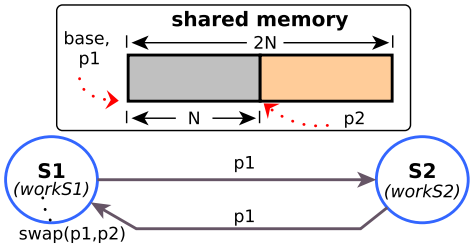


Figure 2. Logical schema of the FastFlow two-stage pipeline described in Listing 1.

As an example, a valid FastFlow program is the one sketched in Listing 1. It implements a two-stage pipeline where the two stages work disjointly on two distinct portions of the same vector in a producer-consumer fashion. The producer (`S1`) allocates a standard vector of size `2N` and then uses two raw pointers to point to two distinct parts of the vector that are swapped at every producer-consumer iteration. Each stage works on a portion of length `N` of the initial vector. The logical schema of this simple producer-consumer use-case is sketched in Figure 2.

In this simple example, there is no guarantee that within the `workS1` or `workS2` functions some wrong accesses to a portion of the vector may produce data-races due to buffer overruns. This kind of implementation would not be possible in the `Rust` language

```
struct Stage1: ff_node_t<float> {
  Stage1():base(2*N) {}
  int svc_init() {
    initialize(base);
    p1=base.data(); p2=p1+N;
    std::swap(p1,p2);
    return 0;
  }
  float* svc(float* in) {
    if(haveToStop(p1,p2))
       return EOS;
    std::swap(p1,p2);
    ff_send_out(p1);
    workS1(p2, N, 10.0);
    return GO_ON;
  }
  std::vector<float> base;
  float *p1,*p2;
} S1;
```

```
struct Stage2: ff_node_t<float> {
  float* svc(float* in) {
        workS2(in, N, 20.0);
        return in;
  }
} S2;

int main() {}
 // creates the pipeline
 ff_Pipe pipe(S1,S2);
 // creates the feedback channel
 pipe.wrap_around();
 // synchronous execution
 if(pipe.run_and_wait_end()<0) {
    error("running pipe\n");
    return -1;
 }
 return 0;
}
```

Listing 1: A simple producer consumer program in FastFlow

because the ownership rule is violated by the concurrent ownership of the vector by the two stages. In Rust, the programmer that wants to implement a similar program is forced to declare two separated vectors and to alternatively move the vectors' ownership through the communication channel connecting the two nodes. Moreover, accesses outside the boundaries of the two vectors is checked at run-time. It is worth noting that, a similar implementation is also possible in C++ but, while in Rust there is basically no other way to implement that program, in C++ there is nothing that may prevent a potentially dangerous implementation using raw pointers.

Concerning the FastFlow parallel library, the point is that the potentially wrong usage of the reference-passing capability approach, which is at the base of the FastFlow programming model, is not checked by the library and the potential faulty behavior is not signaled to the user. The programmer must properly use the provided mechanisms according to the programming model. To alleviate the burden of the programmer, we decided to re-implement the FastFlow library using a language that can enforce reference capability at compile time. In this work, we want to measure the performance impact of using the Rust language with respect to a less-safe C++ implementation. Rust allows static checking at a higher level of abstraction than the one used to check the C++ move semantics. Our uphold that a proper combination of a system-level language with strong static checking features and a structured parallel programming methodology such the one offered by the FastFlow parallel library can significantly help the programmer to produce efficient and portable code with reduced programming effort and shorter time-to-solution.

4. Evaluation

In this section, we show the performance evaluation of using the Rust programming language instead of the C++ one in the implementation of two benchmarks based on two well-known FastFlow parallel patterns, namely the Task-Farm and the Pipeline. We selected these two patterns because they are used within the FastFlow library as basic building blocks for the implementation of other more complex parallel patterns.

4.1. Low-level mechanisms implementation

To have a fair performance comparison between the implementations of the two benchmarks, we need an implementation of the FastFlow communication channel in Rust. Initially, we considered to use the Multi-Producer Single-Consumer (MPSC) unbounded queue provided by the Rust standard library, but we found out that it does not deliver the expected performance, particularly for fine-grained computation. Therefore, we decided to port the C++-based FastFlow lock-free Single-Producer Single-Consumer (SPSC) unbounded queue [2] in Rust. However, instead of writing it from scratch mimicking the same FastFlow implementation, we decided to create a memory-safe Rust interface to the original C++-based FastFlow queue. The name of the Rust interface for the queue is `ff_buffer`[3].

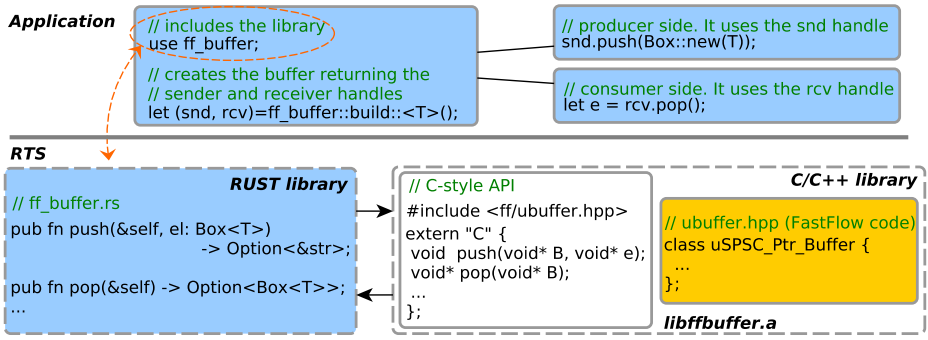


Figure 3. Integration of the FastFlow's unbounded SPSC lock-free queue in Rust.

Figure 3 shows the logical schema of the `ff_buffer` library that we used to integrate the FastFlow queue in Rust. The implementation is composed of two distinct parts: the Rust API providing a memory-safe interface of the queue, and the static C library that exposes the "unsafe" C interface of the C++ implementation. The `ff_buffer` library can be directly compiled as a standard Rust library. Moreover, it is possible to use the *Cross Language Linking Time Optimization*[4] feature of the LLVM compiler infrastructure to reduce the overhead of jumping back and forth between Rust and C++.

Another FastFlow feature we decided to use in the experiments is the ability to automatically pin all the spawned threads to distinct machine cores to improve the application performance when the number of threads is less than or equal to the available

[3] Git repository link https://github.com/lucarin91/ff_buffer
[4] http://blog.llvm.org/2019/09/closing-gap-cross-language-lto-between.html

cores. For this purpose we used the `Rust` third-party library `core_affinity`[5] to set the thread-to-core affinity for all `Rust` threads according to a simple round-robin assignment strategy.

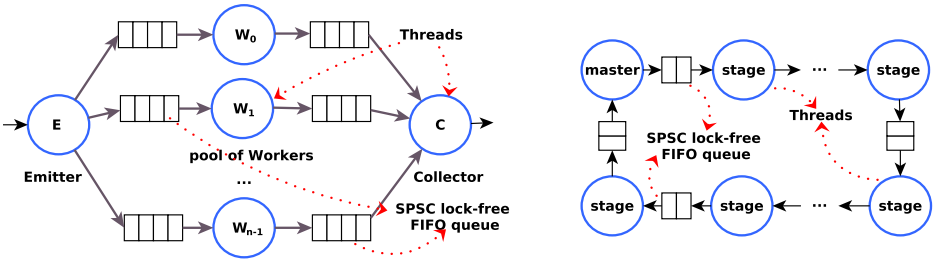


Figure 4. Implementation schema of the Task-Farm pattern (left-hand side) and of the Pipeline with feedback channel pattern (right-hand side).

In Figure 4 are sketched the implementation schemes of the two FastFlow parallel patterns that we used as benchmarks for comparing the performance of the `C++` and `Rust` versions. The one on the left-hand side is the implementation of the Task-Farm pattern where the pool of Workers is composed of sequential nodes. Each node is implemented as a thread. In the tests we executed, each Worker performs a configurable number of floating-point operations on each input data element. The Emitter node is in charge to assign data elements to the Workers according to a pre-defined or user-defined scheduling policy. We considered a simple round-robin assignment. The data elements produced by the Workers are all collected by the Collector node. This test aims to study the scalability of the Task-Farm pattern by varying the number of Worker threads.

On the right-hand side of Figure 4 is shown the Pipeline with feedback pattern as implemented in FastFlow. In the tests we executed, we considered a Master stage (the first one) and a configurable set of other stages. The Master stage is in charge of generating a fixed-length stream of data elements in batches. The other stages of the pipeline chain only forward the input element received to the next stage. The last stage of the pipeline is connected to the Master stage, forming a circular pipeline. This test aims to study the maximum throughput sustained by the Pipeline pattern by varying the number of stages.

4.2. Results

All tests reported in this section were conducted on an Intel Xeon Server equipped with two Intel E5-2695 Ivy Bridge CPUs running at 2.40GHz and featuring 24 cores (12 per socket). Each hyper-threaded core has 32KB private L1, 256KB private L2 and 30MB of L3 shared cache. The machine has 64GB of DDR3 RAM, running Linux 3.14.49 x86 64 with the CPUfreq performance governor enabled and turbo boost disabled. We used the GNU `gcc` compiler version 7.2.0 with the O3 optimization flag enabled and the `rustc` compiler version 1.38.0 with opt-level=3.

The tests were executed ten times, and the values reported in the plots is the average value of all runs. The standard deviation is small (less than 1%) and thus omitted for readability reasons.

[5] https://crates.io/crates/core_affinity

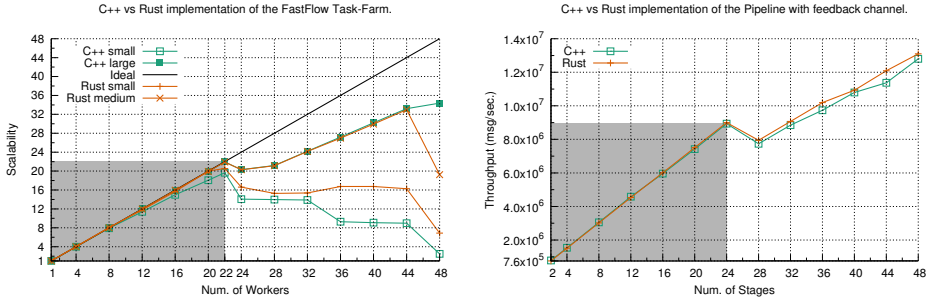


Figure 5. Left:) Scalability of the Task-Farm pattern implementation with two different computation granularities. Right:) Throughput of the Pipeline pattern with feedback channel varying the number of stages.

For the Task-Farm pattern we considered a stream of 50,000 elements and two different per-element computation granularities: *small* (about ∼5 µs), and *large* (about ∼5 ms). On the left-hand side of Figure 5 is shown the scalability of the Task-Farm pattern written in C++ (i.e. FASTFLOW v.3.0.0) and in `Rust`, respectively. The results show that the two versions have similar performance figures both for the *small* and *large* test cases. Both versions exhibit good scalability figures when the number of total threads used (that is equal to the number of Workers plus two) is less than or equal to the number of physical cores of the machine (this is the grey area of the plot). The `Rust` implementation of the benchmark uses a more simple (and aggressive) dequeueing strategy than the one offered by the FASTFLOW library. Moreover, the `Rust` version leverages on the `jemalloc` memory allocator. These two optimizations allow to slightly improve the performance of the `Rust` version in the *small* test case when the number of Workers is high. Conversely, for the *large* test case, the more aggressive polling approach used in the `Rust` implementation produce more overhead when the number of threads is greater than the available logical cores (i.e. the case of 48 Workers).

For the Pipeline test case, we consider a total number of 1M elements divided in an initial batch of 1K elements and 4K small batches each one containing 256 elements. Figure 5 shows the number of messages exchanged per second by varying the number of stages of the pipeline chain. The performance of the two versions is almost the same, and the throughput increases almost linearly with the number of stages with a small drop corresponding to 24 pipeline stages because from that point more threads than physical core are used.

The results obtained demonstrate that there is no significant performance difference between the `C++` and `Rust` versions for the two patterns considered.

5. Related Work and Summary

The `Rust` programming language is attracting increasing interest in the parallel community because of its comparable performance with `C/C++` and its memory safety.

Libraries such as *rsmpi*[6] and *Raycon*[7] are examples of well-known parallel programming libraries that moved from `C/C++` to `Rust`. Rsmpi is a MPI binding for `Rust`, and

[6] https://github.com/bsteinb/rsmpi
[7] https://github.com/rayon-rs/rayon

it permits to use the MPI library from within Rust programs. Raycon is a data parallel library similar to the OpenMP standard. It supports parallel computations such as *map*, *flap-map*, *filter*, *sorting* and *reduce* over Rust collections.

Other research works such as [6] and [9] try to improve and extend the Rust ownership system to better support parallel computations. The former proposes a statically checked communication protocol between threads. The latter proposes an extension of the ownership system where it is possible to specify that the same thread can own multiple times the same variable. Such extension simplifies code writing, especially in an event-based system, while maintaining the same security guarantees.

In this work, we evaluated the impact of statically enforcing the reference-passing semantics used in the FastFlow parallel programming library by using the Rust language features. We evaluated the impact on the performance of a Rust implementation of the Task-Farm and Pipeline pattern as provided by the FastFlow library. The results obtained show that the Rust language can be a valid alternative to the C++ one for implementing the FastFlow parallel patterns with several benefits in terms of programmability. However, more work is needed to build the entire software stack of the FastFlow library.

As future work, we intend to analyze and discuss the implementation of other parallel patterns and in particular of the Map one which poses non-trivial implementation problems if implemented in Rust.

References

[1] M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, and M. Torquati. Design patterns percolating to parallel programming framework implementation. *Int. J. Parallel Program.*, 42(6):1012–1031, 2014.

[2] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati. An efficient unbounded lock-free queue for multi-core systems. In C. Kaklamanis, T. Papatheodorou, and P. G. Spirakis, editors, *Euro-Par 2012 Parallel Processing*, pages 662–673, Berlin, Heidelberg, 2012. Springer.

[3] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. FastFlow: High-Level and Efficient Streaming on multi-core. In S. Pllana and F. Xhafa, editors, *Programming Multi-core and Many-core Computing Systems*, Parallel and Distributed Computing, chapter 13. John Wiley & Sons, Inc, Jan. 2017.

[4] S. Clebsch, S. Drossopoulou, S. Blessing, and A. McNeil. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! 2015, pages 1–12, New York, NY, USA, 2015. ACM.

[5] D. del Rio Astorga, M. F. Dolz, J. Fernandez, and J. D. Garca. A generic parallel pattern interface for stream and data processing. *Concurrency and Computation: Practice and Experience*, pages e4175–n/a, 2017. e4175 cpe.4175.

[6] T. B. L. Jespersen, P. Munksgaard, and K. F. Larsen. Session types for rust. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*, WGP 2015, pages 13–22. ACM, 2015.

[7] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL):66:1–66:34, Dec. 2017.

[8] S. Klabnik and C. Nichols. *Chapter 4: Understanding Ownership*. no starch Press, 2018.

[9] A. Levy, M. P. Andersen, B. Campbell, D. Culler, P. Dutta, B. Ghena, P. Levis, and P. Pannuto. Ownership is theft: Experiences building an embedded os in rust. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*, PLOS '15, pages 21–26. ACM, 2015.

[10] N. D. Matsakis and F. S. Klock, II. The rust language. *Ada Lett.*, 34(3):103–104, Oct. 2014.

[11] Z. Yu, L. Song, and Y. Zhang. Fearless concurrency? understanding concurrent programming safety in real-world rust software. *arXiv preprint arXiv:1902.01906*, 2019.

This page intentionally left blank

Performance

This page intentionally left blank

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200066

AITuning: Machine Learning-Based Tuning Tool for Run-Time Communication Libraries

Alessandro Fanfarillo [a,1], Davide Del Vento [a]
[a] *National Center for Atmospheric Research, Boulder, Colorado, USA*

Abstract. In this work, we address the problem of tuning communication libraries by using a deep reinforcement learning approach. Reinforcement learning is a machine learning technique incredibly effective in solving game-like situations. In fact, tuning a set of parameters in a communication library in order to get better performance in a parallel application can be expressed as a game: *Find the right combination/path that provides the best reward*. Even though AITuning has been designed to be utilized with different run-time libraries, we focused this work on applying it to the OpenCoarrays run-time communication library, built on top of MPI-3. This work not only shows the potential of using a reinforcement learning algorithm for tuning communication libraries, but also demonstrates how the MPI Tool Information Interface, introduced by the MPI-3 standard, can be used effectively by run-time libraries to improve the performance without human intervention.

Keywords. MPI Machine Learning Reinforcement Learning Coarray Fortran

1. Motivaton

Tuning a general-purpose communication library is tightly related to the communication pattern utilized by the application, the network interconnect, the computer architecture, and the problem size. Profilers and other performance analysis tools have improved substantially in recent years and they are now able to provide the user with very accurate and descriptive interpretations of the various bottlenecks in a parallel application. However, most users in the scientific computing community do not have the time or expertise to study and tune the parameters of the communication libraries used by their codes. In fact, optimizing the parameters of communication libraries requires technical knowledge and time to try different configurations. For example, most Message Passing Interface (MPI) implementations offer hundreds of parameters that can provide significant speedup if they are set to their optimal value (which varies depending on the application), compared to the default configuration.

Furthermore, general-purpose communication libraries, like MPI, express several parallel programming models (e.g. one-sided, message-passing, task-based, etc...), and

[1] Corresponding Author: elfanfa@ucar.edu

the optimal setting of a parameter used for a programming model might impact the performance when used on a different application, using a different programming model.

On the other hand, run-time communication libraries usually express fewer parallel programming models than general-purpose parallel programming libraries, and thus the communication pattern exposed by a run-time library can be interpreted and modeled much more easily.

In this work, we explore the use of machine learning techniques to optimize a particular run-time communication library, namely the OpenCoarrays run-time (used by the GNU Fortran compiler to implement the coarray support) and particularly its implementation on top of MPI-3.

Another important goal of this work is to demonstrate how the MPI Tool Information Interface, introduced by the MPI-3 standard, can be used effectively for automatic performance improvements when used by run-time libraries based on MPI-3, such as OpenCoarrays.

2. Related Work

The problem of tuning and auto-tuning communication libraries, like MPI, has been tackled several times in the past, using many different approaches.

In [10], Miceli et al. propose AutoTune, an extension of Periscope [1], an automatic distributed performance analysis tool. This framework tries to optimize a parallel application under many aspects including MPI tuning, thread affinity, and CPU frequency.

In [15], Sikora et al. extend again Periscope as part of the AutoTune project to implement autotuning capabilities for MPI applications. The output of the framework proposed is a set of tuning recommendation that can be integrated into the production version of the code. This tool provides the user with evolutionary algorithms able to heuristically guide the search of the most significant tuning parameters in MPI by executing a reasonable number of experiments.

Pellegrini et al. in [12] propose the use of two machine learning algorithms (decision trees and neural networks), to implement a predictive model that analyzes any MPI input program, and according to gained knowledge of the architecture, produces the value of a set of a predefined runtime parameters that provide optimal speedup. The overall approach proposed by Pellegrini et al. is similar to what we describe in this work, but our machine learning approach and modelization is completely different because it makes use of deep reinforcement learning techniques.

3. (Deep) Reinforcement Learning

The idea behind Reinforcement Learning (RL) is to have a learner called *agent* which interacts with an *environment* through *actions*. The *environment* responds to the *actions* and it presents new situations to the *agent*. The *environment* also gives rise to *rewards*: a numerical representation that the *agent* tries to maximize. The final goal of Reinforcement Learning is to find a *policy*, that maximizes the overall reward for the agent. A *policy* is a mapping from states to probabilities of selecting a certain *action*. Reinforcement Learning methods specify how the *agent* changes its *policy* as a result of experience.

Q-Learning is a reinforcement learning technique. It belongs to the class of model-free methods and tries to estimate the Q-value function using the update equation expressed in 1.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \quad (1)$$

Q-learning is just the Bellman optimality equation applied iteratively to evaluate and improve the Q-value function in a model-free problem, using a greedy policy. In other words, the best update rule to estimate the optimal action-value function Q for a given state, is the quantity that leads to the optimal policy. The optimal policy is the one given by the Bellman optimality equation, which is the max Q among all possible actions in the next state.

The Q-learning algorithm can be implemented by just keeping track of the Q-values of all the visited states in a table, but this is prohibitive for real problem with a large number of states.

Alternatively, one could estimate the Q-value of the states, using various techniques. One of these is called "Deep Q-Learning" and it involves the use of a deep neural network for the estimate. Unfortunately, applying non-linear function approximators to model-free algorithms, such as Q-learning, could cause the Q-network to diverge [16], however there have been works to fix the divergence issue such as the gradient temporal-difference methods like [7] and [8].

The most famous and meaningful example of successful application of deep reinforcement learning is probably [11], where a convolutional neural network has been used to interpret the state of an Atari video game to produce the values of Q for all the possible actions allowed by the game. In the Atari work [11], the stability of the Q-learning algorithm, while using neural networks, is guaranteed by two mechanisms: experience replay and fixed Q-targets. Experience replay is random sampling over the entire experience accumulated and applying an optimization step on the neural network using the samples. This mechanism makes sure to break the temporal correlation of the experience observed by the network, resulting in a better stability and convergence of the algorithm. Q-targets means that the Q values used to compute the updates of the Q-learning algorithm belong to a neural network trained on old values. In [11], the authors use two neural networks, an they switch between the two after a certain number of steps to compute the Q-value for the targets in the Q-learning algorithm.

4. Potential in Communication Library Introspection

Understanding the performance issues of an MPI code is an operation that requires low-level information; for example, knowing how much time is spent in an `MPI_Recv` can help to understand whether the application suffers of poor load balancing or just high communication costs. Such a low-level information is usually hidden into the internal variables of the MPI implementation. For example, a typical information that can be useful to know is *how many messages are in the Unexpected Message Queue waiting to be received?*.

With the new tools information interface introduced in MPI-3, MPI provides a standard way to access performance data contained inside the MPI implementation (called *performance variables*) and internal variables that control the behavior of the implemen-

tation (called *control variables*). An example of a *control variable* is the one that defines the threshold, associated with the message size, that decides whether a message should be sent using the eager or rendezvous protocol.

Although the performance variables are common to any MPI implementation (e.g., Unexpected Message Queue length), the MPI Forum does not specify a direct way to get the status of these variables. The intent of the MPI Tool Information Interface (from now on MPI_T, see Section 4.1) is to enable an MPI implementation to expose implementation-specific details; for this reason is not possible to define variables that all MPI implementations must provide. This approach is called *introspection*. The most common use case for the MPI_T is to provide performance information and control variables to profilers and debuggers in order to help the users understanding issues and bottlenecks in MPI applications.

It is possible to write applications that take advantage of the information provided by MPI_T, but introducing such low-level concepts in user code is not advisable. We believe that the best opportunities to improve the performance of an MPI application using MPI_T are in the run-time communication libraries built on top of MPI. In fact, MPI_T has been already successfully used by run-time communication libraries to select the best algorithm based on the support provided by the MPI implementation. For example, Fanfarillo and Hammond in [5] use the MPI_T to select the best algorithm to implement *events* in OpenCoarrays [4], with a remarkable performance enhancement.

4.1. MPI Tool Information Interface (MPI_T)

MPI_T provides a standard interface to access *performance variables* and *control variables*. For both types of variables, there are several common concepts. In order to access a variable, an handle must be created first. With the handle the MPI implementation can provide low-overhead access to the internal variable.

Control variables allow the use to influence how the MPI implementation works. In order to use a *control variable*, the variable needs to be discovered. MPI provides functions to implement *introspection*, discover how many control variables are available, getting their details and modifying their values. During this work, we found out that it is important to modify all the *control variables* values before calling `MPI_Init`.

Performance variables are usually expressed in terms of queue lengths, waiting times, re-transmission attempts. For example, in a load imbalanced situations, where some processes make send requests before that the corresponding receives have been posted, the length of the unexpected message queue will be longer on some processes than on others. Another typical symptom of load imbalance is the longer time spent in a receive, waiting for the data to arrive. By combining the data with an understanding of how the implementation works, profilers are able to provide clues to the programmer on how to determine the source of the performance problem. The way *performance variables* are accessed is similar to the way *control variables* are managed but *performance variables* require an additional step: the creation of a session. A session enables different parts of the code to access and modify a *performance variable* in a way that is specific to that part of the code. In other words, a session provides a way to isolate the use of a *performance variable* to a specific part of the code. In order to read the value associated with a *performance variable* the creation of handle and session should be performed after calling `MPI_Init`.

4.2. OpenCoarrays

OpenCoarrays [4] is an open-source software project for developing, porting and tuning transport layers that support coarray Fortran compilers. It targets compilers that conform to the coarray parallel programming feature set specified in the Fortran 2008 standard. It also supports several features defined in the Fortran 2018 standard including: *events* for fine-grain synchronization between parallel entities, *failed images* to manage failures, collective/reduction (called *collective*), and a partial implementation of *teams*, used to create independent subgroups of parallel entities. Currently, it is used as the run-time communication library by the GNU Fortran (GFortran) compiler.

OpenCoarrays defines an application binary interface (ABI) that translates high-level communication and synchronization requests into low-level calls to a user-specified communication run-time library. This design decision liberates compiler teams from hardwiring communication-library choice into their compilers and it frees Fortran programmers to express parallel algorithms once, and reuse identical CAF source with whichever communication library is most efficient for a given hardware platform.

Since the first release of OpenCoarrays (August 2014), the widest coverage of coarray features was provided by a MPI based run-time library (LIBCAF_MPI). Because of the one-sided nature of coarrays, the run-time library uses almost exclusively MPI one-sided communication routines based on passive synchronization.

5. AITuning Design

AITuning has been designed as a separate component from run-time communication libraries. Its purpose is to guide the automatic tuning process of the libraries utilizing machine learning techniques. It is written in C++ and it is structured to be completely agnostic of run-time libraries, communication libraries, and machine learning algorithms and paradigms (although RL approaches are well suited for this problem).

5.1. Architecture

[2]The Controller class exposes a set of methods identified by the prefix `AITuning_*` that can be called by the run-time library. The method `AITuning_start(string layer)` takes a string representing the communication layer to be used. This method needs to be called before the initialization of the communication library (in this case `MPI_Init_thread`). In order to plug AITuning in OpenCoarrays without changing the source code of the latter, we decided to use the MPI Profiling Interface. We created wrappers for the MPI functions that AITuning needs to interact with (e.g. `MPI_Init` and `MPI_Finalize`) and called the `AITuning_*` methods from there.

As explained in Section 4.1, *control variables* and *performance variables* needed to be set before and after the actual call to `MPI_Init_thread`, respectively. Once the layer has been passed to the Controller object, a specific CollectionCreator is instantiated using the CollectionCreator object. The actual collection (in our case MPICHCollectionCreator) has predefined lists of control and performance variables that we decided and used for a specific AI component.

[2]A class diagram of the architecture is available on https://github.com/NCAR/AITuning

In order to make AITuning general enough to handle any kind of control and performance variables, we decided to declare the classes `ControlVariable` and `PerformanceVariable` as abstract. In fact, besides the default control and performance variables defined in a specific Collection object (related to a specific communication library implementation), it is possible to define UserDefined Performance Variables. This class of variables allows the user to defined specific performance variables, like the time spent to run the entire application, the time spent to execute a `MPI_Win_flush` and similar. Since they all inherit from the abstract class PerformanceVariable, they can be stored in the CollectionPerformanceVar object. In order to read performance variables, specific objects of the class Probes should be used. This class makes sure that the performance variables read using MPI_T or any other way (user defined included), respect certain criteria, like datatype, precision, and range.

All the performance variables keep track of the values detected during the program execution. At the end of the execution, in a wrapper of `MPI_Finalize`, statistics of the values get collected (e.g. average, max, min, median) and they will form the "state" representation passed to the AI component.

The entire machine learning process is performed in the `MPI_Finalize` wrapper, at the end of the program. The AI components receives a representation of the state of the application, which represents the state of the environment in a reinforcement learning setting. The reward gets computed in the AI component, based on previous data (in particular total_execution_time) and the reinforcement learning algorithm gets trained on the new data and produces a new action, defined as a "change" for a control variable. The new values for the control variables will be used during the next execution of the same application. A detailed description of the training process and AI component is provided in Section 5.2.

Not all the performance variables are the same; a variable like total_time cannot be passed to the RL algorithm as an absolute value. In fact, the same application has very different execution times when run on a different numbers of processes. In AITuning it is possible to declare a performance variable as "Relative". During the first run, the performance variable declared as relative will maintain in memory the absolute value of the quantity they represent. During the other runs, all the values of a relative performance variable are express as the difference between the absolute value obtained during the first run and the current absolute value. For example, if we consider the total execution time as performance variables, a positive value can be seen as a performance improvement, since during the first run the execution time was higher that the new value. This representation allowed us to write easy reward functions based on the results of relative variables.

5.2. Training

As first step, all the values of the performance variables are "standardized" against a reference run. To do so, a first run (or set of runs) is used as a reference for performance variables related to time and to a specific run in a consistent way. For this reason, when AITuning is active, the first run of the application is used to record the performance variables of the application when using a vanilla MPI implementation. The user communicates the first run by setting an environment variable `AITUNING_FIRST_RUN = 1`.

For every run other than the first, the algorithm produces a new action in the form of a "change" on a control variable. Each control variable has a fixed "step" to be used

to change the absolute value of the control variable. For example, the MPICH control variable `MPIR_CVAR_ASYNC_PROGRESS` which controls the use of a helper thread to implement MPI asynchronous progress, can assume only two values: 0 and 1. On the other hand, the variable `MPIR_CVAR_CH3_EAGER_MAX_MSG_SIZE` assumes a numerical value representing the message size threshold to switch from the eager to the rendezvous protocol: in this case AITuning will change its value in predefined steps of 1024.

In every run, the neural network in charge of estimating the Q-value produces an estimate of the Q-value given a certain state provided by the performance variables. At the end of the run, the new reward gets computed and the neural network gets retrained based on the outcome. In order to make the Q-learning stable, we used the replay technique described in Section 3. We pick a random subset of the whole experience accumulated every 200 runs, and we train the neural network on that. We have not implemented the Q-target technique.

5.3. Control and Performance Variables for MPICH

For now, we focused our efforts only MPICH-3.2.1 because of the small number of control and performance variables exposed by the implementation, which made our reinforcement learning algorithm design and training faster. The control variables chosen for MPICH-3.2.1 are `ASYNC_PROGRESS`, `CH3_ENABLE_HCOLL`, `CH3_RMA_DELAY_ISSUING_FOR_PIGGYBACKING`, `CH3_RMA_OP_PIGGYBACK_LOCK_DATA_SIZE`, `POLLS_BEFORE_YIELD`, `CH3_EAGER_MAX_MSG_SIZE`. The only performance variable chosen from MPICH-3.2.1 was `unexpected_recvq_length`, representing the length of the unexpected message queue. We use several user-defined performance variables related to the average and maximum time needed to complete `MPI_Win_Flush`, `MPI_Put`,`MPI_Get`, and total application time. We also added the number of processes used in the run as input parameter.

5.4. Inference

AITuning will be shipped along with OpenCoarrays already trained for several MPI implementations and transport layers (e.g. GASNet). When the user decides to activate AITuning, he/she will compile OpenCoarrays using the PMPI wrapper. At this point, we recommend the user to run their application for at least 20 times. During these 20 runs, the RL algorithm will "explore" the new application and produce the right combination of parameters. During this exploration phase, AITuning may produce a configuration that penalizes the performance. At the end of the 20 runs, AITuning analyzes the results, discards the runs where the performance was penalized, and applies the median over the values of the control variables of the runs that provided good results within 5% from the best (creating an ensemble). Further runs of the same applications with different data input but same number of images will not require additional runs.

6. Experimental Evaluation

In order to train AITuning properly on MPICH-3.2.1, we decided to use two different supercomputers: Cheyenne (NCAR) an SGI machine with InfiniBand network interconnect and Edison (NERSC) a Cray XC30 with Aries interconnect. For the training we

decided to use four main codes parallelized with Coarrays Fortran: 1) CloverLeaf [9], 2) Lattice-Boltzmann code [13], 3) Skeleton Particle-in-cell [2], 4) Parallel Research Kernels [3]. We have run the aforementioned codes using a different number of processes going from 64 to 2048 for a total of 5000 runs.

6.1. Intermediate Complexity Atmospheric Research

The Intermediate Complexity Atmospheric Research (ICAR) [6] model developed at NCAR, is a simplified atmospheric model designed primarily for climate downscaling, atmospheric sensitivity tests, and educational uses. ICAR is a quasi-dynamical downscaling approach that uses simplified wind dynamics to perform high-resolution meteorological simulations 100 to 1000 times faster than a traditional atmospheric model and can therefore be used to better characterize uncertainty across numerical weather prediction models and climate models, and in dynamical downscaling.

In [14], Rouson et al. developed a mini-app of the ICAR model using coarray Fortran, showing great performance improvements. Since then, lead developer of ICAR, Ethan Gutmann, developed a fully functional version of ICAR based on coarray Fortran, which we used for testing AITuning. The version of ICAR we used is a full atmospheric model; the code include computation, communication and IO parts.

6.2. Results Evaluation

In Figure 1, we report the results obtained for ICAR running on Cheyenne using the default "vanilla" configuration set in MPICH-3.2.1, the optimized configuration found by AITuning after running ICAR 20 times, and an human optimized version based on reasonable guesses. The "default" bars represent the total time needed to complete a test case on ICAR using the default settings and in both cases, with 256 and 512 images, it provides the worst performance. On the other hand, the "optimized" version produced by AITuning always leads to the best performance. In both the 256 and 512 images cases, the manual optimization increased the eager limit by an order of magnitude higher than the default while leaving all the other setting as in the default configuration. For the case with 256 images, the optimized version provides 13% performance improvement compared to the vanilla version. For the case with 512 images, the optimized version provide 25% performance improvement over the vanilla version, mostly because of the higher communication cost imposed by the higher number of processes and same problem size (strong scaling).

The most influential tuning parameter for the ICAR test case resulted to be the presence of the asynchronous progress thread. We also noticed that some parameters have a different influence based on the number of processes being used. In particular, the value of `MPICH_POLLS_BEFORE_YIELD` played a much more relevant role in the case with 512 images than in the case with 256 images. This is not surprising because ICAR attempt to overlap computation with communication by using coarray "puts" instead of "gets". For the 256 case, the optimal configuration found by AITuning had `MPICH_POLLS_BEFORE_YIELD` set to the default value 1000, meaning that it was found not relevant. On the other hand, for the 512 images case, AITuning found a value of 1100. We manually changed the value of `MPICH_POLLS_BEFORE_YIELD` by keeping the configurations found by AITuning the same for both cases and found that in the case with

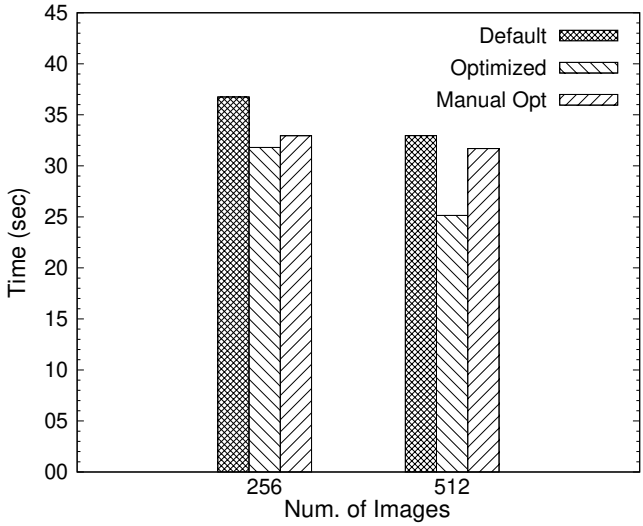


Figure 1. Performance comparison between default and optimized configurations

512 images, a value of `MPICH_POLLS_BEFORE_YIELD` between 1200 and 1500 provides the best performance, so it seems there is still room for improvement.

7. Conclusions and Future Work

In this work, we presented AITuning, a machine learning-base tuning tool for run-time libraries. AITuning has been released under open-source license and it is currently available on github [3]. It currently works with the OpenCoarrays library, but its structure allows it to be extended to any run-time communication library, based on any communication layer. To the best of our knowledge, this paper is a unique contribution because it is the first attempt to try to find the optimal tuning parameters used a deep reinforcement learning algorithm and MPI_T. We tested AITuning and our RL algorithm, carefully designed for MPICH-3.2.1, using a real atmospheric code: ICAR. AITuning was able to produce a configuration of parameters that lead to 13% and 25% performance improvement for the case running on 256 and 512 images, respectively, compared to the default configuration. It also improves performance compared to an expertly tuned configuration, marginally for 256 images and substantially for 512.

In the future, we plan to extend our analysis to other MPI implementations with a higher number of control and performance variables. Furthermore, we will explore more options on the RL algorithm, and potentially other machine learning approaches. In our brief preliminary tests, it has been clear that whatever technique is chosen, it must be very robust to the noise of run-to-run variability. However, finding the best learning algorithm for AITuning is beyond the scope of this paper and left for a future work.

Finally, to better evaluate the results of the tool, we plan to test it on a larger number of machines and on a larger and more diverse set of applications.

[3] https://github.com/NCAR/AITuning

References

[1] Shajulin Benedict, Ventsislav Petkov, and Michael Gerndt. Periscope: An online-based distributed performance analysis tool. In Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 1–16, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[2] Viktor K. Decyk. Skeleton pic codes for parallel computers. *Computer Physics Communications*, 87(1):87 – 94, 1995. Particle Simulation Methods.

[3] R. F. Van der Wijngaart and T. G. Mattson. The parallel research kernels. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, Sept 2014.

[4] Alessandro Fanfarillo, Tobias Burnus, Valeria Cardellini, Salvatore Filippone, Dan Nagle, and Damian Rouson. Opencoarrays: Open-source transport layers supporting coarray fortran compilers. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 4:1–4:11, New York, NY, USA, 2014. ACM.

[5] Alessandro Fanfarillo and Jeff Hammond. Caf Events Implementation Using MPI-3 Capabilities. In *Proceedings of the 23rd Message Passinge Interface Users and Developers Conference*, EUROMPI '16. ACM, 2016.

[6] Ethan Gutmann, Idar Barstad, Martyn Clark, Jeffrey Arnold, and Roy Rasmussen. The intermediate complexity atmospheric research model (icar). *Journal of Hydrometeorology*, 17(3):957–973, 2016.

[7] Hamid R. Maei, Csaba Szepesvári, Shalabh Bhatnagar, Doina Precup, David Silver, and Richard S. Sutton. Convergent temporal-difference learning with arbitrary smooth function approximation. In *Proceedings of the 22Nd International Conference on Neural Information Processing Systems*, NIPS'09, pages 1204–1212, USA, 2009. Curran Associates Inc.

[8] Hamid Reza Maei, Csaba Szepesvári, Shalabh Bhatnagar, and Richard S. Sutton. Toward off-policy learning control with function approximation. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pages 719–726, USA, 2010. Omnipress.

[9] A. C. Mallinson, S. A. Jarvis, W. P. Gaudin, and J. A. Herdman. Experiences at scale with pgas versions of a hydrodynamics application. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 9:1–9:11, New York, NY, USA, 2014. ACM.

[10] Renato Miceli, Gilles Civario, Anna Sikora, Eduardo César, Michael Gerndt, Houssam Haitof, Carmen Navarrete, Siegfried Benkner, Martin Sandrieser, Laurent Morin, and François Bodin. Autotune: A plugin-driven approach to the automatic tuning of parallel applications. In Pekka Manninen and Per Öster, editors, *Applied Parallel and Scientific Computing*, pages 328–342, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[12] Simone Pellegrini, Jie Wang, Thomas Fahringer, and Hans Moritsch. Optimizing mpi runtime parameter settings by using machine learning. In Matti Ropo, Jan Westerholm, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 196–206, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[13] C. Rosales. Porting to the intel xeon phi: Opportunities and challenges. In *2013 Extreme Scaling Workshop (xsw 2013)*, pages 1–7, Aug 2013.

[14] Damian Rouson, Ethan D. Gutmann, Alessandro Fanfarillo, and Brian Friesen. Performance portability of an intermediate-complexity atmospheric research model in coarray fortran. In *Proceedings of the Second Annual PGAS Applications Workshop*, PAW17, pages 4:1–4:4, New York, NY, USA, 2017. ACM.

[15] Anna Sikora, Eduardo César, Isaías Comprés, and Michael Gerndt. Autotuning of mpi applications using ptf. In *Proceedings of the ACM Workshop on Software Engineering Methods for Parallel and High Performance Applications*, SEM4HPC '16, pages 31–38, New York, NY, USA, 2016. ACM.

[16] J. N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690, May 1997.

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200067

Towards Benchmarking the Asynchronous Progress of Non-Blocking MPI Operations

Alexey V. MEDVEDEV
Institute of mechanics, Lomonosov Moscow State University,
a.medvedev@imec.msu.ru

Abstract. This paper discusses the problem of reliable benchmarking of non-blocking MPI communications, both non-blocking point-to-point and non-blocking MPI-3 collective operations. The problem of accurate and practical estimating the level of calculation/communication overlapping (communication hiding efficiency) is discussed. Authors propose the efficiency estimation approach and methodology which is different from previous well-known works. The new IMB-ASYNC benchmark design is proposed. Some practical tests were made on Lomonosov-2 supercomputer using widely used Intel MPI 2017 Update 1 MPI implementation. The results of the tests are discussed, and the future work on IMB-ASYNC testing and development is outlined.

Keywords. MPI, non-blocking, asynchronous, benchmarking

1. Introduction

The non-blocking MPI communications, both point-to-point and MPI-3 non-blocking collectives are important tools which allow the design of MPI-based parallel algorithms that avoid rigid, fully blocking communication patterns. The use of non-blocking MPI operations also makes it possible to introduce algorithm designs which imply calculation/communication overlapping to hide the cross-rank communication latency. The problem in the latter case is that MPI standard does not require from MPI library implementers to make non-blocking communication operations partially or fully asynchronous. This means that for a good level of calculation/communication overlapping some effort from both middleware implementers and parallel code authors may be required.

To estimate which level of asynchronous message passing progress at the same time with intensive calculations can be expected on a particular machine using a particular communication middleware setup, the general purpose benchmarking tools can be used. The popular MPI micro-benchmarks OSU [1] and IMB [2] include only a small subset of scenarios that may help to estimate which calculation/communication overlapping level might be available for some practical applications. The authors faced at least two cases in their work when existing benchmarking tools do not give a full picture: i) in sparse linear algebra applications, there are several research results [4] offering the non-blocking MPI_Iallreduce call usage in Krylov subspace iterative methods to hide the latency of dot product calculations; ii) point-to-point communications with neighbors are also widely used (in their non-blocking forms namely) in sparse linear algebra applications due to good potential of communication

latency hiding. Before the new MPI-based code development, it is nice to estimate which overlapping can be expected in both cases, and how the overlapping depends on a particular middleware/hardware and how it changes at scale: these estimations help a lot at the algorithm choice stage. This necessity is not limited to sparse linear algebra problems.

The goal of this work is to offer the basic and practical design of some new benchmarks which may meet the demand of parallel algorithm developers challenged with the problem of hiding the communication latency and to offer the calculation/communication overlapping efficiency estimation method.

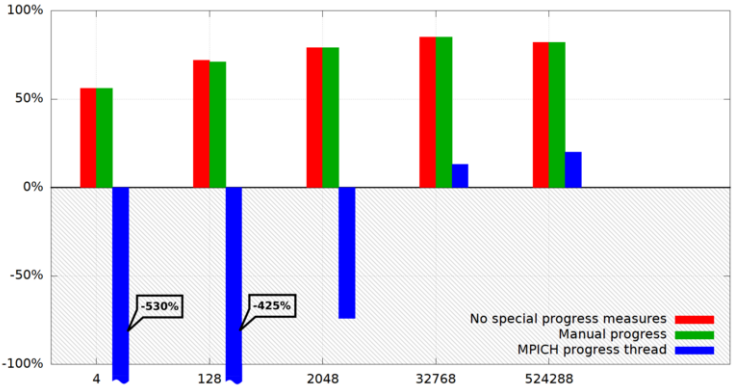


Figure 1. Communication Hiding Efficiency, Eq. (1): 2 nodes, async_pt2pt benchmark.

2. IMB-ASYNC Benchmark

The group of benchmark codes described in this paper is a newly developed extension to the open-source Intel MPI Benchmark 2019 suite, made by authors as a separate project [3]. The name for the benchmark group – IMB-ASYNC – is chosen to match the naming conventions of other IMB benchmark groups. The difference between the benchmarks in question and older parts of IMB suite is that all of their non-blocking variants include tunable computational load simulation procedure. The time for this procedure to produce CPU load can be set up from the command line with ~1 microsecond precision. This makes it possible to estimate calculation/communication overlap carefully for every scenario.

The IMB-ASYNC code includes:

1. Point-to-point benchmarks estimating the pair-wise communications between regular pairs of ranks from the MPI_COMM_WORLD communicator. The ping-pong style communications in each pair are implemented in two ways: "sync_pt2pt" is a sequence of blocking MPI_Send and MPI_Recv calls (reversed in its order in one of the ranks of a pair) and "async_pt2pt" is a sequence of non-blocking MPI_Isend, MPI_Irecv, followed by optional block, simulating high computational load, with MPI_Waitall call finishing the

benchmark iteration. The design of blocking kind of this benchmark is made similar to a well-known "PingPong" benchmark of IMB-MPI1 suite.

2. Point-to-point benchmark estimating the neighborhood communications between a given rank and two or more neighbor ranks. It also has two implementations: blocking ("sync_exch") and non-blocking ("async_exch") flavors, and is equipped with the same optional computational load simulation procedure for a non-blocking kind.
3. Collective benchmark ("sync_allreduce", "async_allreduce") estimating blocking and non-blocking kind of one of most popular collective operations. Non-blocking kind is also equipped with the same optional computational load simulation procedure.

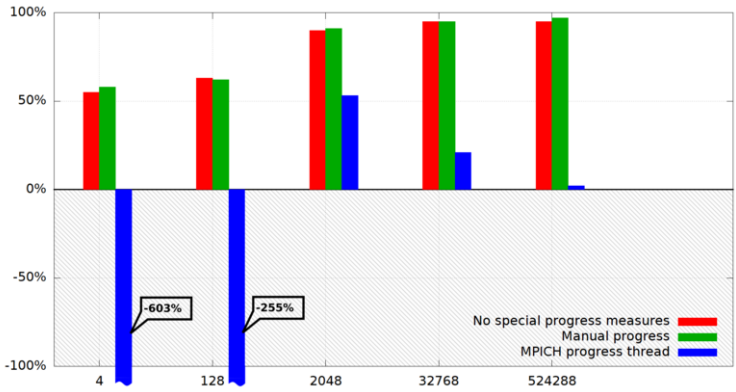


Figure 2. Communication Hiding Efficiency, Eq. (1): 16 nodes, async_pt2pt benchmark.

Typical running session for IMB-ASYNC benchmarks includes at least three benchmark runs:

1. Run the benchmark to calibrate calculations simulating loop. This operation can be made once for the whole testing session. The aim is to find out how many loop iterations make approximately 10 microseconds of computing time on an environment being tested in normal mode. The calibration parameter is later used during the non-blocking communications benchmarking to eliminate possible influence the async progress actions on actual calculation process being done at the same time as the communications.
2. For each benchmark type and message size, run the version of a benchmark which uses blocking MPI communications. This step should typically be done separately from the following one for 3 main reasons: i) it provides a productivity baseline which can be used later to estimate, how effective corresponding non-blocking MPI communications are and how effective the asynchronous progress is; ii) the time of communication execution provides a hint to which amount of time for calculations simulating loop during the following non-blocking run is reasonable; iii) the non-blocking run may require different parameters of MPI environment to be set up (i.e. the case

with dedicated progress thread or threads configuration, which is discussed later).

3. For each benchmark type and message size, run the version of benchmark which uses non-blocking communications and calculations simulating loop, running at the same time. The amount of time the calculations simulating loop is running must be given explicitly in command line, and typically must not be less than pure communication time obtained on a benchmarking step with blocking communications. The loop calibrating parameter, from the first stage of testing, must also be passed explicitly as a command line parameter. Non-blocking version of benchmarks reports as its result not only a wall time of the benchmarking routine, but also calculates the Communication Overhead value, which consists of execution time of all MPI calls plus a rate of calculations simulating loop slowdown according to a given calibration value.

It is the combination of two independent runs of the benchmark that gives good basis to analyze the real asynchronous performance of MPI implementation being tested and the real rate of communication hiding which an application developer may expect in a particular case.

3. Asynchronous Progress Implementation Details

The asynchronous progress of MPI operations may be implemented in different ways. First, the progress can be made "transparently" by communication hardware. However, it is uncommon for up-to-date MPI libraries to have "transparent", hardware-supported asynchronous progress for all possible non-blocking operations and message sizes. Then, there are techniques at the application level which may "stimulate" or "accelerate" the asynchronous progress in some particular cases. The specialized "progression threads" usage (like MPICH_ASYNC_PROGRESS-related technique for MPICH-based implementation) is often discussed in literature. Also, the popular way of software-based progress acceleration is: to do "manual progression" via periodical calls of MPI_Test or MPI_Probe functions during the progress of computations.

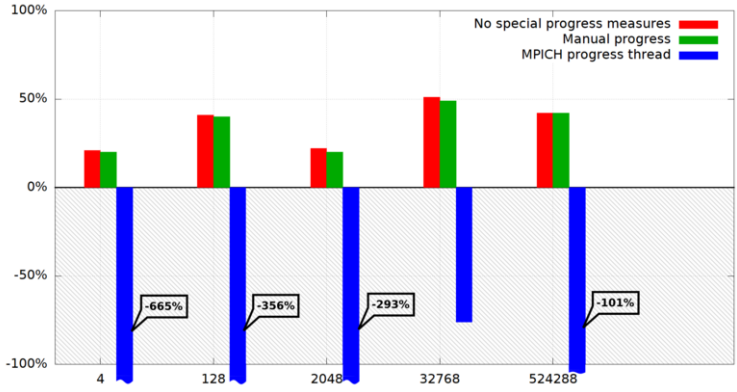


Figure 3. Communication Hiding Efficiency, Eq. (1): 64 nodes, async_pt2pt benchmark.

The progression thread approach is being discussed for a long time. Possible advantages and caveats of the approach were analyzed in [6]. At least one practical implementation is available as a part of well-known MPICH MPI library. Another implementation is discussed in [7]. Modern research on using dedicated cores for progression threads and other related aspects is discussed in [8]. This quick overview shows that the way of accelerating asynchronous message passing progress via dedicated progression threads is an acute research topic.

The approach of periodical calls of MPI_Test or MPI_Probe functions is also worth to be estimated with benchmarking. For this purpose, the calculation simulating loop of the IMB-ASYNC benchmarks includes the section of MPI_Test calls for all the MPI requests appeared in a benchmark. This section can be switched on or off with benchmark command line parameters.

4. Efficiency Estimation Method and Computational Experiment Methodology

The main goal of the estimation is to understand the time benefit one may have by hiding communications behind the calculations for each message. For this purpose, the simulative calculation procedure in the non-blocking versions of benchmarks is turned on. Amount of computing cycles is chosen manually so that communication time was 1.5-2 times less than the computation time (see "Calculation time" table rows). This way to choose calculation cycle length is chosen because it corresponds to the majority of use cases: normally calculation-intensive algorithms have enough CPU workload to hide calculation latency, and the situations when calculation time is comparable with communication time seems to be a corner case of extreme scaling attempts.

The communication overhead which appears in this running mode is then compared to the baseline time of blocking communications to find out the Communication Hiding Efficiency:

$$E = 100\% - (T_{over} / T_{sync}) * 100\% \quad (1)$$

where T_{over} is the communication overhead time in calculation/communication mode, T_{sync} is the baseline.

Table 1. IMB-ASYNC sync_pt2pt, async_pt2pt results on Lomonosov-2. Point-to-point pairwise communications.

2 nodes (28 ranks):	**4**	**128**	**2048**	**32768**	**524288**
Pure communication time, blocking MPI (T_{sync})	3.5	4.9	38.7	613.9	10172.4
Overhead, no special progress measures	1.3	1.3	7.0	136.4	3495.4
Overhead, manual progress (T_{over1})	1.9	1.7	7.4	90.2	214.3
Overhead, MPICH_ASYNC_PROGRESS (T_{over2})	1.5	1.2	7.0	122.6	3289.6
16 nodes (224 ranks):					
Pure communication time, blocking MPI (T_{sync})	7.9	13.4	186.0	2927.1	46630.4
Overhead, no special progress measures	3.5	5.0	17.7	153.2	2436.8
Overhead, manual progress (T_{over1})	3.3	5.1	16.6	153.2	2436.8
Overhead, MPICH_ASYNC_PROGRESS (T_{over2})	55.4	47.4	87.0	2300.4	45772.0
64 nodes (896 ranks):					
Pure communication time, blocking MPI (T_{sync})	8.0	13.4	184.8	2919.1	47465.5
Overhead, no special progress measures	6.3	8.0	143.4	1442.8	27571.3
Overhead, manual progress (T_{over1})	6.4	8.1	147.3	1483.7	27428.5
Overhead, MPICH_ASYNC_PROGRESS (T_{over2})	61.3	61.4	727.0	5147.2	95445.1

The Efficiency of 100% means that thanks to non-blocking communication form usage, all the communications are successfully hidden by computations. This is the best possible case. The worst case is when Efficiency values become negative: it means that combining the non-blocking communications with calculations is even slower than the sequential combination of calculations and blocking communications.

Please note that this Efficiency estimation method is different from the one used in IMB-NBC suite [5]. IMB-NBC compares the communication overhead time in calculation/communication mode with the pure latency of asynchronous communication call without calculations. This way of estimating the efficiency is not quite practical, because the choice is normally made between two modes: synchronous communications serialized with calculations or non-blocking communications with the goal to overlap them with communications. Moreover, in IMB-NBC methodology, it is difficult to estimate correctly which penalty the progress thread introduces since both estimation of baseline time and the benchmark of interest are made during the same execution session, and possible penalty influences both figures. The IMB-NBC suite always use the calculation time equal to baseline communication time. This doesn't give the possibility to adjust the calculation/communication overlap case according to the practical interest of the algorithm designer. Also, IMB-NBC doesn't try to estimate possible efficiency loss due to calculations slowdown when progress thread technique is used to support asynchronous progress.

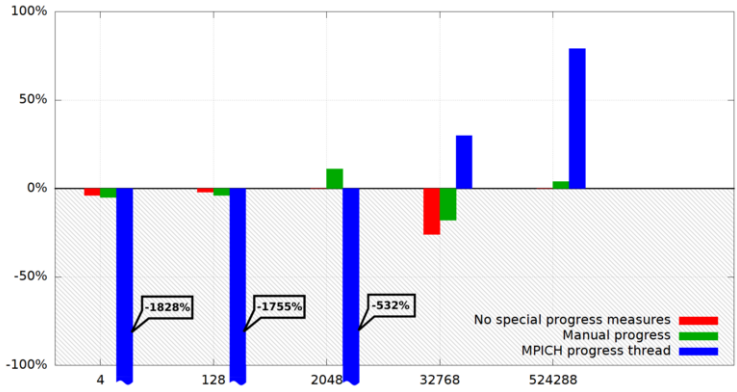


Figure 4. Communication Hiding Efficiency, Eq. (1): 2 nodes, async_allreduce benchmark.

5. Practical Computational Experiments

The benchmark code was tested on Lomonosov-2 supercomputer of the Lomonosov Moscow State University. A short selection of the benchmarking results is given in Table 1 and Table 2.

The full subscription of 2, 16 and 64 computing nodes (14 cores and 14 ranks per node with per-core affinity) was used to estimate the asynchronous progress expectations with Intel MPI 2017.1 library. The baseline is the time taken by a

blocking version of each benchmark (see "Pure communication time, blocking MPI" table rows).

Table 2. IMB-ASYNC sync_allreduce, async_allreduce results on Lomonosov-2. MPI_Allreduce, MPI_Iallreduce communications on the full MPI_COMM_WORLD.

2 nodes (28 ranks):	**4**	**128**	**2048**	**32768**	**524288**
Pure communication time, blocking MPI (T_{sync})	12.0	15.0	46.0	308.9	7221.7
Overhead, no special progress measures	12.5	15.3	46.0	388.2	7241.4
Overhead, manual progress (T_{over1})	12.7	15.7	41.0	365.7	6913.5
Overhead, MPICH_ASYNC_PROGRESS (T_{over2})	231.4	278.9	290.8	216.9	1534.1
16 nodes (224 ranks):					
Pure communication time, blocking MPI (T_{sync})	36.6	35.9	81.1	547.9	8184.8
Overhead, no special progress measures	37.6	39.9	84.5	551.2	7718.8
Overhead, manual progress (T_{over1})	37.0	39.6	69.1	458.3	7331.3
Overhead, MPICH_ASYNC_PROGRESS (T_{over2})	863.3	875.7	834.8	192.8	3136.7
64 nodes (896 ranks):					
Pure communication time, blocking MPI (T_{sync})	60.6	58.7	107.5	655.8	9192.8
Overhead, no special progress measures	60.4	57.3	139.5	823.6	7585.6
Overhead, manual progress (T_{over1})	55.8	55.9	136.4	725.3	7508.4
Overhead, MPICH_ASYNC_PROGRESS (T_{over2})	617.8	626.6	778.1	1262.1	21514.5

Two kinds of progress-support techniques were estimated: the "Manual progress" table rows show the results of enabling manual progression via periodical MPI_Test calls in the benchmark code; the "MPICH_ASYNC_PROGRESS" rows show what the internal progress-thread based technique of Intel MPI 2017.1 library may offer instead.

The above mentioned table rows show the Communcation Overhead value calculated by IMB-ASYNC benchmark as described in section 2 of this article. The first row in Table 1 and Table 2 shows the amount of elements in each MPI message. Message element is MPI_DOUBLE. All the times are given in microseconds.

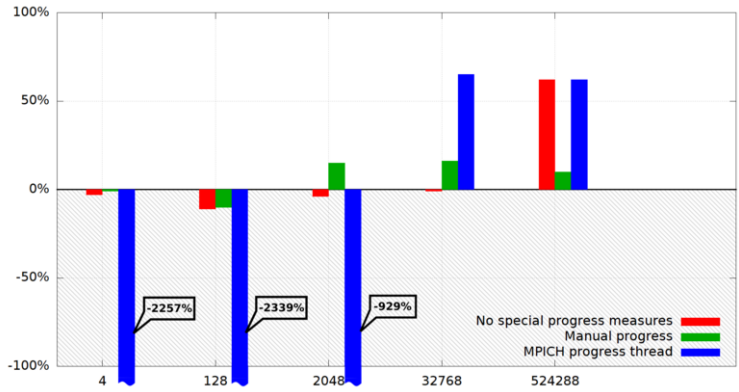


Figure 5. Communication Hiding Efficiency, Eq. (1): 16 nodes, async_allreduce benchmark.

The Communication Hiding Efficiency parameter can be calculated from the data given in Table 1 and Table 2 using Eq. (1). The graphical representation of observed efficiency values is shown on Figures 1-6.

The resulting Communication Hiding Efficiency for point-to-point communications reaches values of 80-90% in experiments without any special progress measures and in experiments with "manual" progression. "Manual" progression don't seem to give any benefit over ordinary runs. Best results are observed for larger message sizes. On smaller message sizes, Communication Hiding Efficiency drops to values about 50%. The experiments at a scale of 64 nodes show worse efficiency: all the observed at this scale values of efficiency are between 20% and 50%.

The figures for MPICH_ASYNC_PROGRESS-enabled experiments are much worse: efficiency values for smaller message sizes are negative, which means the actual slowing down the communication operation, in many cases the communication becomes several times slower. Positive efficiency in this mode observed only for larger message sizes. At scale of 64 nodes, all efficiency values in MPICH_ASYNC_PROGRESS mode are negative.

The resulting Communication Hiding Efficiency for MPI_Iallreduce collective operation appears to be of negative or small positive value for the majority of tested cases. The exception are a few cases with two largest message sizes of this experiment set, where MPICH_ASYNC_PROGRESS mode shows efficiency of 20%-80%, but this efficiency switches to negative values at a scale of 64 nodes.

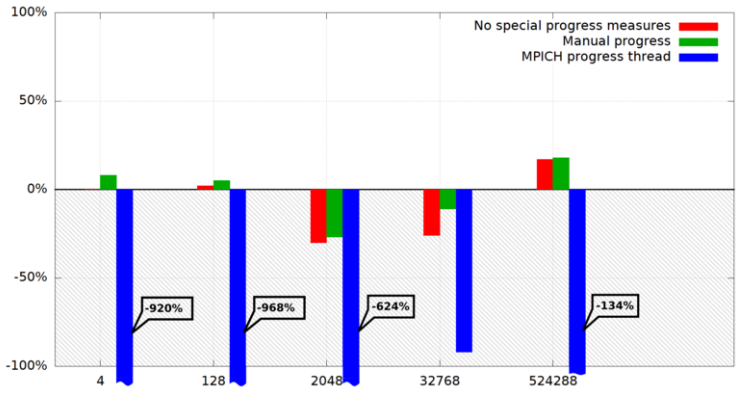


Figure 6. Communication Hiding Efficiency, Eq. (1): 64 nodes, async_allreduce benchmark.

6. Conclusion and Future Work

The presented IMB-ASYNC benchmark suite is an attempt to implement a reliable tool inherited from a well-known and well-tested IMB code for both point-to-point and collective MPI benchmarks. The demand of such a benchmark comes from both application level developers and message-passing middleware developers since the asynchronous message passing progress, implemented both in hardware and software is

a foremost feature of any modern MPI implementation. The calculation/communication overlapping efficiency estimation method is proposed.

The particular results of testing sessions on Lomonosov-2, which are described in section 5 of this article and illustrated by Tables 1-2 and Figures 1-6, may lead to a few conclusions:

1. MPICH_ASYNC_PROGRESS=1 option of Intel MPI library (at least of 2017.1 its version) at many test points gives extremely huge penalty, especially at lower message sizes and at higher scale. Moreover, during the experiments it was noticed, that the MPI performance when the MPICH_ASYNC_PROGRESS is enabled tend to be very variative from one execution to another. Worst cases are even 2 times worse than those given in this paper. This makes it very doubtful that there is any reasonable application of this mode in practice.
2. Manual progression support of non-blocking point-to-point operations with periodical MPI_Test() calls doesn't change anything in asynchronous progression of this type of communication, or the changes are of no practical importance.
3. There is no way to significantly improve the asynchronous progression of non-blocking collective operations over small messages (at least, MPI_Iallreduce()) with Intel MPI 2017.1 on Lomonosov-2 supercomputer. This fact makes serious impact on parallel algorithms choice for new code which is created to be run on Lomonosov-2 system.

As a part of possible future work, it is worth estimating the same metrics for Intel MPI 2019.5 and OpenMPI versions 2, 3 and 4 which are also installed on Lomonosov-2 system to see the full picture. The comparative tests on some other HPC systems are also in our plans.

7. Acknowledgments

The presented work is supported by the RSF grant No. 18-71-10075. The research is carried out using the equipment of the shared research facilities of HPC computing resources at Lomonosov Moscow State University.

References

[1] OSU Micro-Benchmarks, URL: http://mvapich.cse.ohio-state.edu/benchmarks
[2] Intel MPI benchmark, URL: https://software.intel.com/en-us/articles/intel-mpi-benchmarks
[3] IMB-ASYNC benchmark suite, URL: https://github.com/a-v-medvedev/mpi-benchmarks
[4] B. Krasnopolsky, The reordered BiCGStab method for distributed memory computer systems, *Procedia Computer Science* **1** (2010) 213–218
ICCS2010. doi:10.1016/j.procs.2010.04.024.
[5] M.Brinskiy, A.Supalov, M.Chuvelev, E.Leksikov, Mastering performance challenges with the new MPI-3 standard. *The Parallel Universe* **18** (2014), 33–40
URL: https://software.intel.com/sites/default/files/managed/6a/78/parallel_mag_issue18.pdf
[6] T. Hoefler, A. Lumsdaine, Message progression in parallel computing – to thread or not to thread? Proceedings of the 2008 IEEE International Conference on Cluster Computing. IEEE Computer Society, **October 2008**.
[7] A.Ruhela, H.Subramoni, S.Chakraborty, M.Bayatpour, P.Kousha, D.Panda, Efficient Asynchronous Communication Progress for MPI without Dedicated Resources

[8] A.Denis, J.Jaeger, H.Taboada, Progress Thread Placement for Overlapping MPI NonBlocking Collectives using Simultaneous Multi-Threading. *COLOC: 2nd workshop on data locality, in conjuction with EURO-PAR 2018* **Aug 2018**. hal-01888257

Power Management

This page intentionally left blank

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200069

Acceleration of Interactive Multiple Precision Arithmetic Toolbox MuPAT Using FMA, SIMD, and OpenMP

Hotaka YAGI [a,1], Emiko ISHIWATA [a], and Hidehiko HASEGAWA [b]
[a] *Tokyo University of Science, Japan*
[b] *University of Tsukuba, Japan*

Abstract. MuPAT, an interactive multiple precision arithmetic toolbox for use on MATLAB and Scilab, enables users to handle quadruple- and octuple-precision arithmetic operations. MuPAT uses the DD and QD algorithms, which require from 10 to 600 double-precision floating-point operations for each DD or QD operation, which entails corresponding execution time costs. In order to reduce the execution time of vector and matrix operations, we apply FMA, AVX2, and OpenMP to MuPAT by using the MATLAB executable file. Unit stride access is required for high performance and it makes vectorization with AVX2 easier. Larger blocks are suitable for parallelization with OpenMP. That is, AVX2 is suitable for the innermost loop and OpenMP is suitable for the outer loop. One result of adopting the described configuration is that matrix multiplication is nearly 13 times faster in a four-core environment. By using parallel processing in this way, the execution time of some DD vector operations is almost twice that of the original double-precision floating-point operations without parallel processing.

Keywords. DD, Double-Double, MATLAB, AVX2, Multicore

1. Introduction

In floating-point arithmetic, rounding error is unavoidable. The accumulation of rounding errors leads to unreliable and inaccurate results. One of the ways to reduce rounding errors is to use a high-precision arithmetic. For example, the high-precision arithmetic is used for improving the convergence of Krylov subspace methods [1] and is used in semidefinite programming problems [2]. Most high-precision arithmetics are implemented through software emulation such as the QD library [3].

Our team developed *MuPAT*, an open-source interactive *Multiple Precision Arithmetic Toolbox* [4,5] for use with the MATLAB and Scilab computing environments. MuPAT uses the DD (Double-Double) [6,7] and QD (Quad-Double) [3,7] algorithms, which are based on a combination of double-precision arithmetic operations. The high execution time cost is due to the large number of operations. We accelerate the DD arithmetics using FMA [8], AVX2 [8], and OpenMP [9].

[1]Corresponding Author: Department of Applied Mathematics, Graduate School of Science, 1-3 Kagurazaka, Shinjuku-ku, Tokyo 162-8601, Japan; E-mail: 1419521@ed.tus.ac.jp.

FMA (Fused Multiply-Add) can perform a double-precision floating-point multiply-add operation in one step with a single rounding, AVX2 (Advanced Vector Extensions 2) instructions can process four double-precision data at once, and OpenMP enables thread-level parallelism in a shared memory.

2. DD Arithmetic

DD (Double-Double) arithmetic [6,7] is based on an algorithm that enables quasi quadruple-precision arithmetic. A DD number a is represented by a combination of two double-precision numbers a_{hi} and a_{lo} such as $a = a_{hi} + a_{lo}$. According to the DD algorithm, each arithmetic operation of DD requires 10 to 30 double-precision floating-point operations and the order of computation must be maintained.

	$c = DD\ addition\ (a, b)$	$c = DD\ multiplication\ (a, b)$
1.	$s = a_{hi} \oplus b_{hi}$	$p = a_{hi} \otimes b_{hi}$
2.	$v = s \ominus a_{hi}$	$e = fl(a_{hi} \times b_{hi} - p)$
3.	$eh = a_{hi} \ominus (s \ominus v)$	$e = fl(a_{hi} \times b_{lo} + e)$
4.	$eh = eh \oplus (b_{hi} \ominus v)$	$e = fl(a_{lo} \times b_{hi} + e)$
5.	$eh = eh \oplus (a_{lo} \oplus b_{lo})$	$c_{hi} = p \oplus e$
6.	$c_{hi} = s \oplus eh$	$c_{lo} = e \ominus (c_{hi} \ominus p)$
7.	$c_{lo} = eh \ominus (c_{hi} \ominus s)$	

Figure 1. DD addition and multiplication. a, b, and c are DD numbers. The symbols $\oplus$, $\ominus$, and $\otimes$ denote the double-precision floating-point operators and the symbols $+$, $-$, and $\times$ denote mathematical operators. $fl(a \times b + c)$ means FMA.

The algorithms for DD addition and multiplication are shown in Figure 1. The number of double-precision floating-point operations for DD addition is 11. The DD multiplication algorithm utilizes FMA (Fused Multiply-Add). FMA can execute a double-precision multiply-add operation in one instruction with a single rounding. By using FMA instructions, the rounding error is reduced. The number of double-precision floating-point operations for DD multiplication is 7 with FMA and 24 without FMA. Thus, the number of double-precision floating-point operations for the DD multiply-add is 18 (=11+7).

3. Environment of Parallelization

Since the order of computation in DD arithmetic cannot be changed, we consider processing multiple data simultaneously by using data-level parallelism for acceleration. The unit of time of each operation is not changed, but if multiple results can be obtained in one unit of time, then the total execution time is reduced. We applied data-level parallelism to vector and matrix operations.

AVX2 (Advanced Vector Extensions 2) instructions [8] can process four double-precision data in one unit of time. The same arithmetic operations are applied to these four data. To do this, four double-precision data must be prepared on a SIMD register. An AVX2 load instruction can load four double-precision data from a continuous mem-

ory location in one unit of time. However, for a discontinuous memory location, four scalar load instructions are needed. AVX2 instructions cannot sum up the SIMD register elements. The performance may increase four-fold.

OpenMP [9] allows thread-level parallelism on shared memory for a multicore environment. Each thread is a separate process with its own instructions and data. By processing threads with the different cores simultaneously, the performance may be increased by the number of cores. A loop is parallelized by putting a pragma directive above the loop. There are two scheduling methods: block and cyclic scheduling.

We assume the memory references should be in column order, since MATLAB stores data column-wise.

Performance [Gflops/sec] is defined as the number of double-precision floating-point operations [flops] divided by the execution time [sec]. The upper bound of performance is defined as min(computational performance, memory performance$\times$operational intensity). The computational performance [Gflops/sec] is defined as the product of clock frequency for the CPU [Hz] and the number of flops which can be computed in one unit of time [flops/sec]. Performance is increased four-fold using AVX2 and by the number of cores using OpenMP. Memory performance [Gbytes/sec] is defined as 8 bytes/cycle times the product of clock frequency for memory [GHz] and the number of channels. Operational intensity (O. I.) [flops/bytes] is defined as the number of double-precision floating-point operations [flops] divided by the number of memory references [bytes].

We used an Intel Core i7 7820HQ, 2.90 GHz CPU, with LPDDR3-2133 memory and Intel compiler 18.0.3 with options -O2, -fma, -mavx, -fopenmp, and -fp-model precise. The peak computational performance of a single core including FMA is 5.80 Gflops/sec, and that of AVX2 or that of four cores is 23.20 Gflops/sec. Performance is 92.80 Gflops/sec using AVX2 with four cores. The peak memory performance is 34.13 Gbytes/sec because there are two channels.

Performance is bounded by computational performance or memory performance [10]. Performance is bounded by memory performance when operational intensity is 0.17 (= 5.80/34.13) or lower without parallelization, 0.68 (= 23.20/34.13) or lower when using AVX2 or OpenMP, and 2.72 (= 92.80/34.13) or lower when using both AVX2 and OpenMP. When operational intensity is higher than those values, performance is bounded by computational performance.

4. Experiment in DD Arithmetic for Matrix and Vector Operations

4.1. DD Vector Operations

In vector operations, the four elements of a vector are processed simultaneously using AVX2. When using OpenMP, different parts of the vector are processed by each thread. When we compute the inner product with AVX2, we must sum up the four SIMD register elements, with requires three scalar additions. In the case of using OpenMP, since we must sum up the p thread elements, $p-1$ scalar additions are needed. When using both AVX2 and OpenMP, each thread computes a partial sum with a vector of length N/p using AVX2. Then, these partial sums are converted to a global sum.

Table 1 shows the operational intensity (O. I.) and the experimental results of vector operations when the dimension is 4,096,000. In many vector operations, the upper

bound is calculated by the product of memory performance and its operational intensity. According to Table 1, the performances of four operations ($\boldsymbol{y}=\alpha\boldsymbol{x}$, $\boldsymbol{z}=\boldsymbol{x}+\boldsymbol{y}$, $\boldsymbol{z}=\boldsymbol{x}+\boldsymbol{x}$, and $\boldsymbol{z}=\alpha\boldsymbol{x}+\boldsymbol{y}$) with a new vector variable on the left is nearly 20% of the upper bound of performance when using both AVX2 and OpenMP. However, the performances of six operations ($\boldsymbol{x}=\alpha\boldsymbol{x}$, $\boldsymbol{y}=\boldsymbol{x}+\boldsymbol{y}$, $\boldsymbol{x}=\boldsymbol{x}+\boldsymbol{x}$, $\boldsymbol{y}=\alpha\boldsymbol{x}+\boldsymbol{y}$, $\alpha=\boldsymbol{x}^T\boldsymbol{y}$, and $\beta=\boldsymbol{x}^T\boldsymbol{x}$) with no new vector variables on the left are nearly 70% of the upper bound of performance when using both AVX2 and OpenMP. Since the four operations $\boldsymbol{y}=\alpha\boldsymbol{x}$, $\boldsymbol{z}=\boldsymbol{x}+\boldsymbol{y}$, $\boldsymbol{z}=\boldsymbol{x}+\boldsymbol{x}$, and $\boldsymbol{z}=\alpha\boldsymbol{x}+\boldsymbol{y}$ require memory allocation, it is difficult to achieve high performance by parallelization. The other six operations do not have the overhead of allocating memory. Figure 2 shows the performances of $\alpha\boldsymbol{x}$, $\boldsymbol{x}+\boldsymbol{y}$, and $\alpha\boldsymbol{x}+\boldsymbol{y}$ with and without the overhead of allocating memory. As N becomes larger, the differences in performance increase between with and without overhead.

Table 1. Number of double-precision floating-point operations [flops], number of memory references [bytes], operational intensity [flops/bytes], memory requirement, and performances [Gflops/sec] for DD vector operations for N = 4,096,000.

	Flops	Memory references	O. I.	Memory requirement	Serial	AVX2	OpenMP	AVX2& OpenMP
$\boldsymbol{y}=\alpha\boldsymbol{x}$	$7N$	$2N\times16$	0.22	$2N$	1.19	1.51	1.30	1.25
$\boldsymbol{x}=\alpha\boldsymbol{x}$	$7N$	$2N\times16$	0.22	N	2.05	4.78	5.03	5.21
$\boldsymbol{z}=\boldsymbol{x}+\boldsymbol{y}$	$11N$	$3N\times16$	0.23	$3N$	1.96	2.05	2.15	1.67
$\boldsymbol{y}=\boldsymbol{x}+\boldsymbol{y}$	$11N$	$3N\times16$	0.23	$2N$	4.10	5.18	5.56	5.43
$\boldsymbol{z}=\boldsymbol{x}+\boldsymbol{x}$	$11N$	$2N\times16$	0.34	$2N$	2.25	2.25	2.15	2.25
$\boldsymbol{x}=\boldsymbol{x}+\boldsymbol{x}$	$11N$	$2N\times16$	0.34	N	4.10	8.19	7.51	8.34
$\boldsymbol{z}=\alpha\boldsymbol{x}+\boldsymbol{y}$	$18N$	$3N\times16$	0.38	$3N$	2.23	3.21	3.35	2.73
$\boldsymbol{y}=\alpha\boldsymbol{x}+\boldsymbol{y}$	$18N$	$3N\times16$	0.38	$2N$	2.84	7.37	8.57	8.78
$\alpha=\boldsymbol{x}^T\boldsymbol{y}$	$18N$	$2N\times16$	0.56	$2N$	2.30	7.76	8.19	14.18
$\beta=\boldsymbol{x}^T\boldsymbol{x}$	$18N$	$N\times16$	1.13	N	2.30	8.67	8.38	26.33

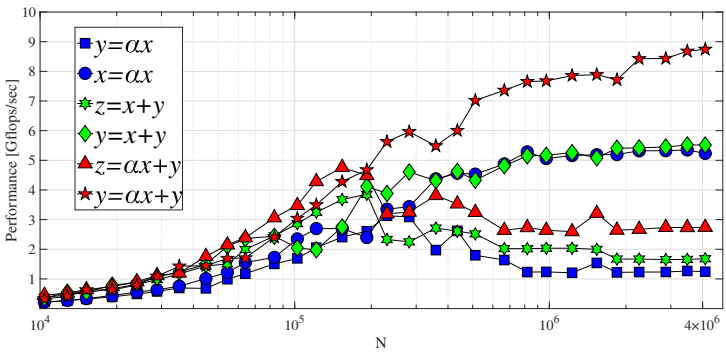


Figure 2. Performances [Gflops/sec] using AVX2 and OpenMP for $\alpha\boldsymbol{x}$, $\boldsymbol{x}+\boldsymbol{y}$, and $\alpha\boldsymbol{x}+\boldsymbol{y}$.

For all ten operations, a higher operational intensity results in a higher performance. For the same operational intensity, performance is higher for a smaller memory require-

ment. It is also important for high performance to reduce the number of memory references and the memory requirements.

In summary, for vector operations, when memory allocation overhead is not required, the performances are almost 70% of the upper bound by using AVX2 and OpenMP. Otherwise, the performances are degraded to 20%.

4.2. DD Matrix-Vector Multiplication

Matrix-vector multiplication $\boldsymbol{y} = A\boldsymbol{x}$ is written as $y_i = \sum a_{ij}x_j$. The operational intensity of matrix-vector multiplication is 1.13, because the number of double-precision floating-point operations is $7N^2 + 11N(N-1)$ and the number of memory references is $(N^2 + 2N) \times 16$, and this operation is limited by memory performance when using both AVX2 and OpenMP. Matrix-vector multiplication has two algorithms, PB and PDOT. The memory references for the matrix A are column order in PB, and those for matrix A are row order in PDOT. Since MATLAB stores data column-wise, the memory references for PB are continuous. Unit stride access can be used for PB.

4.2.1. AVX2

There are four algorithms using AVX2 shown in Figure 3. The order to load elements of matrix A is by column in PDOT_{PB} and PB_{PB}, and by row in $\text{PDOT}_{\text{PDOT}}$ and PB_{PDOT}. Here, a prefix v indicates a vector and the variables are DD numbers. The variables va, vx, and vy hold the four DD numbers. The $vload$ instruction loads four continuous data as $a(i,j)$ to $a(i+3,j)$ or $x(j)$ to $x(j+3)$. The $vmuladd(vy,va,vx)$ instruction performs the multiply-add operation $vy = vy + va * vx$ to compute four elements simultaneously and costs 18 double-precision floating-point operations. The $vmul(va,vx)$ instruction is the multiplication operation $vy = va * vx$ and costs 7 double-precision floating-point operations. The $sum(vy)$ instruction sums the data in the SIMD register and costs 11×3 double-precision floating-point operations.

$\text{PDOT}_{\text{PDOT}}$

```
for (i = 0; i < n; i++)
    vy = {0,0,0,0}     // setzero
  for(j = 0; j < n; j += 4)
                       // load 4 discontinuous data
    va = set(a(i,j), ..., a(i,j+3))
    vx = vload(x(j))   // load 4 continuous data
    vy = vmuladd(vy,va,vx) // vy = vy + va * vx
  y(i) = sum(vy) // sum 4 elements in SIMD register
```

PB_{PDOT}

```
y = setzeros(n,1) // set n×1 vector as zero
for (j = 0; j < n; j += 4)
  vx = vload(x(j))
  for(i = 0; i < n; i++)
    va = set(a(i,j), ..., a(i,j+3))
    vy = vmul(va,vx)  // vy = va * vx
    y(i) = y(i) + sum(vy)
```

PDOT_{PB}

```
for (i = 0; i < n; i += 4)
  vy = {0,0,0,0}
  for(j = 0; j < n; j++)
    va = vload(a(i,j))
              // load 1 element and fill SIMD register
    vx = broadcast(x(j))
    vy = vmuladd(vy,va,vx)
    vstore(y(i),vy)      // store 4 continuous data
```

PB_{PB}

```
y = setzeros(n,1)
for (j = 0; j < n; j++)
  vx = broadcast(x(j))
  for(i = 0; i < n; i += 4)
    va = vload(a(i,j))
    vy = vload(y(i))
    vy = vmuladd(vy,va,vx)
    vstore(y(i),vy)
```

Figure 3. Algorithms using AVX2 for $\boldsymbol{y} = A\boldsymbol{x}$.

Table 2. Instructions and performances [Gflops/sec] for $\boldsymbol{y} = A\boldsymbol{x}$ for N=2,500.

	Computation	Load	Store	Serial	AVX2	OpenMP	AVX2& OpenMP
PDOT	N^2 *muladd*	$2N^2$ *load*	*N store*	1.25	-	4.96	-
PB	N^2 *muladd*	*N load* $2N^2$ *load*	N^2 *store*	3.55	-	12.77	-
$\text{PDOT}_{\text{PDOT}}$	$N^2/4$ *vmuladd* *N sum*	*N setzero* N^2 *load* $N^2/4$ *vload*	*N store*	-	1.64	-	6.11
PDOT_{PB}	$N^2/4$ *vmuladd*	$N/4$ *setzero* $N^2/4$ *vload* $N^2/4$ *broadcast*	$N/4$ *vstore*	-	4.96	-	10.58
PB_{PDOT}	$N^2/4$ *vmul* $N^2/4$ *sum* $N^2/4$ *add*	$N/4$ *vload* $5N^2/4$ *load*	$N^2/4$ *store*	-	3.89	-	14.72
PB_{PB}	$N^2/4$ *vmuladd*	*N broadcast* $N^2/2$ *vload*	$N^2/4$ *vstore*	-	11.97	-	25.63

It is clear from comparing PDOT with PB in Table 2 that unit stride access is required to achieve high performance. Since the performance for PB_{PB} (11.97) is increased almost four-fold from that shown for PB (3.35), and that for PDOT_{PB} (4.96) is also increased four-fold from the PDOT value (1.25) shown in Table 2, it is clearly important to use the *vmuladd* instruction instead of *muladd* in order to increase performance. To improve performance with the *vmuladd* instruction, the *vload* and *vstore* instructions must also be used.

4.2.2. AVX2 and OpenMP

The performance of PB_{PB} using AVX2 in the innermost loop was the highest as shown in Section 4.2.1. Since we assume that the innermost loop i is parallelized by AVX2, we parallelize the outer loop by OpenMP.

As in Table 2, when using AVX2 and OpenMP, the performances of PB_{PB} (11.97→25.63) and PDOT_{PB} (4.96→10.58), which have improved performance with AVX2, are only twice as high as the case of just AVX2. By using AVX2 and OpenMP, the performances of $\text{PDOT}_{\text{PDOT}}$ (1.64→6.11) and PB_{PDOT} (3.89→14.72), which did not improve much with AVX2, is nearly four times higher than the case of just AVX2. Since the OpenMP can parallelize and accelerate regardless of the order of memory references, the performances of $\text{PDOT}_{\text{PDOT}}$ and PB_{PDOT} can be increased by using OpenMP. The performance of PB_{PB} is the highest for using AVX2 and OpenMP, 25.63 Gflops/sec, and it is almost 7 times faster than before parallelization.

As shown in Table 2, even using both AVX2 and OpenMP, the execution time cannot be reduced 16-fold (AVX2 $\times$ four cores) compared to without parallelization. Since the operational intensity of matrix-vector multiplication is 1.13, which is lower than 2.72, matrix-vector multiplication is limited by memory performance when using AVX2 and OpenMP. Since the upper bound of performance is 38.53 Gflops/sec, which is calculated using operational intensity $\times$ memory performance, the performance 25.63 Gflops/sec is 67% of the upper bound of performance.

As for DD matrix-vector multiplication, when using the AVX2 *vmuladd*, *vload* and *vstore* instructions with OpenMP applied to the outer loop, the performance can reach 67% of the upper bound.

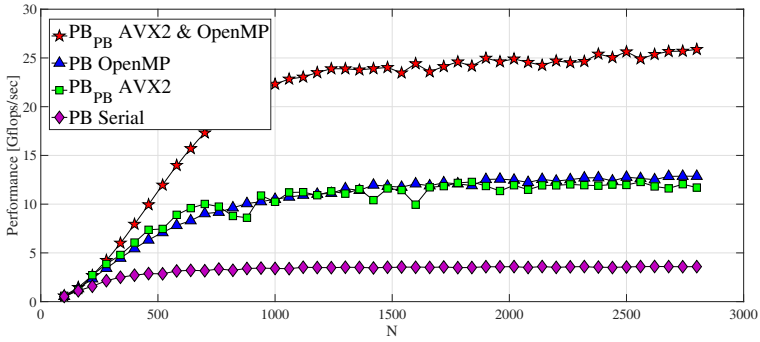


Figure 4. Performances [Gflops/sec] for Serial, AVX2 or OpenMP, and AVX2 and OpenMP for $\boldsymbol{y} = A\boldsymbol{x}$.

4.3. DD Matrix-Matrix Multiplication

Matrix multiplication $C = AB$ is written as $c_{ij} = \sum a_{ik} b_{kj}$ with three nested loops i, j, and k. Since operational intensity for matrix multiplication is quite high, at $O(N) = 18N^3/(3N^2 \times 16)$, this operation is limited by computational performance. As we have seen in Section 4.2, unit stride access is essential to achieve high performance. If the innermost processed loop is the k or j loop, then unit stride access cannot be performed, because MATLAB stores data in column-wise order. Since the index of the innermost loop for matrix multiplication must be i, there are two implementation algorithms: MP and PDOT. MP uses j-k-i order and PDOT uses k-j-i order for the loops.

AVX2 is easily applied to the loops in both algorithms, MP and PDOT. In order to parallelize using *vload* and *vstore* instructions, the loop of index i should be processed as a vector, in which case, its performance increases almost four-fold, as shown in Table 3. One of the remaining loops, with index j or k, will be parallelized by OpenMP. Figure 5 shows the four algorithms according to which loops are parallelized.

Table 3. Instructions and performances [Gflops/sec] for $C = AB$ for N = 2,500.

	Additional Instructions	Serial	AVX2	OpenMP block	OpenMP cyclic	AVX2& OpenMP block	AVX2& OpenMP cyclic
MP		3.60	13.73	13.04	12.88	46.71	45.58
MP$_{PB}$	N^2 *load*/*store* N^2 *sum*	-	-	13.04	13.08	29.46	29.36
PDOT		3.67	12.25	12.79	12.62	15.06	15.02
PDOT$_{PDOT}$	N *load*/*store* N *sum*	-	-	7.55	7.76	8.46	8.54

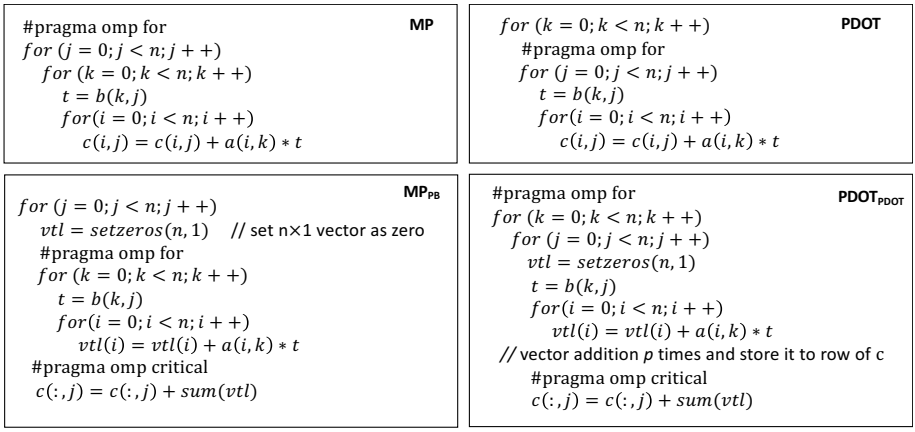


Figure 5. Algorithms using OpenMP for $C = AB$.

MP and PDOT are easily parallelized with putting the "#*pragma omp for*" directive above an intended *for* statement, as shown in Figure 5. All the threads in $\mathrm{MP_{PB}}$ and $\mathrm{PDOT_{PDOT}}$ can potentially process and update the same data location in parallel. To avoid this problem, in each thread, we defined thread-local vector *vtl* for holding a partial sum as a private variable. Thread-local vector *vtl* requires memory equal to the product of $N \times 16$ and the number of threads. Accumulation to a global sum from each partial sum is processed serially by inserting a "#*pragma omp critical*" directive. Each $sum(vtl)$ in Figure 5 costs $11pN$ double-precision floating-point operations, where p denotes the number of threads. In Figure 5, $c(:, j)$ denotes the j-th column of array c. MP and PDOT can be processed in serial without OpenMP. However, executing $\mathrm{MP_{PB}}$ and $\mathrm{PDOT_{PDOT}}$ requires using OpenMP.

In the case of OpenMP, as shown in Table 3, the performances for MP and PDOT are almost 13 Gflops/sec, or about four times higher than without parallelization, but that for $\mathrm{PDOT_{PDOT}}$ is about 8 Gflops/sec, which is lower than other cases. There is almost no difference between block and cyclic scheduling. $\mathrm{PDOT_{PDOT}}$ needs one $sum(vtl)$ for each innermost loop, total cost of $sum(vtl)$ becomes $11pN^3$ double-precision floating-point operations. $\mathrm{MP_{PB}}$ also needs $sum(vtl)$, but its total cost is $11pN^2$ double-precision floating-point operations because of once for each nested loop. Since DD Matrix multiplication requires $18N^3$ double-precision floating-point operations, the overhead for $\mathrm{PDOT_{PDOT}}$ is extremely large and greater than original computations. Since the additional overhead for parallelization in MP is much less than that for $\mathrm{MP_{PB}}$, as shown in Figure 5, the performance for MP was higher than that of $\mathrm{MP_{PB}}$, as shown in Table 3. It is clear that less additional overhead for parallelization is required for high performance.

When using both AVX2 and OpenMP, there is a difference in performance between PDOT and MP for large N, as shown in Figure 6. Since the all elements of c_{ij} are updated N times in PDOT, but a column of c_{ij} are updated N times in MP. The data locality of MP is higher than that of PDOT. Thus, the performance of MP is higher than that of PDOT.

Since operational intensity for matrix multiplication is larger than 2.72, the upper bound of performance for matrix multiplication is 92.80 Gflops/sec. The DD matrix multiplication has a 16-fold increase, because this operation is limited by computational per-

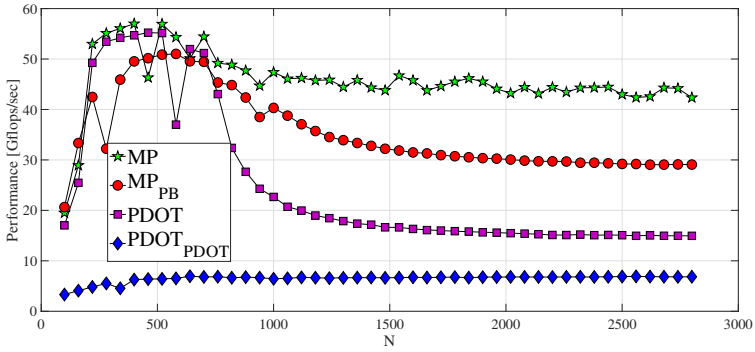


Figure 6. Performances [Gflops/sec] using AVX2 and OpenMP in block scheduling for $C = AB$.

formance. Although MP demonstrates the best performance 46.71 Gflops/sec, 50% of the upper bound of performance and 13-fold higher than without parallelization (46.71/3.60).

When operational intensity is high and the outer loop is parallelized by using OpenMP with less additional overhead, the operation is much accelerated. Thus, in order to use both AVX2 and OpenMP, it is important to vectorize the innermost loop by using AVX2 and parallelize outer loops by using OpenMP while avoiding the same memory location being updated by different threads.

Table 4. Execution time [sec] in double-precision and DD precision. N=4,096,000 for vector operations, and N=2,500 for matrix-vector multiplication.

	$\boldsymbol{x} = \alpha\boldsymbol{x}$	$\boldsymbol{y} = \boldsymbol{x} + \boldsymbol{y}$	$\boldsymbol{y} = \alpha\boldsymbol{x} + \boldsymbol{y}$	$\alpha = \boldsymbol{x}^T\boldsymbol{y}$	$\boldsymbol{y} = A\boldsymbol{x}$
Double	0.0028	0.0042	0.0042	0.0027	0.0022
DD (AVX2 & OpenMP)	0.0055	0.0083	0.0084	0.0052	0.0044
DD / Double	1.96	1.98	2.00	1.93	2.00

5. Conclusion

In response to demands for ways to facilitate high-precision arithmetic with an interactive computing environment, we developed MuPAT on Scilab/MATLAB. MuPAT uses DD and QD arithmetics that require large numbers of double-precision floating-point operations. Executing DD arithmetic operations takes 10 to 30 times the execution time of double-precision floating-point operations, due to the heavy computation load and the need to maintain computation order.

We utilized computation offloading to call an outer C function with the MATLAB executable file, and parallelized the computation by using AVX2 and OpenMP. By using a C executable with MATLAB, the code becomes platform dependent, but we intended to achieve fast computation using data-level parallelism such as with AVX2 and OpenMP instead of platform independence. We used an FMA (Fused Multiply-Add) based algorithm to reduce rounding errors.

AVX2 (Advanced Vector Extensions 2) executes operations on four double-precision numbers simultaneously. To achieve high performance, it requires that vector-

ized fused multiply-add operations and load/store instructions be used. Unit stride access is essential for using vector load/store instructions. OpenMP enables the same operations to process different data within threads on different cores. To avoid the same data location from being accessed by different threads, we should apply OpenMP to as large a block as possible.

Thus, in order to use both AVX2 and OpenMP, it is important to vectorize the innermost loop by using AVX2 and parallelize outer loops by using OpenMP while avoiding the same memory location being updated by different threads. By utilizing both AVX2 and OpenMP, the performance of the matrix-vector multiplications became 25.63 Gflops/sec (67% of the upper bound), and the performance of matrix multiplication became 46.71 Gflops/sec (50% of the upper bound). Each DD arithmetic operation requires 10 to 30 double-precision floating-point operations, however the execution time of these DD operations for vector operations and matrix-vector multiplication in parallel processing became only about twice that of the original double-precision floating-point operations without parallel processing.

The performance of these DD operations is bounded by memory performance. It is possible to compute many more operations in the same time if no additional data are required. The execution times of $\boldsymbol{y} = \boldsymbol{x} + \boldsymbol{y}$ and $\boldsymbol{y} = \alpha\boldsymbol{x} + \boldsymbol{y}$ are the same. These facts mean that parallel processing provides us more accurate results and/or processes a much larger workload for during the same time without an extra cost.

Acknowledgment

The authors would like to thank the reviewers for their valuable comments. This research was supported by a grant from the Japan Society for the Promotion of Science (JSPS: JP17K00164).

References

[1] T. Saito, E. Ishiwata, and H. Hasegawa, Analysis of the GCR method with mixed precision arithmetic using QuPAT, *Journal of Computational Science* **3** (2012), 87-91.

[2] H. Waki, M. Nakata, and M. Muramatsu, Strange behaviors of interior point methods for solving semidefinite programming problems in polynomial optimization, *Computational Optimization and Applications* **53** (2012), 823-844.

[3] Y. Hida, X. S. Li, and D. H. Baily, QD arithmetic: algorithms, implementation, and application, Technical Report LBNL-46996, Lawrence Berkeley National Laboratory, 2000.

[4] S. Kikkawa, T. Saito, E. Ishiwata, and H. Hasegawa, Development and acceleration of multiple precision arithmetic toolbox MuPAT for Scilab, *JSIAM Letters* **5** (2013), 9-12.

[5] MuPAT on MATLAB. `https://www.ed.tus.ac.jp/1419521/index.html`

[6] T. J. Dekker, A floating-point technique for extending the available precision, *Numerische Mathematik* **18** (1971), 224-242.

[7] J.- M. Muller, et al, Handbook of Floating-Point Arithmetic, *Birkhäuser*, 2010.

[8] Intel Intrinsics Guide. `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`

[9] P. S. Pacheco, An Introduction to Parallel Programming, *Morgan Kaufmann*, 2011.

[10] S. Williams, A. Waterman, and D. Patterson, Roofline: An insightful visual performance model for multicore architectures, *Communications of the ACM* **52** (2009), 65-76.

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200070

Dynamic Runtime and Energy Optimization for Power-Capped HPC Applications

Bo WANG [a], Christian TERBOVEN [a] and Matthias MÜLLER [a]
[a] *IT Center at RWTH Aachen University, Germany*

Abstract. Future large-scale high-performance computing clusters will face a power wall where the peak power draw of these clusters exceeds the maximal power-supplying capability of the surrounding infrastructure. To use the limited power budget efficiently, we developed a dynamic strategy to tackle execution time imbalance issues through power shifting and frequency limitation. By applying this strategy to NPB OpenMP benchmarks, we succeed in a continuous enforcement of power draw under a specified power cap. At the same time, execution time is reduced by up to 12.8% and the energy to solution is reduced by up to 12.3%, compared to a native power strategy.

Keywords. power capping, performance optimization, energy reduction, power shifting, core frequency limitation

1. Introduction

Large-scale high-performance computing (HPC) clusters face a power wall where their peak power draw exceeds the power supplying capacity of the surrounding infrastructure [1]. The power draw of these clusters has to be limited in order to avoid hardware damage. Under this constraint, utilizing the power budget efficiently and minimizing the execution time of running jobs are required in order to improve the clusters' job throughput.

Execution of a parallel job suffers frequently from imbalance among parallel tasks. In this work, a parallel task can be an MPI process or an OpenMP thread. Each task may reach a global synchronization call, like barrier, with distinct delays. The imbalance can be caused by inherited load imbalance of the job, but can also be caused by power capping at runtime.

Due to variations at manufacturing, processors have distinct power efficiencies, defined as the number of watts needed to execute certain operations. Enforcing a power cap on these processors causes distinct performance loss, like floating-point operation rate [FLOP/s]. The loss happens because of the different amounts of power need be cut to remain under the power cap[2]. Therefore, power-capping causes an additional runtime execution imbalance.

Because of diverse factors causing an execution imbalance, it is difficult to analyze and handle the imbalance issue before the execution. In this work, we developed a dynamic power and frequency management (DPF) method to detect, analyze and handle

runtime imbalances issue. Applying DPF to NPB OpenMP benchmarks, we achieved up to 12.8% performance improvement and saved up to 12.3% energy compared to a native strategy.

The remainder of the paper is structured as follows: in Section 2, we describe related work briefly. In section 3, we illustrate and analyze execution imbalance issues in detail. Based upon the imbalance observations, we introduce the DPF method to tackle these issues in section 4. In the subsequent Section, we evaluate DPF with NPB benchmarks in comparison with a native management strategy. In the last section, we conclude this work.

2. Related work

Several large scale clusters are facing a power wall issue and several works had been contributed to investigate and optimize performance of power-capped applications and clusters[3][2][4][5][6][7].

Routree et al. [2] observed that power capping causes performance imbalance among processors of the same model since processors have distinct power efficiencies. This kind of imbalance causes execution time imbalance of parallel tasks.

The authors of [6] and [7] introduced a static power budgeting method to handle the imbalance issue. They measured and documented the processors' power efficiencies. Then they increased the power budget of less power-efficient hardware to improve the overall performance. However, the static methods are limited in scalability and usability since each processor needs to be measured and characterized accordingly.

[4] and [5] introduced dynamic methods to tackle the disadvantages of the static methods separately. Marathe et al. [5] archived performance optimization through fine-grained management of power budget, thread concurrency, and core clock rate. Gholkar et al. [4] improved performance through careful power shifting. Both works require online power and frequency monitoring.

Our DPF implementation differs from the other dynamic methods[4] and [5]. DPF assumes the power budget for a job is dynamically adjustable at runtime [8]; DPF avoids hardware monitoring as far as possible; DPF manages hardware with limitations instead of direct manipulation on power or frequency.

3. Platform and Power Capping

In this section, we present the hardware platform where our measurements were conducted at first. We present power values and performance characteristics of the processors. In the end, we illustrate scenes of a power-capped cluster where this work contributes to.

3.1. Hardware Platform and Software

Measurements in the following sections were conducted on a computing node which is chosen randomly from the CLAIX-2016 system at RWTH Aachen university. This node possesses two processors of Intel Xeon E5-2650 V4. Each processor has 12 cores

with hyper-threading deactivated, attached with 64 GB DRAM. The processors can be clocked up to 2.5 GHz with activated Turbo Boost.

The thermal designed power (TDP) of each processor is 105 watts while the power can be throttled down to 53 watts according to the runtime average power limitation (RAPL) setting[9]. RAPL is a technology introduced by Intel which enables power measurements. The power values provided by RAPL are well verified in many aspects [10],[11],[12],[13]. On the other hand, RAPL enables power capping where a user can specify a power value and a time window. RAPL enforces that the average power draw of the time window remains under the specified power value, regardless of what kind of operations are being performed.

RAPL manages power of a processor in three domains, the PKG, PP0 and DRAM domain[1] [14]. The PKG domain is in charge of power management of core and uncore area where arithmetic logic units (ALU) and last-level cache are located respectively. The DRAM domains manages power of DRAM. In this work, we only adjust the PKG domain since it has a high power draw compared to the DRAM domain. In addition, power capping the DRAM domain decreases the execution performance seriously since memory bandwidth is the performance bottleneck for many scientific-technical applications.

Compared to the per-processor power capping through RAPL, each core of the processors can be adjusted independently because of the intergrated dynamic voltrage and frequency scaling (DVFS), and fully integrated voltage regulator (FIVIR)[15] technologies. In this work, we limit the maximal frequency using Linux */sys/devices/system/cpu/cpui/cpufreq/scaling_max_freq* interface instead of setting a concrete frequency. Through the frequency limitation, hardware is allowed to choose a concrete frequency to meet a power cap automatically and flexibly.

We employ the NPB OpenMP [16] benchmarks of the size C and a home-grown synthetic benchmark (this benchmark will be introduced in the Section 5.1). The benchmarks in our measurements always occupy all available cores with parallel tasks, i.e., 24 threads.

3.2. Power Efficiency Variation of Hardware

Due to manufacturing variations, processors of the same product line can have diverse power efficiencies. We define power efficiency as the power draw of a processor performing certain operations: the higher the power draw, the less efficient the processor is. Figure 1 illustrates power efficiencies of the two processors of our testbed. The power draw of each benchmark differs. On processor 0, *ft* draws more than 90 watts while *is* consumes 61 watts in average. In particular, processor 0 is about 10% less power efficient than processor 1 (processor 0 consumes 8 watts more than processor 1 in average.).

We observed a similar power variation in many other RWTH HPC cluster's nodes with different degree of variations.

3.3. Performance Variation under Power Capping

The enforcement of a power cap causes performance degradation, i.e. applications will perform slowly. We measured relative execution time extension of NPB benchmarks

[1]The PP0 domain is not supported on our platform.

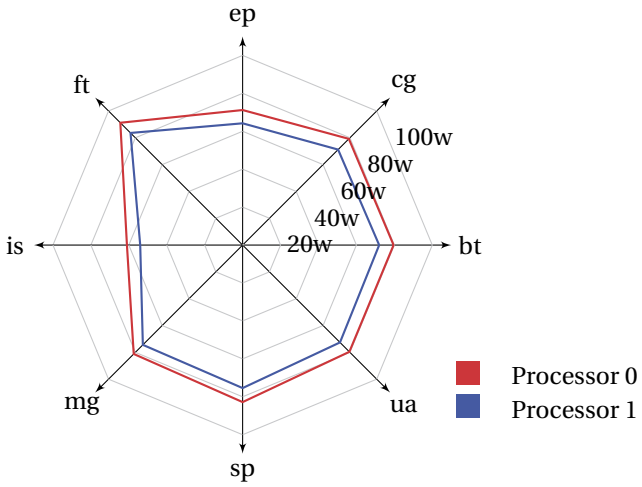


Figure 1. Power draw of two processors

capped at 53 watts compared to executions at TDP. Measurements were conducted on sockets one after another. Results are illustrated in Figure 2.

The amount of performance degradation depends on many factors. It depends on applications' peak power draw. The higher the peak power is, the more power needs to be capped and the more performance will be throttled. Since *ft* has a higher power draw than *is* illustrated in Figure 1, *ft*'s performance degradation is greater than *is*, as illustrated in Figure 2.

The degradation also depends on the performed operations by the applications, like memory-bound or compute-bound operations. *bt* and *sp* have a similar peak power draw in Figure 1, but the performance degradation differs a lot (12% vs. 24% on processor 0) since the measured L3 cache miss rates of *bt* and *sp* are quite different(2E-4 and 1.2E-3 misses per second).

In particular, the degradation depends on the power efficiency of the underlying processors. On a power-inefficient processor, more power needs to be capped compared to on a power-efficient processor. The frequency of the inefficient processor is throttled greatly and the performance degradation is tendentiously high, as illustrated by Figure 2. Processor 0 has more performance loss than processor 1 for all benchmarks.

3.4. Dynamic Power Budgeting on a Power-capped Cluster

Because of infrastructure limitations, power supplying can be insufficient for a cluster running at its peak power draw. On such a cluster, power budgeting methods were developed [8][17] to accelerate executions as far as possible. The methods schedule power to jobs dynamically and according to jobs' states. For an individual job, power budget varies from time to time. In this context, the DPF method needs to track the jobs' power budget and enforces the average power draw under the current budget.

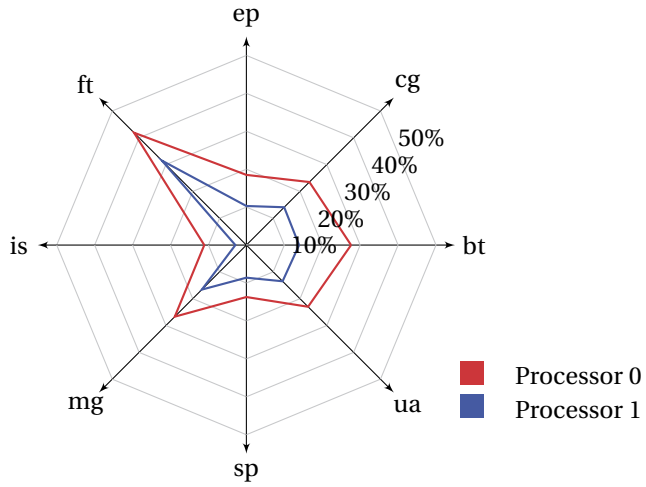


Figure 2. Relative time extension or performance degradation of executions power capped to 53 watts compared to capped to 105 watts

4. Dynamic Resource Management to Eliminate Imbalance

An execution of a parallel job on a power-capped cluster may suffer from an imbalanced execution time among parallel tasks. In this section, we justify and present our dynamic algorithm to tackle the imbalance issue and improve the jobs' performance.

4.1. Why Dynamic Management?

An imbalanced execution of a job can be caused by many factors. The job may have inherited load imbalance, i.e. parallel tasks obtain uneven loads. The load imbalance is individual to an application even to an input dataset.

In addition, the execution imbalance can also be caused by varied performance of power-capped processors as illustrated is Figure 2. The degree of imbalance is determined by the processors and the power cap.

Because of individual imbalances and runtime factors, a static analysis that predicts and handles imbalance before an execution is complex and imprecise. In contrast, a dynamic runtime imbalance tracking and handling are more promising. Assuming that a job consists of iterative executions of parallel regions and the imbalance remains constant among iterations, we track the imbalance of the previous region, calculate and distribute resources in a way to eliminate the imbalance for the next iteration. This dynamic algorithm is a light-weight solution since it does not require any prerequisite knowledge of the hardware or the software.

4.2. Load Imbalance Detection and Elimination

Normally, an HPC application executes one or multiple parallel regions iteratively. A parallel region is defined as a section of execution between two global synchronization calls, such as barriers. The entire execution time T_{app} can be calculated as a sum:

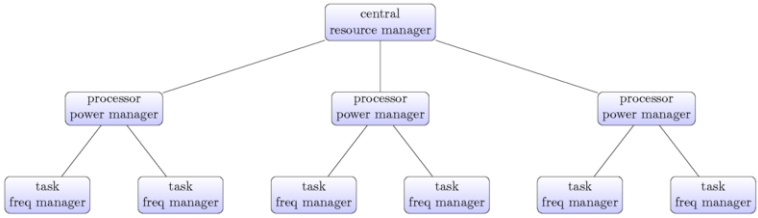


Figure 3. Two-level resource management

$$T_{app} = \sum T_r \tag{1}$$

T_r is the execution time of a parallel region r. Under the assumption that synchronization time is negligible, T_r is determined as

$$T_r = max(T_r^t), \forall t \in parallel\ tasks \tag{2}$$

namely by the slowest (critical) task.

T_{app} can be reduced by accelerating the critical task of each parallel region r through a high power budget or a high core frequency. Since a power-capped application has a limited power budget, the required power budget needs to be moved or shifted from other tasks. If the overall power cap remained and the execution is accelerated, the job's energy consumption will be reduced.

4.3. Power Shifting and Frequency Limitation to Minimize Execution Time

We developed a two-level resource management approach, the DPF, to minimize T_{app}, illustrated by a tree in Figure 3. The tree design is constructed according to the available resource management technologies, RAPL and DVFS. On top of the tree a central resource manager monitors available power-budget for the job. In the middle, a processor manager manages its own power draw using RAPL. At the bottom, a core manager manages its own frequency through DVFS.

The managers are integrated into each parallel task which are bound to physical cores. In general, the task with ID 0 of a job is the *central resource manager*. A single task on each processor manages the processor's power. Besides, each task is a core *freq manager*.

The computation time T^p of a processor within the parallel region r is defined as the maximum computation time of parallel tasks executing on the processor in Eq. (3).

$$T_r^p = max(T_r^j), \forall j\ runs\ on\ processor\ p \tag{3}$$

A critical processor $\overline{p}$ is the processor with the maximal T_r^p. Executions on $\overline{p}$ can be accelerated through enlarging its power budget PB. The required power enlargement FP (free power) needs be collected from other sockets through Eq. (4).

$$FP = \sum (SP,\ if\ T_r^p < \alpha \cdot T_r^{\overline{p}}\ and\ PB^p > MinPB,\ \forall p \in P) \tag{4}$$

SP is a predefined step power (e.g. one watt). $MinPB$ is the minimal power budget inherited in hardware to assure a stable operation. $\alpha \in (0,1)$ is a threshold that determines

when power can be shifted. α is a sensitive parameter: a high α causes corrupted resource adjustment frequently due to the tiny runtime deviation; a low α eliminates improvement potential since the resource rescheduling occurs rarely.

A critical task $\bar{t}$ within a processor is the task with the maximal execution time T_r^t. $\bar{t}$ should always run without any frequency limitation while the frequency of other tasks can be limited. In this way, more power will be allocated to the critical task. The frequency limitation is calculated using Eq. (5):

$$F_{new} = \begin{cases} 0, & \text{if } T_r^t \geq T_r^p \\ F_{current} + 1, & \text{if } T_r^t \leq \beta \cdot T_r^p \\ F_{current}, & \text{otherwise} \end{cases} \tag{5}$$

Here, the frequency setting F is similar to ACPI P-states [18] where the higher F is, the lower the actual frequency of the hardware is. β is a sensitive scaling parameter similar to α in Eq. (4). In the first case where $T_r^t \geq T_r^p$, the frequency is reset to the valid maximum. In the second case, the tasks' frequency is limited to a lower level. Otherwise, the current frequency is retained.

Through an execution, once a parallel regions r is encountered, the central resource manager checks and updates the job's current power budget. If the power budget is changed, the new budget is distributed evenly among processors. Frequency limitation of each core is reset to the valid maximum. Otherwise, power budget and frequency will be recalculated for each processor and for each core.

After a parallel region execution, each task stops its execution time. The time values are collected from *task freq managers* to the *central resource manager*. Algorithm 1 illustrate the implementation.

4.4. Implementation and Overhead

We implemented the DPF for common MPI and OpenMP jobs. To eliminate the expenditure in recognizing parallel regions, we employ OMPT [19] and PMPI [20] tools for automatic detection.

Since a parallel region can be small and the resource-scheduling time overhead compared to the execution time can be high, we introduced three techniques to eliminate the overhead:

- If the execution of an identified region is shorter than 0.01 seconds, no resource recalculation takes place.
- DPF tracks current resource settings. If the newly-calculated settings are identical to the current settings, no resource scheduling takes place.
- Hardware setting occurs locally and in parallel. Each task manages its own core frequency. A single task of each processor enforces the power cap.

In particular, the DPF only tracks time consumption. Monitoring of other hardware settings, like actual power draw and actual frequency, is unnecessary since it is irrelevant to calculate settings of the next step. Using this method, monitoring overheads can be eliminated essentially.

Algorithm 1: Resource management

```
1  Function resource_scheduling()
2      if taskID = centralManagerID then
3          calculate and distribute power budget for each processor;
4      if taskID = processorManagerID then
5          receive and set power cap;
6          calculate and distribute freq;
7      set freq;
8      start time measurement;
9  Function time_collection()
10     stop time measurement;
11     send time to processor manager;
12     if taskID = processorManagerID then
13         receive time;
14         calculate processors' critical time;
15         send processor critical time to the central manager;
16     if taskID = centralManagerID then
17         receive time;
18 Function job_execution()
19     Parallel
20         call resource_scheduling();
21         doing some operations;
22         call time_collection();
```

5. Evaluation

At the beginning, we illustrate how power and clock rate are managed for a synthetic benchmark. Then, we present the overhead introduced by DPF. At the end, we evaluate energy and execution time reduction through DPF compared to a native strategy with NPB benchmarks[2].

5.1. Dynamic Power and Clock Rate Management

As described in previous sections, the dynamic resource manager should a) enforce the power draw under a specified power cap continuously, b) react to a changed power limitation quickly and c) manage resources to eliminate execution imbalance. We verified the management process with a synthetic benchmark on the hardware platform.

The synthetic benchmark executes a parallel region in a loop. The iterations are divided into three phases. In the first phase, power cap for each processor is changed from 80 watts to 70 watts (from 160 watts to 140 watts for two processors). In the second

[2]The EP, UA and LU benchmarks are not suitable for the investigation since they only have very short regions, or a single parallel region or inconsistent execution time.

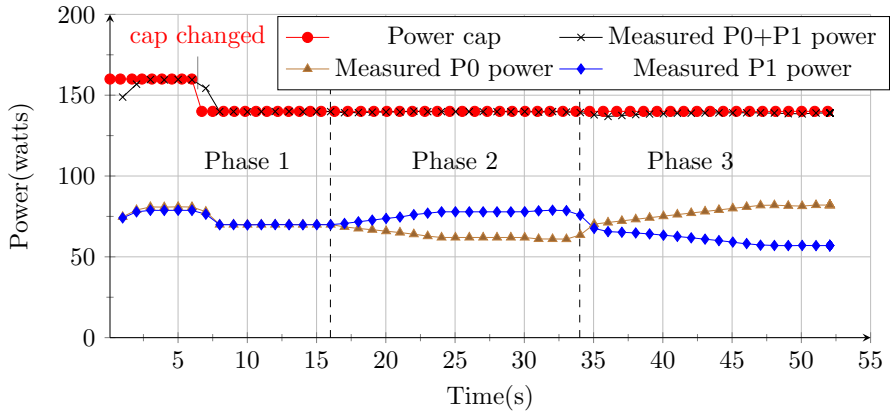


Figure 4. Run-time power monitoring

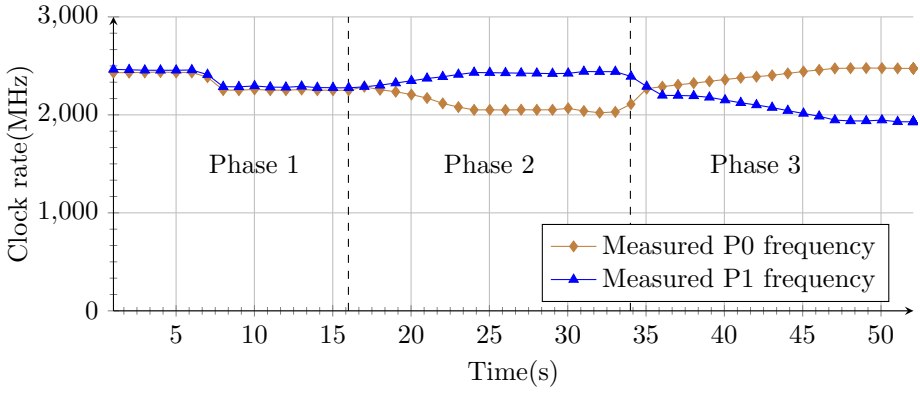


Figure 5. Run-time processor frequency monitoring

phase, tasks on processor 1 obtain 25% more load than the others. In the third phase, load of the two processors is reversed.

During the benchmark execution, we sampled the actual power draw and core frequency at 1 Hz. The results are illustrated in Figure 4 and Figure 5. As expected, the power draw remains permanently under the configured power cap, except the second after the cap is throttled from 80 to 70 watts (the measured power amounts to 77 watts). This violation is due to the interleaving of the sampling and power-adjusting points.

The measured power and frequency of phases 2 and 3 in Figures 4 and 5 illustrate that the resources are scheduled to eliminate load imbalance. In phase 2, more power is shifted to processor 1 whose cores are clocked up. In phase 3, power is shifted back to processor 0 immediately after a new load imbalance is detected.

5.2. Overhead of the Resource Management

Since the time overhead of DPF can be critical for short parallel regions, we evaluate the overhead with NPB OpenMP benchmarks.

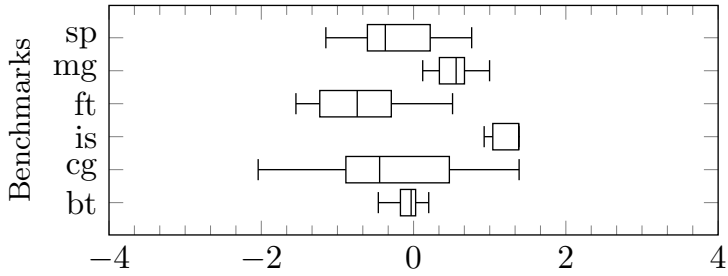


Figure 6. Measured DPF relative overheads in percent

During the measurements, power was capped to TDP (105 watts) on each processor. The actual execution is not power capped at all. However, DPF calculates power and frequency settings continuously and may set the hardware rarely.

We started each benchmark ten times with or without DPF and recorded the execution time. The overheads calculated as

$$Overhead = \frac{T_{DPF}}{T_{NODPF}} - 1$$

is illustrated in Figure 6. T_{DPF} and T_{NODPF} are the execution time with DPF and the average execution time without DPF, respectively.

The executions with DPF have some deviations up to $\pm 2\%$. Regardless of the deviations, the DPF overhead amounts to lower than 2% in the worst case, and in average lower than 1%.

5.3. Execution Time and Energy Consumption Optimization

We attached DPF to NPB OMP benchmarks of size C for validation. Capping to 55, 60, 65 and 70 watts, we measured execution time and energy consumption. Figure 7 illustrates normalized DPF time and energy compared to the values of a native strategy. The native strategy sets equal and constant power caps among processors.

For most benchmarks the achieved execution time improvements are higher than the observable deviation illustrated in Figure 6, except the *cg* at 65 watts and *sp* at 70 watts. The average improvements amount to about 4% for all power caps. In some cases, executions can be accelerated by up to 12.8%. At the same time, energy can be saved by up to 12.31%.

6. Conclusion

Because of the high power draw of an HPC cluster, the power needs be capped to avoid hardware damage in the future. However, power capping causes performance variation among processors due to their distinct power efficiencies.

In this work, we introduced a dynamic method, the DPF, that schedules power and limits frequencies to tackle the performance variation issue. DPF monitors runtime and hardware in a light-weight way. It succeeds in remaining the power draw under a specified power cap. At the same time, it accelerates executions and reduces energy consump-

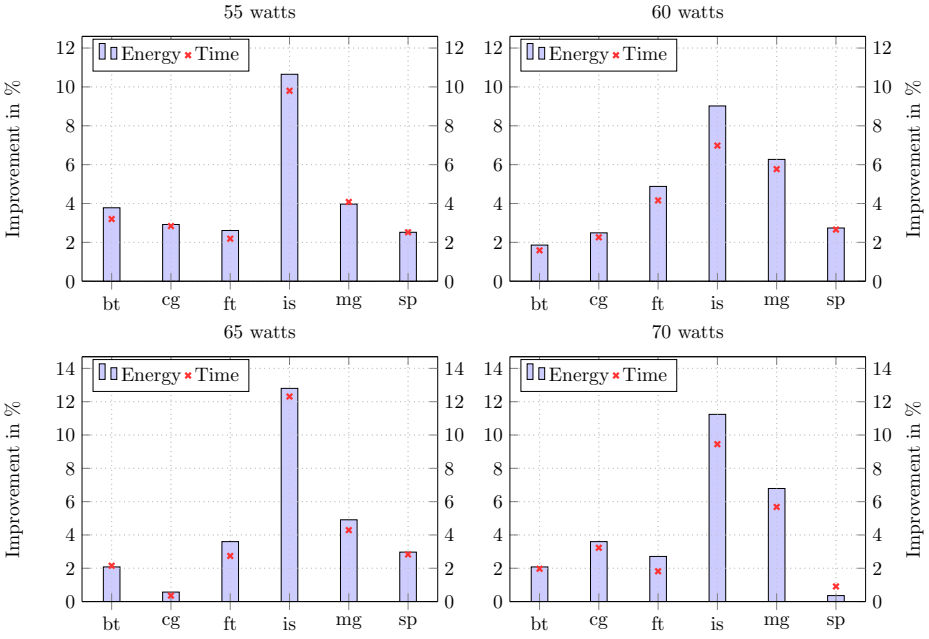


Figure 7. Improvements provided by DPF for executions with NPB OMP benchmarks

tion. For instance, applying DPF on NPB benchmarks executions were accelerated up to 12.8% and the energy consumption was reduced up to 12.3%.

In the future, we will improve the DPF for large-scale executions where hardware variation is more significant and DPF overheads need be reduced much more.

Acknowledgment

(Part of) this work was performed under the POP project and has received funding from the European Union's Horizon 2020 research and innovation programs under grant agreement 824080.

References

[1] Exascale Initiative Steering Committee. A decadal plan for providing exascale applications and technologies for doe mission needs. `https://permalink.lanl.gov/object/tr?what=info:lanl-repo/lareport/LA-UR-11-02200`, 2010.

[2] Barry Rountree, Dong H Ahn, Bronis R De Supinski, David K Lowenthal, and Martin Schulz. Beyond dvfs: A first look at performance under a hardware-enforced power bound. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 947–953. IEEE, 2012.

[3] Tapasya Patki, David K Lowenthal, Barry Rountree, Martin Schulz, and Bronis R De Supinski. Exploring hardware overprovisioning in power-constrained, high performance computing. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 173–182. ACM, 2013.

[4] Neha Gholkar, Frank Mueller, Barry Rountree, and Aniruddha Marathe. Pshifter: Feedback-based dynamic power shifting within hpc jobs for performance. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 106–117. ACM, 2018.

[5] Aniruddha Marathe, Peter E Bailey, David K Lowenthal, Barry Rountree, Martin Schulz, and Bronis R de Supinski. A run-time system for power-constrained hpc applications. In *International conference on high performance computing*, pages 394–408. Springer, 2015.

[6] Neha Gholkar, Frank Mueller, and Barry Rountree. Power tuning hpc jobs on power-constrained systems. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, pages 179–191. ACM, 2016.

[7] Yuichi Inadomi, Tapasya Patki, Koji Inoue, Mutsumi Aoyagi, Barry Rountree, Martin Schulz, David Lowenthal, Yasutaka Wada, Keiichiro Fukazawa, Masatsugu Ueda, et al. Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2015.

[8] Bo Wang, Dirk Schmidl, Christian Terboven, and Matthias S Müller. Dynamic application-aware power capping. In *Proceedings of the 5th International Workshop on Energy Efficient Supercomputing*, page 1. ACM, 2017.

[9] Howard David, Eugene Gorbatov, Ulf R Hanebutte, Rahul Khanna, and Christian Le. Rapl: memory power estimation and capping. In *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, pages 189–194. IEEE, 2010.

[10] George Stantchev, William Dorland, and Nail Gumerov. Fast parallel particle-to-grid interpolation for plasma pic simulations on the gpu. *Journal of Parallel and Distributed Computing*, 68(10):1339–1349, 2008.

[11] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. Measuring energy consumption for short code paths using rapl. *ACM SIGMETRICS Performance Evaluation Review*, 40(3):13–17, 2012.

[12] Huazhe Zhang and H Hoffman. A quantitative evaluation of the rapl power control system. *Feedback Computing*, 2015.

[13] Daniel Hackenberg, Thomas Ilsche, Robert Schöne, Daniel Molka, Maik Schmidt, and Wolfgang E Nagel. Power measurement techniques on standard compute nodes: A quantitative comparison. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 194–204. IEEE, 2013.

[14] Martin Dimitrov. Intel power governor. `https://software.intel.com/en-us/articles/intel-power-governor`, 2012. [Online; accessed 19-July-2019].

[15] Edward A. Burton, G Schrom, Fabrice Paillet, Jonathan Douglas, William J. Lambert, Kaladhar Radhakrishnan, and Michael Hill. Fivr fully integrated voltage regulators on 4th generation intel core socs. pages 432–439, 03 2014.

[16] Nas parallel benchmarks. `https://www.nas.nasa.gov/publications/npb.html`. [Online; accessed 1-July-2019].

[17] Daniel Ellsworth, Tapasya Patki, Martin Schulz, Barry Rountree, and Allen Malony. Simulating power scheduling at scale. In *Proceedings of the 5th International Workshop on Energy Efficient Supercomputing*, page 2. ACM, 2017.

[18] Acpi specification. `https://uefi.org/sites/default/files/resources/ACPI_6_2.pdf`.

[19] Alexandre Eichenberger, John Mellor-Crummey, Martin Schulz, Nawal Copty, John DelSignore, Robert Dietrich, Xu Liu, Eugene Loh, and Daniel Lorenz. Ompt and ompd: Openmp tools application programming interfaces for performance analysis and debugging. In *International Workshop on OpenMP (IWOMP 2013)*, 2013.

[20] Sava Mintchev and Vladimir Getov. Pmpi: High-level message passing in fortran77 and c. In *International Conference on High-Performance Computing and Networking*, pages 601–614. Springer, 1997.

Programming Paradigms

This page intentionally left blank

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200072

Paradigm Shift in Program Structure of Particle-in-Cell Simulations

Takayuki UMEDA [1]
Institute for Space-Earth Environmental Research, Nagoya University, Japan

Abstract. Performance measurement of the particle-in-cell (PIC) method for collisionless plasma is made on the strong scaling of the thread-level parallelism with OpenMP. The conventional program structure of the PIC method, in which a single loop statement involves an iteration through the list of particles, is compared with the new program structure, in which outer multiple loop statements involve iterations through spatial grid cells and the most inner single loop statement involves an iteration through the list of particles. The present strong scaling measurement shows that the new program structure improves both performance and scalability of the PIC code from the conventional program structure. The new code runs about three times faster than the conventional code without sorting of the list of particles.

Keywords. high performance computing, particle-in-cell method, space plasma, kinetic simulation, thread-level parallelism, performance measurement

Introduction

The Particle-In-Cell (PIC) method has now become used widely in various scientific fields. The PIC was first developed for plasma physics by geophysicists, radio scientists, and nuclear-fusion scientists. The standard numerical schemes for the PIC method for studying collisionless space plasma were established in early 1980s [1,2].

In the general PIC method, force fields are defined on grid cells, while particles move in the grid cells in accordance with the equation of motion. This is why the method is called the "particle-in-cell" method. Performance tuning of the PIC method is an issue in high-performance computing, since the PIC method solves the time development of both Eulerian variables (force fields) and Lagrangian variables (position and velocity of particles).

The force field at the position of a particles is interpolated from the force fields on neighbor grid cells. The zeroth moment (mass and charge) and the first moment (momentum and electric current) are assigned to neighbor grid cells by using the similar procedure to the interpolation of fields. In these operations, an index of a grid cell in which a particle belongs to is computed from the position of the particle. When particles are placed independently of grid cells, data access to the arrays of force fields and the arrays of moments becomes random, which causes a cache miss. The cache miss can be

[1]Corresponding Author: Institute for Space-Earth Environmental Research, Nagoya University, Nagoya 464-8601, Japan; E-mail: umeda@isee.nagoya-u.ac.jp.

suppressed and the computational performance of the PIC method can be improved by sorting of the list of particles in accordance with the order of the index of grid cells [3,4].

In the present study, a program structure used in molecular dynamic (MD) simulations and astrophysical N-body simulations is implement into the PIC method, in which each grid cell has a sorted list of particles for computing particle-particle interactions faster [5]. The program structure of the new PIC method has outer multiple loop statements and a most inner single loop statement that involve iterations through spatial grid cells and an iteration through the list of particles, respectively [6]. On the other hand, the program structure of the conventional PIC method has a single loop statement that involves an iteration through the list of particles only [1,2,3,4].

Performances of three different programs of the PIC simulations for collisionless plasma are measured. The first is the conventional program structure, in which particles are placed randomly in the simulation domain. The second is also the conventional PIC method but the index of particles is sorted in accordance with the order of the indexes of grid cells at each time step. The third program has the new program structure, in which the index of particles is sorted in accordance with the order of the indexes of grid cells at each time step. A measurement of strong scaling is performed to study how the new program structure improve the performance and scalability of the PIC method on a single compute node with the thread-level parallelism with OpenMP. Note that performances on massively parallel computers with the process-level parallelism by Message Passing Interface library (e.g., Refs.[7,8,9]) and on Graphic Processing Unit with OpenCL (e.g., Ref.[10]) are out of scope of the present study.

1. Overview of numerical schemes

The PIC method for collisionless space plasma solves the first-principle equations, i.e., the Maxwell equations for electromagnetic fields (1) and the Newton equations of motion for charged particles (2),

$$\left.\begin{aligned} \nabla\times\mathbf{B} &= \mu_0\mathbf{J}+\frac{1}{c^2}\frac{\partial\mathbf{E}}{\partial t}\\ \nabla\times\mathbf{E} &= -\frac{\partial\mathbf{B}}{\partial t}\\ \nabla\cdot\mathbf{E} &= \frac{\rho}{\epsilon_0}\\ \nabla\cdot\mathbf{B} &= 0 \end{aligned}\right\} \tag{1}$$

$$\left.\begin{aligned} \frac{\mathrm{d}\mathbf{r}_p}{\mathrm{d}t} &= \mathbf{v}_p\\ \frac{\mathrm{d}}{\mathrm{d}t}(\gamma_p\mathbf{v}_p) &= \frac{q_p}{m_p}\left\{\mathbf{E}(\mathbf{r}_p)+\mathbf{v}_p\times\mathbf{B}(\mathbf{r}_p)\right\} \end{aligned}\right\} \tag{2}$$

where $\mathbf{E}$, $\mathbf{B}$, $\mathbf{J}$, ρ, μ_0, ϵ_0 and c represent the electric field, the magnetic field, the current density, the charge density, the magnetic permeability, the dielectric constant and the speed of light, respectively. The quantities $\mathbf{r}$, $\mathbf{v}$, γ, q and m represent the position, the velocity, the Lorentz factor, the charge and the mass, with the subscript p being the p-th particle. The current density and the charge density in Eq.(1) must satisfy the continuity equation for charge,

$$\frac{\partial \rho}{\partial t} + \nabla \cdot \mathbf{J} = 0. \tag{3}$$

The continuity equation for charge (3) satisfies both of Maxwell equation (1) and Boltzmann (Vlasov) equation for particle species s,

$$\frac{\partial f_s}{\partial t} + \mathbf{v} \cdot \frac{\partial f_s}{\partial \mathbf{r}} + \frac{q_s}{m_s} (\mathbf{E} + \mathbf{v} \times \mathbf{B}) \cdot \frac{\partial f_s}{\partial \mathbf{v}} = 0, \tag{4}$$

which describes the statistical behavior of the equations of motion (2).

The Maxwell equations (1) are solved with the Finite Difference Time Domain (FDTD) method [11], in which electromagnetic fields on staggered grid cells are computed with the second-order leap-frog time-integration together with the second-order central difference. The equations of motion for charged particles (2) are solved with the second-order leap-frog time-integration based on the Buneman-Boris scheme [12,13]. The Sokolov interpolation scheme [14] is used to interpolate electromagnetic fields on neighbor grid cells to particle positions. The current density is computed with the charge conservation scheme [15] which exactly satisfies the continuity equation for charge (3).

The present PIC code is parallelized in two levels. As the first-level thread parallelism, the computation of velocity and position of particles in accordance with the equations of motion (2) and the computation of current density in accordance with the continuity equation for charge (3) are parallelized with OpenMP. The "particle decomposition" with the standard message passing interface (MPI) library is also adopted as the second-level process parallelism, in which the computation of velocity and position of particles and the computation of current density for electrons and positively-charged ions are decomposed into two processes. Note that the domain decomposition with MPI as the third-level parallelism is out of scope of the present study.

2. Program structures

The PIC code for collisionless plasma has four kernels. The first kernel updates the velocity of charged particles, as shown in Figures 1 and 3. The second kernel update the position of charged particles and computes the current density, as shown in Figures 2 and 4. The third kernel update electromagnetic fields based on the Maxwell equations. The fourth kernel sorts the index of particles. Since the load of the third kernel is generally less than 0.1% of the total load of the entire kernels, the performance of the field kernel is not of interest in the present study.

In the conventional PIC method, a single loop involving an iteration through the index of particles, `p`, is used to compute the current density, the velocity and the position of particles. Figures 1 and 2 show FORTRAN programs of the velocity and current kernels, respectively. At lines 3–5 in Figures 1 and 2, the indexes of grid cells nearest the p-th particle, `(i,j,k)` are obtained based on the position vector of the p-th particle, `(x,y,z)`. Here, `x(p)` is the x-coordinate of the p-th particle position, and `i` is used as an index in a 3D arrays, `(fx,fy,fz)` and `(cjx,cjy,cjz)`. At lines 6–9 in Figures 1 and 2, a set of weights, `(w1,w2,w3,...)`, is computed by the numerical interpolation at the position of the p-th particle.

In Figure 1, the acceleration vector of the p-th particle, `(ax,ay,az)`, are computed from forces on the neighbor grid cells, `(fx,fy,fz)`, by using the set of weights,

(w1,w2,w3,...) as at lines 10–12. Then, the velocity vector of the p-th particle, (vx,vy,vz), is updated at lines 13–15. In Figure 2, the current density due to the motion of the p-th particle is accumulated on neighbor grid cells by using the set of weights, (w1,w2,w3,...) as at lines 10–19. These loop statements are thread-parallelized with OpenMP. In the accumulation of the current density, the reduction operation of the arrays (cjx,cjy,cjz) on each thread is needed. Note that a collective communication (i.e., MPI_Allreduce()) among particle species s is also necessary to compute the total current density in the second-level parallelism of the particle decomposition with MPI.

The position of particles, (x,y,z), is generally independent of the index of particles, p. The access to the arrays fx,fy,fz at lines 10–12 in Figure 1 corresponds to random load. The access to the arrays cjx,cjy,cjz at lines 10–19 in Figure 2 is random load and store. Hence, cache misses are often caused in these operations. When the list of particles is sorted in accordance with the position of particles, then the access to the arrays fx,fy,fz and cjx,cjy,cjz is sequential and the cache miss can be suppressed.

In the new program structure, quadruple loop statements are used, as shown in Figures 3 and 4, in three dimensional simulations. (Note that triple and double loop statements are used in two- and one-dimensional simulations, respectively.) The outer three loop statements involving iterations through k, j, and i are thread-parallelized with the loop collapsing of OpenMP. This is because there is a possibility that the number of threads on many-core architectures is larger than the number of iteration (i.e., number of grid cells) in a single spatial dimension. The number of particles, np, on each grid cells, (i,j,k), and the pointer to (i.e., the index of) the head of the particle list on the grid cell, psta, are used in the new program structure. The most inner loop statement (at line 5 and 9 in Figure 3 and 4, respectively) involves an iteration through the list of particles from psta(i,j,k) to psta(i,j,k)−1+np(i,j,k). Hence, the index of particles, p, must be sorted in accordance with the order of the indexes of grid cells, (i,j,k).

In the new program structure, it is clear that the access to the arrays fx,fy,fz at lines 10–13 in Figure 3 is sequential load. The access to the arrays cjx,cjy,cjz at lines 25–34 in Figure 4 is also sequential store and load. In the most inner loop statement in Figure 4, the current density due to the motion of the p-th charged particle is accumulated into temporary scalar variables rather than arrays to use cache memory more efficiently as seen at lines 14–23. Note that the "static" schedule for the DO directive of OpenMP with a chunk size of one is adopted. This is because the number of particles on each grid cells is generally nonuniform, and a load imbalance is often caused with a large chunk size.

It is noted that there are additional operations in the new program structure, i.e., computation of the number of particles on each grid cells and the pointer to the head of particle list, and the sorting of the index of particles. In the present study, a counting sort parallelized with OpenMP [16] is used, in which the counting and the sorting operations are performed simultaneously.

```
1  !$OMP DO
2  DO p=1,np
3    i = x(p)
4    j = y(p)
5    k = z(p)
6    w1=...
7    w2=...
8    w3=...
9      :
10   ax = w1*fx(i,j,k)+w2*fx(i+1,j,k)+w3*fx(i,j+1,k)+...
11   ay = w1*fy(i,j,k)+w2*fy(i+1,j,k)+w3*fy(i,j+1,k)+...
12   az = w1*fz(i,j,k)+w2*fz(i+1,j,k)+w3*fz(i,j+1,k)+...
13   vx(p) = vx(p)+ax*dt
14   vy(p) = vy(p)+ay*dt
15   vz(p) = vz(p)+az*dt
16 END DO
17 !$OMP END DO
```

Figure 1. Program structure of the conventional PIC method in FORTRAN for computing the velocity of particles.

```
1  !$OMP DO REDUCTION(+:cjx,cjy,cjz)
2  DO p=1,np
3    i = x(p)
4    j = y(p)
5    k = z(p)
6    w1=...
7    w2=...
8    w3=...
9      :
10   cjx(i,j,k) = cjx(i,j,k) + w1*vx(p)
11   cjy(i,j,k) = cjy(i,j,k) + w1*vy(p)
12   cjz(i,j,k) = cjz(i,j,k) + w1*vz(p)
13   cjx(i+1,j,k) = cjx(i+1,j,k) + w2*vx(p)
14   cjy(i+1,j,k) = cjy(i+1,j,k) + w2*vy(p)
15   cjz(i+1,j,k) = cjz(i+1,j,k) + w2*vz(p)
16   cjx(i,j+1,k) = cjx(i,j+1,k) + w3*vx(p)
17   cjy(i,j+1,k) = cjy(i,j+1,k) + w3*vy(p)
18   cjz(i,j+1,k) = cjz(i,j+1,k) + w3*vz(p)
19     :
20 END DO
21 !$OMP END DO
```

Figure 2. Program structure of the conventional PIC method in FORTRAN for computing the current density.

```
1  !$OMP DO COLLAPSE(3) SCHEDULE(static,1)
2  DO k=1,nz
3   DO j=1,ny
4    DO i=1,nx
5     DO p=psta(i,j,k),psta(i,j,k)-1+np(i,j,k)
6       w1=...
7       w2=...
8       w3=...
9          :
10      ax = w1*fx(i,j,k)+w2*fx(i+1,j,k)+w3*fx(i,j+1,k)+...
11      ay = w1*fy(i,j,k)+w2*fy(i+1,j,k)+w3*fy(i,j+1,k)+...
12      az = w1*fz(i,j,k)+w2*fz(i+1,j,k)+w3*fz(i,j+1,k)+...
13      vx(p) = vx(p)+ax*dt
14      vy(p) = vy(p)+ay*dt
15      vz(p) = vz(p)+az*dt
16    END DO
17   END DO
18  END DO
19 END DO
20 !$OMP END DO
```

Figure 3. Program structure of the new PIC method in FORTRAN for computing the velocity of particles.

3. Performance measurement

In the present performance measurement, three different programs of the two-dimensional PIC code are used. "Program 1" is the conventional PIC code where the program structure has a single loop statement involving an iteration through the list of particles as shown in Figures 1 and 2. At the initial state, particles are placed randomly in the simulation domain. "Program 2" is the conventional PIC code as well, but the index of particles is sorted in accordance with the order of the indexes of grid cells at each time step. "Program 3" has the new program structure, which has outer multiple loop statements involving iterations through the spatial grid cells and the most inner single loop statement involving an iteration through the list of particles as shown in Figures 3 and 4. The indexes of particles must be sorted in accordance with the order of the indexes of grid cells at each time step to use the new program structure.

The performances of the these programs are measured on a single compute node that has 512 GB of DDR4 shared memory and a dual Xeon E5-2697 v4 processor (Broadwell, 2.3 GHz). The processor has 18 compute cores and a total of 36 processes are executable on a single node, with the hyper-threading technology disabled. The Intel Parallel Studio XE Cluster Edition Ver.17.0.1.132 is installed on the system. The compiler option used in the present performance measurement is "`-ipo -ip -O3 -xCORE-AVX2 -qopenmp`."

The number of grid cells is fixed to $N_x = 500$ and $N_y = 500$. The number of particles is also fixed to $N_p = 25,000,000$ for each of particle species, i.e., positively-charged ion and electrons. The total size of the job corresponds to ≈ 4 GB including temporary work arrays. The elapsed time for 100 timesteps is measured. One process is

```
1  !$OMP DO COLLAPSE(3) SCHEDULE(static,1) &
   !$OMP REDUCTION(+:cjx,cjy,cjz)
2  DO k=1,nz
3   DO j=1,ny
4    DO i=1,nx
5     cjx1=0.0; cjy1=0.0; cjz1=0.0
6     cjx2=0.0; cjy2=0.0; cjz2=0.0
7     cjx3=0.0; cjy3=0.0; cjz3=0.0
8       :
9     DO p=psta(i,j,k),psta(i,j,k)-1+np(i,j,k)
10      w1=...
11      w2=...
12      w3=...
13        :
14      cjx1 = cjx1 + w1*vx(p)
15      cjy1 = cjy1 + w1*vy(p)
16      cjz1 = cjz1 + w1*vz(p)
17      cjx2 = cjx2 + w2*vx(p)
18      cjy2 = cjy2 + w2*vy(p)
19      cjz2 = cjz2 + w2*vz(p)
20      cjx3 = cjx3 + w3*vx(p)
21      cjy3 = cjy3 + w3*vy(p)
22      cjz3 = cjz3 + w3*vz(p)
23        :
24    END DO
25    cjx(i,j,k) = cjx(i,j,k) + cjx1
26    cjy(i,j,k) = cjy(i,j,k) + cjy1
27    cjz(i,j,k) = cjz(i,j,k) + cjz1
28    cjx(i+1,j,k) = cjx(i+1,j,k) + cjx2
29    cjy(i+1,j,k) = cjy(i+1,j,k) + cjy2
30    cjz(i+1,j,k) = cjz(i+1,j,k) + cjz2
31    cjx(i,j+1,k) = cjx(i,j+1,k) + cjx3
32    cjy(i,j+1,k) = cjy(i,j+1,k) + cjy3
33    cjz(i,j+1,k) = cjz(i,j+1,k) + cjz3
34      :
35   END DO
36  END DO
37 END DO
38 !$OMP END DO
```

Figure 4. Program structure of the new PIC method in FORTRAN for computing the current density.

used for each of the two particle species. The number of processes per compute node is fixed to two, and a measurement of strong scaling is performed by changing the number of threads per process.

Figure 5 shows the result of the performance measurement. Panels (a), (b), (c), and (d) show the strong scaling of the "velocity" kernel, "current" kernel, "sort" kernel, and

the entire program, respectively. The circles, squares and "x"-mark correspond to the results of Programs 1 (conventional program structure with unsorted list of particles), 2 (conventional program structure with sorted list of particles), and 3 (new program structure), respectively.

The velocity kernel of all the programs scales very well, suggesting that the random load does not affect the scalability of OpenMP. The velocity kernel of Program 2 runs about two times faster than that of Program 1 due to the sorted list of particles. The velocity kernel of Program 3 runs about two times faster than that of Program 2 due to the new program structure.

The current kernel of Program 1 scales until eight threads per process. The performance of the current kernel of Program 1 becomes worse with sixteen and eighteen threads per process because of a cache miss in the random store. The current kernel of Program 2 runs about two times faster than that of Program 1 due to the sorted list of particles (until eight threads per process). The current kernel of Program 2 scales well. However, an overhead of the reduction of OpenMP becomes larger as the number of threads is larger. The scalability of the current kernel of Program 3 is not excellent, because an overhead of the reduction of OpenMP (~ 0.1 sec per timestep) is included in the performance measurement. However, the current kernel of Program 3 runs about four times faster than that of Program 2.

The sort kernel (of Programs 2 and 3) scales well until eight threads per process. The scalability becomes worse with sixteen and eighteen threads per process due to the prefix sum of the counting sort, which cannot be parallelized. Note that Program 1 does not use the sort kernel.

The ratio of the load of the velocity kernel to the current kernel in Program 1 is about 1:2. The ratio of the load of the velocity kernel to the current kernel to the sort kernel in Program 2 is about 2:4:3. The ratio of the load of the velocity kernel to the current kernel to the sort kernel in Program 3 is about 1:1:2. Program 2 runs about 1.5 times faster than Program 1. Program 3 runs about two times faster than Program 2.

4. Conclusion

Since the PIC method solves the time development of both of Eulerian and Lagrangian variables, its performance tuning has been an issue in high-performance computing. Cache misses are often caused in the random access to arrays from particles, because the indexes of particles are generally independent of the position of particles, i.e., the indexes of arrays. By sorting the list of particles in accordance with the order of the indexes of grid cells, both of the performance and the scalability of the PIC method can be improved substantially.

The conventional program structure has a single loop statement involving an iteration through the list of particles only. In the present study, a new program structure is also implemented into the PIC method, which has multiple loop statements involving iterations through both of the indexes of spatial grid cells and the list of particles. The new program structure also improved the performance of the PIC code from the conventional program structure. The new code runs about three times faster than the conventional code without sorted list of particles and two times faster than the conventional code with sorted list of particles.

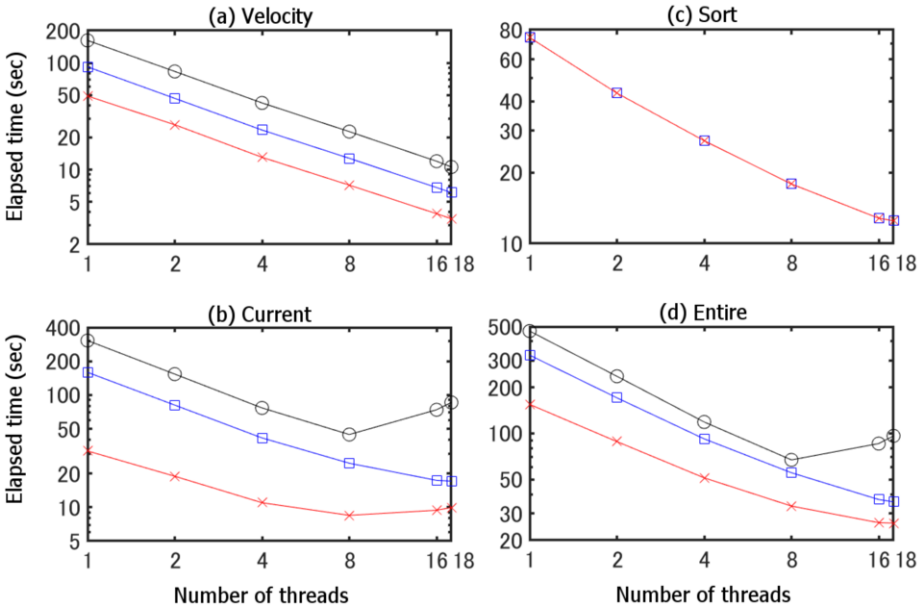


Figure 5. Strong scaling of the PIC code. Elapse time of (a) "velocity" kernel, (b) "current" kernel, (c) "sort" kernel, and (d) the entire program for 100 timesteps with $N_x = 500$ and $N_y = 500$ and $N_p = 25,000,000 \times 2$ (ions and electrons). The circles, squares and "x"-mark correspond to the results of Programs 1 (conventional program structure with unsorted list of particles), 2 (conventional program structure with sorted list of particles), and 3 (new program structure), respectively.

The overhead of the reduction of OpenMP in the accumulation of current density of a single particle into arrays is visible in the present performance measurement with a large number of threads per process. Use of a loop tiling with multi-color ordering instead of the reduction in the thread-level parallelism with OpenMP is left as a future task, which is expected to improve the scalability of the current kernel.

Acknowledgements

This work was partly supported by MEXT/JSPS Grant-In-Aid (KAKENHI) for Scientific Research (B) No.JP19H01868. This work was also conducted as a joint research program at Institute for Space-Earth Environmental Research, Nagoya University.

References

[1] R. W. Hockney, J. W. Eastwood, *Computer Simulation Using Particles*, McGraw-Hill, New York (1981).
[2] C. K. Birdsall, A. B. Langdon, *Plasma Physics via Computer Simulation*, McGraw-Hill, New York (1985).
[3] V. K. Decyk, S. R. Karmesin, A. deBoer, P. C. Liewer, Optimization of particle-in-cell codes on reduced instruction set computer processors, *J. Comput. Phys.*, **10** (1996), 290–298.
[4] K. J. Bowers, Accelerating a particle-in-cell simulation with a hybrid counting sort, *J. Comput. Phys.*, **173** (2001), 393–411.

[5] W. Mattson, B. M. Rice, Near-neighbor calculations using a modified cell-linked list method, *Comput. Phys. Commun.*, **119** (1999), 135–148.
[6] H. Nakashima, Y. Summura, K. Kikura, Y. Miyake, Large scale manycore-aware PIC simulation with efficient particle binning, *Proceedings of 2017 IEEE International Parallel and Distributed Processing Symposium* (2017), 202–212.
[7] B. Di Martino, S. Briguglio, G. Vlad, P. Sguazzero, Parallel PIC plasma simulation through particle decomposition techniques, *Parallel Comput.*, **27** (2001), 295–314.
[8] S. Briguglio, B. Di Martino, G. Fogaccia, G. Vlad, Hierarchical MPI+OpenMP implementation of parallel PIC applications on clusters of symmetric multiprocessors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface; Proceedings 10th European PVM/MPI User's Group Meeting 2003*, *Lect. Notes Comput. Sci.*, **2840** (2003), 180–187.
[9] R. Hatzky, Domain cloning for a particle-in-cell (PIC) code on a cluster of symmetric-multiprocessor (SMP) computers, *Parallel Comput.*, **32** (2006), 325–330.
[10] X. Saez, A. Soba, J. M. Cela, E. Sanchez, F. Castejon, Particle-In-Cell algorithms for Plasma simulations on heterogeneous architectures, *Proceedings of 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing* (2011), 385–389.
[11] K. S. Yee, Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media, *IEEE Trans. Antenn. Propagat.*, **AP-14** (1966), 302–307.
[12] O. Buneman, Time reversible difference procedures, *J. Comput. Phys.*, **1** (1967) 517–535.
[13] J. P. Boris, Relativistic plasma simulation-optimization of a hybrid code, *Proceedings of 4th Conference on Numerical Simulation of Plasmas* (1970), 3–67.
[14] I. V. Sokolov, Alternating-order interpolation in a charge-conserving scheme for particle-in-cell simulations, *Comput. Phys. Commun.*, **184** (2013), 320–328.
[15] T. Umeda, Y. Omura, T. Tominaga, H. Matsumoto, A new charge conservation method in electromagnetic particle simulations, *Comput. Phys. Commun.*, **156** (2003), 73–85.
[16] T. Umeda, S. Oya, Performance comparison of parallel sorting with OpenMP, *Proceedings of 2015 Third International Symposium on Computing and Networking* (2015), 334–340.

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200073

Backus FP Revisited: A Parallel Perspective on Modern Multicores

Alessandro DI GIORGIO [a] and Marco DANELUTTO [a] [1]
[a] *Dept. Computer Science, Univ. of Pisa*

Abstract. We discuss an open source implementation of Backus FP formalism in C++. Our implementation preserves all the nice formal properties of the original language. The implementation is fully C++17 compliant and leverages standard concurrency mechanisms. It provides linear scalability on state-of-the-art shared memory multi cores. By preserving the possibility to use all the rules of the associated "algebra of programs" described by Backus more that 40 years ago, the C++ FP implementation is a natural candidate to be used to introduce parallel programming concepts in core parallel computing courses.

Keywords. parallel patterns, algorithmic skeletons, code refactoring, structured parallelism

1. Introduction

Backus Turing award lecture [Bac78] dates back in the late '70 but provides different features that may be very interesting in these days characterized by the pervasive presence of different kind of parallel devices. Backus designed a programming framework (FP) aimed at relieving the programmers from the burden of explicitly controlling traffic through the Von Neumann bottleneck via memory references. Computations in FP are represented by compositions of functions that may be i) data combiners (just re-shaping data), ii) data transformers (e.g. arithmetic functions) and iii) higher order functional (an apply-to-all and an insert functions that represent kind of map and reduce computations).

As an example, **trans** represents the transpose data combiner: given a sequence of sequences it returns the sequence of sequences made by the corresponding items in the original inner sequences. The **distr** and **distl** data combiners get a sequence and an object and return the sequence of sequences made of the items of the original sequence paired (in a new sequence) with the object. The two variants represent distribution of the right object into the left sequence and vice-versa. Higher order functions include $[f,g]$, the higher order function that builds a sequence of two items obtained applying f and g respectively on the input item, α that is the *apply-to-all* higher order function, applying the function parameter to all items in the input sequence, and finally / that is the *insert* higher order function, "summing up" all items in the input sequence by means of the parameter function (see Fig. 1 for the main FP function definition).

The typical code shown to illustrate FP features is the code implementing matrix multiplication. In FP data is represented in sequences, enclosed in angle brackets ($\langle \ldots \rangle$). A matrix will be therefore represented as a sequence of sequences (the matrix rows). In order to provide the matrix multiplication code, first we define the inner products as:

Higher order fuctions
$\alpha f : \langle x_1 \dots x_n \rangle \equiv \langle f(x_1) \dots f(x_n) \rangle$
$/\oplus : \langle x_1 \dots x_n \rangle \equiv \langle x_1 \oplus \dots \oplus x_n \rangle$
$[f,g] : x \equiv \langle f(x), g(x) \rangle$
$f \circ g : x \equiv f(g(x))$
Fuctions (data transformers)
$binop :< x,y >\equiv x\ binop\ y$
Data combiners
distl : $\langle a, \langle x_1 \dots x_n \rangle \rangle \equiv \langle \langle a, x_1 \rangle \dots \langle a, x_n \rangle \rangle$
distr : $\langle \langle x_1 \dots x_n \rangle, a \rangle \equiv \langle \langle x_1, a \rangle \dots \langle x_n, a \rangle \rangle$
rotl : $\langle x_1 \dots x_n \rangle \equiv \langle x_2 \dots x_n, x_1 \rangle$
rotr : $\langle x_1 \dots x_n \rangle \equiv \langle x_n, x_1 \dots x_{n-1} \rangle$
trans : $\langle \langle < x_1 \dots x_n \rangle \langle y_1 \dots y_n \rangle \rangle \equiv \langle \langle x_1, y_1 \rangle \dots \langle x_n, y_n \rangle \rangle$
$i : \langle x_1 \dots x_i \dots x_n \rangle \equiv x_i$
All functions are $\perp$ preserving. Whenever x is or contains $\perp$ then $f : x = \perp$ for any f

Figure 1. Main FP components

$$IP \equiv (/+) \circ (\alpha\times) \circ \textbf{trans}$$

The computation of the inner product applied to a sequence of two sequences representing the two vectors may be described by the following (rewriting) steps:

$$\begin{aligned} &(/+) \circ (\alpha\times) \circ \textbf{trans} : \langle \langle 1,2,3 \rangle, \langle 4,2,2 \rangle \rangle \rightarrow \\ &(/+) \circ (\alpha\times) : \langle \langle 1,4 \rangle, \langle 2,2 \rangle, \langle 3,2 \rangle \rangle \rightarrow (/+) : \langle 1,4,6 \rangle \rightarrow 11 \end{aligned}$$

Then the matrix multiplication (input is a sequence of two matrices, each represented as a sequence of sequences (rows)) may be defined as follows:

$$\text{MM} \equiv \underbrace{(\alpha\alpha IP)}_{compute\ code} \circ \underbrace{(\alpha\ \textbf{distl}) \circ \textbf{distr} \circ [1, \textbf{trans} \circ 2]}_{data\ routing\ code}$$

In the MM code, the right part represents the computation needed to prepare the data for the actual computation part ($(\alpha\alpha IP)$, that is apply IP on all the sequences build of a row of the first matrix and a column of the second one, as prepared byu the data routing code from the initial pair of matrices). Actually, the definition of IP may be used in place of the IP call in MM, which leads to the expression:

$$\underbrace{(\alpha\alpha((/+) \circ (\alpha\times)}_{compute\ code} \circ \underbrace{\textbf{trans})) \circ (\alpha\ \textbf{distl}) \circ \textbf{distr} \circ [1, \textbf{trans} \circ 2]}_{data\ routing\ code}$$

with an even longer "data routing" part on the right and a correspondingly longer "computational" part on the left.

In his work, Backus stressed the fact FP may be used as an *algebra* of programs, with different rules that can be used to transform programs into functionally equivalent, syntactically different programs. Despite the fact parallel execution of programs was not considered in the paper, the higher order functions and the data combiners may be interpreted as parallel operations. Different researchers pointed out that FP programs naturally express parallel computations (e.g. [WB94,M1]). We claim that the idea of

separating data combiners from actual computations may be useful to express different kind of optimizations.

In this paper, we discuss a framework (fpar) providing Backus FP as an embedded parallel DSL in C++. The implementation leverages the modern features recently included in C++ as well as different existing libraries for parallelism support (OpenMP) and to implement immutable data structures (see Sec. 2.0.3). We will show that programs written in fpar may be automatically parallelized achieving proper speedups on small size shared memory multi-cores using standard C++ mechanisms (threads) and state-of-the-art parallel programming frameworks (OpenMP). In addition, we will discuss how we may apply known and proven correct program transformations that actually improve application performance by coarsening parallel computations (map fusion rule) or improving the data combiner usage (zip rules).

The usage of refactoring rules preserving the functional semantics while changing/improving non functional properties of programs is of great importance. The availability of such refactoring rules has been proven to be a viable solution to explore alternative implementations of parallel applications even before actually starting their coding, especially in the framework of structured parallel programming [BHD+13,GD18, MRR12]. The implementation of fpar preserves all the properties of Backus' FP framework and, in particular, it can be used to show how different refactoring rules may be applied to simple numerical computations such that the rules improve different kind of non functional properties (performance, as shown in Sec. 3, as well as data locality or load balancing, not covered in this paper). Overall this provides the possibility to run simple exercise in classroom whose complexity is far less than the complexity involved in running patterned applications such as those developed using different C++ based parallel programming frameworks [dRADFG17,Rei07].

Finally, we want to point out an additional argument in favour of the usage of FP, related to the utilization of data parallel accelerators. FP code exposes the data transformations needed to subsequently execute map and reduce functionals. This can be exploited while targeting GPU accelerators. In fact the composition of combiners may be used to optimize data transfers to and from GPU accelerators, and the apply-to-all and insert functionals naturally define proper and efficient GPU kernels. Despite we do not discuss explicitly in this paper these aspects, they may contribute to the development of automatic parallelization of programs targeting both CPU cores (as we demonstrated) and GPU cores.

The paper contribution can be summarized as follows.

- We introduce a modern C++ implementation of Backus' FP targeting shared memory multi-cores via OpenMP and we
- We discuss an experiment parallelizing a simple neural network training code. The parallelization comes for free after turning classical imperative code into FP code.
- We discuss how a trivial application of some "algebra of programs" transformation may be used to improve the performance of the original application FP code.
- Finally, we show how decently grained FP computations scale on state-of-the-art shared memory multicores.

2. Implementation

fpar is an implementation of Backus FP in C++17. The implementation is provided as an open source library and available on github[2]. The library is actually provided as an header only package as the compiler optimization techniques may greatly benefit from the simultaneous compilation of both library and business logic (user) code.

2.1. Data types

fpar aims at reproducing the key features of FP as faithful as possible. In particular, data types are implemented via a single variadic template class `Object` that encapsulates a `variant` type:

```
template <typename... Ts>
class Object {
  private:
    variant<monostate, Ts..., flex_vector<Object<Ts...>>> _obj;
  public:
    template <typename T> Object(const T& obj) : _obj(obj) {}
    template <typename T> operator T () const { return get<T>(_obj); }
};
```

This enables the possibility to express polymorphic objects that can assume values of one of the types specified in the instance of the template or, recursively, sequences of such objects. Finally, a further alternative is the empty type $\perp$ which is represented as an instance of `monostate` that is also conveniently denoted as the constant expression `Bottom`. The advantage of using this technique is twofold: on one hand the use of variant enforces type safety [std19], on the other hand the possibility of identifying alternatives of the possible types as variadic arguments frees the implementation from a fixed set of available types. However, programmer has to declare the types of the items eventually appearing in sequences before actually using them. As an example, if we want to have integers and floating point numbers in a sequence, we must use the following code:

```
using namespace fpar;
using Number = Object<int, double>;
Sequence<Number> X = {0.0, 42, 1.0, 23.0};
```

2.2. Functions

All of the basic arithmetic and logic operators and functions to manipulate and access sequences are implemented. In addition, higher order functions, called functionals, are also provided. It is worth pointing out that all of them are unary functions that take and return (constant references to) `Object` instances. Therefore, an n-ary function takes a sequence of n objects acting as multiple arguments (e.g., the `plus` operator takes the sequence of the two needed operands).

Since the available types are decided by the programmer, all of the functions and functionals provided by fpar are function templates that take as template parameter the instantiated `Object` class. Some of them also have an additional template parameter that specifies the kind of execution (parallel or sequential). As an example, the following code defines a function that squares all items in a sequence in parallel:

[2] `https://github.com/alessandrodgr/fpar`

```
auto square =
    apply_to_all<par_exec, Number>([](const Number& x) { return (x*x); });
```

In this case, an OpenMP parallel for is used to implement the `apply_to_all`. Parallelism degree used in parallel computations of fpar may be fixed through `OMP_NUM_THREADS` variable, as usual. If a `seq_exec` was used as first template parameter of the apply-to-all, the application of a square function on a sequence would have been performed sequentially. In other cases, such as the `insert` (foldr) and `condition` functionals, the mechanisms used for the parallel evaluation strategy are the ones provided by the standard library (`thread`, `async`, `future`).

2.3. Immutability

In order to respect the "functionality" of the FP framework, data managed by the fpar library are implemented using an immutable data structure implementation [Pue17], provided by the immer library[3].

An alternative version does not use immutable data structures requiring a little bit more attention while coding applications, but providing better performances when executing functionals in `par_exec` mode. It is worth pointing out fpar pays a penalty in terms of performance w.r.t. non fpar equivalent code, mainly due to data type boxing that, besides usual overhead, impairs automatic vectorization opportunities. We are currently working to overcome this limitation.

However, the usage of immutable data structures, in addition to the const-correctness enforced by the constant reference parameter passing, gives two main advantages: i) in many cases it rules out eventual data races that otherwise the programmer should take care of, and ii) it keeps the semantics of parallelized constructs unchanged.
This last property is a consequence of the fact that functions with no side effects naturally introduce independence among the tasks executing the constructs in parallel and, since these constructs are parallelized via embarrassingly parallel algorithms (parallel for) task independence is a prerequisite for the correctness of the results.

The only case where purity is not enough to guarantee the correctness of the result is the `insert` functional (foldr), where also commutativity and associativity of the reducing function is asked [MRR12].

Finally, the main consequence of keeping unchanged the semantics of these constructs is that all of the laws and theorems given by the "algebra of programs" also hold for parallel programs. Therefore, fpar programs performance can be optimized using two different, not necessarily disjoint, approaches:

- parallelization of constructs
- simplification of programs via algebraic laws

As an example, the first kind of optimization can be applied to the inner product function *IP* presented in Sec. 1, whose fpar implementation is:

```
auto ip =
  (insert<par_exec>(add, Number(0)) *
    (apply_to_all<par_exec, Number>(mul) *
      trans<Number>))(x);
```

[3] `https://github.com/arximboldi/immer`

As explained before, the parallelization happens for the `insert` and `apply_to_all` functionals, since their execution flag is set to `par_exec`. However, in this case we can further optimize the program by applying, for example, the rule for zip introduction ($\alpha\ f \circ \text{trans} \equiv \text{zip}\ f$):

```
auto ip =
  (insert<par_exec>(add, Number(0)) *
    (zip<par_exec, Number>(mul))(x);
```

Doing so, the amount of computation along with memory operations is drastically reduced, resulting in a better performing program equivalent to the original one. Also notice that in the original code there was a sequential part computing the transposition of the two input vectors (`trans`), while the transformed code is fully parallelized, except for the function composition.

3. Experimental validation

We first discuss an experiment aimed at demonstrating the applicability of the refactoring rules typical of the FP framework and the possibility to achieve notable performance increases through refactoring. In this experiment, we parallelized a simple Neural Network training code with fpar. The original code is written as a loop iterating steps that include matrix multiplications, matrix differences and matrix items transformations. The single iteration can be expressed in FP considering the application of the following steps, working on different input and temporary data sequences: a matrix multiplication, an α, a **zip**[4], another α, a second **zip**, a second matrix multiplication and eventually a final **zip**. These phases eventually result in the following excerpt of C++ code:

```
auto out = (apply_to_all<par_exec, Number>(sigmoid) *
            apply_to_all<par_exec, Number>(ip(W)))(X);

auto err = (apply_to_all<par_exec, Number>(sub_op<double, Number>) *
            trans<par_exec, Number> *
            construct<seq_exec, Number>({constant<Number>(Y), id<Number>}))(out);

auto delta = (apply_to_all<par_exec, Number>(mul_op<double, Number>) *
              trans<par_exec, Number> *
              construct<seq_exec, Number>({
                 constant<Number>(err), apply_to_all<par_exec, Number>(sigmoidder)
              }))(out);

auto WDelta = (apply_to_all<par_exec, Number>(ip(delta)) *
                trans<par_exec, Number>)(X);

W = (apply_to_all<par_exec, Number>(add_op<double, Number>) *
      trans<par_exec, Number> *
        construct<seq_exec, Number>({constant<Number>(W), id<Number>}))(WDelta);
```

The code computes the same results of the original, C++ only, sequential code and achieves decent speedups on single socket multi-core systems with 64bit Linux 4.15 and

[4]the **zip** combiner may be defined in FP as follows: **zip** $f \equiv (\alpha\ f) \circ$ **trans**

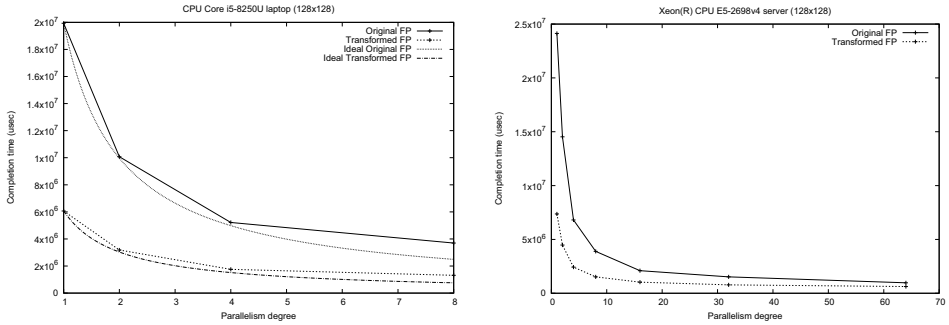


Figure 2. Original and transformed code T_C on a i5 laptop (4 core, 2 way hyper threading) left and on a dual Xeon(R) CPU E5-2698 v4 server (40 cores, 2 way hyper threading) (right). X axis: parallelism degree, Y axis: T_C in $\mu secs$

g++ 9.0.1. Leveraging on the FP formal background, the application represented by FP code may be rewritten using different rules of the algebra, namely:

- $(\alpha\ f) \circ (\alpha\ g) \equiv \alpha(f\ \circ\ g)$ (map fusion)
- $\alpha\ f \circ \text{trans} \equiv \text{zip}\ f$ (zip intro)
- $(\text{zip}\ f) \circ [\alpha\ g \circ 1,\ \alpha\ h \circ 2] \equiv \text{zip}\ (f \circ [g \circ 1, h \circ 2])$ (zip generalize)

In principle, the transformation in the C++ code may be performed automatically, may be following an approach such as the one proposed in [GD17] for more generic and high level parallel pattern applications.

By manually applying the rules mentioned above, we obtain the code listed below that turns out to compute the correct results (as expected, due to the proven correctness of the transformation rules used) but also to compute results with better performance w.r.t. the original code.

```
auto out =
  apply_to_all<par_exec, Number>(
    construct<seq_exec, Number>({id<Number>, sigmoidder}) *
    sigmoid * ip1(W)
  )(X);

auto delta = (zip<par_exec, Number>(sub_and_mul) *
          construct<seq_exec, Number>({constant<Number>(Y),
          id<Number>}))(out);

W = (zip<par_exec, Number>(
  add_op<double, Number> *
    construct<seq_exec, Number>({select<2, Number>,
    ip1(delta) * select<1, Number>})
) * construct<seq_exec, Number>({
        trans<par_exec, Number>,
        constant<Number>(W)
    }))(X);
```

Fig. 2 shows some results we achieved running our FP version of the neural network training code on different architectures. Fig. 2 (right) shows the completion time (T_C) with input matrix size 256x256 on a I7 laptop with 4 cores, with 2-way hyper-threading. Measured and ideal times are shown. Transformed code performs better both in absolute

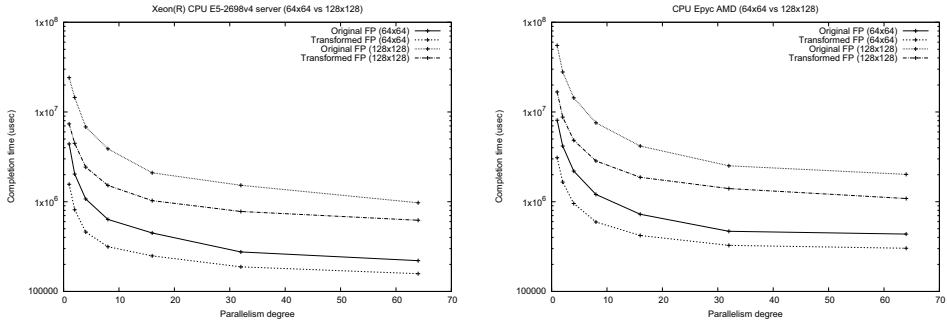


Figure 3. Effect of computational grain: T_C relative to different matrix sized (64x64 and 128x128) on a Xeon PHI KNL (64 cores, 4 way hyper threading) and on an AMD Epyc 7661 (2x32 core, 2-way hyper threading) original and transformed versions. Physical cores only have been used.

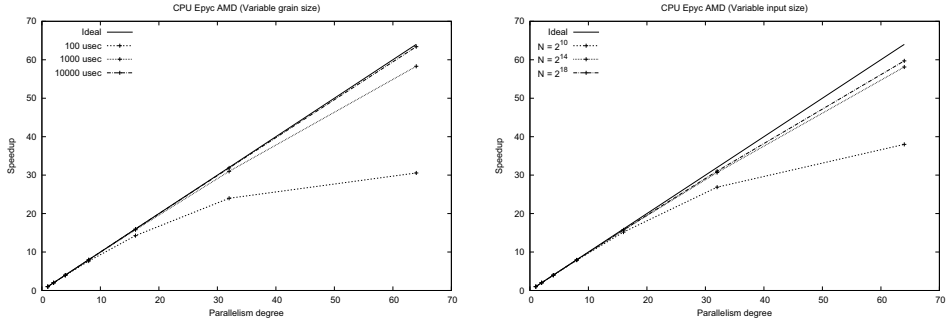


Figure 4. Speedup of typical FP code (AMD Epyc 7551, 64 cores, 2 way hyper-threading). Effect of variable business logic weight (left) and input size (right) in the computation of a synthetic FP application. Speedup computed w.r.t. plain C++17 sequential code.

times and in scalability w.r.t. original FP code. Fig. 2 (left) shows scalability results on a Xeon E5 v4 server relative to 1024x1024 matrix size, confirming the kind of results observed on smaller parallelism degrees on the I7 processor. Eventually, Fig. 3 shows impact of computational grain. The completion times show (log scale on the Y-axis) are relative to an experiment run on both an Intel Xeon PHI KNL [MMM+17] and an AMD Epyc 7551 with different input matrix sizes: 256x256 and 1024x1024. In both cases, the transformed version performs much better than the original one, either stopping scaling after the original version (256x256 version) or even not stopping improving times with parallelism degree while the original version actually stops quite early.

The numbers shown here are good when comparing the two different versions of the code. However, in absolute they demonstrate quite an amount of overhead derived from the ”pure” implementation of FP. As an example, the usage of immutable data structures–while greatly simplifying the overall parallelism management–introduces a considerable overhead with respect to the very same computations implemented using plain `std::vector` data type. We therefore run a different set of experiments aimed at showing strong and weak scaling properties of `fpar`. Using synthetic applications, we looked for the typical computational grain needed to go closer to ideal speedups and to the effect of working on larger and larger data structures. The results are summarized in Fig. 4. The left plot shows that close to ideal speedup can actually be achieved when

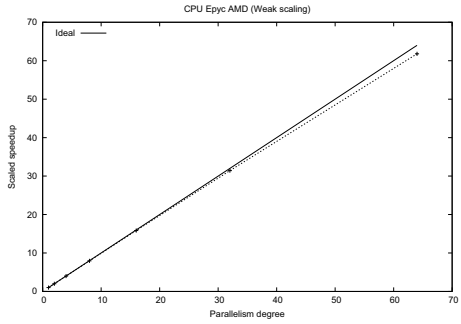


Figure 5. Weak scalability of typical FP code (AMD Epyc 7551, 64 cores, 2 way hyperthreading). Scaled speedup computed w.r.t. plain C++17 sequential code.

the grain of the parallel computation (time spent in a single parallel activity) is close or higher to the milliseconds. This allows to conclude that our FP implementation is definitely not fine grain but still can achieve decent strong scalability. The right plot shows speedups achieved when increasing the size of the processed data. Fig. 5 shows typical results achieved with the synthetic applications in terms of weak scalability.

4. Conclusions

In this paper we briefly discussed a modern open source implementation of Backus' FP exploiting pure C++17: fpar. We discussed how fpar implementation preserves all the nice properties of Backus algebra of programs and how fpar can be used to improve performance of parallel programs through the application of simple program refactoring rules from FP. fpar implementation demonstrated fairly good scalability on state-of-the-art shared multicore architectures. The experiments run with the synthetic applications also demonstrate that fpar implementation exploits medium to coarse grain parallelism pretty efficiently on the same state-of-the-art shared memory parallel architectures. Although other modern programming languages include some of the FP features discussed and exploited in this work, the clean and minimal design of FP supported the efficiency achieved by fpar. Other functional programming languages (Haskell and Erlang, just to mention two well know and widely adopted languages) also support parallelism at different levels and regularities. Refactoring techniques have also been designed to improve or introduce parallelism [BLH12]. Some of the advantages of these languages are that FP programs can be easily expressed via, for example, point-free programming and moreover, they natively support immutable data structures and purity without the need of external libraries. Therefore, they are probably more optimized than fpar with respect of these techniques and features that were artificially tuned inside of our C++17 implementation, as showed in Sec. 2. However the proper usage of these much more sophisticated languages requires different/more significant effort than the one required to understand and use fpar, especially for the parallelisation of programs that in our library is achieved just by setting a flag. Last but not least, the clean and easy to understand implementation of fpar makes it suitable to be adopted in parallel programming courses to demonstrate refactoring techniques for parallel programming.

Acknowledgements This work has been partially funded by Univ. of Pisa PRA_2018_66 DECLware: Declarative methodologies for designing and deploying applications

References

[Bac78] John Backus. Can programming be liberated from the von neumann style?: A functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, August 1978.

[BHD+13] Christopher Brown, Kevin Hammond, Marco Danelutto, Peter Kilpatrick, Holger Schöner, and Tino Breddin. *Paraphrasing: Generating Parallel Programs Using Refactoring*, pages 237–256. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[BLH12] Christopher Brown, Hans-Wolfgang Loidl, and Kevin Hammond. Paraforming: Forming parallel haskell programs using novel refactoring techniques. In Ricardo Peña and Rex Page, editors, *Trends in Functional Programming*, pages 82–97, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[dRADFG17] David del Rio Astorga, Manuel F. Dolz, Javier Fernández, and José Daniel García. Supporting advanced patterns in grppi, a generic parallel pattern interface. In *Euro-Par 2017: Parallel Processing Workshops - Euro-Par 2017 International Workshops, Santiago de Compostela, Spain, August 28-29, 2017, Revised Selected Papers*, pages 55–67, 2017.

[GD17] Leonardo Gazzarri and Marco Danelutto. A tool to support fastflow program design. In Sanzio Bassini, Marco Danelutto, Patrizio Dazzi, Gerhard R. Joubert, and Frans J. Peters, editors, *Parallel Computing is Everywhere, Proceedings of the International Conference on Parallel Computing, ParCo 2017, 12-15 September 2017, Bologna, Italy*, volume 32 of *Advances in Parallel Computing*, pages 687–697. IOS Press, 2017.

[GD18] Leonardo Gazzarri and Marco Danelutto. Supporting structured parallel program design, development and tuning in fastflow. *The Journal of Supercomputing*, May 2018.

[MMM+17] Nicholas Malaya, Damon McDougall, Craig Michoski, Myoungkyu Lee, and Christopher S. Simmons. Experiences porting scientific applications to the intel (knl) xeon phi platform. In *Proc.line of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, PEARC17, pages 40:1–40:8, New York, NY, USA, 2017. ACM.

[MRR12] Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[M1] Mihaela Maliţa and Gheorghe M. Ştefan. Backus language for functional nano-devices. In *CAS 2011 Proceedings (2011 International Semiconductor Conference)*. IEEE press, 2011.

[Pue17] Juan Pedro Bolívar Puente. Persistence for the masses: Rrb-vectors in a systems language. *Proc. ACM Program. Lang.*, 1(ICFP):16:1–16:28, August 2017.

[Rei07] James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[std19] std::variant — cppreference.com, 2019.

[WB94] Clifford Walinsky and Deb Banerjee. A data-parallel FP compiler. *J. Parallel Distrib. Comput.*, 22(2):138–153, 1994.

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200074

Multi-Variant User Functions for Platform-Aware Skeleton Programming

August ERNSTSSON [a] and Christoph KESSLER [a] [1]
[a] *PELAB, Dept. of Computer and Information Science*
Linköping University, 581 83 Linköping, Sweden

Abstract. Today's computer architectures are increasingly specialized and heterogeneous configurations of computational units are common. To provide efficient programming of these systems while still achieving good performance, including performance portability across platforms, high-level parallel programming libraries and tool-chains are used, such as the skeleton programming framework SkePU. SkePU works on heterogeneous systems by automatically generating program components, "user functions", for multiple different execution units in the system, such as CPU and GPU, from a high-level C++ program. This work extends this multi-backend approach by providing the possibility for the programmer to provide additional variants of these user functions tailored for different scenarios, such as platform constraints. This paper introduces the overall approach of multi-variant user functions, provides several use cases including explicit SIMD vectorization for supported hardware, and evaluates the result of these optimizations that can be achieved using this extension.

Keywords. Skeleton programming, SkePU, Heterogeneous computing, Multi-variant user functions, Vectorization

1. Introduction

Programming of complex multi-core and heterogeneous computer architectures can be a difficult task, especially when there is a desire to fully and efficiently utilize the available processing resources. Managing the required workload distribution, synchronisation, and data management often requires expert knowledge and long-time experience. This is especially true if also *performance portability* is desired, as different systems can vary widely in terms of both the number and types of processing cores, as well as in other characteristics such as memory hierarchy.

High-level parallel programming frameworks aim to improve on this situation by reducing the user-facing complexity of programs. A small number of highly optimized but still general programming building blocks are presented through a high-level interface. This category of frameworks include application specific languages, PGAS (Partitioned Global Address Space) interfaces, dataflow models, and more, but most importantly for this paper: the *skeleton programming* [4] concept, borrowing the higher-order operations of functional programming such as `map` and `reduce`, and implemented as an abstraction level that is portable across both multi-core and heterogeneous computers and larger supercomputer clusters. Skeleton programming uses generic building blocks encoding

common computational *patterns* as the high-level programming interface. Examples of such patterns are often divided into two categories: *data parallel* patterns such as the aforementioned map and reduce, and *task parallel* patterns including task farming and parallel divide-and-conquer, among others.

The core contribution of this paper is a generalization of the variant selection mechanism for the skeleton programming framework SkePU, where the problem-specific, sequential user code used to customize a skeleton at skeleton instantiation can be provided in several variants, some of which might even be platform-specific. This is done in a general-purpose programming environment, which differentiates the approach from existing domain-specific variant selection [8]. Our work is also tightly integrated with a platform modeling system [10] allowing build-time lookup of eligible variants going beyond only algorithmic choice or minor variations in performance tuning parameters. The approach is powerful and flexible enough to allow selection based on hardware architecture, levels of heterogeneity, software installations, and more.

Relevant background on SkePU is introduced in Section 2, followed by the idea and implementation of the core contribution in Section 3. We present several use cases in Section 4 and experimental evaluation results in Section 5. We discuss related work in Section 6 and conclusions and future work in Section 7.

2. Background: SkePU

One example of a skeleton programming framework is *SkePU*[2] [6, 7], a C++ compiler toolchain and runtime library implementing data-parallel skeleton programming. SkePU targets heterogeneous systems with multiple *backends*, such as multi-core CPU, GPUs using either CUDA or OpenCL, or even a "hybrid" combination of several backends at once [15]. Each skeleton pattern defined in SkePU (one of `Map`, `Reduce`, `MapReduce`, `Scan`, or `MapOverlap`) is instantiated with a *user function*, typically a small piece of code that is applied once for each element in the input data at run-time. This creates a skeleton instance that can be called like a normal C++ function, but internally provides automatic backend selection and data management (using *smart containers*) across backends.

The SkePU framework performs source code analysis of the input program by an external source-to-source compiler. This tool locates SkePU skeleton instances used by the programmer, and identifies the user functions that are needed to instantiate them. The parameter signature and body of these functions is used to generate fully instantiated backend wrappers and kernels for each user function and skeleton instance (including distinct source files for GPU execution). Each user function therefore has a number of (implicit) *implementation variants* available for it, and the runtime library part of SkePU will select automatically among these variants at program execution time.

The main contribution of this paper is the extension of the variant selection system in SkePU to be a *user-facing* feature of the framework, as presented in the next section.

[2] http://www.ida.liu.se/labs/pelab/skepu/

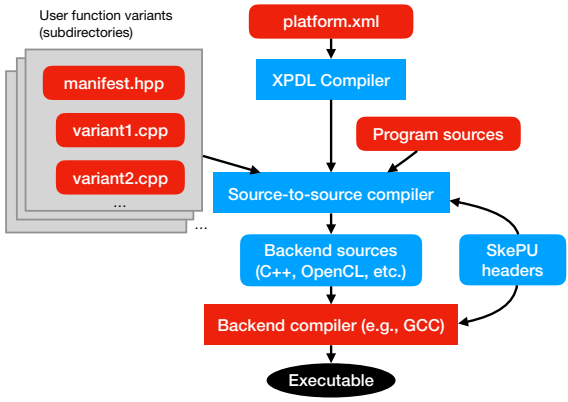


Figure 1. Overview of the components involved in SkePU variant selection and subsequent build process.

3. Idea and Implementation

There are multiple scenarios where a user function with a singular definition can be too restrictive for the purposes of performance: use cases include algorithms with different tradeoffs in time complexity versus memory complexity (some platforms may have very limited memory space available per execution thread), instruction set architecture differences such as native double or half precision floating point arithmetics, the existence of SIMD vector instructions, or other hardware-accelerated implementations of common computations. Since these attributes are *constrained* on the underlying platform, the software-defined code variants must somehow be declared compatible only with the appropriate hardware configurations. For this we employ a combination of *language attributes*, annotations at source-code level that are recognized by the SkePU source-to-source compiler, in addition to the *platform description language* XPDL [10].

A platform description (such as the one given in Listing 1) is supplied to the SkePU source-to-source compiler and depending on the attributes in the model, user function variants are either included or removed from the resulting program. In this example, the user function variant in Listing 3 requires the *Intel AVX* extension to the instruction set. The list of variants for each user function and their prerequisites for inclusion are declared in a *manifest file* (example given in Listing 4). Here XPDL metaprogramming queries or other statically evaluated expressions can be used. As the model in Listing 1 declares the platform to support this extension (line 7 in Listing 1), this vectorized variant will be included for variant selection at run-time. In cases where library or binary compatibility is not required for the extension, this filtering of eligible variants can also happen at run-time, as long as the XPDL model is available for querying. This approach is preferred when a single program executable might run on different hardware configurations.

User function variants are defined externally from the main source file. The variants are placed in individual source files in subdirectories, following a standard naming schema, with one directory for each user function. A *component implementation descriptor* file defines the hardware platform and run-time requirements for each variant. See Figure 1 for an illustration of the workflow: the outlined rectangles denote directories in the file system and the filled rectangles represent files.

Listing 1: XPDL model for an Intel Xeon Gold 6130 CPU. Please refer to XPDL publications [10] and documentation for details about the syntax.

```
<?xml version="1.0" encoding="UTF-8"?>
<xpdl:model xmlns:xpdl="http://www.xpdl.com/xpdl_cpu"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.xpdl.com/xpdl_cpu xpdl_cpu.xsd ">
  <xpdl:component type="cpu" />
  <xpdl:cpu name="Intel_Xeon_Gold_6130" num_of_cores="16"
    num_of_threads="32" isa_extensions="avx avx2">
  <xpdl:group prefix="core_group" quantity="16">
    <xpdl:core frequency="2.1" unit="GHz" />
    <xpdl:cache name="L1" size="32" unit="KiB" set="16" />
    <xpdl:cache name="L2" size="1" unit="MiB" set="16" />
  </xpdl:group>
  <xpdl:cache name="L3" size="22" unit="MiB" set="1" />
  <xpdl:power_model type="power_model_Gold_6130"></xpdl:power_model>
  </xpdl:cpu>
</xpdl:model>
```

4. Use Cases

In this section we present two use cases in detail: user function vectorization and multi-variant components with the `Call` skeleton. We also provide further examples for application of multi-variant components at the end of the section.

4.1. Vectorization Example

As an example of where user function variants are applicable, consider instruction set extensions for SIMD vectorization. These extensions allow the processor to compute the same instruction in parallel over multiple data items, even from a single thread. Many compilers today are *auto-vectorizing* [11–13], but this optimization requires a number of preconditions to be satisfied, such as the correct data alignment and no pointer aliasing; and even then, additional compiler flags are often required. For a high-level parallel program such as a SkePU application, aggressive inlining and loop unrolling must also be applied by the backend (external to SkePU) compiler before there is even an opportunity for auto-vectorization.

For the aforementioned reasons, vectorization is a good motivational use case for multi-variant user functions. Consider the SkePU program in Listing 2. The program performs element-wise addition of two vectors using the SkePU `Map` skeleton with arity 2. The user function `add` is trivial, with two inputs (one from each vector) and the function body returning the sum of the two elements. This user function is straight-forward for the SkePU source-to-source compiler to handle when generating output for all backends: sequential CPU, OpenMP, CUDA, and OpenCL; it is just a matter of copying the function body. However, by this approach, the CPU backends will not be guaranteed optimal performance in the case of the hardware platform supporting SIMD ISA extensions. As such, it makes sense to provide a variant of `add` and make it available for run-time selection.

Listing 2: A SkePU program performing element-wise vector addition.

```
float add(float a, float b) { return a + b; }

int main(int argc, char *argv[])
{
  const size_t size = N; // multiple of 8
  auto vector_sum = skepu2::Map<2>(add);
  skepu2::Vector<float> v1(size), v2(size), res(size);
  vector_sum(res, v1, v2);
}
```

Listing 3: Variant of the `add` user function with explicit vectorization.

```
#pragma skepu vectorize 8
void add(float* c, const float *a, const float *b)
{
  __m256 av = _mm256_load_ps(a);
  __m256 bv = _mm256_load_ps(b);
  __m256 cv = _mm256_add_ps(av, bv);
  _mm256_store_ps(c, cv); // return by pointer
}
```

Listing 4: Manifest file for user function `add`.

```
skepu::VariantList {
  skepu::Variant("add_avx",
    skepu::Requires(
      xpdl::includes<xpdl::cpu_1::isa_extensions, xpdl_avx>::value
    ), skepu::Backend::Type::CPU
  )
};
```

Listing 3 contains a variant of `add` that is defined in a separate file as outlined in Section 3. This file is referenced from the manifest, as seen in Listing 4. In this case, there needs to be a *block* of eight elements available for the function to enable the use of SIMD instructions, which is different in signature to the default variant.[3] This variant uses compiler intrinsic functions which map directly to Intel AVX instructions. The elements in this variant are passed and returned by pointer, and the component implementation descriptor contains the specification of how many elements it accepts in one block (here illustrated by an inline `pragma`). The elements in the array have to be copied to intermediate vector registers before computation.

[3]The need for framework support in this example is not a universal trait; user function variants can be defined with the same signature and even without any required platform constraints.

4.2. Generalized Multi-variant Components with the Call Skeleton

The version 2 revision of SkePU [7] introduced an atypical skeleton construct known as `Call`. The Call skeleton, unlike all other skeleton constructs in SkePU and other typical skeleton programming libraries, does not encode a computational pattern, but rather is an entry point for a self-contained *component* for arbitrary computations. This construct is highly useful in SkePU for two main reasons: firstly, not all computations can be efficiently expressed as data-parallel algorithms, which is the type of patterns present in SkePU, and it is desirable to let generic computations integrate with the smart container and backend selection and tuning systems within SkePU. Secondly, the optimal way to structure computations is in general different for different parallel backends; there needs to be a way to provide *variants* also for these non-skeleton computations.

A common class of computations that fit the above criteria are sorting algorithms. Another example is the fast Fourier transform (FFT) [19], which has several highly optimized implementations available at library level. In cases such as FFT, an instance of `Call` can instantiated with a naive sequential FFT algorithm as the default user function, and additional user function variants are specified as shown in Figure 1 and implemented as thin wrappers over libraries such as FFTW for CPU and CuFFT for Nvidia GPUs. Both the backend type and the presence of libraries in the target system is specified and taken into account for variant selection.

4.3. Other Use Cases

There are a number of other use cases for when multi-variant user functions can be useful for improving performance portability. Below are some suggestions: The user can specify a hand-optimized user function variant to be used only with a certain backend, such as CUDA (declared via the platform attribute in the user function's component implementation descriptor), while the generic auto-generated user function is used for all other backends. Even within the same backend and the same platform constraints, complex user functions may offer multiple variants implementing the same computation by different algorithmic approaches. Selection between the variants can be controlled by input size and shape, as well as other run-time properties such as idle resources and memory pressure. See e.g. the CellSort sorting algorithm [9] where the algorithm used is closely coupled to the characteristic architecture and instruction set of the Cell processor. When SkePU skeletons are invoked from a language other than C++, components that have a variant defined for that language would have lower overhead due to bridging and data representation and would open up for improved compiler optimization.

5. Performance Evaluation

We present performance evaluations for two distinct use cases for multi-variant user functions: vectorization of `Map`-type skeleton applications on real and complex numbers, and specialization of the algorithms used in the user function of a stencil-type image filtering operation using `MapOverlap`.

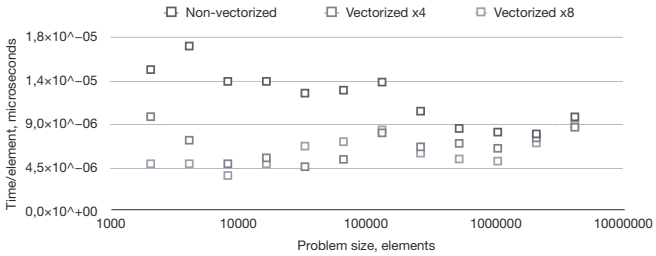


Figure 2. Element-wise vector addition, three variants. Execution time normalized (per element).

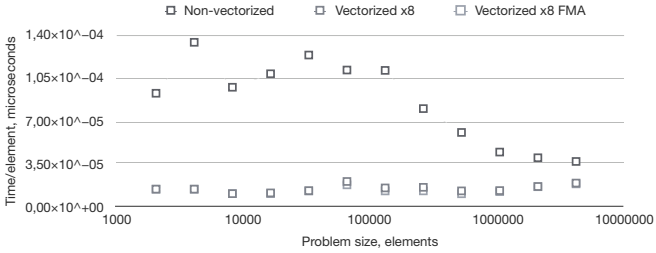


Figure 3. Element-wise complex vector multiplication, three variants. Execution time normalized (per element).

5.1. Vectorization

To demonstrate the performance gained from vectorization of user functions in a scenario in which automatic compiler optimization might be prohibited, we test the example from Section 4.1 using the Intel C++ Compiler v.18.0.1. `-O3` level optimization is enabled for all benchmarks, and the results are presented as the average of 100 runs. All computations are performed on single-precision floating point data. The target system uses Intel Xeon Gold 6130 processors. Two vectorization scenarios are evaluated:

Element-wise vector addition: Three variants are compared: no vectorization, and vectorization by a factor of four and eight, respectively.

Element-wise vector multiplication of complex numbers: Complex numbers stored in struct-of-arrays format, with four input data containers in total. Three versions are tested: no vectorization, factor eight direct vectorization, and a refactored vectorized version using fused multiply add (FMA) vector instructions.

For scalar element addition, the results show that there is always a benefit of vectorization if available. However, as seen in Figure 2 the overhead of loading and storing vector registers is significant when there is only one vector instruction to compute. The choice between four element vector instructions and eight element variants does not matter as much, as the best performer is inconsistent. It is clear that more computation is required to get the most out of manual vectorization.

We also evaluate complex number multiplication (Figure 3). The complex numbers are stored in cartesian form and multiplied element-wise according to $(a+bi)\times(c+di) = (ac-bd)+(ad+bc)i$. There are more vector instructions to amortize the register transfer overhead over in this case, even though the number of inputs is doubled. An alternate version with FMA instructions provides more efficient computation but at the cost of reducing this amortization factor.

Table 1. User function variants for median filtering.

Variant	Time complexity	Memory complexity	Dependencies
Double loop	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	None
Histogram	$\mathcal{O}(n + \|D\|)$	$\mathcal{O}(\|D\|)$	None
`qsort`	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	C standard library

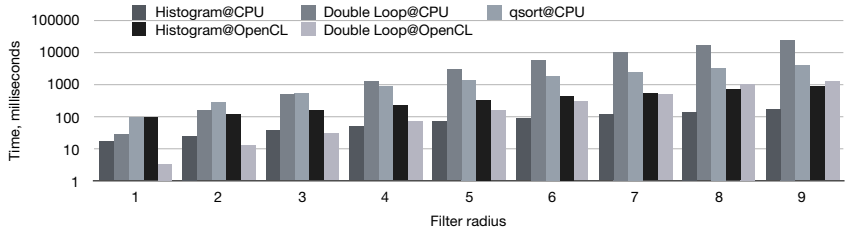


Figure 4. Median filtering using different median computation algorithms.

5.2. Median Filtering

To demonstrate and evaluate the application of multi-variant user functions to provide different algorithmic approaches to the same computation, we look at the median filtering operation on images. For each pixel in the output image, the filter selects the *median* value of all pixels in a region surrounding the corresponding pixel in the input image. The region is defined by a *radius*, the same in both x and y dimensions. Using the `MapOverlap` skeleton, the image filter is then implemented directly by providing the median-finding algorithm as the user function. This can be done in several ways: by sorting the elements in the region, brute-force counting search, or by a histogram collection, among others. The characteristics of the aforementioned three approaches are compared in Table 1 (in the table, n denotes input size and $|D|$ denotes the size of the value domain).

A comparison of execution times for the different variants is presented in Figure 4. The OpenCL variants target a single NVIDIA Tesla K20c GPU. The radius is varied in the range 1-9 pixels, but note that this has an effect in two dimensions and will scale the input region in the user function quadratically. The input image is fixed at 512×512 pixels, in 24-bit RGB format. The results show that there is no algorithm that is optimal across both backends; we even see that, on the GPU, the best variant varies with the filter radius.

6. Related Work

High-level parallel programming using skeletons or patterns [4] allows to model semantics as well as parallelization-relevant properties (such as type of parallelism, data access pattern, data locality constraints) of a computation using special predefined generic constructs (called skeletons or patterns) at a level of abstraction that is clearly above that of source code (such as OpenMP, OpenCL or CUDA). Existing skeleton programming frameworks include SkePU [6, 7], FastFlow [1], Marrow [14], GrPPI [5], Thrust [3] and others.

None of these skeleton programming frameworks considered automated, platform-specific operator specialization for multi-element groups in skeleton instantiations or calls. Lift, [18] on the other hand is a framework consisting of a functional pattern-based programming language, a compiler and an intermediate representation with pre-defined skeleton-like constructs for the hierarchical, functional modeling of data-parallel computations. It allows for (cost-model directed) rewriting of Lift IR trees by a design space exploration process to automatically take into account platform-specific structures such as SIMD operations, data transfers and data layout transformations, which can be expressed by OpenCL-specific constructs. While Lift is more general than our method, it requires the programmer to specify skeleton instances as a hierarchically nested functional decomposition of multiple primitive operators. In contrast, our approach is based on the simpler SkePU programming API, which is more high-level and does not require special tooling nor automated design space exploration nor an explicit intermediate representation.

PetaBricks is another framework which also exposes algorithmic variant ("choice") selection [2, 16]. In contrast to SkePU, PetaBricks is task-oriented with a more involved run-time scheduling system, and does not integrate a platform modeling subsystem into the toolflow.

It is also possible to take a more domain-specific approach. *SLinGen* [17] is a generative programming environment for linear algebra which outputs optimized C code, including optional vectorization driven by intrinsics. The *Cl1ck* system for matrix computations [8] focuses on generating multiple alternative application variants for a single operation.

The limitations of compiler auto-vectorization are explored by Larsen et al. [11] who also suggest improvements to the programming language and environment to facilitate the optimization in more scenarios.

7. Conclusions and future work

Introducing multi-variant user functions increases the performance portability aspect of SkePU programs by allowing the (expert) user to supply optimized source code for different target architectures. The extension is optional to use and not source breaking, and does not impact the programmability of the SkePU framework.

The multi-variant user functions is a part of the multi-variant component programming model developed within the EXA2PRO[4] project. Definition and declaration of user function variants will follow the general component declaration syntax in the EXA2PRO compilation workflow and, together with SkePU skeletons as components themselves, provide *nested* component selection in the EXA2PRO run-time system.

Acknowledgements

This work has been partly funded by the EU Horizon2020 project grant no. 801015 *EXA2PRO* and Swedish National Graduate School in Computer Science (CUGS).

[4] https://exa2pro.eu

References

[1] Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: high-level and efficient streaming on multi-core. In: in Programming Multi-core and Many-core Computing Systems, ser. Parallel and Distributed Computing, S. Pllana, p. 13 (2012)

[2] Ansel, J., Chan, C., Wong, Y.L., Olszewski, M., Zhao, Q., Edelman, A., Amarasinghe, S.: Petabricks: A language and compiler for algorithmic choice. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '09, pp. 38–49. ACM, New York, NY, USA (2009). DOI 10.1145/1542476.1542481

[3] Bell, N., Hoberock, J.: Thrust: A productivity-oriented library for CUDA. GPU Computing Gems, Jade Edition (2011)

[4] Cole, M.I.: Algorithmic skeletons: Structured management of parallel computation. Pitman and MIT Press, Cambridge, Mass. (1989)

[5] del Rio Astorga, D., Dolz, M.F., Fernández, J., García, J.D.: A generic parallel pattern interface for stream and data processing. Concurrency and Computation: Practice and Experience **29**(24), e4175 (2017). DOI 10.1002/cpe.4175. E4175 cpe.4175

[6] Enmyren, J., Kessler, C.W.: SkePU: A multi-backend skeleton programming library for multi-GPU systems. In: Proceedings of the fourth international workshop on High-level parallel programming and applications, pp. 5–14. ACM (2010)

[7] Ernstsson, A., Li, L., Kessler, C.: SkePU 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. International Journal of Parallel Programming pp. 1–19 (2017). DOI 10.1007/s10766-017-0490-5

[8] Fabregat-Traver, D., Bientinesi, P.: Automatic generation of loop-invariants for matrix operations. In: 2011 International Conference on Computational Science and Its Applications, pp. 82–92 (2011). DOI 10.1109/ICCSA.2011.41

[9] Gedik, B., Bordawekar, R.R., Yu, P.S.: Cellsort: High performance sorting on the cell processor. In: Proceedings of the 33rd International Conference on Very Large Data Bases, VLDB '07, pp. 1286–1297. VLDB Endowment (2007)

[10] Kessler, C., Li, L., Atalar, A., Dobre, A.: XPDL: Extensible Platform Description Language to Support Energy Modeling and Optimization. In: Proc. 44th International Conference on Parallel Processing Workshops, ICPP-EMS Embedded Multicore Systems, in conjunction with ICPP-2015 (2015). DOI 10.1109/ICPPW.2015.17

[11] Larsen, P., Ladelsky, R., Lidman, J., McKee, S.A., Karlsson, S., Zaks, A.: Parallelizing more loops with compiler guided refactoring. In: 2012 41st International Conference on Parallel Processing, pp. 410–419 (2012). DOI 10.1109/ICPP.2012.48

[12] Levine, D., Callahan, D., Dongarra, J.: A comparative study of automatic vectorizing compilers. Parallel Computing **17**(10), 1223 – 1244 (1991). DOI https://doi.org/10.1016/S0167-8191(05)80035-3

[13] Maleki, S., Gao, Y., Garzarán, M.J., Wong, T., Padua, D.A.: An evaluation of vectorizing compilers. In: 2011 International Conference on Parallel Architectures and Compilation Techniques, pp. 372–382 (2011). DOI 10.1109/PACT.2011.68

[14] Marques, R., Paulino, H., Alexandre, F., Medeiros, P.D.: Algorithmic skeleton framework for the orchestration of GPU computations. In: Euro-Par 2013 Parallel Processing, vol. LNCS 8097, pp. 874–885. Springer (2013)

[15] Öhberg, T., Ernstsson, A., Kessler, C.: Hybrid CPU–GPU execution support in the skeleton programming framework SkePU. The Journal of Supercomputing (2019). DOI 10.1007/s11227-019-02824-7

[16] Phothilimthana, P.M., Ansel, J., Ragan-Kelley, J., Amarasinghe, S.: Portable performance on heterogeneous architectures. In: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, pp. 431–444. ACM, New York, NY, USA (2013). DOI 10.1145/2451116.2451162

[17] Spampinato, D.G., Fabregat-Traver, D., Bientinesi, P., Püschel, M.: Program generation for small-scale linear algebra applications. In: Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018, pp. 327–339. ACM, New York, NY, USA (2018). DOI 10.1145/3168812

[18] Steuwer, M., Remmelg, T., Dubach, C.: Lift: A functional data-parallel IR for high-performance GPU code generation. In: Proc. CGO 2017, Austin, USA. IEEE (2017)

[19] Yasuhito Ogata, Toshio Endo, Naoya Maruyama, Satoshi Matsuoka: An efficient, model-based cpu-gpu heterogeneous fft library. In: 2008 IEEE International Symposium on Parallel and Distributed Processing, pp. 1–10 (2008). DOI 10.1109/IPDPS.2008.4536163

Scalability Analysis

This page intentionally left blank

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200076

POETS: Distributed Event-Based Computing - Scaling Behaviour

Andrew BROWN[a,1], Mark VOUSDEN[a], Alex RAST[a], Graeme BRAGG[a], David THOMAS[b], Jonny BEAUMONT[b], Matthew NAYLOR[c], and Andrey MOKHOV[d]
[a] *Electronics and Computer Science, University of Southampton, UK*
[b] *Electrical and Electronic Engineering, Imperial College London, UK*
[c] *Computer Laboratory, University of Cambridge, UK*
[d] *Electrical and Electronic Engineering, University of Newcastle, UK*

Abstract. POETS (Partially Ordered Event Triggered Systems) is a significantly different way of approaching large, compute intensive problems. The evolution of traditional computer technology has taken us from simple machines with tiny memory and (by todays standards) glacial clock speeds, to multi-gigabyte architectures running orders of magnitude faster, but with the same fundamental process at the heart: a central core doing one thing at a time. Over the past few years, architectures have appeared containing multiple cores, but exploiting these efficiently in the general case remains a 'holy grail' of computer science. POETS takes an alternative approach, made possible only today by the proliferation of cheap, small cores and massive reconfigurable platforms. Rather than program explicitly the behaviour of each core and each communication between them, as is done in conventional supercomputers, here the programmer defines a set of relatively small, simple behaviours for the set of cores, and leaves them to get on with it - with the right behavioural definitions, the system 'self-organises' to produce the desired results.

Keywords. Multicore/manycore systems, Heterogeneous systems, Accelerators

1. Introduction

Moores Law[1]: the number of transistors on a chip doubles every 18 months or so. Dennard scaling[2]: as transistors get smaller, the power density stays constant, so dissipated power remains proportional to area. Koomeys Law[3]: the number of computations per joule of energy dissipated increases in line with Moores Law.

These principles have guided commentary on the computing industry for a long while. Two are exponentials, (and no exponent is sustainable indefinitely in nature), and the other runs into trouble in the opposite direction: semiconductor device physics cannot avoid leakage and quantum effects forever. However, they are all - quite soundly - based on physical effects, and are the domain of the fabrication engineer.

A parallel problem is the continued absence of any general theory of parallel computing. There are multiple academic publications on theoretical aspects of various parallel computing models, but the *general* problem remains hard. Technology gives us a *new* Moores Law: the number of *cores* on a silicon platform rises exponentially and starts to push at the boundaries of manageability - a new roadblock, alongside the power wall, the memory wall, process spread....[7]. In a conventional parallel system, huge swathes of data are moved around to benefit from the compute capabilities afforded by multiple processors. Bubbles in pipelines must be filled. Every cycle of

[1] Corresponding author: Andrew Brown, Electronics and Computer Science, University of Southampton, , Hampshire, SO17 1BJ UK; adb@ecs.soton.ac.uk

every thread must produce useful data: The Beast must be fed. The choreography of this dance is controlled - designed - by the software architect, and in the vast majority of cases the *complexity* issue is side-stepped by making much of the compute functionality exact duplicates of some cornerstone behaviour. What cannot be side-stepped by this technique are the *relative* costs of communications and compute. As computation grows in size, so too do the necessary support datastructures, and the proportion of wallclock spent communicating increases unhelpfully at the expense of the time spent computing. Fabrication technology is realising exa-scale *compute*, but simultaneously exposing the problems intrinsic to exa-scale *communication*.

Concurrent **event-based computing** is an approach intended to address simultaneously the complexity and the communication problems. The foundation work in this space has been reported previously in this conference series[4] and elsewhere [5-6,8]. In essence, the idea is that vast numbers of tiny compute units, each with a small amount of state, interconnected by a narrow but fast (hardware brokered) communications fabric, carrying information in small, fixed size packets, can provide far superior performance in terms of cost and power dissipation - and in some cases, also compute capability. In this paper, we discuss firstly the concept in general terms, and then provide an outline of a prototype architecture, designed to exploit the idea of computation based around an *un*choreographed non-deterministic 'packet storm'. We then provide some initial physical scaling measurements derived from two application areas that have been implemented on the event-based architecture.

2. The concept

Without loss of generality, consider the numerical solution of some physical matrix-based (discrete grid) problem using an iterative process - Gauss-Seidl or Jacobi, for example. Note there is no requirement for regularity or any kind of dimensional planarity. The solution process will consist of some number of embedded loops, or some kind of traversal sequence, moving over the data points of the grid in some trajectory[2] determined by the programmer. At each point, the local state is updated by a

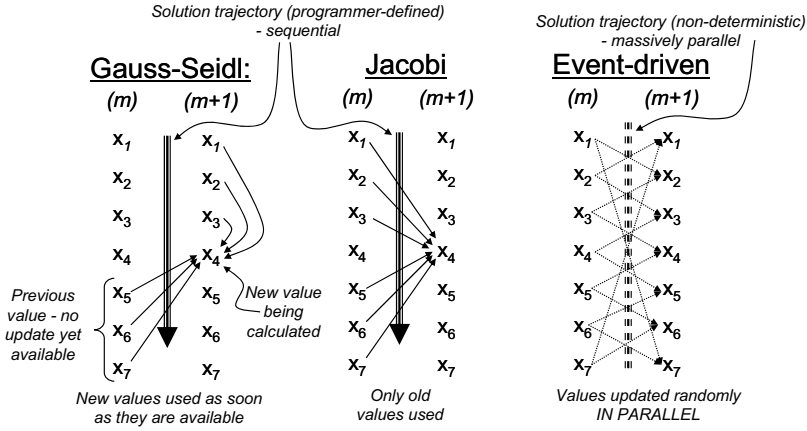


Figure 1: Gauss-Seidl, Jacobi and event-based relaxation

[2] By "solution trajectory", we mean the movement of the overall system state, as opposed to individual atomic data flows.

function of some set of physically adjacent states, and computation moves on. The solution trajectory is deterministic, and dictated by the programmer. It is not until the system reaches some form of numerical equilibrium that we assign physical meaning to the numerical results. If we have a single thread machine, we may use Gauss-Seidl for a fast convergence. If we have multiple cores available, we can use Jacobi (over) relaxation and double-buffer the data, achieving still faster overall performance. The numerical solution sequence *at each grid point* converges more slowly than the comparable trajectory in the Gauss-Seidl regime, because Jacobi is using data that is *less fresh* than Gauss-Seidl. However, both these approaches have controlled, deterministic solution trajectories, and this control is a waste of compute if all we are interested in is the asymptotic solution. The goal here is to do away with this component of determinism, which saves communication time, and thereby exploit the physical parallelism available more efficiently (because we are not paying for control). Ultimately, this (ideally) gives constant scaling.

Consider an alternative approach: Each grid point - there may be millions - has a compute unit associated with it. Each compute unit maintains knowledge of its own state, plus ghosts of its logical neighbours. Leaving aside the starting and stopping problems (described later), the behaviour of each unit is almost trivial. It does nothing until it receives notification (a data packet) telling it that one of its logical neighbours has changed state. On receipt of such a notification, the unit recomputes its own state. *If* the state has not changed, the unit returns to quiescence. *If* it has, the unit asynchronously broadcasts this fact to its logical neighbours (unacknowledged data-push). It is easy to see that once this process starts, a packet storm will develop quite quickly, as each unit continually re-evaluates its own state and broadcasts the change. Some packets will be delayed: the design intention is that the wallclock cost of computing a state update is small, but it cannot be zero, and it certainly cannot be relied upon to be uniform over the system. The notion of simulated time across the compute fabric cannot be defined in any meaningful way whatsoever. How can we achieve useful compute in these circumstances? Some units will be computing with 'stale' data, but we don't mind, because 'fresher' values will be along in short (wallclock) order. We have wasted a (trivial) amount of compute, but this is the price for not having to impose (and pay for) high-level data choreography. The solution *trajectory* is non-deterministic, but has no physical meaning anyway in any compute regime; only the *asymptotic* numerical solution is stable and physically meaningful. This state of affairs obviously depends on the numerical properties of the equation set; some are wildly unstable and unsuitable for this technique. At present, we have a loose formalism for deciding if a technique is suitable: if any change of state caused by a packet arrival unconditionally results in the decrease of (some numerical definition of) energy, then the process will terminate. This is not an all-embracing criterion, and further study is needed. However, the size of the application space for which this approach *is* useful is large and growing.

Event-based processing is not a new concept; space constraints preclude a useful bibliography. What *is* timely is the ability of technology - now - to provide us with sufficient numbers of processing units that the architecture can be made to usefully fit the problem, rather than the other way around.

3. A prototype architecture

Event-based computing is appropriate for systems that can be decomposed into a discrete mesh, albeit one with sometimes millions of nodes. Many important

engineering problems[4] fall into this category. POETS introduces a system based on linking an event-based abstract problem definition to an event-based physical compute platform. From the perspective of abstract application definition, a problem consists of an *arbitrary* graph of **devices**. A device captures the behaviour of a vertex in the discrete distributed model of the physical system (it could be a point on a wire-mesh model of a thermal system, or a single CFD point). From the perspective of abstract *compute*, the system consists of a large number (O(millions)) of extremely small, cheap compute units. These are interconnected by a fixed, fast packet-based communication infrastructure. The packets are small (64 bytes) and entirely hardware-mediated. There is no MPI-like software message layer. The arbitrary application graph is mapped onto the fixed hardware graph by initialisation software (called the Orchestrator), and thereafter device can talk to *logical* neighbour devices logically transparently via hardware. Central points of this system:

- It is computationally asynchronous: there is no central 'overseer' clock.
- The state memory is distributed throughout the physical system, and devices have no visibility of any memory other than that which is local to them.
- Communication is via short, hardware brokered packets. Packet transits are non-deterministic (once launched, the sender loses visibility of the packet, and until it physically arrives, the receiver has no visibility or knowledge of the impending arrival. Packets can take an unpredictable amount of time to arrive, and *in extremis* it is possible for the communication stream to be non-transitive.

By far the most significant aspect of the system lies in the way packets are communicated. In any packet-based communications system with finite internal buffering, if material is injected into the infrastructure faster than it is removed, something must give: either the communications system must refuse to accept further packet injections, or packets must be dropped. In POETS, packet launch is proscribed until and unless the hardware can guarantee (at least part of) the route is open. Whilst this does not *solve* the problem of local congestion, it *moves* it to the point at which it can be most responsibly addressed: the sending component. The sender can

- Abandon the send attempt.
- Repeat the attempt at some future (real) time.
- Modify the packet and try again.

Although (ultimately) guaranteeing data delivery, it is easy to see how this can contribute to the data shear that can lead to non-transitivity.

3.1 The hardware platform

The underlying system platform consists of a six-layer hierarchy - see figure 2 - not dissimilar to the GPGPU stack.

At the highest level, a POETS **system** consists of a set of physical **boxes**. Each box contains a **mothership** (an X86 conventional machine) and a set of **boards**. A board hosts a DE5 development system of 6 FPGAs Every subsequent layer in the system is synthesized on the FPGA, and so can easily be modified. The FPGA contains a fixed (inasmuch as anything is fixed on an FPGA) graph of **mailboxes** and **ports**. The latter connect the cross-board mailboxes The former contains a number of **slots** (currently 4) that play host to a dynamic stream of 64 byte **packets**.

Each mailbox is connected (register mapped) to a synthesized RISC V core (250MHz), which is itself hyperthreaded. The current system (recall everything is synthesized) uses 32 bits to address the threads, limiting the maximum thread count to 4G [9].

3.2 The software stack

The computational problem, from the perspective of the domain-specific user, is of an *arbitrary* graph of application **devices**. The user defines the **application graph** in terms of *named* vertices (devices), each device presenting a set of *numbered* pins, and each pin may be connected to an arbitrary set of pins on other devices (and itself, if need be). The user may also define a **supervisor**. This is a kind of uber-device, the design intent of which is to oversee and facilitate command, control and data exfiltration. Figure 3 illustrates this. The important point here is that the mapping of devices to threads is decided by configuration software (the **Orchestrator**). Each mothership contains an instance of the supervisor (so the number of supervisor instances is dictated by the hardware). The mapping of supervisor instance to device subset is controlled by the Orchestrator. The supervisor behaviour must be defined by the user in the absence of hard knowledge of which device subset it will be overseeing - although the supervisor can always interrogate the device graph and find out.

3.3 Executing an application

What, then, constitutes the definition of an application graph? The application programmer defines the POETS graphs as two components: the graph topology and the device behaviour. The intent (hope?) is that the emergent behaviour of these components will produce the desired result - refer to the non-deterministic solution trajectory outlined in the previous section.

Graph topology is defined conventionally as a set of named, typed device instances with numbered (typed) pins, plus a set of pin-to-pin connections. Pins may only connect to pins of identical type.

Device behaviour is defined by a set of handlers. A hardware thread may play host to a number of (logical) devices (nominally 1024, but this figure is largely arbitrary). Multiple devices per thread represents an area of local temporal sequentialisation in the overall dataflow, so *prima facie* is to be avoided. Resident on each thread is a software skeleton (called the **softswitch**) which is effectively a spinner, interrogating the mailboxes attached to its host core and forwarding packets to the target device. (All the devices mapped to a specific thread share a hardware (32-bit) address. 1024 devices/thread gives a theoretical hard total system size of 4T devices.)

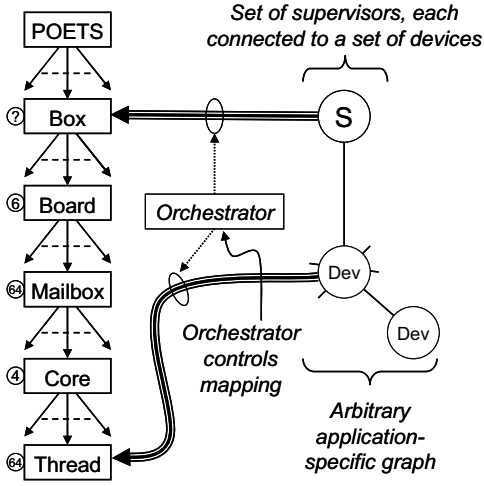


Figure 2: The POETS hardware stack

Each device contains a small state space (further subdivided into static properties and mutable state). Any incoming packets to a device are passed to the handler (invoked by the softswitch): the precise behaviour is domain-specific and user defined (the programmer embeds fragments of C into the device handler definitions), but in general the device handler - as a consequence of the incident packet -

may (optionally) change the internal device state and/or emit packets of its own to its (logical) device neighbours and/or supervisor.
Note that the user (or any external source) may inject packets into the device graph via the Orchestrator - (MPI) - supervisor path.
The Orchestrator is an asynchronous, heterogeneous MPI universe, resident on the set of motherships (plus any other processors connected to the MPI backbone). The Orchestrator controls the configuration of the system. Within its own datastructures, it contains

- A model of the available POETS hardware platform (vertex capacities, capabilities and connectivity).
- A model of the (abstract) application graph (devices, pins and types, device and supervisor behaviours).

It is responsible for

- Mapping the device graph to the thread set/graph (this single phase encapsulates the most numerically intensive functionality of the Orchestrator, and draws heavily from the world of IC placement, assignment and routing).
- Labeling the logical devices with a hardware address.
- Assembling the code fragments describing device behaviour and the device state space definitions with the softswitch skeleton, cross-compiling and linking the composite source with the low-level RISC-V library to produce the binary code (to be executed on the RISC-V threads), and downloading these binaries to the target cores.

Further details of note:

- The RISC-V has a Harvard architecture, and so the data space memory maps produced by the Orchestrator are obviously thread unique (and thus a function of the device:thread mapping), but the instruction space in each core is shared by all the threads on that core. This is not as restrictive as it might appear - in intended use, the vast majority of the devices will be of very few types, so the Orchestrator can ensure that all the devices on a core are of the same type without undue stress on the mapping penalty function. (This issue draws from the openMP GPU thread affinity problem).
- The Orchestrator part of the MPI universe is itself multi-threaded, and so can spin off the cross-compilers in a set of conventional X86 threads.

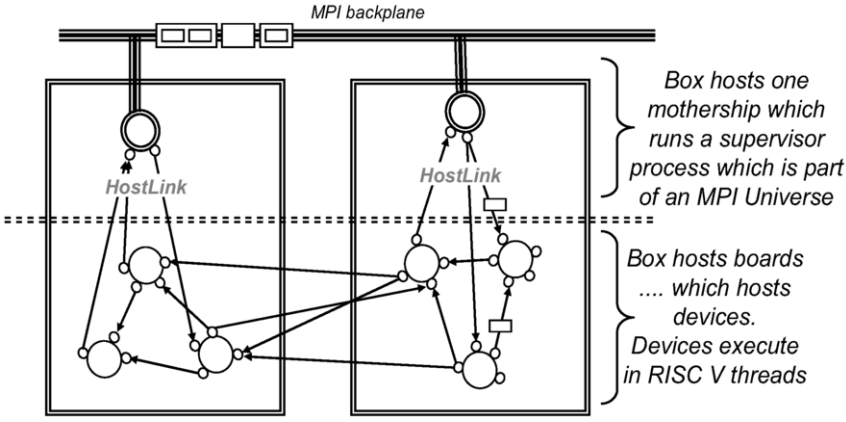


Figure 3: Supervisors and devices

4. Performance: scaling behaviour

Two example application domains are presented here: solving the heat equation, and an example from computational chemistry.

4.1 The heat equation

The heat equation (section 2) $\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2}$ may be canonically discretised to give us $u_i^{(n+1)} = u_i^{(n)} + D\frac{\partial t}{\partial x_i^2}\left(u_{i-1}^{(n)} - 2u_i^{(n)} + u_{i+1}^{(n)}\right)$. A steady state solution of this equation has the temperature of each grid point with mutable state (temperature) as an average of the temperatures of its logical neighbours. (Time varying forcing heat sources necessitate the introduction of thermal *capacities* which complicate the point unnecessarily here.)

4.1.1 Knowing when to stop

Solving the equation numerically is an iterative process. In a conventional computing environment, some limit function looks to establish if the overall or average change in temperature value per iteration step has fallen below some pre-defined value; once this situation is detected, the system is deemed to have converged. In a packet-storm based system, this notion is less well defined, as individual packet latencies may vary wildly, and the time taken to notify the outside world of a putative convergence can be many times larger than an individual packet lifetime. Here we compromise:

Like the conventional approach, we ignore temperature changes below a pre-defined value, so the system eventually stops sending packets. However, the individual devices have no knowledge that this has occurred as they have no notion of time. We introduce the idea of a **heartbeat**: a software-implemented idle detection method that is fully defined by the application writer in the handlers that they provide. (We use the term "heartbeat" because there is no clock-like regularity implied.)

Heartbeats are a type of packet that is emitted frequently (see below); each device counts how many heartbeats it has received, the count being reset any time the device receives a packet from one of its logical neighbours. When this count reaches a pre-defined limit, the device emits an "end" packet to the supervisor. This packet also contains the device current temperature, fulfilling the role of data exfiltration. An end packet can be cancelled at any time prior to all the supervisors flagging finished, should a device receive any subsequent packets from its logical neighbours.

In our initial implementation, we generate heartbeats asynchronously at the thread level. Each device has a user-defined *OnIdle* handler that may be executed by the softswitch when there is no other work to do (no packets to send or receive). We usurp the "first" device on each thread to count the number of times this softswitch handler is executed. When this reaches a pre-defined limit, a heartbeat is sent to each other device on the same thread, bypassing the mailbox. Two counters are required as an individual device has no knowledge of any packets received by other devices in the same thread.

4.1.2 Heat equation – performance

Figure 4 shows the wall-clock execution time a series of simulations of *n*-by-*n* two-dimensional heated plates on a POETS system and a single-threaded 3.8 GHz Intel i7

machine. On the POETS engine, a device calculates the temperature for a single point and convergence is detected using Heartbeats as described in 4.1.1. Devices on POETS are currently mapped to threads naïvely. Near-linear scaling is observed between 6,400 and 78,400 devices (with an anomaly at 16,900 devices). There is a discontinuity between 78,400 and 96,100 devices where the simulation fails to converge. We currently have no explanation for this. Near-linear scaling continues between 96,100 and 1,000,000 devices, albeit at a greater wallclock time.

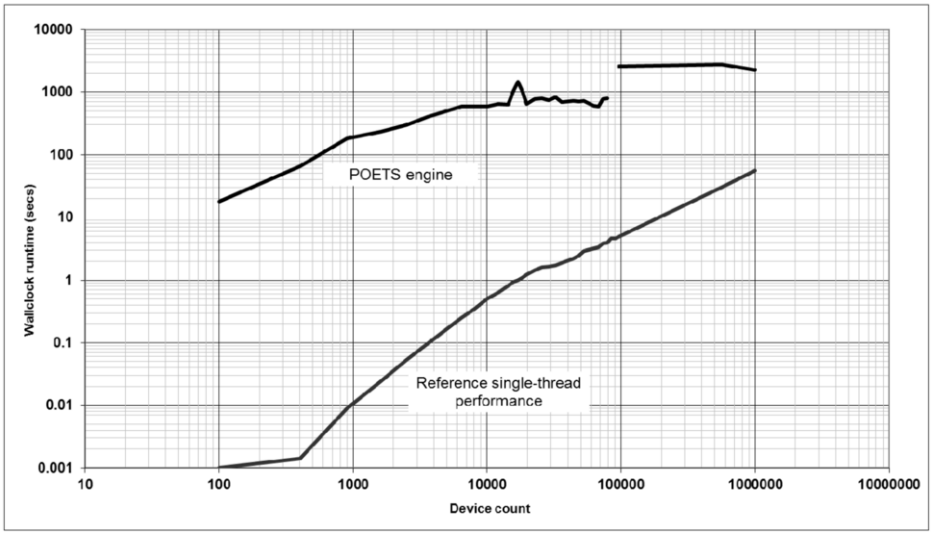


Figure 4: Heat plate simulation performance

4.2 Computational chemistry

The explosion of compute capability over the past decades has had a transformative effect on what may be achieved, and few fields have benefitted as much as computational chemistry: by solving the equations of motion of individual atoms and molecules, the demonstrated emergent behaviour is effectively that of a chemical reaction, with all the complexity that that implies. We live in interesting times: yes, we can compute the trajectories of individual atoms, and so simulate real chemical interactions, but to extract physically meaningful results requires the reaction trajectories of millions of particles followed over billions of timesteps. Even by the standards of the compute resources available today, such an undertaking is hugely expensive, and techniques are constantly being developed to make the undertaking less costly. Two strategies come together to provide a significant increase in what may be achieved in this area: Dissipative Particle Dynamics (DPD) and POETS.

4.2.1 Dissipative particle dynamics

Interesting chemistry usually (but not always) involves large organic molecules, where a carbon backbone folds in complex ways, depending on its surroundings and the ligands attached to side-chains. Usually, 'interesting' behaviour is a function of some gross stereochemical attribute of the system, not the detail: there is no point in following the behaviour of *each atom* in a -CH_3 group, because the relationship

between the three hydrogens and the central carbon is unlikely to change significantly, no matter what happens to the rest of the molecule in the large. Without loss of (too much) generality, then, we can replace the four-atom subsystem with a single pseudo-particle - call it a **bead**. This idea of locally replacing relatively inflexible and internally uninteresting subgroups of atoms can be extended, sometimes cutting down the number of individual elements in a molecule by half an order of magnitude. As each individual atom in a bead itself contributes several degrees of freedom to any simulation, this represents a considerable decrease in the computational load.
The system under simulation usually consists of some number of large, complicated organic molecules, modeled by a set of beads. The beads are interconnected by Hookean and angular bonds, (representing chemical bonds), and usually immersed in some environment (water?) where each water molecule is represented by a single bead. (For reasons that are beyond the scope of this paper, systems incorporating electric charge do not analyze well in DPD). The simulation consists of integrating Newtons' equations of motion for each bead, marching forwards in discrete time steps. The forces acting on each bead at each time step are relatively simple:

- Some bead-bead repulsive force
- Some dissipative (damping) force
- Some random (thermal) force

Within 'sensible' limits, the gross behaviour of the overall system is quite insensitive to the exact numerical form of the force-fields.

4.2.2 The compute environment

Clearly this problem is amenable to parallelisation. The traditional supercomputer approach (using MPI) to this kind of simulation is to tile space with three-dimensional cuboids (wrapping round the boundaries to give a continuous periodic physical model), map each cuboid to a compute core, and to give each core responsibility for simulation of the interactions of the beads within that cuboid. Movement of beads across cuboid boundaries is handled by means of 'ghost' layers, and the simulation *rate* (the ratio of simulated time to wallclock time) is some function of the resources available to the core, the size of the system under simulation, and the number of beads per core. None of this is particularly novel, but the ideas map elegantly onto the POETS architecture, where we can easily and cheaply bring to bear many thousands of individual cores.

4.2.3 Dissipative particle dynamics - performance

Figure 5 below shows the computational cost of a simulation of two immiscible liquids. There is no termination configuration, the simulation is uninteresting and is simply allowed to run for the same number of timesteps for each point on the figure. For comparison, the sequential line is generated on a single thread, single core, 3GHz Intel i7 machine. The POETS line is generated from a small POETS system, containing 6144 threads. The wallclock cost of the simulation is (almost) flat up to 6144 devices, showing that the parallelism is (almost) perfect. The slight slope is due to the physical latency of moving packets about the system - communication costs. At 6144, the system is forced to start doubling up on the number of devices/thread - see earlier comments about serialization in the softswitch - and the runtime cost immediately doubles. Another discontinuity is visible at about 12000 devices, and thereafter the performance degenerates as network congestion starts to take its toll.

5. Final comments

These are small systems (the next system to be built is under construction - this will be an order of magnitude larger, and will move the inflections in figure 5 to the right correspondingly). Even though network congestion has an effect on the performance, in both examples, the system continues to function (section 3). However much traffic is injected into the communications fabric, the system waits locally until the network is drained by computation, and processing continues.

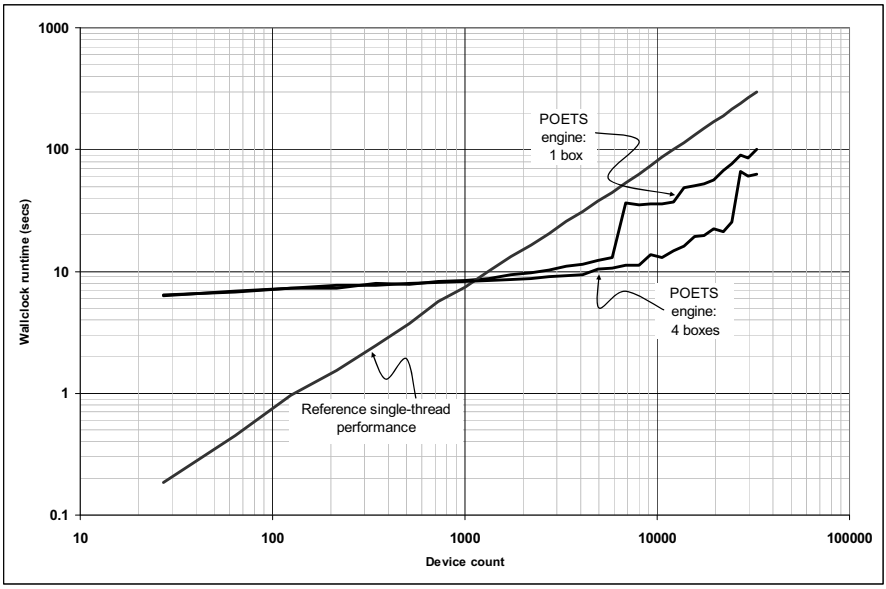


Figure 5: POETS DPD performance

References

[1] G.E. Moore, "Cramming more components onto integrated circuits", Electronics, **38** no 8, 1965
[2] R.H. Dennard, F. Gaensslen, H-N. Yu, L. Rideout, E. Bassous, A. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions" IEEE Journal of Solid State Circuits. **SC-9** (5) 1974.
[3] J. Koomey et al "Implications of Historical Trends in the Electrical Efficiency of Computing", IEEE Annals of the History of Computing, **33** (3): 46-54, doi:10.1109/MAHC.2010.28
[4] A.D. Brown et al, "Distributed event-based computing", International conference on parallel computing, ParCo'17, Bologna, September 2017.
[5] S.B. Furber et al, "Overview of the SpiNNaker system architecture", IEEE T Computers, **62**, no 12, Dec 2013, pp2454-2467, doi 10.1109/TC.2012.142
[6] E. Painkras et al, "SpiNNaker: A 1W 18-core System-on-Chip for Massively-Parallel Neural Network Simulation", IEEE Journal of solid-state circuits, **48**, no 8, pp 1943-1953. doi:10.1109/JSSC.2013.2259038
[7] M. McCool, A.D. Robinson and J Reindeers, Elsevier, Structured Parallel Programming, ISBN 978-0-12-415993-8
[8] A.D. Brown et al "SpiNNaker: Neuromorphic simulation - quantitative behaviour", IEEE T Multi-Scale Computing, 4, no 3, July 2018, pp450-462, doi 10.1109/TMSCS.2017.2748122
[9] M.F. Naylor et al "Tinsel: a manythread overlay for FPGA clusters" International Conference on Field Programmable Logic and Applications (FPL) 9-13 September, 2019.

Acknowledgements

This work was supported by the UK Engineering and Physical Sciences Research Council grant EP/N031768/1

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200077

Towards High-End Scalability on Biologically-Inspired Computational Models

Dario DEMATTIES [a], George K. THIRUVATHUKAL [b,c] Silvio RIZZI [c]
Alejandro WAINSELBOIM [e] B. Silvano ZANUTTO [a,d]

[a] *Universidad de Buenos Aires, Facultad de Ingeniería, Instituto de Ingeniería Biomédica, Ciudad Autónoma de Buenos Aires, Argentina*
[b] *Computer Science Department, Loyola University Chicago, Chicago, Illinois, United States*
[c] *Argonne National Laboratory, Lemont, Illinois, United States*
[d] *Instituto de Biología y Medicina Experimental-CONICET, Ciudad Autónoma de Buenos Aires, Argentina*
[e] *Instituto de Ciencias Humanas, Sociales y Ambientales, Centro Científico Tecnológico-CONICET, Ciudad de Mendoza, Mendoza, Argentina*

Abstract. The interdisciplinary field of neuroscience has made significant progress in recent decades, providing the scientific community in general with a new level of understanding on how the brain works beyond the store-and-fire model found in traditional neural networks. Meanwhile, Machine Learning (ML) based on established models has seen a surge of interest in the High Performance Computing (HPC) community, especially through the use of high-end accelerators, such as Graphical Processing Units(GPUs), including HPC clusters of same. In our work, we are motivated to exploit these high-performance computing developments and understand the scaling challenges for new–biologically inspired–learning models on leadership-class HPC resources. These emerging models feature sparse and random connectivity profiles that map to more loosely-coupled parallel architectures with a large number of CPU cores per node. Contrasted with traditional ML codes, these methods exploit loosely-coupled sparse data structures as opposed to tightly-coupled dense matrix computations, which benefit from SIMD-style parallelism found on GPUs. In this paper we introduce a hybrid Message Passing Interface (MPI) and Open Multi-Processing (OpenMP) parallelization scheme to accelerate and scale our computational model based on the dynamics of cortical tissue. We ran computational tests on a leadership class visualization and analysis cluster at Argonne National Laboratory. We include a study of strong and weak scaling, where we obtained parallel efficiency measures with a minimum above 87% and a maximum above 97% for simulations of our biologically inspired neural network on up to 64 computing nodes running 8 threads each. This study shows promise of the MPI+OpenMP hybrid approach to support flexible and biologically-inspired computational experimental scenarios. In addition, we present the viability in the application of these strategies in high-end leadership computers in the future.

Keywords. MPI, OpenMP, Central Processing Units(CPUs), biologically-inspired computational models, neuroscience, irregular computation, sparse computation.

Introduction

Neuroscience has undoubtedly provided a more in-depth understanding of brain organization in the last decades. Nevertheless, mainstream Artificial Intelligence (AI) research is yet to incorporate these advancements in their models. This fact could be attributed–at least in part–to the success accomplished by some AI approaches–such as Deep Convolutional Neural Networks–which have achieved classification accuracy levels without precedent in the last years. Despite this, some researchers from the AI community recognize that in order to overcome current AI limitations and to create intelligent machines it will be necessary to understand and mimic the brain [1,2]. In such sense, to better understand and explore more deeply how the brain may process information it is essential to use more complex and biophysically accurate neuron and network models than the ones that are prevalent today.

The model of Hodgkin-Huxley (HH) [3]–for example–simulates synaptic receptors and ion channels explicitly. Nevertheless, the more interesting the biological mechanisms, the more limited they are by the size and complexity of the networks. There are some alternative models such as spiking model [4] and the integrate-and-fire model [5] which have been proposed as a simplification of the HH model. Such models demand less computational power, but are not able to directly simulate the biological dynamics present in ion channels. On the other side, we find deep learning (DL) applications [6] that are partially inspired by the biology of the visual ventral pathway, which have dramatically improved the state-of-the-art in many AI domains while ignoring–at the same time–important biological facts and giving priority to computational efficiency and classification accuracy.

Finding the appropriate level of detail in modeling the brain seems to be the holy grail to disentangle the mysteries of animal behavior. In [7] we introduced a biologically inspired and fully unsupervised neurocomputational approach following sequence learning mechanisms applied in [1], and gathering what are–under our point of view–only relevant neuro-anatomical and neuro-physiological facts in order to process information in cortical tissue. In such work we simulated columnar organization, spontaneous micro-columnar formation, adaptation to contextual activations and Sparse Distributed Representations(SDRs) produced by means of partial N-Methyl-D-aspartic acid (NMDA) depolarization. Our pyramidal neuron model dissociated proximal from distal dendritic branches. Proximal dendrites acted as a homogeneous set receiving only afferent information. Information in proximal dendrites determined a bunch of neural units in a Cortical Column (CC) which could be activated depending on the previous activations in the same as well as in neighboring CCs. Distal dendrites–on the other hand–received only lateral and apical information acting as independent detectors. Distal dendritic information pre-activated neural units putting them in a

predictive state in order to receive future afferent information. Some important remarks in reference to the neurocomputational approach are: (i) proximal afferent dendrites do not determine a neuron to fire, instead, they bias its probability of doing so, (ii) distal dendritic branches are independent computing elements that contribute to somatic firing by means of dendritic spikes, and (iii) prediction failures in the network produce a phenomenon called Massive Firing Event (MFE) which manifests with the activation of many neurons in a CC impairing SDRs formation. The model's feature abstraction capabilities showed promising phonetic invariance and generalization attributes, improving the performance of a Support Vector Machine (SVM) classifier for monosyllabic, disyllabic and trisyllabic word classification tasks in the presence of environmental disturbances such as white noise, reverberation, and pitch and voice variations. The work aimed to gather only biologically relevant aspects avoiding loading simulations with excessive computational burden and–at the same time–capturing the essence of the information processing properties of the cortex.

With these points in mind, certain aspects were taken into account in order to pursue the implementation of our computational model. Firstly, the biological plausibility of our model freed us from the need to compute gradients. Even though there are important works supporting the idea that *credit assignament*–the ultimate goal of backpropagation–could be a phenomenon happening in cortical tissue [8], we pondered that it is unknown whether *teaching signals* exist in the brain. Furthermore, there is not enough evidence to include a too complex mechanism in our model yet. Instead, we decided to be conservative in this respect. Secondly, prevalent DL frameworks are mainly biased towards GPU parallelization on CUDA cores. Albeit those frameworks have been extremely optimized to take the maximum advantage especially from NVIDIA cards, too many conditions have to be satisfied in order to obtain the best performance. Moreover, there exists an acute specialization of such technologies towards the precise development of certain DL frameworks with little room for innovative and specifically biologically plausible implementations. In that sense, one of the biggest problems in such approaches arises when trying to implement neural populations with sparse or random connectivity structures. Those implementations–strongly demanded in biologically plausible modelling–compromise coalescence in GPU cards and seriously impair performance [9].

Following this line, we implemented our model in C++14 using Object-oriented programing (OOP) paradigm and parallelized it by means of a hybrid strategy using MPI and OpenMP (Fig. 1). The OOP paradigm gave us a powerful tool to compose modular structures allowing the management of complex computational graphs. MPI enabled our model to run on distributed memory systems in a coherent and stable way. Finally, OpenMP provided a fine grained distribution of workload inside each computing node with the option to schedule the OpenMP threads dynamically. This allowed to manage different options of thread affinity and to vary the number of threads in each computing node, among other options.

The measurements of scaling efficiency returned by our tests allow us to claim that this parallelization strategy is a promising procedure to approach new computational implementations, with more biological plausibility and with more irregular and unstructured data-sets in high-end leadership computer resources.

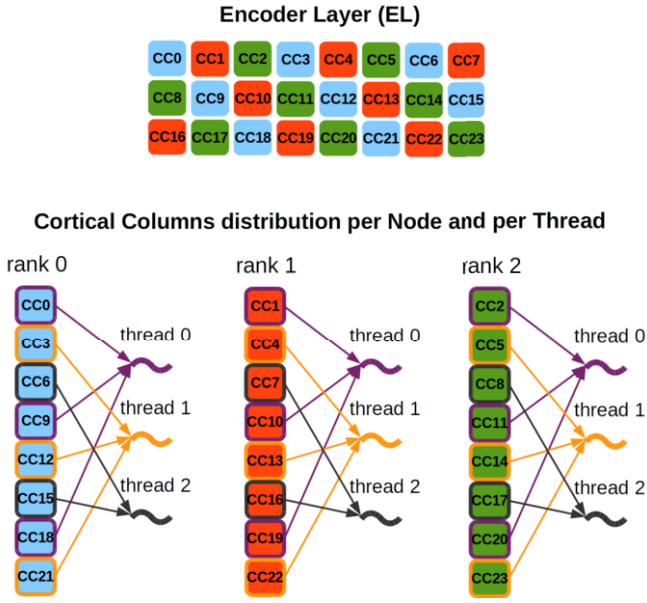


Figure 1. Encoder Layer (EL) MPI+OpenMP parallelization.

Related Work

The computational effort demanded by the brain's specifications force us to consider only those physiological and anatomical features which are key for information processing avoiding loading computational simulations with unnecessary biological detail. In the same direction, parallelization strategies have to be as highly qualified as to face the challenges presented by the implementation of new–biologically accurate–computational approaches on HPC resources.

Brain-Inspired Artificial Neural Network (ANN) Computational Approaches

The development of ANNs is classified in three generations regarding their computational units [10,11]. In 1943, the first generation of ANNs came from McCulloch and Pitts [12]. The authors introduced neurons as computational units which received binary inputs through associated weights and produced threshold dependent binary outputs. Important derivations from such ANNs are multilayer perceptrons, Hopfield nets and Boltzman Machines.

In the second generation, neural units are computational elements whose outputs represent a continuous set of possible values obtained by means of activation functions applied to the weighted sum of the inputs. Common activation functions are sigmoid, polynomial or exponential functions. Examples of these networks are feedforward and recurrent sigmoidal neural nets. An extremely important feature of these networks is that they support learning algorithms based on gradient descend–such as the popular backpropagation. Finally, the real-valued outputs of networks of this generation are interpreted as firing rates in real neurons.

Important behavioral and physiological evidence though made *firing rate interpretation* questionable, and gave rise to the third generation of ANNs which employ *spiking neurons*–or *integrate and fire neurons*–as computational units [3,4,5]. These models–unlike the second generation models–use timing of single action potential–or spikes–to encode information. Including the concept of time in their processing model, Spiking Neural Networks(SNNs) could capture neural behavior more accurately than traditional neural networks. Unlike traditional ANNs, the main idea is that neurons in a SNN do not fire at each cycle, and rather they fire only if a membrane potential reaches its threshold.

In spite of such compelling modeling approach, threshold circuits like the ones introduced by the first generation could be seen as abstract models for digital computation on networks of spiking neurons. In such sense, one bit in active state could be interpreted as a neuron firing within certain short time window and the same bit in inactive state could be interpreted as the same neuron non-firing within such time window [13]. This coding strategy provides a good model for a network of spiking neurons as long as firing times among pre and postsynaptic neurons are synchronized within a few msec time window. There is evidence supporting the fact that this strongly synchronized activity does really occur within the nervous system [14,15].

In such sense, there are new algorithmic developments [1,7] which instead of modeling precise timing activations, prioritize the different roles of proximal and distal dendritic configurations incorporating important physiological and anatomical phenomena, such as the consideration of dendritic branches as active processing elements, the microcolumnar organization in cortical tissue and the sparse patterns of activation in the neocortex–among others. Almost all ANNs, such as those used in deep learning [6] and spiking neural networks [10], use artificial neurons without considering active dendrites and with an unrealistic low number of synapses. These facts suggest that these ANNs are missing fundamental functional properties present in the brain.

CPUs and GPUs for ANN Large Simulations

In the realm of biologically plausible computational models the CPU/GPU dichotomy is not clearly defined. In [9] for example, the authors analyzed the advantages and drawbacks of the CPU and GPU parallelization in different shared memory parallel paradigms, such as OpenMP, Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL) of mean-firing rate neurons. The authors inspected different speed limiters such as floating point precision, thread configuration, data organization and connectivity structure of the networks. Parallel CPU implementations greatly benefited smaller networks, mostly because of cache effects. Large networks–on the other hand–benefited from the GPU only if they demanded memory beyond the available on CPU caches, otherwise an OpenMP implementation was highly preferred. The authors compared several structure representations on the different parallel frameworks showing that on CPUs, these representations reached almost the same computation time. On GPUs instead, the performance was significantly affected by violations of coalescence induced by heterogeneous data structures. Finally, the most serious

problem appeared when the network had a sparse or random connectivity structure, i.e. neurons received connections randomly from other neurons, and not in an organized or ascending order. As the authors pointed out, this totally broke down the performance of GPU implementations, while CPUs were only slightly affected. This was perhaps the strongest argument against GPU implementations of mean-firing rate neural networks, since this sparse connectivity is a repeated pattern in biological networks as well as it is in the computational model presented in this paper.

Materials and Methods

In this paper we introduce a parallelization strategy with great independence on data coalescence and show how it scales efficiently on distributed memory systems while running our biologically-inspired computational model which simulates cortical dynamics with highly sparse and random connectivity profiles [7].

Our group pursued the implementation of a completely unsupervised and biologically inspired computational model–the Encoder Layer (EL) in [7]–which incorporated key properties from the mammalian cortex and returned phonetic features that improved the classification accuracy levels of the SVM algorithm in word classification tasks. This happened in the presence of noise, reverberation and pitch and voice variations not present during training [7]. In this paper we introduce the parallelization strategy applied to the Encoder Layer (EL) code which is approached by means of a hybrid MPI and OpenMP paradigm and through the use of MPI I/O parallel file system with Checkpoint and Restart capacity in the training stage where there is total flexibility in terms of the number of ranks with which the execution is restarted (Fig. 1).

We performed all computational experiments on Cooley [16], a visualization and analysis cluster at Argonne National Laboratory in which we executed scaling tests on the EL–the central algorithm in our model–using up to 64 nodes (one MPI rank per node) and up to 8 OpenMP threads per node/rank. We performed strong and weak scaling tests and measured scaling efficiency.

We parallelized the Encoder Layer (EL) in a way that each MPI rank ends up with one or more CCs and the CCs in each rank are distributed among different OpenMP threads. Fig. 1 shows a hypothetical distribution of CCs in an EL with 3 by 8 (24) CCs among three MPI ranks with three OpenMP threads per rank. Certain ranks could take care of a different number of CCs depending on the number of MPI ranks as well as the number of CCs in the EL. Each MPI rank distributes its CCs among different threads in the same fashion.

Information among MPI ranks must be transferred in each time step. We gather all the information corresponding to the CCs in each rank and then use MPI Bcast function to transmit such information using a special communication protocol by means of which we specify the boundaries in the information corresponding to each CC. By means of this strategy, each MPI rank has to call MPI Bcast just once in order to transmit its data.

The EL uses MPI I/O parallel file system to save its status. Each MPI rank gathers all the data corresponding to its CCs in the EL and puts such data in a

`std::stringstream` class. Then such MPI rank communicates the part of the file it will use to the other MPI ranks in order to store the data without interfering with the other ranks in the MPI environment. Finally, each MPI rank saves all its data with a unique call to MPI Write using the complete `std::stringstream`. An EL with a different number of ranks can load the same file without affecting the final results.

In this work, each MPI rank runs in a different node and keeps all the data that corresponds to the EL object. Although this general EL data is replicated in each MPI rank, this is not significant compared to the data corresponding to the CC objects. Each MPI rank keeps only the data for those CCs which are under its charge.

Results

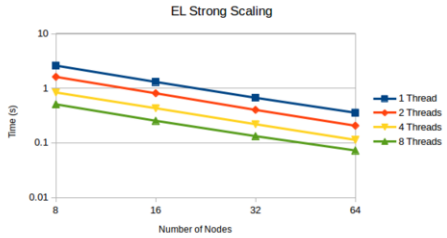
In this paper we tested the scaling efficiency of the parallelization strategies used in the EL by means of *strong* and *weak* scaling tests (Fig. 2). We conducted our tests on Cooley, a cluster to analyze and visualize data produced on high-end supercomputers at Argonne National Laboratory. Cooley has 126 compute nodes; each node has 12 CPU cores. The entire system has a total of 47 terabytes of system RAM. The Cooley node configuration has an Intel Haswell architecture with two 2.4 GHz Intel Haswell E5-2620 v3 processors (6 cores per CPU, 12 cores total), 384GB RAM, FDR Infiniband interconnect and 345GB local scratch space.

Figs. 2a and 2b show the strong scaling capacity of our code in terms of number of processing elements used for the task. In these tests we constrained the code to run one MPI rank per node. Each MPI rank spreads a specific number of threads through the different CPUs in its corresponding node as shown in Fig. 1. The problem size stayed fixed and the number of processing elements was increased. Straight lines in Fig. 2a show–at first–a good scaling capacity. Such fact is confirmed by Fig. 2b which shows the strong scaling efficiency [1].

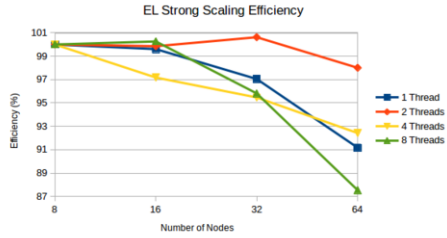
In order to avoid scaling efficiency degradation, the EL has to keep certain number of CCs per OpenMP thread. We tested the scaling running on up to 64 nodes with 8 OpenMP threads each, since the more nodes you incorporate, the more Inter-Process Communication (IPC) load you have. In order to keep a considerable number of CCs per OpenMP thread, we generated an EL with 16384 CCs. In the worst scenario there were 64 computing nodes with 8 OpenMP threads each (512 threads), the model ended up distributing 32 CCs per OpenMP thread. Each CC in this model had 225 neural units to reach a total of 3686400 neural units and 1706803200 synapses in the EL.

As can be seen in Fig. 2b, the larger amount of computing nodes (MPI ranks) with the consequent growth of MPI IPC load did not affect the strong scaling efficiency of the model which was above 87% when running 8 threads per node, but was above 97% when running two threads per node for 64 nodes.

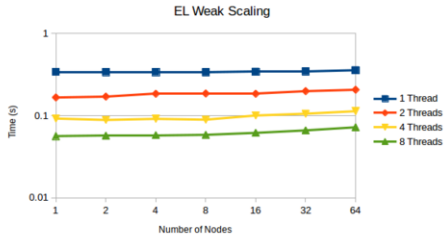
[1]Let t_1 be the amount of time to complete a work unit with 1 processing element, and t_N the amount of time to complete the same unit of work with N processing elements, the Strong Scaling Efficiency is: $t_1/(N * t_N) * 100$



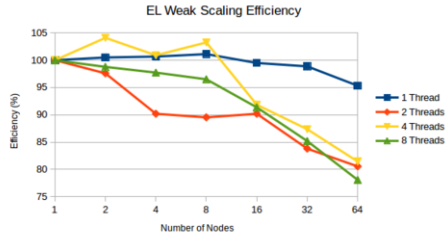
(a) Strong Scaling. Run time vs. the number of nodes for different number of threads per node.



(b) Strong Scaling. Efficiency vs. the number of nodes for different numbers of threads per node. Race line (reference) is taken at 8 computing nodes.



(c) Weak Scaling. Run time vs. the number of nodes for different number of threads per node.



(d) Weak Scaling. Efficiency vs. the number of nodes for different numbers of threads per node.

Figure 2. Strong and Weak scaling tests of the EL algorithm on Cooley nodes. Each MPI rank runs in a different node with 1, 2, 4 and 8 OpenMP threads running in each rank.

In reference to Weak Scaling, in order to keep the ratio of CCs per OpenMP thread constant, we used increasing EL sizes and kept a ratio of 32 CCs per OpenMP thread. Figs. 2c and 2d show the weak scaling performance of the model. In this case the problem workload assigned to each processing element stayed constant and additional elements were used to solve a larger total problem. The horizontal lines in Fig. 2c show–at first–a good scenario. As can be seen in Fig. 2d, the scaling efficiency was always above 75% [2]. These measures show that the model parallel execution was not affected by MPI IPC load as the number of computing nodes increased. This scenario was specially evident for the case of one OpenMP thread in whose case the worst efficiency was above 95%.

Discussion and Conclusion

In this paper we show how parallelization strategies with great independence on data coalescence, scale efficiently on distributed memory systems while running

[2]Let t_1 be the amount of time to complete a work unit with 1 processing element, and t_N the amount of time to complete N times the same unit of work with N processing elements, the Weak Scaling Efficiency is: $t_1/t_N * 100$

a biologically inspired computational model with highly sparse and random connectivity profile.

Algorithmic implementations strongly based on Single Instruction, Multiple Data (SIMD) parallel computing architectures, impose important restrictions on memory data alignment. OpenMP threads in shared memory systems are instead highly independent and powerful processing abstractions which can perform complex tasks with eventual vectorization optimizations when possible.

Our findings show that the parallelization strategies used in this work present good and robust scaling efficiency even in the face of intensive IPC loads. Such behavior can be kept in a sustained manner. Achieving load balance in distributed memory systems is extremely expensive, given the amount of IPC overload needed. In this manner, we claim that the best way to balance computational load among computing elements is to try to confine as much computational load as possible in a shared memory unit without far exceeding the concurrent threading or hyperthreading capacity and/or cache memory capacity provided by a node. Once such conditions are satisfied, it is relatively straightforward to spawn a number of threads in which the work could be distributed. Once in a shared memory system, OpenMP threads are much lighter than MPI processes, and they do not need complex communication methods to share data. Furthermore, OpenMP threads can manage load balancing efficiently and automatically since OpenMP manages dynamic parallel schedule on its own. This is highly desirable, specially in a simulation environment in which individual modules–such as CCs in our cortical model–are not uniformly analogous in terms of size and or connectivity. In MPI instead, the programmer has to deal with load balancing using intensive IPC which is highly expensive especially when the communication is carried out among processes in different nodes. On the other side, OpenMP threads suffer from false sharing in the CPUs caches, but with a highly flexible parallelization scheme as the one used in section Materials and Methods, the user can flexibly vary the parallelization granularity as to achieve the best performance, avoiding that each thread exceeds the quota of cache memory available in each CPU.

In Fig. 2 the phenomenon of *super-linear* speedup is present for several cases. In [17] the authors pointed out that: The superlinear speedup performance in persistent algorithms occurs mainly due to the increased cache resources in the parallel computer architectures, the prefetching of shared variables in shared memory organization, or better scheduling in heterogeneous environments. The effects of the shared memory architectures also impact the performance behavior of the granular and scalable algorithms. We endorse such statement and consider it sustainable as a general explanation for our case. Nevertheless, we also consider that more in depth analysis of memory utilization using profiling tools will be needed in the future.

The scenario in which the computational burden assigned to each shared memory system is distributed among a set of highly lightweight, flexible and dynamic OpenMP threads, is really favorable in a context in which the number of CPUs sharing memory increases specially in high-end supercomputers. In such respect and in the face of the good results returned by our experiments, we evaluate as viable the implementation of these parallelization strategies in high end supercomputers in the future.

We thus claim that this work introduces parallelization strategies whose flexibility and robustness are particularly useful in overly variable and biologically inspired computational scientific scenarios whose modelization approaches can vary dramatically in different biologically accurate implementations strategies in which there are erratic network structures with highly sparse and random connectivity profiles.

References

[1] Jeff Hawkins and Subutai Ahmad. Why neurons have thousands of synapses, a theory of sequence memory in neocortex. *Frontiers in Neural Circuits*, 10:23, 2016.

[2] Adam H. Marblestone, Greg Wayne, and Konrad P. Kording. Toward an integration of deep learning and neuroscience. *Frontiers in Computational Neuroscience*, 10:94, 2016.

[3] A.L. Hodgkin and A.F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *Bulletin of Mathematical Biology*, 52(1):25 – 71, 1990.

[4] Eugene M. Izhikevich, Joseph A. Gally, and Gerald M. Edelman. Spike-timing dynamics of neuronal groups. *Cerebral cortex*, 14 8:933–44, 2004.

[5] E. M. Izhikevich. Which model to use for cortical spiking neurons? *IEEE Transactions on Neural Networks*, 15(5):1063–1070, Sept 2004.

[6] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015.

[7] Dario Dematties, Silvio Rizzi, George K. Thiruvathukal, Alejandro Wainselboim, and B. Silvano Zanutto. Phonetic acquisition in cortical dynamics, a computational approach. *PLOS ONE*, 14(6):1–28, 06 2019.

[8] Jordan Guerguiev, Timothy P. Lillicrap, and Blake A. Richards. Towards deep learning with segregated dendrites. In *eLife*, 2017.

[9] Helge lo Dinkelbach, Julien Vitay, Frederik Beuth, and Fred H Hamker. Comparison of gpu- and cpu-implementations of mean-firing rate neural networks on parallel hardware. *Network: Computation in Neural Systems*, 23(4):212–236, 2012. PMID: 23140422.

[10] Wolfgang Maass. Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, 10(9):1659 – 1671, 1997.

[11] Samanwoy Ghosh-Dastidar and Hojjat Adeli. Third generation neural networks: Spiking neural networks. In Wen Yu and Edgar N. Sanchez, editors, *Advances in Computational Intelligence*, pages 167–178, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[12] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. 1943. *Bulletin of mathematical biology*, 52 1-2:99–115; discussion 73–97, 1990.

[13] Leslie G. Valiant. *Circuits of the Mind*. Oxford University Press, Inc., New York, NY, USA, 1994.

[14] M. Abeles, Hagai Bergman, E Margalit, and Eilon Vaadia. Spatiotemporal firing patterns in the frontal cortex of behaving monkeys. *Journal of neurophysiology*, 70 4:1629–38, 1993.

[15] W. Bair, C. Koch, W. T. Newsoine, and K. H. Britten. Reliable temporal modulation in cortical spike trains in the awake monkey. *Proceeding of Dynamics of Neural Processing*, pages 84–88, 1994.

[16] Cooley | Argonne Leadership Computing Facility. https://www.alcf.anl.gov/user-guides/cooley. Accessed: 2018-11-24.

[17] S. Ristov, R. Prodan, M. Gusev, and K. Skala. Superlinear speedup in hpc systems: Why and when? In *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 889–898, Sept 2016.

Scientific Visualization

This page intentionally left blank

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200079

GraphiX: A Fast Human-Computer Interaction Symmetric Multiprocessing Parallel Scientific Visualization Tool[1]

Re'em Harel [a,b] and Gal Oren [c,d]

[a] *Department of Physics, Bar-Ilan University, IL52900, Ramat-Gan, Israel*
[b] *Israel Atomic Energy Commission, P.O.B. 7061, Tel Aviv 61070, Israel*
[c] *Department of Computer Science, Ben-Gurion University of the Negev, P.O.B. 653, Be'er Sheva, Israel*
[d] *Department of Physics, Nuclear Research Center-Negev, P.O.B. 9001, Be'er-Sheva, Israel*

Abstract. Scientific visualization tools are essential for the understanding of physical simulation, as it gives a visualization aspect of the simulated phenomena. In the past years, data produced by simulations join the big-data trend. To maintain a reasonable reaction time of the user's commands, many scientific tools tend to introduce parallelism schemes to their software. As the number of cores in any given architecture increases, the need for software to utilize the architecture is inevitable. Thus, GraphiX - a scientific visualization tool parallelized in a shared-memory fashion via OpenMP version 4.5 was created. We chose Gnuplot as the graphical utility for GraphiX due to its speed as it is written in C. GraphiX parallelism scheme's work-balance is nearly perfect and scales well both in terms of memory and amount of cores. We achieved a maximum of 560% speedup with 16 cores while visualizing approx 3 million cells.

Keywords. Visualization, HCI, OpenMP, GUI, NUMA, SMP, ParaView, VisIt, MATLAB, multi-core

1. Introduction

Scientific visualization tools play an important role in the understanding of simulated physical data [2], exploring the data produced and debugging the simulation itself. This data is produced by various scientific simulations and is analyzed by placing the data in some visual context. Among these scientific simulations are computational fluid dynamics, molecular dynamics and so forth. Nowadays, many scientific visualization tools can be used in a variety of ways to visualize data as heat maps, contours, isosurfaces, three-dimensional and unstructured meshes. One important aspect these tools must take into account is how *fast* the tool can process the user's command or data and produce a visual image [3].

As the demand for simulation resolution increases, the amount of data produced by simulation also increases [4], i.e the data that needs to be processed by the visualization

[1] This work was supported by the Lynn and William Frankel Center for Computer Science. Computational support was provided by the NegevHPC project [1].

tool, combined with the complicated ways to represent the data, leads to long response time, thus harming the user's experience. One approach visualization tools may take to shorten the response time is introducing the distributed-memory parallelism schemes [5] to the tool, such as done in *VisIt* [6], *ParaView* [7], *Tecplot* [8] and many more.

The distributed-memory model consists of multiple processes with different address space, that may coordinate in some manner to perform a task. These processing units may reside on completely different computer nodes using MPI (Message Passing Interface) [9] to communicate with one another. With this approach, data is automatically read, processed, and if needed rendered, in a distributed manner. Thus, dividing the workload and the data between the processing units and decreasing the response time results in an improvement of the HCI (Human-Computer Interaction).

2. Previous Work

Many current scientific visualization tools ease the work of the scientist. Some of these visualization tools provide the graphical aspect and it is in the scientists' duty to write the data-parsing and plotting methodology. These tools are fast in response time and flexible in their options (as the scientist has full command on how to plot). However, these tools are less scalable, even at the presence of multi-core hardware, as they are work in a serial fashion. For example, *MATLAB* [10], which is a numerical computing environment and programming language developed by MathWorks. Among *MATLAB*'s various features are multi-dimension plotting, contour generation, histograms, vector fields and more. However, besides the parallel computing toolbox [11], *MATLAB* is a serial software. In the similar well-known open-source mimic, *Matplotlib* [12] is a Python plotting library with a similar syntax to *MATLAB*'s plotting commands. *Matplotlib* is capable of two-dimension plots with different options such as color-maps, histograms and more. Nevertheless, is still not essentially parallel.

There are many more tools with the same rationale - focusing on providing a fast-response graphic visualization of data such as *Gnuplot* [13] , *GNU Octave* [14] etc. However, as previously mentioned, these tools require manual parsing of the data, and specifically producing (via command-line or code) the wanted plot. To further ease the job of the scientist, some scientific visualization tools provide the processing and parsing of the data, and already include built-in scientific-relevant options such as contours, iso-surfaces, color-maps, mesh generation and more. In contrary to the previous tools, these introduced with scalable parallelization schemes to the parsing and rendering stages. For example, *VisIt* is an open-source, scalable, interactive, parallel up-to three-dimensional visualization tool developed by Lawrence Livermore National Laboratory (LLNL). *VisIt* supports multiple operating systems such as Unix, Windows and Mac, and multiple scientific data formats. Users can manipulate and change their data by applying different operators and mathematical expressions on the data, save their results and images and even produce animations. Moreover, users can use the tool to have a better understanding of their data and even use it to debug their code. *VisIt*'s parallelism scheme [15] is based on the distributed-memory model. The most frequent parallel mode in *VisIt* is where data is partitioned and distributed over the different processing units - the MPI tasks. Each MPI task is responsible for the analysis of the data, i.e on the different operators applied to the data. Additionally, the MPI tasks are responsible for the rendering of its chunk of data and the resulting images from each task are composite together. In most of the cases, the parallelism behind *VisIt* is in an embarrassingly parallel fashion, meaning there is no

need for communication between the processing units. However, in cases where the data needs to be shared among the processing units due to streamlining calculation or volume rendering, the processing units will coordinate and communicate via the MPI API. Another scalable, parallel visualization tool that is based on *VisIt* and extends its parallelization scheme is *VisIt-OSPRay* [16]. The rationale behind this system is to visualize hundreds of gigabytes and even terabytes of data efficiently on regular processors (Intel Xeon [17]) and co-processors (Intel Xeon-Phi [18]). *VisIt-OSPRay* implements a hybrid parallelization scheme, that includes both a distributed-memory model (processes) and a shared-memory model (threads). In the final stage of the visualization - the final image composition, the composition in the same node will be done by using threads thus, minimizing communication overhead and using the shared memory between the threads while the composition between different nodes will be done via MPI. A similar tool to *VisIt* is *ParaView* [19]. This tool is an open-source, multi-platform data analysis, parallel up-to three-dimensional visualization tool developed by Los Alamos National Laboratory. *ParaView*, similarly to *VisIt*, was developed to visualize both small datasets which are suited for laptops, personal computers, etc. and large scale datasets that are suited for HPCs. *ParaView* was designed in layers: The most basic layer is the visualization toolkit (VTK), which provides the data representation, algorithms and the connection between the two. The second layer of *ParaView*'s design is the parallel extension to the first layer (VTK). The parallel layer allows the execution of the algorithms on shared and distributed memory machines. The third layer is the graphical user interface (GUI) itself which provides the transparency of the visualization and the rendering. *ParaView* supports many options such as contours and isosurfaces, vector fields and more. *ParaView*'s parallelism scheme [7] is based on the distributed-memory model, and works in the same work fashion as *VisIt*. *ParaView* implements its parallelism scheme the same way *VisIt* uses MPI. Each MPI task reads a portion of the data, processes it, and if needed will render the data in a parallel manner. The communication between the MPI tasks is handled by the internal modules, i.e every algorithm is implemented in a parallel manner and contains the communication schemes. Figure 1 presents a common way to implement a visualization tool with MPI.

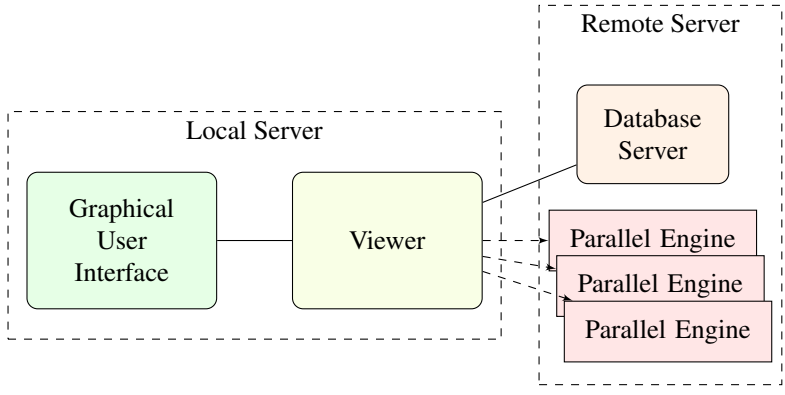


Figure 1. Visualization tool scheme with distributed memory parallelism in *ParaView* and *VisIt*.

3. Problem Formulation

As seen in section 2, the distributed-memory model is a common visualization approach tools take to enhance the user's experience. However, the use of distributed computing and the work-fashion in those tools has some major disadvantages [20]:

- **Distributed Computing Overhead:** When executing commands on the whole dataset, the processing units need to communicate and synchronize with one another - a situation that creates communication overhead. Furthermore, executing a parallel visualization on a Symmetric multiprocessing (SMP) or Non-uniform memory access (NUMA) architecture leads to unnecessary communication overhead, as there is no need for such communication as they can all share the same physical memory and can access it.
- **Slow HCI:** In many cases when there is a need for *immediate* visualization, even of relatively large amount of data [$\sim$ second], the data can actually fit on a single socket [21] - both in terms of memory and computational power - and thus the distribution of the domain on many nodes in order to utilize more cores is counter-effective. Furthermore, the creation of MPI processes even on the same socket is longer than spawning threads, which can be done in the shared-memory model. Additionally, initializing the GUI and Viewer results in spawning MPI processes which of course is time-consuming. However, one can leave the GUI open and save the initializing time but it is not recommended as it is resource-wasteful.
- **Non-Optimized Resource Utilization:** As the current distributed visualization tools use a parallel engine that launches the processes to the nodes throughout all of the tool usages - regardless of actual service - it also implies that the computational resources are in many cases idle but still allocated.

However, in the past years, multi-core architectures become more and more common in desktops, laptops, mobile, etc [22]. The multi-core architecture provides a way for software developers to introduce parallel schemes such as the shared-memory parallelization [23], thus decreasing the software's response time and allowing more complicated operations. The shared-memory model consists of processing units that share the same address space, allowing the processing units to exchange data and communicate with minimal overhead.

As the number of those cores and the amount of their RAM increase [24], the need for distributing the data on different processing units decreases. Thus, introducing shared-memory model parallel schemes can lead to faster response time and optimized resource management [25]. Regarding scientific visualization, distributed tools were created first and foremost for complicated and extreme high-resolution simulations and are suited for the HPC arena. On the contrary, simplified visualization tools were created for visualization of lower to medium resolution than the latter and are suited for desktops computers or single socket of NUMA computers. This distinction is very common in the sciences work fashion, and as such of great interest to be improved in both cases. Due to the increased usage of multi-core architectures in all computational architectures since the year of 2005 [22], the gap between these two types of visualization tools can be reduced by introducing shared-memory parallel schemes to the tools, and most urgently to the desktop and single-socket NUMA suited ones. Therefore, we created *GraphiX* a Fast HCI SMP scientific visualization tool. Figure 2 illustrates the need for such tool.

The existing gap between HPC and PC in terms of simulations, PC will usually run small scale simulations (low dimension) and HPC will run big scale simulation (high resolution, high dimension), which leads to a need for a tool that can benefit from the common architecture - multi-core. Thus, suitable for both architectures and bridging the gap between the two.

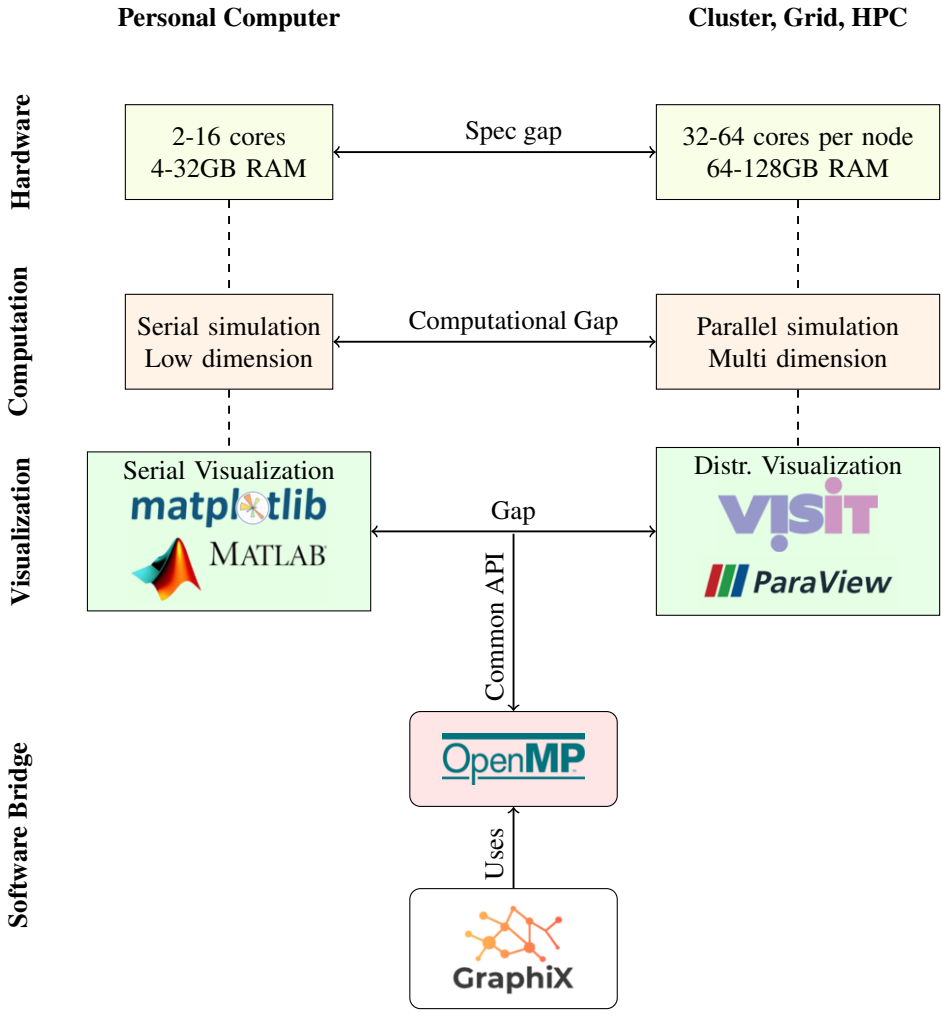


Figure 2. Illustration of the need for GraphiX.

4. GraphiX

GraphiX is a fast HCI-suited scientific visualization tool for both SMP and NUMA up to three-dimension. GraphiX supports several ways to visualize the data such as volume mesh representation, color-maps, contours, x-axis/y-axis mirroring, presenting data related to the mesh, and more.

GraphiX graphical utility is based on Gnuplot [26]. We chose *Gnuplot* because it is open-source and written in C and C++ while *MATLAB* is a proprietary software and not open-source. Additionally, *MATLAB* cannot be parallelized in shared or distributed memory thus, choosing this tool as the graphical utility is impractical. Although *Matplotlib* is open-source and can be parallelized (with threads, not OpenMP) but less effective than parallelization in C, *Gnuplot* is more suited for OpenMP.

Gnuplot [13] is an interactive, multi-platform up-to three-dimensional visualization tool. Unlike *VisIt* and *ParaView*, *Gnuplot* is not parallel in any way and is a command-line driven visualization tool, rather than GUI driven. Nevertheless, it is a fast visualization tool written in C/C++ and was created to allow scientists to visualize different functions and data interactively and non-interactively. It also supports different interactive screen displays such as Qt, wxWidgets, x11 or system-specific. Users can also direct their plots to different file types such as pdf, eps, gif, jpeg, LaTeX, svg and more.

GraphiX GUI is written in Python and the heavy computational operations such as mesh creation, contour line calculations, and color-map interpolations are written in *Cython* [25]. *Cython* is a programming language designed to give C-like performances while maintaining the simplicity of Python syntax. In cases of large data and many operations, we used OpenMP under *Cython*.

Parallelism schemes were introduced to two main modules inside *GraphiX*. The first module, as discussed above, is responsible for reading and parsing the initial data (creating the polygon's coordinates, contours calculations, etc.), and creating the *Gnuplot*'s commands that will later produce the visual image. For the second module, Gnuplot's source code was partially introduced with shared-memory parallelism via OpenMP [23], specifically the source code that creates the polygons (the mesh and color-map) which is the most time-consuming part as will be discussed in section 5. The rationale behind introducing *Gnuplot* with OpenMP is explained in section 3, which is minimizing the overhead and optimizing the use of the common modern architecture - multi-core processing.

GraphiX workflow consists of four main modules: GUI, Controller, File Handler and Gnuplot communicator along with *Gnuplot*'s source code. Figure 3 illustrates the main modules and work-flow.

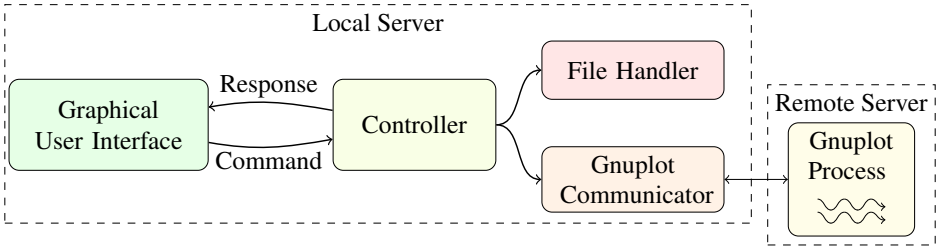


Figure 3. GraphiX's work-flow.

4.1. Graphic User Interface

The Graphic User Interface module handles the user's requests and interactions. As seen in figure 4 and in section 4, *GraphiX* can produce contours, color-maps, axis mirroring,

along with providing the cell's physical data when clicked. The GUI is written in Python with PyQt [27] as the visualization kit. The GUI contains a window (the *Viewer*) that is connected automatically to a *Gnuplot* process. This means that when *Gnuplot* displays the plot it is automatically drawn on to the *Viewer*. Figure 4 presents the GUI part, which includes all the different options such as contours, skipping to the next plot, showing the physical data of a cell and more. In figure 5 different simulations are plotted on the GUI, with *Gnuplot* as the graphical utility. Among the plots are Sedov-Taylor simulation (blast wave) mesh presentation and pressure color-map, and Rayleigh-Taylor instability mesh presentation and density color-map.

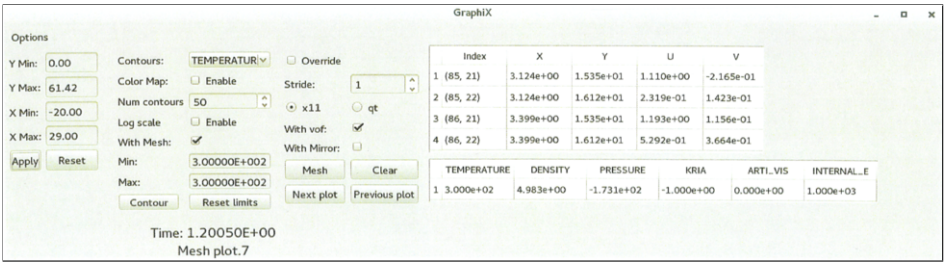
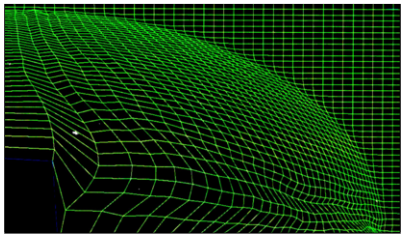
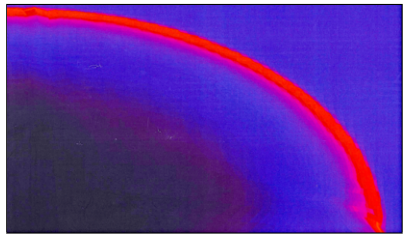


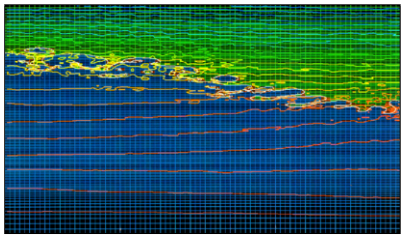
Figure 4. Graphix Graphic User Interface.



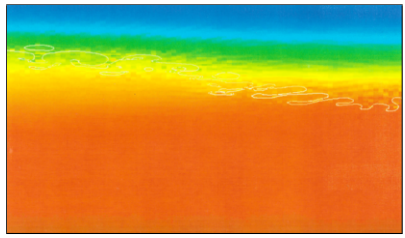
(a) Sedov-Taylor simulation.



(b) Sedov-Taylor density color map.



(c) Rayleigh-Taylor pressure contours.



(d) Rayleigh-Taylor pressure color map.

Figure 5. GraphiX visualization on different simulations.

4.2. Controller

The Controller module is the main module that connects all the other modules. The module stores all the data (coordinates, physical data such as pressure and temperature and more) of the plot. Additionally, the creation of contour lines, color-map and the mesh is done in this module. As mentioned, *GraphiX* is written in Python. Due to this, operations that require heavy calculations such as contours, mesh and color-map creation

may lead to long response time (compared to low-level programming languages such as C). To cope with this problem and speedup this process, these modules were written in *Cython*.

When a user executes a command the GUI sends the request to the Controller. The Controller then executes the appropriate action via the File Handling module. Finally, the Controller will send the file name, and if needed more data, to *Gnuplot*'s communicator (see below 4.4) that will later send the appropriate execution command to *Gnuplot*.

4.3. File Handling

The File Handling module is the module that handles two main parts. The first part reads and parses (if needed) the user's data. The second part creates temporary files (in-memory file system */tmp/* to maximize the I/O operations) that contain the commands *Gnuplot* will later execute. For example, producing a mesh in *GraphiX* is usually done via *Gnuplot*'s polygon objects. To create the appropriate polygon objects the coordinates are parsed and connected in some manner. Afterward, a temporary file is created where each line defines a *Gnuplot* polygon object. Finally, *Gnuplot*'s *load* command is executed on the temporary file and the mesh is presented on the *Viewer*. Currently *GraphiX* supports only the VTK input file format. However, it is possible to extend this part and support additional formats.

4.4. Gnuplot Communicator and Parallel Source Code

Gnuplot Communicator and Source Code module consists of two separate modules that are strongly connected. Because *Gnuplot* is a command-line based visualization tool, the first module, the communicator, opens a *Gnuplot* process shell (command-line) and is in charge of sending the *Gnuplot* commands such as the *load* command, the *plot* command etc. Furthermore, the module receives messages back from the *Gnuplot* process. For example, clicking coordinates that provide the cell physical data.

The second module consists of *Gnuplot*'s modified OpenMP parallel source code. It was found that producing color-maps is the most computationally intensive and time-consuming part in *Gnuplot*. Producing color-maps is done by creating polygons with some value that represents its' color. *Gnuplot* creates polygons by creating a linked-list of objects (objects can be polygons, rectangles etc.). Each time a new polygon is created it is added in some manner to the linked list. Thus, to speed-up the process of creating the polygons the function *load_file*, which in fact creates the linked-list of polygons, was introduced with shared-memory parallelism - OpenMP. As the *Gnuplot load* command is executed to produce the color-map, the parallelism scheme was introduced to the function *load_file* that parses each line of the file and creates the linked-list of polygons accordingly. The parallelism scheme is based on dividing the linked-list to the working threads, i.e each thread that is spawned by the OpenMP run-time environment is responsible on parsing the specific file line and eventually creating a polygon that is part of its own *private* and independent linked-list. Finally, once all the threads finish creating their linked-lists, the *master-thread* links them together. In NUMA architectures the OpenMP run-time environment may execute the threads on processing units (cores) that reside on different sockets, resulting in more frequent false-sharing [28], thus reducing the speedup gained. To overcome this [29], thread affinity and placement were included

in the parallelization schemes using OpenMP 4.5 [30]. The following figure 6 illustrates the parallelization scheme, given N polygon objects (the yellow rectangle) and K threads the workload will be:

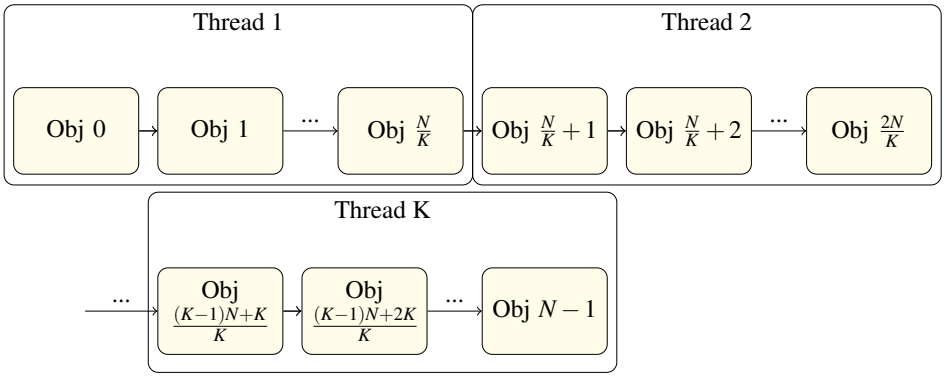


Figure 6. Illustration of Gnuplot parallelization scheme.

5. Parallelism and Performances Evaluation

Evaluating *GraphiX*'s parallelism scheme (or in other words *Gnuplot*'s modified parallel source code) in terms of speed and memory scaling was done by creating a file that contains many polygon-objects and executing *GraphiX* color-map command, which as mentioned is the most time consuming operation (in *Gnuplot*'s source code it is the *load_file* function). First, evaluating *GraphiX*'s thread-scaling capability was tested with {1, 2, 4, 8, 16, 32} threads with a 500MByte file which is roughly 3,850,000 polygons. *GraphiX* was executed on the NUMA architecture with two different options of a new OpenMP 4.5 feature, the thread affinity with the options of *close* and *spread*. The thread affinity *close* option spawns new threads as close as possible to the master thread, thus utilizing the cache-usage and local memory, while the *spread* option spreads the threads across the machine (on different sockets) as much as possible, thus utilizing the memory-bandwidth. As one can see from figure 7, spawning threads close to the master thread yields better speedup, as the algorithm behind the creation of the polygons is better utilized with cache-sharing. The parallelism scheme (with *close*) scales well until 8 cores. Although there is a slight speedup with 16 threads compared to 8, it was found that 8 threads on the NUMA architecture yield the optimal results in terms of performance per dollar. It is also notable that nowadays most desktops and laptops have 8 to 16 cores in a SMP architecture. This further indicates that the parallelism scheme is suited for SMP architecture as *close* ensures the threads are created within the same socket. Additionally, the two trends intersect at 32 threads as this is the number of cores on the machine, thus there is no meaning for *close* or *spread* as they perform in the same way. Furthermore, we included the well-known theoretical Amdahl's law graph to demonstrate that the speedup of *Gnuplot* total execution is almost the same, as the function *load_file* takes approximately 99% of the total execution time.

To evaluate *GraphiX*'s memory-scaling capabilities, similarly to evaluating the thread-scaling capability, *GraphiX* color-map option was tested with files of sizes

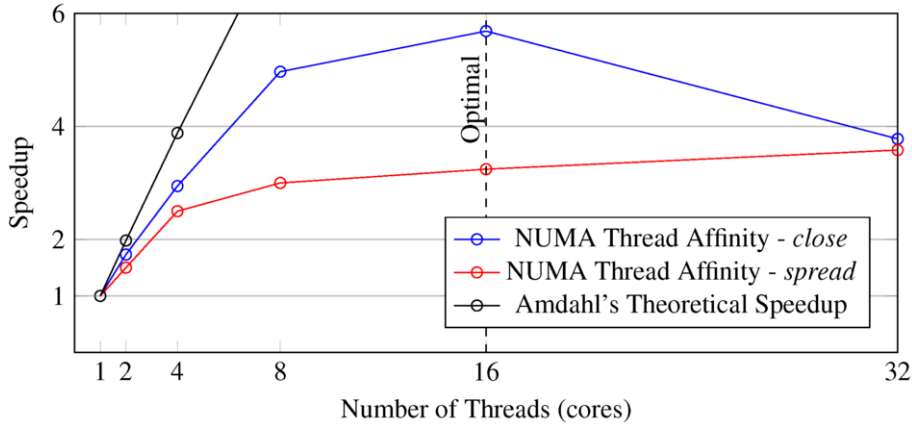


Figure 7. Speedup of the *GraphiX* color-map creation with 500MByte file compared to serial *GraphiX*.

100MByte (∼770,000 polygons), 500MByte (∼3,850,000 polygons), 1GByte (∼7,700,000 polygons), 2GByte (∼15,200,000 polygons) and 4GByte (∼30,000,000 polygons).

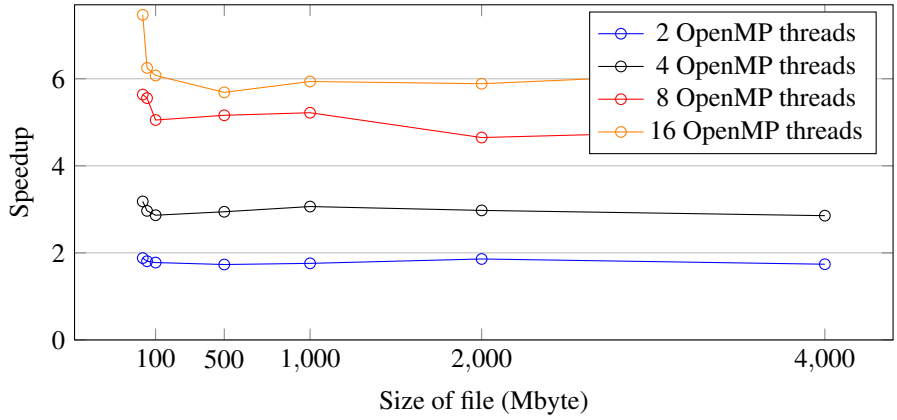


Figure 8. Speedup of the *GraphiX* color-map creation over different file sizes compared to serial *GraphiX* execution with thread affinity *close*.

To evaluate *GraphiX* workload between threads, it was compiled with *Scalasca* [31] - a tool that analyzes and measures a programs runtime behavior. One of the features in *Scalasca* is to identify performance bottlenecks - specifically in our case, the work-balance between the threads - and to verify that there are no bottlenecks in Gnuplot's modified source code. As seen in figure 9 on the third column, the execution time of each thread in the OpenMP region is nearly the same, pointing out that the work-balance between all 16 threads is the same.

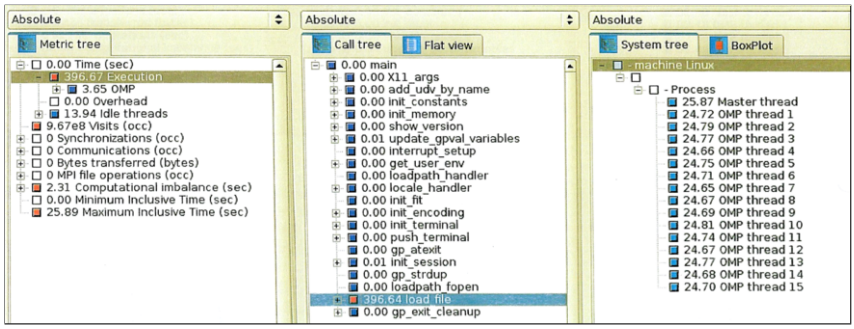


Figure 9. *GraphiX* Parallelisem with 16 threads, exhibiting near perfect load-balancing.

6. Conclusions

As the trend of multi-core architectures is getting more and more popular, the introduction of shared-memory parallelism scheme to software is necessary in order to utilize this architecture. Scientific visualization tools are no exception to this introduction, thus *GraphiX*, a fast two/three dimension visualization tool suited for every multi-core architecture, was created. The most time-consuming option found in GraphiX was the colormap, thus OpenMP directives were introduced to the tool. As shown in section 5 the parallelism scheme's work-balance is perfect and scales well with both the problem size and the number of threads, and achieve a speedup of ∼5.6 at peak with 16 cores.

References

[1] NegevHPC Project. www.negevhpc.com. [Online].
[2] Johannes Kehrer and Helwig Hauser. Visualization and visual analysis of multifaceted scientific data: A survey. *IEEE transactions on visualization and computer graphics*, 19(3):495–513, 2013.
[3] Melanie Tory and Torsten Moller. Human factors in visualization research. *IEEE transactions on visualization and computer graphics*, 10(1):72–84, 2004.
[4] Leading examples of using VisIt at scale. visitusers.org@VisIt_top_50. [Online].
[5] Ewa et al. Deelman. The future of scientific workflows. *The International Journal of High Performance Computing Applications*, 32(1):159–175, 2018.
[6] VisIt homepage. wci.llnl.gov/simulation/computer-codes/visit. [Online].
[7] James Ahrens, Berk Geveci, and Charles Law. Paraview: An end-user tool for large data visualization. *The visualization handbook*, 717, 2005.
[8] WA Bellevue. Tecplot. 360 2018 users manual, 2018.
[9] Marc Snir, Steve Otto, Steven Huss-Lederman, Jack Dongarra, and David Walker. *MPI–the Complete Reference: The MPI core*, volume 1. MIT press, 1998.
[10] MATLAB. https://www.mathworks.com/products/matlab.html. [Online].
[11] Parallel Computing Toolbox. www.mathworks.com/products/parallel-computing.html. [Online].
[12] John D Hunter. Matplotlib: A 2d graphics environment. *Computing in science & engineering*, 9(3):90, 2007.
[13] Gnuplot homepage. www.gnuplot.info. [Online].
[14] John W Eaton, David Bateman, and Soren Hauberg. *GNU Octave manual*. Network Theory Ltd. Bristol, UK, 2002.
[15] Hank Childs. Visit: An end-user tool for visualizing and analyzing very large data. 2012.
[16] Qi Wu, Will Usher, Steve Petruzza, Sidharth Kumar, Feng Wang, Ingo Wald, Valerio Pascucci, and Charles D Hansen. Visit-ospray: Toward an exascale volume visualization system. In *EGPGV*, pages 13–23, 2018.

[17] Intel Xeon website. https://www.intel.com/content/www/us/en/products/processors/xeon.html. [Online].
[18] George Chrysos. Intel® xeon phiâĎć coprocessor-the architecture. *Intel Whitepaper*, 176, 2014.
[19] ParaView website. https://www.paraview.org/. [Online].
[20] Marc Buffat, Anne Cadiou, Lionel Le Penven, and Christophe Pera. In situ analysis and visualization of massively parallel computations. *The International Journal of High Performance Computing Applications*, 31(1):83–90, 2017.
[21] Christoph Lameter et al. Numa (non-uniform memory access): An overview. *Acm Queue*, 11(7):40, 2013.
[22] David Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.
[23] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press, 2008.
[24] Comparison Charts for Intel Core Desktop Processor Family. www.intel.com/content/www/us/en/support/articles/000005505/processors.html. [Online].
[25] E Wes Bethel, Hank Childs, and Charles Hansen. *High Performance Visualization: Enabling Extreme-Scale Scientific Insight*. CRC Press, 2012.
[26] Thomas et al. Williams. gnuplot 5.3. 2017.
[27] Mark Summerfield. *Rapid GUI programming with Python and Qt: the definitive guide to PyQt programming*. Pearson Education, 2007.
[28] Josep Torrellas, HS Lam, and John L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.
[29] Christian Terboven, Dirk Schmidl, Henry Jin, Thomas Reichstein, et al. Data and thread affinity in openmp programs. In *Proceedings of the 2008 workshop on Memory access on future processors: a solved problem?*, pages 377–384. ACM, 2008.
[30] Ruud Van der Pas, Eric Stotzer, and Christian Terboven. *Using OpenMP The Next Step: Affinity, Accelerators, Tasking, and SIMD*. MIT Press, 2017.
[31] Markus Geimer, Felix Wolf, Brian JN Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, 2010.

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200080

When Parallel Performance Measurement and Analysis Meets In Situ Analytics and Visualization

Allen D. MALONY [a], Matt LARSEN [b] Kevin HUCK [a] Chad WOOD [a]
Sudhanshu SANE [c] Hank CHILDS [c]
[a] *Oregon Advanced Computing Institute for Science and Society (OACISS)*
[b] *Lawrence Livermore National Laboratory*
[c] *Department of Computer and Information Science University of Oregon*

Abstract.
Large scale parallel applications have evolved beyond the tipping point where there are compelling reasons to analyze, visualize and otherwise process output data from scientific simulations *in situ* rather than writing data to filesystems for post-processing. This modern approach to in situ integration is served by recently developed technologies such as *Ascent*, which is purpose-built to transparently integrate runtime analysis and visualization into many different types of scientific domains. The TAU Performance System (*TAU*) is a comprehensive suite of tools that have been developed to measure the performance of large scale parallel libraries and applications. TAU is widely-adopted and available on leading-edge HPC platforms, but has traditionally relied on post-processing steps to visualize and understand application performance. In this paper, we describe the integration of Ascent and TAU for two complementary purposes: Analyzing Ascent performance as it serves the visualization needs of scientific applications, and visualizing TAU performance data at runtime. We demonstrate the immediate benefits of this in situ integration, reducing the time to insight while presenting performance data in a perspective familiar to the application scientist. In the future, the integration of TAU's performance observations will enable Ascent to reconfigure its behavior at runtime in order to consistently stay within user-defined performance constraints while processing visualizations for complex and dynamic HPC applications.

Keywords. HPC, performance measurement, runtime visualization.

1. Introduction

Parallel applications developed for large-scale, high-performance computing (HPC) continue to increase in sophistication and complexity. To a great extent, this is driven by the advances in computational modeling of scientific and engineering phenomena that will demand the next-generation hardware technologies fueling the HPC evolution. The ability of applications to harness the greater

computational resources of HPC systems will deliver results of finer precision, higher resolution, and more significant predictive power. The challenge, of course, is to develop applications that can maximize the performance potential of present and future HPC platforms. This represents the flip side of the sophistication and complexity problem. Applications will need advanced parallel programming, language, runtime system, and communication interface technologies to produce programs that can utilize the heterogeneous many-core processors, multi-level memory architecture, and fast interconnect hardware effectively and do so in a performance portable manner.

The dual nature of what defines HPC application success — advanced scientific outcomes and highly-efficient execution — is reflected in tools created to further enhance that success, again in the context of HPC sophistication and complexity. For example, analysis and visualization tools are central to the understanding of science and engineering simulation results. The last 30 years has seen a steady progression from tools generating analysis and visualization products *post hoc* to those running *in situ* with the application [1]. The reasons are consequential of simulation fidelity and HPC scale, making it increasingly intractable to save and process huge modeling data offline [2]. In a similar manner, the importance of parallel performance measurement, analysis, and visualization tools is central to understanding and tuning applications on HPC machines. Contemporaneous to the transition of *in situ* analysis and visualization, runtime performance data introspection, analytics, and feedback are becoming more relevant in performance systems. Again, the reasons are due to HPC idiosyncrasies, including larger performance data size, more factors affecting performance variation, and non-deterministic performance dynamics as a result of asynchronous execution, all making *post mortem* performance methods less productive.

Like *brothers from a different mother*, we consider in this paper the opportunities for the inter-operation of a parallel performance system with an *in situ* analysis and visualization framework. Interestingly, the shared HPC heritage positions these tools today in a place that begs for their integration and supports it. We will demonstrate the merits of the endeavor by focusing on two leading efforts: the *Ascent in situ* project [3] and the *TAU Performance System*® [4]. Ascent is being developed by a multi-institution effort funded by the U.S. Department of Energy (DOE) Exascale Computing Project (ECP) [5] to deliver in situ analysis and visualization technology ECP application teams. TAU provides portable, robust performance measurement and analysis of HPC applications and systems.

There are three perspectives that we will investigate, the first two of which we will describe in this paper. One perspective looks at the use of TAU to instrument, measure, and analyze the Ascent infrastructure. TAU is particularly suited to observing the execution of large-scale software [6] and can directly be applied to characterize the performance of Ascent components. Ascent's performance will correlate with the application-specific analytics and visualization requirements for which it is being used. Based on the performance analysis, Ascent developers will be able to understand inefficiencies and optimize performance for specific usage scenarios.

Another perspective involves the use of Ascent for application performance analytics and visualization. Here our interest is to gather and process perfor-

mance data online (from TAU's measurement of the application measurement and systems-level information) and utilize Ascent's infrastructure to analyze and visualize the data *in situ*. Specifically, we program Ascent's runtime with filters to interface with TAU performance measurements, application-specific values, and system information during periods when Ascent is invoked.

It is reasonable to assume that TAU and Ascent will co-exist in HPC platforms and applications. Thus, an outcome in pursuing the two perspectives above is to validate the cross-leveraging of Ascent and TAU technologies. A third, intriguing perspective comes from more tightly-coupled integration of TAU and Ascent whereby they are being used cooperatively in online tuning and adaptive control of the application. We envision this taking several forms. For instance, suppose that the user constrains *in situ* analytics and visualization to take no more than 10% of the application's execution time. TAU could be used to measure the performance of both the application and Ascent, thereby informing the Ascent infrastructure when corrective action is necessary.

Another possibility is to develop joint analytics that take into account a combination of performance data, application variables, and other execution state to guide policies concerning how the application should advance. Innovations taking place in both Ascent and TAU for supporting application *triggers* [7], *feedback* mechanisms, and autonomic management make this especially salient for integration purposes. Furthermore, there are strong motivations to extend Ascent's and TAU's operation to scientific workflows where *in situ* concerns of computational productivity and performance efficiency involve interactions between multiple simulation modules and workflow components.

Our plan is to evaluate these perspectives with benchmark applications taken from the ECP proxy applications project. These include two of the applications that are part of the Ascent test programs, LULESH and Cloverleaf3D. We ran our experiments on large-scale HPC machines at DOE national laboratories. The main research objectives are to investigate effective methods and explore development strategies for the integration of two state-of-the-art runtime infrastructures for HPC, principally Ascent for *in situ* analytics and visualization and TAU for parallel performance measurement and analysis.

2. Applied Technologies

2.1. Ascent

Ascent [3] is a library for in situ visualization and analysis. Simulation geometry and results are passed to Ascent at runtime in order to generate periodic analysis results without the need to write much larger simulation data to disk for post-mortem analysis. It differs from other in situ libraries in its focus on "flyweight processing," meaning small API, small binary size, small execution overhead, small memory footprint, and few dependencies on other libraries. Ascent supports zero-copy in situ (meaning that it can share memory with a simulation code), and supports parallelism both within a node and across nodes. Its parallelism within a node comes from incorporating the VTK-m project [8], which focuses on

delivering portable performance across many-core architectures for visualization and analysis algorithms. It has been demonstrated good performance on 16,384 cores of the Oak Ridge Titan machine [9], 16,384 GPUs on Lawrence Livermore's Summit machine, and 2,048 GPUs on Oak Ridge's Summit.

2.2. TAU and PerfStubs

The TAU Performance System [4] is a portable profiling and tracing toolkit for performance analysis of parallel programs written in Fortran, C/C++, Java, and Python. TAU is capable of gathering performance information through system-interrupt-based sampling and/or instrumentation of functions, methods, basic blocks, and statements. The instrumentation can be inserted in the source code automatically with a TAU specific compiler wrapper based on the Program Database Toolkit (PDT) [10], dynamically using DyninstAPI [11], at runtime in the Java Virtual Machine or Python runtime, or manually using the instrumentation API (application programming interface). TAU measurements represent first-person, per-OS (operating system) thread measurements for all processes in a distributed application, such as an MPI application. TAU measurements are collected as profile summaries and/or a full event trace.

While application developers are willing to instrument their source code for performance measurement or correctness testing, they are frequently reluctant to add a build dependency for their library or application. The TAU library has many useful features, however can be complex to configure for a given system, and has several configuration options that are mutually exclusive and may require multiple configurations and builds for a given performance experiment. Also, a library such as Ascent is meant to be integrated into larger simulation applications and a complex configuration/build process for "optional" features will prevent adoption of these technologies. Finally, many applications already include some instrumentation to provide rudimentary performance measurement for the purposes of reporting at the end of execution. For these reasons, we have developed a simple instrumentation interface library called *PerfStubs* that attempts to resolve API symbols at link time (using weak symbol overrides) or at runtime (using the dynamic library loader). PerfStubs itself is a small library (one source file) with no additional build dependencies and can quickly be installed on a system.

If the PerfStubs symbols are not defined in the application symbol table at runtime, the instrumentation API will check to see if the function pointer is defined (not-null) and if not, return – an acceptable amount of negligible overhead. If the symbols are resolved at the program startup process, function pointers are assigned the resolved addresses and the PerfStubs API will "feed" any performance measurement tool that implements the tool interface. TAU includes the `tau_exec` script that will preload the TAU shared object libraries and provide the symbol implementations needed by the *perfstubs* interface. Other measurement libraries (e.g. APEX, Score-P, Caliper) could also implement the simple API and be used with the interface. Because the instrumentation interface is pre-processor macro-based, it can be entirely removed at compile time if the *PerfStubs* API is not desired. In fact, the Ascent library already has implemented its own macro-based instrumentation, and the PerfStubs API was easily integrated into that code, as well as into specific places in the Ascent code base, as described in Section 3.

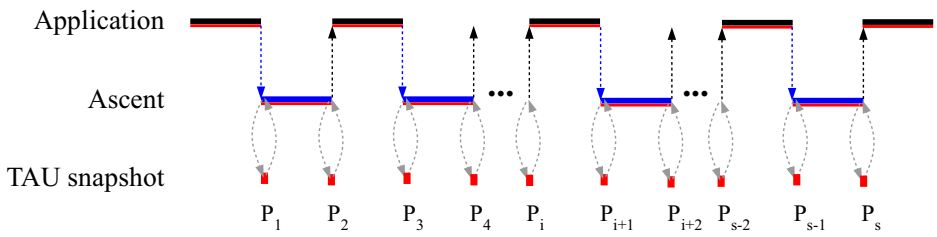


Figure 1. The application (black line) and Ascent (blue line) execute synchronously, with TAU performance measurements (red line) possibly enabled. The transition points between them is an opportunity to link in TAU performance data analysis, including the passing of information to Ascent in a friendly manner for further processing. Profile snapshots are an example of online performance data processing.

2.3. Integration Model

The Ascent operational design provides the basis for the strategy we pursued for TAU integration. Specifically, Ascent is invoked synchronously by the application at certain places during its execution. Ascent then operates while the application is halted. Upon completion of its work, Ascent returns to the application and it continues. This process repeats until the application finishes. Adding TAU to the mix is straightforward. First, in general, we are interested in performance measurement of both the application and Ascent. This is enabled through TAU's instrumentation and measurement mechanisms. Second, gaining access to performance data at runtime is possible with new TAU's plugin architecture. The application/Ascent transition points present an opportunity to look at the current measurements, run analytics, and pass results to Ascent for further processing. In essence, the transitions are used as triggers for TAU analysis.

Our integration design is demonstrated in Figure 1. Shown is a sequence of phases of application execution (black line) followed by Ascent execution (blue line). The dashed arrows indicate the transition points. The red line indicates TAU performance measurement taking place during both the application and Ascent. Dashed gray lines further highlight calls to the TAU plugin (red box) at the beginning and end of Ascent processing. For example, the plugin could be capturing a snapshot of the present performance data state, designated as P_i. These snapshots could be stored for post-mortem analysis and/or processed online.

Profile snapshots can be used to isolate the application's performance from Ascent's performance. From Figure 1, we can use P_i, P_{i+1}, and P_{i+2} to compute the performance for the application phase by "subtracting" P_i from P_{i+1} for every event and metric measured. Similarly, we can compute the performance for the Ascent phase by subtracting P_{i+1} from P_{i+2}. This is similar to the procedure we implemented in the examples described in Section 3. If the TAU plugin stored the computed performance for each phase, it is further possible to compare between phases to detect certain features or changes in performance behavior that might reflect application dynamics.

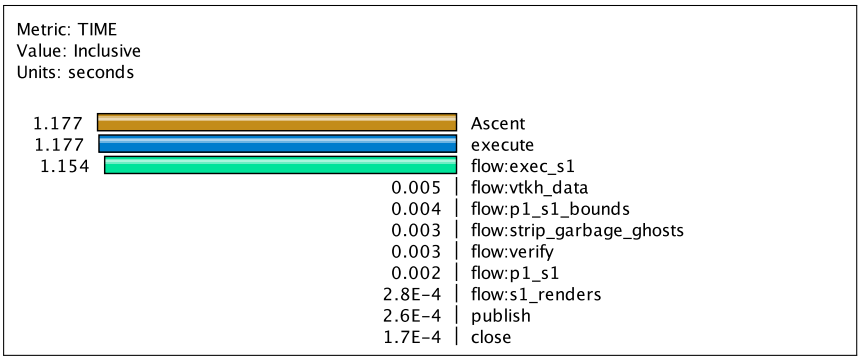


Figure 2. TAU profile data from a representative run of Lulesh integrated with Ascent. MPI and Lulesh timers have been filtered out for space considerations. Only data from rank 0 is shown.

3. Application Examples

3.1. Measuring Ascent with TAU

The first goal of the Ascent-TAU integration is to measure the Ascent library for the purpose of performance evaluation and eventually, guided execution based on performance characteristics (future work). As described in Section 2.2, the Ascent library includes its own instrumentation macros. It was straightforward to integrate `perfstubs_start()` and `perfstubs_stop()` API calls into these macro definitions that are frequently used when Ascent is integrated into a simulation. In addition, primary entry points to the Ascent library were instrumented, such as the `Ascent()` object constructor, `open()`, `close()`, `publish()` and `execute()` operations as well as the `Flow` operation pipeline executed by the `execute()` function. A TAU static phase [12] was also defined around the region of code where the simulation pauses execution in order for Ascent to render simulation output. Figure 2 shows an example TAU profile measurement of the Ascent library integrated with the Lulesh application.

3.2. Visualizing TAU data with Ascent

The second goal of the integration is to use the Ascent library to visualize performance data from the application. This can be achieved using different perspectives. For example, the application performance data can be rendered as a collection of stacked bar charts, representing the per-process performance profile from each of the MPI ranks. Figure 3 (left) shows the performance data from MPI ranks, represented as stacked bar charts. Each color represents a different timed region of the application, showing only the top 5 contributors (the rest are aggregated).

However, a much more interesting perspective is shown when the performance data is projected in the scientific domain. Figure 3 (right) shows the respective time spent in the main computational loop for all sub-domains at the end of the last iteration. What had started as a regular grid has been distorted due to the nature of the Lagrangian computation. Interestingly, the MPI rank computing

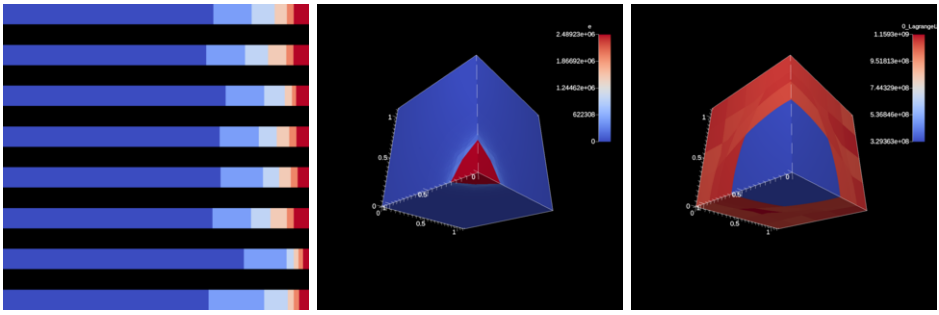


Figure 3. The figure on the left shows output from Lulesh 2.0.3, running with 8 ranks. Each stacked bar represents the performance profile of each rank at the end of the 10th iteration. The other two figures show simulation output from Lulesh 2.0.3 running with 64 MPI ranks, after iteration 4264. The middle figure shows the energy value at the end of the simulation, the right figure shows the relative time spent in the main computation loop of the simulation for each process in the domain, where each process is assigned one of the 4x4x4 (distorted) subdomains.

the region with the largest energy level also spends the least amount of time in the computation.

3.3. Performance Comparison

To further demonstrate the Ascent-TAU integration, we use an in situ algorithm used for flow analysis and visualization. Lagrangian analysis is an in situ data reduction operator used to capture the behavior of time-varying computational fluid dynamics (CFD) simulations. Lagrangian analysis involves the placement of particles and the calculation of particle trajectories across the entire spatial domain. Particle trajectories are calculated using vector fields generated by the simulation code.

In our study, we consider two Lagrangian analysis techniques which offer different workload characteristics. The first Lagrangian analysis technique, referred to as $Lagrangian_{MPI}$, is communication-based and requires all processes to synchronize every cycle. This method involves exchanging particles between nodes as they cross spatial boundaries during the calculation of particle trajectories. The second Lagrangian analysis technique, referred to as $Lagrangian_{BTO}$, is a communication-free method. This algorithm chooses to discard particles that exit the local node's spatial domain.

Our experiments use a hydrodynamics proxy application Cloverleaf3D and are run on Summit[1] at Oak Ridge National Laboratory. In each test, we use 48 MPI tasks across 8 nodes, with each MPI task using a single GPU for particle advection. $Lagrangian_{MPI}$ uses MPI to exchange particles between ranks every cycle. For each technique we considered two workloads for number of particles used: 1.56M and 12.48M. The grid resolution of the Cloverleaf3D simulation is set to 232^3 and we execute 50 cycles each of 0.01 step size. In each case, we save the particle trajectory locations after 10 cycles, i.e., 5 rounds of I/O over 50 cycles.

[1]For Summit technical specs, see https://www.olcf.ornl.gov/summit/

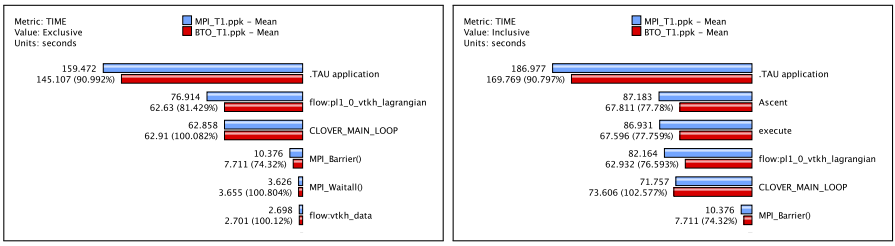


Figure 4. Exclusive and Inclusive time comparisons between MPI and BTO methods. The analysis shows that the BTO method (red) is faster because it generates less synchronization at `MPI_Barrier` and is less computationally expensive in the `pl1_0_vtkh_lagrangian` flow step in analysis.

Lagrangian analysis uses particle advection capabilities from the VTK-m library and is available for use via Ascent.

Figure 4 shows a performance comparison between the two methods when simulating the larger number of particles used (12.48M). As can be seen in both the exclusive (time not including sub-routines/-timers) and inclusive (time including sub-routines/-timers) measurements, the $Lagrangian_{BTO}$ method is less computationally expensive, and therefore less time consuming. the $Lagrangian_{BTO}$ method is also less I/O intensive, and a majority of the difference is summarized by the comparison of the time spent in the `flow:pl1_0_vtkh_lagrangian` step of the processing pipeline.

4. Related Work

Typically, performance data is visualized and represented in the physical and/or logical context of the hardware and/or software resources used in the simulation. Data is organized by processes and threads, and visualized with respect to nodes, network topologies and CPU architectures. Scalasca is a powerful performance system that has extended support in its Cube 3D analysis [13] to show how performance data is distributed across a parallel execution using a computational topology base on a cube topology. TAU provides similar capabilities by mapping performance data to network coordinates captured as metadata [14]. Husain and Gimenez's work on Mitos [15] and MemAxes [16] use memory hierarchy and architecture metadata to provide the context for performance measurements. BoxFish [17] also demonstrated the value of visualizing projections of performance data from multi-dimensional coordinate systems, providing a hierarchical data model for combining visualizations and interacting with data.

Huck et al. [18] integrated performance data with simulation output in order to project the performance data into the scientific domain. However, that technique required post-processing of both the performance and simulation data and did not allow for *in situ* processing. Using the Scalable Observation System [19], performance data was aggregated over SOS and queries were executed to extract performance data and generate VTK output files [20]. Using a similar approach, fusion simulation performance data was aggregated and exported to VTK files [21]. The authors of those papers were forced to re-define the physical

domains in order to map the performance data and/or map the performance data to physical and/or logical coordinates of the allocation. In contrast, the work described in this paper has the ability to reuse the scientific domain defined for visualizing the simulation data. Weber [22] has also visualized performance data at runtime, albeit in a similar perspective to post-mortem trace visualization. Sanderson [23] is the closest work related to this paper, in that they visualized performance data at runtime in the scientific domain.

5. Conclusion and Future Work

In this paper, we have presented the integration between the Ascent in situ visualization and analysis library and the TAU Performance System. We instrumented the Ascent library with an instrumentation coupling library to understand its performance characteristics with TAU, and used the Ascent library to visualize TAU performance data during runtime of proxy applications. We used the TAU instrumentation to compare two Lagrangian analysis implementations on the Summit system. In terms of future work, we believe our approach is very relevant to nascent cost modeling efforts in the scientific visualization community. Among these are works to optimize algorithms [24,25], as well as fit in situ algorithms within time and power budgets [26,27,28]. In each of these efforts, the researchers studied performance a priori, and then used the findings to direct their algorithms. This limits the relevance of their approaches to the cases where they can perform performance studies, digest results, and calibrate their algorithms. With performance measurements, this process could be automated, meaning that researchers could develop algorithms that adapt during runtime and with no a priori performance studies.

References

[1] A. Bauer et al. In Situ Methods, Infrastructures, and Applications on High Performance Computing Platforms, a State-of-the-Art (STAR) Report. In *Eurographics Conference on Visualization (EuroVis)*, June 2016.

[2] U. Ayachit et al. Performance Analysis, Design Considerations, and Applications of Extreme-scale In Situ Infrastructures. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 79:1–79:12, November 2016.

[3] M. Larsen et al. The ALPINE In Situ Infrastructure: Ascending from the Ashes of Strawman. In *In Situ Infrastructures for Enabling Extreme-scale Analysis and Visualization (ISAV)*, pages 42–46, November 2017.

[4] S. Shende and A. Malony. The TAU Parallel Performance System. *International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[5] P. Messina. The Exascale Computing Project. *Computing in Science and Engineering*, 19(3):63—67, 2017.

[6] A. Malony et al. Methods and Strategies for Parallel Performance Measurement and Analysis: Experiences with TAU and HPCToolkit. In *Performance Tuning of Scientific Applications*. CRC Press, 2010.

[7] M. Larsen et al. A Flexible System for In Situ Triggers. In *In Situ Infrastructures for Enabling Extreme-scale Analysis and Visualization (ISAV)*, pages 1–6, November 2018.

[8] K. Moreland et al. VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures. *IEEE Computer Graphics and Applications (CG&A)*, 36(3):48–58, May/June 2016.

[9] J. Kress et al. Comparing the Efficiency of In Situ Visualization Paradigms at Scale. In *ISC High Performance*, pages 99–117, Frankfurt, Germany, June 2019.
[10] K. Lindlan et al. A tool framework for static and dynamic analysis of object-oriented software with templates. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, SC '00, Washington, DC, USA, 2000. IEEE Computer Society.
[11] W. R. Williams et al. Dyninst and mrnet: Foundational infrastructure for parallel tools. In *Tools for High Performance Computing 2015*, pages 1–16. Springer, 2016.
[12] A. D. Malony et al. Phase-based parallel performance profiling. In *PARCO*, pages 203–210, 2005.
[13] D. Lorenz et al. Extending scalasca's analysis features. In *Tools for High Performance Computing 2012*, pages 115–126. Springer Berlin Heidelberg, 2013.
[14] W. Spear et al. An approach to creating performance visualizations in a parallel profile analysis tool. In *Euro-Par 2011: Parallel Processing Workshops*, volume 7156 of *Lecture Notes in Computer Science*, pages 156–165. Springer Berlin Heidelberg, 2012.
[15] B. Husain et al. Relating memory performance data to application domain data using an integration api. In *Proceedings of the 2nd Workshop on Visual Performance Analysis*, page 5. ACM, 2015.
[16] A. A. Gimenez et al. Memaxes: Visualization and analytics for characterizing complex memory performance behaviors. *IEEE Transactions on Visualization and Computer Graphics*, 2017.
[17] K. Isaacs et al. Exploring performance data with boxfish. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1380–1381. IEEE, 2012.
[18] K. A. Huck et al. Linking performance data into scientific visualization tools. In *2014 First Workshop on Visual Performance Analysis*, pages 50–57, Nov 2014.
[19] C. Wood et al. A scalable observation system for introspection and in situ analytics. In *Proceedings of the 5th Workshop on Extreme-Scale Programming Tools*, pages 42–49. IEEE Press, 2016.
[20] C. Wood et al. Projecting performance data over simulation geometry using sosflow and alpine. In *Programming and Performance Visualization Tools*, pages 201–218, Cham, 2019. Springer International Publishing.
[21] J. Y. Choi et al. Coupling exascale multiphysics applications: Methods and lessons learned. In *2018 IEEE 14th International Conference on e-Science (e-Science)*, pages 442–452, Oct 2018.
[22] M. Weber et al. Online performance analysis with the vampir tool set. In *International Workshop on Parallel Tools for High Performance Computing*, pages 129–143. Springer, 2017.
[23] A. Sanderson et al. Coupling the uintah framework and the visit toolkit for parallel in situ data analysis and visualization and computational steering. In *High Performance Computing*, pages 201–214, Cham, 2018. Springer International Publishing.
[24] V. Bruder et al. Prediction-based load balancing and resolution tuning for interactive volume raycasting. *Visual Informatics*, 2017.
[25] M. Larsen et al. Performance Modeling of In Situ Rendering. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC16)*, pages 24:1–24:12, Salt Lake City, UT, November 2016.
[26] M. Dorier et al. Adaptive performance-constrained in situ visualization of atmospheric simulations. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 269–278, Sept 2016.
[27] M. Dorier et al. Damaris/viz: A nonintrusive, adaptable and user-friendly in situ visualization framework. In *2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*, pages 67–75, Oct 2013.
[28] S. Labasan et al. PaViz: A Power-Adaptive Framework for Optimizing Visualization Performance. In *Proceedings of EuroGraphics Symposium on Parallel Graphics and Visualization (EGPGV)*, pages 1–10, Barcelona, Spain, June 2017.

Stream Processing

This page intentionally left blank

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200082

Seamless Parallelism Management for Video Stream Processing on Multi-Cores

Adriano Vogel [a,1], Dalvan Griebler [a,c], Luiz Gustavo Fernandes [a], Marco Danelutto [b]
[a] *School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, Brazil*
[b] *Computer Science Department, University of Pisa, Italy*
[c] *Laboratory of Advanced Research on Cloud Computing (LARCC), Três de Maio Faculty (SETREM), Três de Maio - Brazil*

Abstract. Video streaming applications have critical performance requirements for dealing with fluctuating workloads and providing results in real-time. As a consequence, the majority of these applications demand parallelism for delivering quality of service to users. Although high-level and structured parallel programming aims at facilitating parallelism exploitation, there are still several issues to be addressed for increasing/improving existing parallel programming abstractions. In this paper, we aim at employing self-adaptivity for stream processing in order to seamlessly manage the application parallelism configurations at run-time, where a new strategy alleviates from application programmers the need to set time-consuming and error-prone parallelism parameters. The new strategy was implemented and validated on SPar. The results have shown that the proposed solution increases the level of abstraction and achieved a competitive performance.

Keywords. Parallel Programming, Domain-Specific Language, Stream Processing, Autonomic Computing, Self-adaptive Systems, Seamless Computing.

1. Introduction

A significant amount of applications/systems must gather and analyze data in real-time [2]. Processing continuous stream sequences and responding fast enough requires powerful machines and robust runtimes/languages. Performance optimization for stream processing applications concerns parallelism, which is important because computer architectures have multiple processing units per chip. Therefore, performance gains are usually conditioned to parallel executions.

We have seen the emergence of parallel programming frameworks and libraries for stream processing, such as Intel TBB [11], StreamIt [13] and, FastFlow [4,1]. However, the programming abstractions provided by the parallel programming frameworks remain arguably complex for application programmers[2], which are more concerned with the developing of stream processing algorithms than implementing low-level techniques for

[1]Corresponding Author: adriano.vogel@edu.pucrs.br

[2]The separation of concerns covers the skills and aspects for different types of programmers. Application programmers are software developers focused on the algorithm design and implementation while system programmers are focused on better using computational resources.

exploiting the parallel architecture. Recently, SPar[3] [6] was created for providing additional parallel programming abstractions on stream parallelism targeting multi-core architectures.

Although FastFlow was supported with abstraction concerning parallelism and energy in [12,3], we believe that opportunities exist for novel higher and ready to use parallelism abstractions in the stream parallelism domain. In this work, we aim at providing additional abstractions regarding the definition of the degree of parallelism. In the context of stream processing, manually defining and statically using a degree of parallelism throughout the execution is not suitable. Defining the degree of parallelism tends to be a complicated and time-consuming task because the programmer has to run the same program several times to decide which is the optimal configuration.

Moreover, a significant part of stream processing applications requires recurrent optimizations at run-time. Mainly because stream processing applications have load fluctuations (*e.g.,* performance, environment, or input rates). Consequently, static/unchangeable executions can lead to inefficient resources usage (waste) or poor performance. One way to respond to fluctuations is by adapting the degree of parallelism to improve the performance and/or the efficiency of stream processing applications. Regarding adaptation to load fluctuations, a conventional approach for handling it could be a proactive one, attempting to predict the future load. The challenge is that it is very difficult to predict performance peaks due to the combination of input temporal changes, irregular behavior, and different workload patterns. In this scenario, reactive approaches that are effective by reacting fast, accurately, and run with low computational complexity are a potential solution for enabling suitable adaptations to runtimes.

Abstracting the definition of parallelism configurations is an opportunity for simplifying the process of running parallel applications. In previous work, we presented a new latency-aware self-adaptive strategy [15], where we demonstrated how the degree of parallelism impacts in the latency of stream items. We also provided abstractions [8,14] that enable users/programmers to express service-level objectives (SLO), such as energy bounds, system utilization, and throughput. These implemented strategies require the definition of a target performance or SLOs. Yet, this can be a usability challenge since users/programmers may have no performance/system expertise. However, it is challenging for a completely abstracted strategy to make adaptation decisions without user hints. For instance, approaches that require a definition of a target performance or service objective can decide by comparing such parameters to the actual system/application state.

In this paper, the main contributions can be summarized as the following: 1) We provide a new fully abstracted self-adaptive strategy for the autonomic management of the parallelism in stream processing applications, this new strategy seamlessly manages the parallelism by detecting workload fluctuations; 2) A characterization and comparison of the decision making of the new strategy with respect to other solutions; 3) A validation of the proposed solution with video stream processing applications in terms of performance and resources consumption.

This paper is organized as follows. The background scenario is presented in Section 2. The proposed solution is shown in Section 3. Then, Section 4 shows the experimental results of this paper and Section 5 discusses aspects related to the proposed solution. Finally, the conclusion is presented in Section 6.

[3]SPar home page: https://gmap.pucrs.br/spar

2. Context

The scenario of this study is related to extending SPar DSL features, SPar is briefly described in Subsection 2.1. Moreover, Subsection 2.2 presents relevant related approaches.

2.1. SPar Overview

SPar [6] provides a standard C++ annotation interface, fully compatible with the host language and compiler. In Spar, programmers are invited to simply add annotations on their source code with C++ attributes that represent stream parallelism properties. Then, the compiler interprets the annotations added and generates parallel code with source-to-source transformations.

SPar provides five attributes to exploit key aspects of stream parallelism. The *ToStream* attribute represents the beginning of a stream region, the code block between the *ToStream* and the first *Stage* will run as the first processing stage. More *Stages* can be created inside the *ToStream*. The *Input* attribute allows programmers to define the data to be processed inside a stream region. In contrast, the *Output* attribute is used to define the processing results produced. *Replicate*[4] is the attribute used to define the degree of parallelism. In the code example shown in Listing 1, the data type is a "string" and the input stream comes from a file (read in line 3). This code block is a loop with iterations and a new stream item is read and computed (line 6) on each iteration. In line 5, the attribute *Replicate* defines the degree of parallelism with 4 replicas, which is the static number of replicas used during the entire execution. Finally, in line 8 an output is produced. Figure 1 represents the activity graph with 3 stages of the parallel execution implemented in the runtime according to the annotations introduced in Listing 1.

```
1 [[spar::ToStream]] while(1){
2   std::string data;
3   read_in(data);
4   if(stream_in.eof()) break;
5   [[spar::Stage,spar::Input(data),spar::
        Output(data),spar::Replicate(4)]]
6   { compute(data); }
7   [[spar::Stage,spar::Input(data)]]
8   { write_out(data); }
9 }
```

Listing 1 SPar code example.

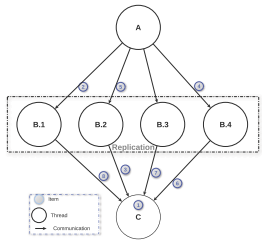


Figure 1. Parallel activity graph.

2.2. Related Approaches

In the related literature exist studies for adaptivity on stream processing. Noteworthy, Sensi *et al.*[12] present a programming interface and runtime called *NORNIR*, which aims at predicting performance and power consumption. *NORNIR* manages the system for maintaining a given power consumption and/or a performance goal. The execution

[4]The term replicate refers to the degree of parallelism in SPar, here the number of replicas and degree of parallelism are used interchangeably.

is managed by adapting system configurations (*e.g.,* number of cores, clock frequency) at run-time. In addition, Matteis and Mencagli [10] presented elastic properties for data stream processing, their goal was to improve performance and energy consumption. The proposed model was implemented along with the FastFlow runtime using one controller thread for monitoring the environment as well as for triggering changes.

Gedik *et al.* [5] and Heinze *et al.* [9] address distributed stream systems. Our approach in contrast targets parallelism abstraction for stream parallelism in multi-core systems. The algorithm implementations provided by related works arguably do not provide sufficient abstractions for application programmers. Differently, our goal is to provide new parallelism abstractions for parallel stream processing applications that is ready-to-use. Our solution require no additional configuration nor drivers installation. Additionally, we propose an improved evaluation of the overhead caused by the adaptivity as we measured the performance and memory consumption. The solution is also compared to the regular static executions.

3. Seamless Parallelism Management

Defining a performance goal is presumably easier for application programmers than defining a low-level parameter of the runtime library. Therefore, in previous works [14,8] we presented strategies that abstracted from users the need to set parallelism parameters related to the number of replicas. The parallelism abstraction was achieved by monitoring the actual application performance and responding to performance violation by continuously adapting the number of replicas. In listing 2 is shown a SPar example with the solution proposed in [14], where the difference compared to the Listing 1 is that the definition of the number of replicas inside the *Replicate* attribute was no longer required. Regarding the adaptation at run-time, in Figure 2 is shown the solution that creates a pool of replicas and dynamically changes the status of the replicas (active, suspended).

```
1 [[spar::ToStream]] while(1){
2   std::string data;
3   read_in(data);
4   if(stream_in.eof()) break;
5   [[spar::Stage,spar::Input(data),spar::
        Output(data),spar::Replicate()]]
6   { compute(data); }
7   [[spar::Stage,spar::Input(data)]]
8   { write_out(data); }
9 }
```

Listing 2 SPar code example.

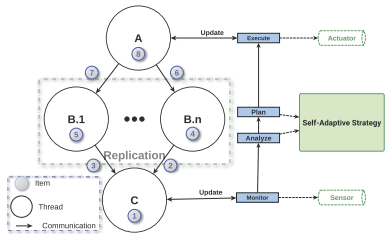


Figure 2. Autonomous Parallelism.

The previous proposed strategies [14,8] require from users the input of performance hints for adapting the number of replicas. However, low-level performance aspects tend to be complex for application programmers. Additionally, stream processing applications are usually long running and with significant load fluctuations, where temporal changes could require different performance objectives. Consequently, we propose a new strategy to manage the execution in an autonomous and seamless way. This new strategy abstracts from users the parameters set. This solution enables a fully seamless execution, which is

achieved by a new decision strategy that monitors the application, detects changes in the workload and performs optimizations in the number of replicas used.

The new decision strategy workflow implements a sensor with a *monitor* running inside the last stage, while the parallelism is adapted by the actuator running in the first stage. The decision (D) whether the number of replicas should be adapted is performed by the Analyze and Plan phases with the following steps: 1) stores data regarding the application performance collected by the monitor; 2) After the execution starts, when it has a minimum of three (a number defined from empirical tests for having a balance between fast and accurate decisions) performance results from monitor iterations, compares this previously collected data with the current performance; 3) If the current performance is significantly lower than the previous one, a new replica (R) is activated (D1); 4) If the current performance is significantly higher than previous results, an active replica is suspended (D2); 5) After the monitor executed 10 iterations with performance results, the regulator enters a new phase where it has more performance data for deciding, which tends to improve the decisions accuracy. Then, for the sake of stability, the average of the previous three throughput collected is compared to the average throughput from all historical data.

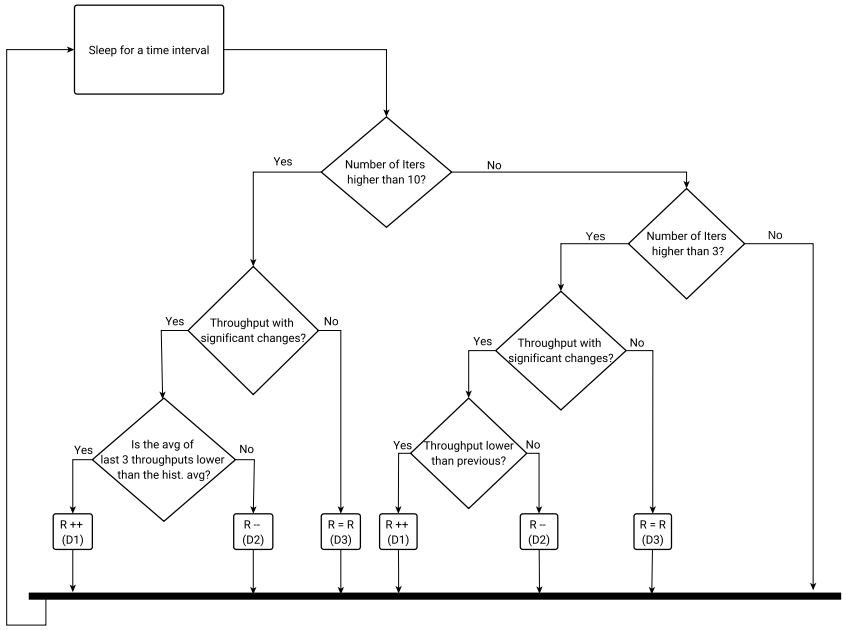


Figure 3. Overview of the Analyze and Plan phases.

Figure 3 shows a high-level representation of decision phases and iterations performed. It is important to note that in addition to decisions 1 and 2 (D1 and D2), there is also the D3 that is performed when the decision is for maintaining the same number of replicas. Moreover, the self-adaptive strategy runs continuously and decides if the number of replicas should be adapted. Although the strategy runs several times and changes the configuration, the adaptations do not affect the regular computations of the appli-

cation. In fact, while the application is running, the strategy periodically runs and then sleeps for a time interval. In this study, we consider 1 second as the default sampling time interval, which allows the strategy to achieve a suitable level of sensitivity to workload fluctuations. Too frequent adaptations can cause instability, while too high sampling times can result in unresponsiveness to changes. Also, two is the minimum number of replicas in a replicated stage, which is a value for minimum parallelism. The maximum number of replicas is defined by the self-adaptive strategy by detecting the machine configuration. The maximum number of replicas is set to at most one application thread per hardware thread, also counting threads from other sequential stages (*e.g.,* Read, write).

4. Evaluation

This section characterizes the new strategy comparing to other solution and to parallel static executions. The new strategy is also evaluated in terms of performance and memory utilization.

4.1. Methodology

The proposed solution was evaluated by implementing it to existing parallel stream processing applications. In fact, real-world applicability was the key criterion used to select the applications. We also selected them based on different characteristics and QoS requirements. In this work, two applications were tested. The first is **Lane Detection** that is an application used on autonomous vehicles to detect road lanes, which is using for maintaining the car on the road. This is performed by reading a video feed from a camera. The road lanes are detected through a sequence of operations where the parallel implementation is like an assembly line composed of three stages, where the second stage is stateless and therefore replicated [7].

Person Recognition is the other tested application that is used to recognize people in video streams. It starts by receiving a video feed and detecting the faces. The faces that are detected are then marked with a red circle and then compared with the training set of faces. When the face detected matches the database one, the face is marked with a green circle. Person recognition's performance was evaluated with a MPEG-4 video (1.36MB - 640x360 pixels) using a training set of 10 images with faces to be recognized in the video [7].

4.2. Characterization

The new seamless strategy is characterized and compared to an existing one [14] that requires a manual definition of a target performance, which was defined to a throughput of 50. The experiments shown here and in the next section were carried out on a multi-core machine equipped with 32 GB of memory, a dual-socket Intel Xeon CPU 2.40GHz (12 cores- 24 threads). The operating system used was Ubuntu Server, G++ v. 5.4.0 with the -O3 compilation flag. The parallel version used the on-demand scheduling policy that is suitable for stream processing, which improves the load balancing by distributing one item to each replica. Moreover, in order to avoid overhead, the emitter and collector stages were placed on dedicated physical cores.

The seamless strategy behavior is characterized in Figure 4 using the Lane Detection application and the input workload was a file of 260 MB [14]. The experiment demon-

strates the throughput and the number of replicas used by each strategy in parallel executions. Moreover, the self-adaptive strategies are compared to static executions running with a fixed number of replicas. For the sake of visual clarity, we only show representative results of static executions with 10 and 20 replicas.

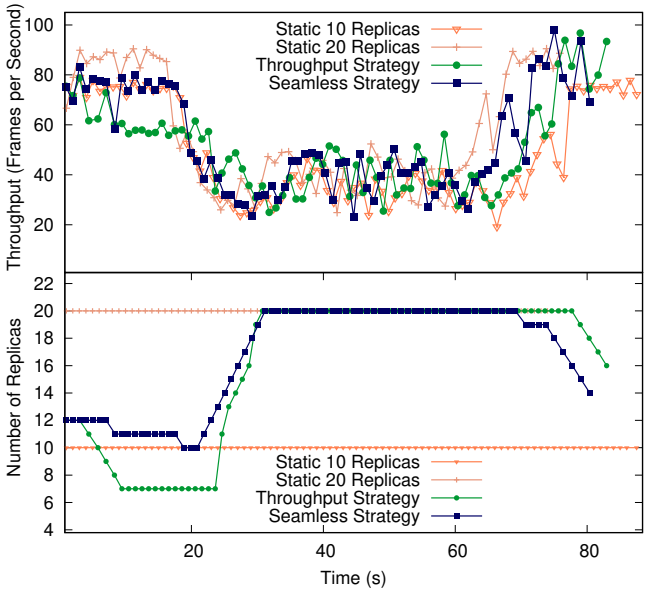


Figure 4. Characterization - Parallel Executions.

In Figure 4, we can observe throughput fluctuations caused by the input workload [14]. The executions with a static number of replicas also presented throughput fluctuations, which emphasizes that the oscillations were caused by input workload instead of the self-adaptive strategies. Regarding the proposed seamless performance strategy, it is important to note that after the first iterations, the throughput increased because of the workload fluctuation. As a consequence, the parallelism actuator changed the number of replicas from 12 to 11. Noteworthy, considering the workload fluctuations around the middle of the execution, the actuator responded to this fluctuation by increasing the number of replicas between the seconds 21 and 36. Another event that highlights the correct sensitivity of this strategy is that the number of replicas was reduced when the execution entered a new phase that increased the throughput (near the second 70).

Comparing the strategies, it is possible to note a similar performance trend caused by the input workload. The strategy based on a manual target performance presented a short settling time, which is notable in the adaptation of the number of replicas after the second 20. The seamless performance strategy required more time to respond to workload fluctuations, which can impact negatively on those applications that demand very fast adaptations. Moreover, it is possible to note in Figure 4 that the seamless performance strategy had a slightly lower execution time, which occurred because this execution had a higher throughput in the first seconds by using more parallel replicas.

4.3. Performance and Overhead

A relevant evaluation of the proposed solution concerns the performance achieved and the resources consumption. The static executions have a simpler runtime that does not perform any adaptation. The advantage tends to be in theory a higher performance. On the other hand, static executions are unresponsive to workload or resources changes. In some cases, with a specific number of replicas, we have seen that static executions achieved the highest performance. However, manually finding the best performing number of replicas configuration is a time-consuming and sometimes counter-productive task. In this section, we present the performance of the adaptive solution compared to static execution. The performance metric is the average throughput, which is a result considering the number of processed items divided by the total time taken in an entire execution. Observe that this is different from the previous performance characterization, where the throughput was collected during different time-steps.

In Table 1 is shown the throughput and memory usage of adaptive and static executions in the Lane Detection application. It is notable that the throughput and memory utilization increases with more replicas. The Seamless performance strategy achieved a slightly higher throughput than the throughput strategy. Comparing to the static executions, the static using more than 16 replicas achieved a higher performance, but these executions also consumed more memory space. The performance of the Seamless strategy is less than 5% lower than the best static execution.

Execution	Average Throughput (FPS)	Memory Usage (MBytes)
Static 10 Replicas	47	807
Static 12 Replicas	48.26	1368
Static 14 Replicas	49.14	1276
Static 16 Replicas	50.31	1648
Static 18 Replicas	50.89	1799
Static 20 Replicas	52.11	2228
Throughput Strategy (50)	48.57	1272
Seamless Strategy	49.67	1327

Table 1. Lane Detection Application

In Table 2 is presented the throughput and memory usage of the Person Recognizer application. In this case, a different performance trend can be seen. The Throughput strategy achieved higher performance, while the Seamless strategy again achieved a throughput similar to the best static executions. Regarding memory usage, the self-adaptive strategies used more memory space on the Person Recognizer application.

5. Discussion

When evaluating higher level abstractions, they often tend to present less performance. However, the best static configuration varies from machines, applications, and workloads. Therefore, tuning all these parameters can be error-prone, time consuming, and may become instantly suboptimal in phase changes or fluctuations. Consequently, a seamless strategy that reacts to workload changes can be a suitable solution that achieves a compromise between abstractions and performance.

Execution	Average Throughput (FPS)	Memory Usage (MBytes)
Static 10 Replicas	12.64	193.6
Static 12 Replicas	12.72	212.60
Static 14 Replicas	12.96	222.10
Static 16 Replicas	12.94	232.10
Static 18 Replicas	13.29	262
Static 20 Replicas	13.22	293.8
Throughput Strategy (15)	13.78	487.7
Seamless Strategy	12.99	448.4

Table 2. Person Recognition Application

There may be overheads as seen in Section 3. A self-adaptive strategy has additional monitoring and actuators entities. For instance, monitoring has a computational cost, but it occurs concurrently while worker replicas are computing tasks. Thereby, the parallel execution is not suspended for monitoring because the monitor runs inside the last stage, which periodically and asynchronously collects statistics. For instance, considering the machine used in the experiments, it took in average only 523 nanoseconds for the monitor implemented in C++ to measure the application throughput. In this case, such a minor amount of time is negligible. The design choices combined with effective mechanisms implemented in the runtime library resulted in a low overhead regarding performance without significantly consuming memory resources.

Moreover, adapting the parallelism of applications at run-time brings additional concerns about safety, which relates to the state and ordering of stream processing applications. Safety is important to ensure that an application can be changed at run-time while preserving its correctness. In SPar, stateless stages can be replicated by default, while stateful executions would require synchronizing a shared internal state. If the ordering of data items is required, the last stage orders the items in SPar. Consequently, self-adapting the parallelism of a stateless stage easily maintains stream items ordered because the last stage is still sequential. Moreover, another aspect of safety is that a worker replica is only suspended after it finishes its computations.

6. Conclusion and Future Work

In this study, we have seen aspects related to the complexities of abstracting parallelism and autonomously managing parallelism configurations at run-time. The new proposed strategy that abstracts the need to set the parallelism and performance configuration shown to be effective. However, the strategy that uses a target performance was able to react faster by comparing the actual performance to the target one.

The alternative that required the definition of a target performance increases the flexibility at the price of additional complexities. On the other hand, running an application transparently increases the abstraction level, but tends to provide less flexibility and lower performance. Some users/programmers may have performance expertise, in which case they may customize their execution by setting system parameters and target performance. However, the provided strategy for seamless execution is designed for users/programmers with no performance and system expertise. Regarding the experimental results, it is important to note that the performance slightly varied among the tested applications, but the trend was similar: the self-adaptive Seamless strategy achieved a com-

petitive performance. Consequently, an implication from the experimental results is that self-adaptivity is suitable for seamlessly managing parallelism configurations.

This study is also limited in some aspects, the implemented strategies control applications with only one replicated stage, use parallel applications with a more complex structure is a future goal. Additionally, our proposed Seamless strategy was validated only with video stream processing applications. Although the applications are representative of stream processing, a different performance trend may be seen under other application characteristics. In the future, we aim at porting related solutions from the literature to our context for comparing them to our strategies.

Acknowledgment This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nivel Superior - Brasil (CAPES) - Finance Code 001, Univ. of Pisa PRA_2018_66 "DECLware: Declarative methodologies for designing and deploying applications", the FAPERGS 01/2017-ARD project called PARAELASTIC (No. 17/2551-0000871-5), and the Universal MCTIC/CNPq N 28/2018 project called SParCloud (No. 437693/2018-0).

References

[1] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. *Fastflow: High-Level and Efficient Streaming on Multicore*, chapter 13, pages 261–280. Wiley-Blackwell, 2014.

[2] H. Andrade, B. Gedik, and D. Turaga. *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. Cambridge University Press, 2014.

[3] D. De Sensi, T. De Matteis, and M. Danelutto. Simplifying Self-Adaptive and Power-Aware Computing with Nornir. *Future Generation Computer Systems*, 87:136–151, 2018.

[4] FastFlow. FastFlow (FF) Website, 2019. last access in Fev, 2019. URL: http://calvados.di.unipi.it/.

[5] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu. Elastic Scaling for Data Stream Processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1447–1463, Jun 2014.

[6] D. Griebler, M. Danelutto, M. Torquati, and L. G. Fernandes. SPar: A DSL for High-Level and Productive Stream Parallelism. *Parallel Processing Letters*, 27(01):1740005, March 2017.

[7] D. Griebler, R. B. Hoffmann, M. Danelutto, and L. G. Fernandes. Higher-Level Parallelism Abstractions for Video Applications with SPar. In *Proceedings of the International Conference on Parallel Computing*, ParCo'17, pages 698–707, Bologna, Italy, September 2017. IOS Press.

[8] D. Griebler, A. Vogel, D. De Sensi, M. Danelutto, and L. G. Fernandes. Simplifying and implementing service level objectives for stream parallelism. *The Journal of Supercomputing*, Jun 2019.

[9] T. Heinze, L. Roediger, A. Meister, Y. Ji, Z. Jerzak, and C. Fetzer. Online Parameter Optimization for Elastic Data Stream Processing. In *Proceedings of ACM Symposium on Cloud Computing*, pages 276–287. ACM, 2015.

[10] T. D. Matteis and G. Mencagli. Keep Calm and React with Foresight: Strategies for Low-latency and Energy-efficient Elastic Data Stream Processing. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming*, pages 13:1–13:12, 2016.

[11] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, 2007.

[12] D. D. Sensi, M. Torquati, and M. Danelutto. A Reconfiguration Algorithm for Power-Aware Parallel Applications. *ACM Transactions on Architecture and Code Optimization*, 13(4):43:1–43:25, Dec 2016.

[13] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proceedings of the International Conference on Compiler Construction*, pages 179–196, 2002.

[14] A. Vogel, D. Griebler, M. Danelutto, and L. G. Fernandes. Minimizing Self-Adaptation Overhead in Parallel Stream Processing for Multi-Cores. In *Euro-Par 2019: Parallel Processing Workshops*, Lecture Notes in Computer Science, page 12, Göttingen, Germany, August 2019. Springer.

[15] A. Vogel, D. Griebler, D. D. Sensi, M. Danelutto, and L. G. Fernandes. Autonomic and Latency-Aware Degree of Parallelism Management in SPar. In *Euro-Par 2018: Parallel Processing Workshops*, Lecture Notes in Computer Science, pages 28–39, Turin, Italy, August 2018. Springer.

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200083

High-Level Stream Parallelism Abstractions with SPar Targeting GPUs

Dinei A. ROCKENBACH [a,1], Dalvan GRIEBLER [a,c], Marco DANELUTTO [b] and Luiz G. FERNANDES [a]

[a] *School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre – Brazil.*
[b] *Computer Science Department, University of Pisa (UNIPI), Pisa, Italy.*
[c] *Laboratory of Advanced Research on Cloud Computing (LARCC), Três de Maio Faculty (SETREM), Três de Maio – Brazil*

Abstract. The combined exploitation of stream and data parallelism is demonstrating encouraging performance results in the literature for heterogeneous architectures, which are present on every computer systems today. However, provide parallel software efficiently targeting those architectures requires significant programming effort and expertise. The SPar domain-specific language already represents a solution to this problem providing proven high-level programming abstractions for multi-core architectures. In this paper, we enrich the SPar language adding support for GPUs. New transformation rules are designed for generating parallel code using stream and data parallel patterns. Our experiments revealed that these transformations rules are able to improve performance while the high-level programming abstractions are maintained.

Keywords. Parallel Programming, Domain-Specific Language, C++11 Attributes, Parallel Patterns, Stream Processing, GPGPU, GPU Programming

1. Introduction

Stream processing applications are present in different domains and are receiving renewed attention in the last decade, mostly because of the importance of stream processing in the core of big data and Internet of Things technologies [4]. In addition to that, the ubiquitous presence of parallel hardware architectures [16] led researchers to develop new tools focused on stream parallelism [22,2,9]. In recent studies [23,13,1,21,7], data parallelism has been exploited via proper software extensions to take advantage of the emerging massively parallel architectures such as GPUs (Graphics Processing Units), which were intentionally designed for data parallelism.

Parallel programming libraries [1,21] offer good performance but lower-level programming abstractions. To meet higher-level abstractions, some tools [23,13,7] prefer to focus on compiler techniques to alleviate the parallel programming burden of GPUs. The problem is that they still require code refactoring to properly exploit the parallelism in

[1]Corresponding Author: {dinei.rockenbach, dalvan.griebler}@edu.pucrs.br

stream processing applications. Alternatively to these options, the SPar[2] domain-specific language [9] provides a productive parallel programming model without adding significant performance overheads for multi-cores [11]. Although that SPar is demonstrating a good compromise between productivity and performance among different stream processing applications for multi-core architectures [11,10,12], automatic code generation for heterogeneous architectures composed of CPU and GPU is still not supported. Our recent investigations using SPar to annotate stream parallelism for multi-cores with manually programming data parallelism for GPUs have shown promising performance results [20]. In this paper, we present an extension to SPar language for supporting GPU data parallelism. We also discuss the compiler transformation rules and evaluate them in a set of experiments. Therefore, our contributions may be summarized as follows:

- we introduce new SPar attributes that enrich the expressiveness and semantics of the language;
- we design new compiler transformation rules suitable to implement both stream and data parallel patterns after proper source code annotations;
- we describe experiments aimed at assessing performance of our transformation rules targeting code generation for heterogeneous parallel architectures.

This paper is organized as follows: Section 2 presents the related work. Section 3 presents two new attributes for the SPar language. Section 4 presents the definitions and compiler transformation rules for the *Map* parallel pattern based on the new attributes. In Section 5 we perform a performance evaluation of the transformation rules. Finally, Section 6 present our conclusions and future work.

2. Related Work

Skeleton-based frameworks like FastFlow [2] and DSLs like StreamIt [22] provide different programming approaches and levels of abstraction to developers. Libraries like Intel's TBB (Threading Building Blocks) [18] also offer support to the parallel implementation of stream processing applications by instantiating the *Pipeline* parallel pattern. Support for GPGPU in FastFlow [3,1] focus in *Stencil* parallel pattern but also allows the implementation of *Map*, *Reduce*, and its combinations. Another C/C++ library based on algorithm skeletons/parallel patterns is represented by SkelCL [21]. On SkelCL, user defined kernels are passed as string to the parallel pattern classes like *Map*, *Zip* (a special case of gather [16]), *Reduce*, and *Scan*. These functions are combined with skeleton code to generate the final OpenCL kernels.

StreamIt [22] is a new imperative programming language focused on stream processing applications. There are only the works of [23] and [13] that extended StreamIt to support CUDA code generation and so far is no longer updated. Sarek (Stream ARchitecture using Extensible Kernels) [5] is a customized language for writing GPGPU kernels in the OCaml language. There is also SkePU 2 [7], which provides a source-to-source compiler tool and a parallel runtime. The source code can be compiled by any C++11 compiler to produce a sequential executable. SkePU 2 compiler generates all CUDA and OpenCL kernel code.

[2]SPar home page: https://gmap.pucrs.br/spar

SPar is unique in the stream parallelism domain regarding the level of abstraction and code intrusion. While sharing similar streaming concepts with FastFlow and StreamIt as well as a slightly similar approach as SkePU 2 (that uses C++11 attributes for some advanced features), no other study aims to provide stream parallelism support without requiring code refactoring and restructuring. There are only solutions that aim for sequential code maintainability for data parallelism like OpenMP [6] and OpenACC [17].

3. New Attributes for SPar

SPar offers five standard C++11 attributes that application programmers may use to annotate the source code [14]. Two of them are "identifiers" (ID): `ToStream` annotation delimits the streaming region and `Stage` delimits each of the computation "phases" (or *stages*). The other three are "auxiliary" (AUX) attributes: `Input` and `Output` are used to specify the stream items, while `Replicate` is used to specify the degree of parallelism for a stage. Listing 1 demonstrates how these attributes are used to annotate sequential source code. This example is computing the Mandelbrot Streaming application. `ToStream` marks where the stream parallelism region starts and also refers to the stream generator stage. Inside the stream computation there are two `Stage`s annotations identifying the stream operators. The data stream dependencies are specified through the `Input` and `Output` attributes. `Replicate` in line 5 indicates the degree of parallelism for that specific stage, running the amount of replicas given as argument in the attribute. The last `Stage` simply shows line by line the Mandelbrot image. It cannot be replicated because `ShowLine` is a stateful operator.

The current SPar attributes are closely related to the stream parallelism domain. Also, they do not express any semantics of the data parallelism properties. Therefore, we created a novel attribute called `Pure` to be used along with the `Stage` attribute list or as identifier inside `Stage` annotated regions. This attribute indicates that the annotated code block is a pure function, "whose output depends only on its input, and does not modify any other system state" [16]. In SPar, a `Stage` or code block will be considered a pure function when it satisfies the following statements to guarantee correct use and correct code generation:

1. The `Pure` region should not have any side effects (*i.e.*, mutation on non-local variables).
2. The `Pure` region should not have execution order dependency (*i.e.*, depending on the values modified by previous iterations).
3. The `Pure` region should not access any global variable that are not listed in the `Input` attribute.

From the programmer perspective, the `Pure` attribute is just another attribute allowing to identify data parallelism inside the `Stage`. On the other hand, the compiler transformation rule identifies that this region/function can be computed in parallel over multiple data. It is up to the compiler decide which parallel architecture (GPU or multi-core) generate the stream parallelism with data parallelism code. Section 4 will describe the design of the compiler transformation rules to target data parallelism for GPUs.

In our previous work, we evaluated different parallel programming models when implementing stream and data parallelism combined [19]. One lesson learned is that

fine-grained stream processing may not generate enough workload to properly exploit massively parallel architectures such as GPUs. Thus, some stream processing applications may not provide the expected performance scalability when using GPUs. For these cases, we are providing the possibility to express stream batches in SPar through the new auxiliary attribute for the `ToStream`, named `Batch`. The programmer can specify as argument the size of the batch with literal or integer variable. In principle, this is the amount of stream items to be computed at once by the subsequent stages, which can be or not a `Pure` stage. In short, `Batch` will now allow programmers to define the stream item granularity.

```
1  void mandel(int dim,int niter,double init_a,double init_b,double step) {
2    [[spar::ToStream, spar::Batch(size), spar::Input(dim, niter, init_a,
       init_b, step)]]
3    for (int i=0; i<dim; i++) {
4      unsigned char *img = new unsigned char[dim];
5      [[spar::Stage, spar::Pure, spar::Input(dim, niter, init_a, init_b,
       step, i, img), spar::Output(img), spar::Replicate(workers)]]
6      for (int j=0; j<dim; j++) {
7        double im = init_b + (step * i);
8        double cr;
9        double a = cr = init_a + step * j;
10       double b = im;
11       int k = 0;
12       for (k=0; k<niter; k++) {
13         double a2 = a * a;
14         double b2 = b * b;
15         if ((a2+b2) > 4.0) break;
16         b = 2 * a * b + im;
17         a = a2 - b2 + cr;
18       }
19       img[j] = (unsigned char) 255-((k*255/niter));
20     }
21     [[spar::Stage, spar::Input(img, dim, i)]] {
22       ShowLine(img, dim, i);
23       delete img;
24     }
25   }
26 }
```

Listing 1: Mandelbrot Streaming annotated with SPar using the new attributes.

Observe that none of these attributes are actually related to underlying parallel architecture. They were intentionally designed to express data parallelism properties such as data granularity (`Batch`) and single instruction for multiple data (`Pure`). If we compare to existing data parallel programming models such as OpenMP [6], `Batch` has a meaning to OpenMP *chunk* and `Pure` has a meaning similar to OpenMP *parallel for* where every computation inside the region can be performed in parallel and independently. For this work, data parallelism will be purposely exploited in GPUs. However, these new attributes are also open for further investigations and research on multi-core and cluster parallel architectures. The central point is that the programmer is no longer obliged to reason about the parallel architecture details when developing its application such as required by CUDA or OpenCL. SPar's compiler and transformation rules handle this complexities in place of programmers through its high-level annotation-based language.

Listing 1 exemplifies the use of our new attributes in the existing SPar annotations. We are just adding the `Pure` attribute in the `Stage` annotation in line 5 of Listing 1 because the `for` loop in line 6 is a pure function. Moreover, we inserted the `Batch` attribute in line 2, allowing the control of the stream granularity. It is worth point out that the application latency and throughput are directly impacted by the use of this attribute. However, the programmer may test and choose the best configuration (size of the batch) that fits the performance requirements. Section 5 will discuss the performance impacts of these attributes.

4. New Compiler Transformation Rules for SPar

In his PhD Thesis, Griebler [8,9] designed the original structure of the SPar language. The SPar attributes are combined in annotation schemas, which trigger transformation rules in the compiler. These transformation rules are based on previous definitions. We present current SPar definitions and transformation rules for *Pipeline* and *Farm* parallel patterns and built upon those to generate novel definitions and transformation rules for the *Map* parallel pattern.

To express the definitions and transformation rules, Griebler created a particular notation: `ToStream` and `Stage` attributes are represented by T_{id} and S_{id}, where *id* represents a numeric identifier. `Input`, `Output`, and `Replicate` attributes are represented by I_i, O_i, and R_n, respectively. I_i and O_i may contain a list of typed variables a_i, and n denotes the integer number of replicas for `Replicate` argument. To denote a code block with one or more statements it is used $\square_{id}$. The scope of the sentence is denoted by $\{...\}$. The annotations that contain one identifier attribute and optionally a list of auxiliary attributes, are denoted using [[...]] [14].

The current definitions and transformation rules of SPar [9] are generating the stream parallel patterns *Pipeline* and *Farm*. They are implemented in the SPar compiler for transforming the annotated code into C++ code with calls to the FastFlow library, which provide classes and built-in functions for implementing these parallel patterns. Griebler uses functional semantics to define the *Farm* and *Pipeline* patterns: $farm(E,W,C)$ has arguments E (Emitter, the stream item scheduler), W (Worker, that compute stream items), and C (Collector, which gather results/stream items from the workers), where E, C, and W receive a $\square_{id}$ as argument; and $pipe(S_1,S_2,...)$ has two or more stages, which can be $\square_{id}$ or $farm$ instances. The current SPar transformation rules can generate a combination of these patterns based on the annotation schema.

In this paper, we focus in the combination of data stream and data parallel patterns. First, we concentrate only in the *Map* pattern, as it is the simplest and widely used pattern for data parallelism [16]. Using functional semantics, we defined this pattern as: $data = map(\square^{p}_{id})$, where $\square^{p}_{id}$ is the pure function or code wrapper that computes over multiple data independently and transforms them into $data$. This $data$ can be a list, vector, or an array of data.

Before introducing our novel definitions and transformation rules, we extend the previous SPar notation: P_i denotes a `Pure` attribute and $\forall_{id}(\square_{id})$ denotes a `for` statement [14] with a code block. The `Batch` attribute is not discussed in this section since it only changes the data management and does not interferes in the pattern generation.

There are six definitions presented in [9] related to the transformation rules for generating *Pipeline* and *Farm* parallel patterns from SPar annotations. Table 1 presents our

new definitions aimed at supporting the transformation rules with the *Map* pattern. The changes with respect to the definitions from [9] are highlighted in blue color.

Table 1. Definitions for transformation rules adapted from [9].

D_0	A *generic stage* ψ is a $\square$ annotated with S that contains in its attribute list R_n and O_i and therefore requiring a further $\square$ gathering its results.
D_1	A $\square$ may appear as a *pipe* stage, as an E or C stage in a $farm$ or as the *map* function if its annotation list S does not contain the attribute R_n.
D_2	A $\square$ where the first statement is a $\forall_{id}$ annotated with a S followed by P in its attribute list becomes a *map*.
D_3	A $\square$ with an annotation list S containing an R_n attribute may appear as a W stage in a $farm$ or as the parameter of a *map*.
D_4	When D_1 and D_2 applies on a $\square$, a *map* is instantiated as a *pipe* stage.
D_5	When D_2 and D_3 applies on a $\square$, a *map* is instantiated inside the W stage of the $farm$.
D_6	A $\forall(\square)$ annotated with only P inside a S's code block becomes a *map* nested into a *pipe*'s S or $farm$'s W.
D_7	T is a *map* when a $\square$ has $\forall_0$ as the first statement annotated with T, where right after this $\forall_0$ there is only a single $\square$ which is a $\forall_1$ annotated with S and contains P in its attribute list.
D_8	A T is a $farm$ when the first S annotation contains R_n in the attribute list of a maximum two S.
D_9	A T is a *pipe* when the first S does not have R_n in the attribute list or when there are more than two Ss.
D_{10}	A $farm$ is a stage of *pipe* when D_7 cannot be applied and $\square$ is annotated with S that contains R_n in the attribute list.

From the original SPar transformation rules [9], we take the fourth transformation rules as an example to demonstrate the combination of stream and data parallelism. Adding P_i and considering a $\forall(\square)$ as the code block of the first S in the transformation rule 4 from [9], we can apply D_2 and D_4 to obtain Rule 1. In this case, we combine the *Map* and *Pipeline* patterns. Each stream item produced by the first *pipe* stage instantiate the *map* to exploit data parallelism.

$$[[T_0]]\{\square_0, [[S_0, P_i]]\{\forall(\square_1)\}\} \Rightarrow pipe(\square_0,\ map(\square_1)) \tag{1}$$

Similarly, if we take transformation rule 3 from [9], add P_i and consider a $\forall(\square)$ as the code block of the first S, we can apply D_2, D_3, and D_5 to obtain Rule 2. In this case, a new parallel pattern is generated, combining *Farm* with workers instantiating the *Map* pattern.

$$\begin{gathered}[[T_0]]\{\square_0, [[S_0, O_i, R_n, P_i]]\{\forall_0(\square_1)\}, [[S_1]]\{\square_2\}\} \\ \Downarrow \\ farm(E(\square_0),\ W(map(\square_1)),\ C(\square_2))\end{gathered} \tag{2}$$

Adding P_i in the fifth rule from [9], with $\forall(\square)$ as the code block, we can apply D_2, D_3, and D_5 and obtain Rule 3. This Rule combines three parallel patterns: *Pipeline*, *Farm*, and *Map*. The *pipe* is generated based on D_9. The $farm$ appears as a *pipe* stage (based on D_10) and the *map* pattern comprises the $farm$'s worker stage (W), according to D_5.

$$[[T_0]]\{\square_0, [[S_0]]\{\square_1\}, [[S_1, R_n, P_i]]\{\forall(\square_2)\}\}$$
$$\Downarrow \tag{3}$$
$$pipe(\square_0,\ farm(E(\square_1),\ W(map(\square_2))))$$

Definition D_6 allows P to be employed as ID attribute, which provides more flexibility to SPar application. If only part of the last `Stage` from Rule 3 is a pure function, P_i could be applied in this specific code block, as demonstrated by Rule 4.

$$[[T_0]]\{\square_0, [[S_0]]\{\square_1\}, [[S_1, R_n]]\{\square_2, [[P_i]]\{\forall(\square_3)\}\}\}$$
$$\Downarrow \tag{4}$$
$$pipe(\square_0,\ farm(E(\square_1),\ W(\square_2, map(\square_3))))$$

Rule 5 applies D_7 to generate a single *map* pattern from a T attribute. The presence of the `Pure` attribute in this specific code structure simplifies the implementation and allows the exploitation of a pure data parallelism.

$$[[T_0]]\{\forall_0([[S_0, P_i]]\{\forall_1(\square_0)\})\} \Rightarrow map(\square_0) \tag{5}$$

5. Performance Evaluation

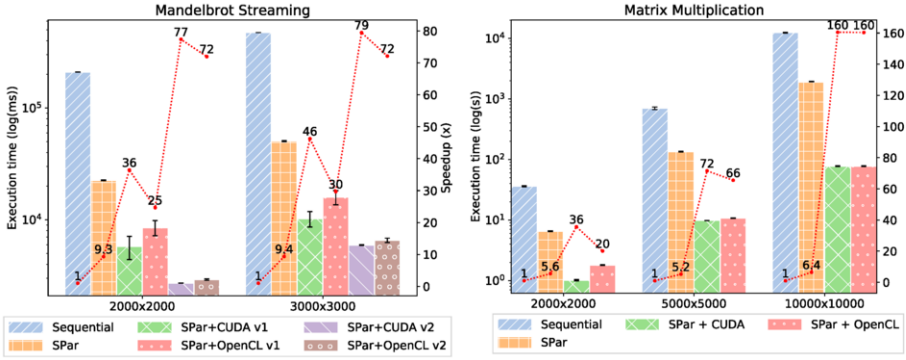
To support data parallelism for GPUs in SPar, we decided to generate CUDA and OpenCL code. Although we aim to offer multi-GPU support, we focused in a single GPU in these experiments. The transformation rules were created to generate parallel patterns, however, CUDA and OpenCL does not offer support for a structured parallel programming approach. Therefore, we implemented the *Map* pattern by transforming the pure function $\square^{p}_{id}$ into a GPU kernel, where each thread launched goes throughout this code wrapper. Then, inside this GPU kernel each thread get its global index and computes over a different data index. Consider N as the number of iterations of the annotated $\forall$ and $max\_threads$ the maximum number of threads per block available in the GPU. On the absence of the `Batch` attribute, we launch N threads divided in $N/max\_threads$ blocks. When the `Batch` attribute is used, we launch $N * batch\_size$ threads on each kernel call. In this case, we modify the previous and next computation stages to generate and consume stream items of size $batch\_size$. Prior to implementing the transformation rules in the compiler, we evaluated them by generating the CUDA and OpenCL code manually when our transformation rules were triggered. Therefore, this performance evaluation was carried out by manually performing the work that would be done by the compiler. We want to show that our transformation rules could work in a future compiler implementation.

To integrate stream parallelism on the multi-core and data parallelism with CUDA, we added a `cudaStream` object on each stream item to properly define dependencies between data transfer and kernel function calls. For the OpenCL runtime, we added a `cl_kernel`, a `cl_command_queue`, and a `cl_event` object on each stream item. The `cl_kernel` are not thread-safe [15] and must be allocated for each thread. The `cl_command_queue` allows overlapping kernel and memory copies between different

stream items and the `cl_event` is used to synchronize asynchronous calls between different pipeline stages.

The experiments ran in a server machine that has an Intel(R) Core(TM) I9-7900X @ 3.3GHz (10 cores and 20 threads), 32GB of RAM memory and two Titan XP GPUs (although we use only one of them in this experiments) with compute capability 6.1 and each one has 12GB of memory. The system was running on Ubuntu OS (kernel 4.15.0-43-generic). All programs were compiled using -O3 compiler flags. The software used were G++ 9.1, NVCC 10.0.130, OpenCL 1.2, SPar, and FastFlow. We chose the best degree of parallelism and batch sizes by empirical testing the applications under different configurations. The SPar implementations ran with 20 worker replicas and versions combining SPar with CUDA or OpenCL ran with 10 worker replicas in the annotated regions with the `Replicate` attribute. Each version was executed five times and the average execution time is plotted, while error-bars show the standard deviation.

We present experiments using two pseudo-applications: Mandelbrot Streaming and Matrix Multiplication. We focused in traditional HPC metrics such as execution time and speedup to observe the applications scalability and performance.



(a) Mandelbrot Streaming.

(b) Matrix Multiplication.

Figure 1. Experiments Results.

We tested two workloads for the Mandelbrot Streaming application: generating 2000x2000 and 3000x3000 fractal images, both with a maximum of 100,000 iterations per single pixel. This fractal image size represent 4,000,000 and 9,000,000 numbers between -2.125-1.5 and 0.875+1.5. The annotation schema presented in [8] ("SPar" in Figure 1a) shows 9.3× and 9.4× of speedup with respect to the sequential version for our workloads. The simplest modification is to insert the `Pure` as auxiliary attribute in the first `Stage` annotation. This annotation schema triggers Transformation Rule 2 and generates the *Farm* with *Map* pattern. This version shows 36× and 46× speedup for the CUDA runtime ("SPar+CUDA v1" in Figure 1a), and 25× and 30× speedup for the OpenCL runtime ("SPar+OpenCL v1") with respect to the sequential times. These versions presented an unexpectedly high standard deviation, which is due to data transfer between CPU and GPU.

As demonstrated by our previous study, a single Mandelbrot line does not generate enough workload to fully utilize the GPU [19]. Therefore, we can add `Batch` attribute in the `ToStream` annotation to achieve further performance improvements, as demonstrated

in Listing 1. Using a batch size of 30 for this annotation schema yields 77× and 79× speedup for the CUDA runtime ("SPar+CUDA v2" in Figure 1a), and 72× speedup in both workloads for the OpenCL runtime ("SPar+OpenCL v2"). Each stream item of these batch versions are calculating 30 lines of the Mandelbrot set in a single kernel call. The performance improvement is explained by this batch of lines utilizing the massive parallelism of the GPU.

We discuss here the matrix multiplication presented in [8] as an example of data-parallel algorithms. We ran this experiment with matrices of 2000x2000, 5000x5000, and 10000x10000 32-bit elements. The `Pure` attribute can be added to the `Stage` annotation of [8] to trigger Transformation Rule 5. It generates a single *Map* pattern for this annotation schema.

The SPar annotated version presented in [8] for multi-core architecture ("SPar" in Figure 1b) achieved 5.6×, 5.2×, and 6.4× speedup with respect to the sequential version in our tests. Adding the `Pure` attribute in the `Stage` annotation yields 36×, 72×, and 160× speedup for the CUDA runtime ("SPar+CUDA") in the three workloads. For the OpenCL runtime ("SPar+OpenCL") this modification yielded 20×, 66×, and 160× speedup.

6. Conclusion

In this paper, we enriched the expressiveness of the SPar language to target data parallelism for GPUs, which can in the future be extended to multi-core and cluster architectures. After, we created new compiler transformation rules for generating the *Map* parallel pattern along with the existing stream parallel patterns. Lastly, we carried out a performance evaluation using two pseudo-applications. The outcome is that the language simplicity was maintained (Listing 1) while performance improvements were obtained with respect to only generating parallel code to multi-core without data parallelism support (Figure 1). Using this work's transformation rules, we obtained very similar performance results with respect to our previous work [20], where these applications were fine tuned and manually programmed.

We aim in the future add new definitions and transformation rules that can potentially support more data parallel patterns. We also intend to implement these transformation rules in the SPar compiler to automatically generate GPU parallel code based on our high-level annotations. Given the many challenges of GPU programming, we intend to propose or use an intermediate library such as SkePU and SkelCL to support functional data parallel patterns for GPUs in C++ that are fully compatible with stream parallelism to alleviate the compiler work.

Acknowledgment This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nivel Superior - Brasil (CAPES) - Finance Code 001, Univ. of Pisa PRA_2018_66 "DECLware: Declarative methodologies for designing and deploying applications", the FAPERGS 01/2017-ARD project called PARAELASTIC (No. 17/2551-0000871-5), and the Universal MCTIC/CNPq N 28/2018 project called SParCloud (No. 437693/2018-0). We also thank LARCC for the computing resources.

References

[1] M. Aldinucci, M. Danelutto, M. Drocco, P. Kilpatrick, C. Misale, G. Peretti Pezzi, and M. Torquati. A parallel pattern for iterative stencil + reduce. *Journal of Supercomputing*, pages 1–16, 2016.

[2] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. *FastFlow: high-level and efficient streaming on multi-core*, chapter 13, pages 261–280. John Wiley & Sons, 1st edition, 2017.

[3] M. Aldinucci, G. P. Pezzi, M. Drocco, C. Spampinato, and M. Torquati. Parallel visual data restoration on multi-GPGPUs using stencil-reduce pattern. *International Journal of High Performance Computing Applications (IJHPCA)*, 29(4):461–472, 2015.

[4] H. C. M. Andrade, B. Gedik, and D. S. Turaga. *Fundamentals of Stream Processing*. Cambridge University Press, New York, USA, 2014.

[5] M. Bourgoin, E. Chailloux, and J.-L. Lamotte. Efficient Abstractions for GPGPU Programming. *International Journal of Parallel Programming*, 42(4):583–600, Aug. 2014.

[6] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, Jan. 1998.

[7] A. Ernstsson, L. Li, and C. Kessler. SkePU 2: Flexible and Type-Safe Skeleton Programming for Heterogeneous Parallel Systems. *International Journal of Parallel Programming*, 46(1):62–80, Feb. 2018.

[8] D. Griebler. *Domain-Specific Language & Support Tools for High-Level Stream Parallelism*. PhD thesis, Faculdade de Informática - PPGCC - PUCRS, Porto Alegre, Brazil, June 2016.

[9] D. Griebler, M. Danelutto, M. Torquati, and L. G. Fernandes. SPar: A DSL for High-Level and Productive Stream Parallelism. *Parallel Processing Letters*, 27(01):1740005, Mar. 2017.

[10] D. Griebler, R. B. Hoffmann, M. Danelutto, and L. G. Fernandes. Higher-Level Parallelism Abstractions for Video Applications with SPar. In *Parallel Computing is Everywhere, Proceedings of the International Conference on Parallel Computing*, ParCo'17, pages 698–707. IOS Press, Sept. 2017.

[11] D. Griebler, R. B. Hoffmann, M. Danelutto, and L. G. Fernandes. High-Level and Productive Stream Parallelism for Dedup, Ferret, and Bzip2. *International Journal of Parallel Programming*, pages 1–19, Feb. 2018.

[12] D. Griebler, R. B. Hoffmann, J. Loff, M. Danelutto, and L. G. Fernandes. High-Level and Efficient Stream Parallelism on Multi-core Systems with SPar for Data Compression Applications. In *XVIII Simpósio em Sistemas Computacionais de Alto Desempenho*, pages 16–27. SBC, October 2017.

[13] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke. Sponge: Portable Stream Programming on Graphics Engines. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, pages 381–392. ACM, 2011.

[14] ISO/IEC. *ISO/IEC 14882:2017 - Programming languages – C++*. International Organization for Standardization, Geneva, Switzerland, 5 edition, Dec. 2017. https://www.iso.org/standard/68564.html.

[15] The Khronos Group. *The OpenCL Specification*, Oct. 2018. v2.2-8.

[16] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier Science, 2012.

[17] OpenACC-Standard.org. *The OpenACC® Application Programming Interface*, Nov. 2018. v2.7.

[18] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Series. O'Reilly Media, 2007.

[19] D. A. Rockenbach, C. M. Stein, D. Griebler, G. Mencagli, M. Torquati, M. Danelutto, and L. G. Fernandes. Stream Processing on Multi-Cores with GPUs: Parallel Programming Models' Challenges. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2019.

[20] C. M. Stein, D. Griebler, M. Danelutto, and L. G. Fernandes. Stream Parallelism on the LZSS Data Compression Application for Multi-Cores with GPUs. In *27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, Pavia, Italy, February 2019. IEEE.

[21] M. Steuwer, P. Kegel, and S. Gorlatch. SkelCL - A portable skeleton library for high-level GPU programming. In *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1176–1182. IEEE, May 2011.

[22] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In R. N. Horspool, editor, *Compiler Construction*, pages 179–196, Berlin, Heidelberg, 2002. Springer.

[23] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil. Software pipelined execution of stream programs on GPUs. In *Proceedings of the 7th International Symposium on Code Generation and Optimization*, CGO '09, pages 200–209, Mar. 2009.

Mini-Symposia

Energy-Efficient Computing on Parallel Architectures (ECOPAR)

This page intentionally left blank

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200085

Energy-Efficiency Evaluation of FPGAs for Floating-Point Intensive Workloads

Enrico CALORE [a,1], Sebastiano Fabio SCHIFANO [a,b]
[a] *INFN Ferrara, Italy*
[b] *University of Ferrara, Italy*

Abstract. In this work we describe a method to measure the computing performance and energy-efficiency to be expected of an FPGA device. The motivation of this work is given by their possible usage as accelerators in the context of floating-point intensive HPC workloads. In fact, FPGA devices in the past were not considered an efficient option to address floating-point intensive computations, but more recently, with the advent of dedicated DSP units and the increased amount of resources in each chip, the interest towards these devices raised. Another obstacle to a wide adoption of FPGAs in the HPC field has been the low level hardware knowledge commonly required to program them, using Hardware Description Languages (HDLs). Also this issue has been recently mitigated by the introduction of higher level programming framework, adopting so called High Level Synthesis approaches, reducing the development time and shortening the gap between the skills required to program FPGAs wrt the skills commonly owned by HPC software developers. In this work we apply the proposed method to estimate the maximum floating-point performance and energy-efficiency of the FPGA embedded in a Xilinx Zynq Ultrascale+ MPSoC hosted on a Trenz board.

Keywords. HPC, Energy, EuroEXA, FPGA, Roofline

1. Introduction

Despite the relevant programming effort to program FPGAs using Hardware Description Languages (HDLs), these devices were already used in the past as accelerators, in the context of physics simulations and scientific computing in general. Anyhow, FPGAs were used just for specific applications, where the performance benefit could be of several order of magnitude, with respect to the use of ordinary general purpose processors [1,2], or whenever strong timing constraint were required, in order to justify the programming efforts.

In addition to the programming complexity, applications developed to use FPGAs, once designed for a specific architecture, could not be trivially ported to different FPGAs and a complete re-implementation was commonly required to run on other kind of processors, such as CPUs.

Thus, programming complexity and a weak code portability had been historically a consistent barrier towards a wide adoption of FPGAs in the context of the HPC scien-

[1]Corresponding Author: Enrico Calore, INFN Ferrara, Via Saragt 1, 44121 Ferrara, Italy; E-mail: enrico.calore@fe.infn.it.

tific community. Nowadays, thanks to the consolidation of higher level programming approaches, mainly thanks to better transcompilers and synthesis tools, FPGAs usage is being investigated also in a wider area of applications [3]. FPGAs, can now be programmed using languages such as OpenCL [4] or High-Level Synthesis (HLS) paradigms, sometimes referred to as "C synthesis", allowing for an algorithmic description of the programmer desired behavior, which can be interpreted and transcompiled by automatic software tools into a Register-Transfer Level (RTL) design using an HDL. HDL can later be synthesized to the gate level using a logic synthesis tool.

Clearly, such a higher level of abstraction, often translate to inefficiencies and a waste of FPGA resources wrt a low level manual programming of the HDL code. Despite of this, the highly reduced programming effort required, in conjunction with a faster design space exploration, and a much higher software portability, make this approach a very attractive one. In particular, for pre-existing applications developed for ordinary CPUs and accelerators, the possibility to use directive based languages, allowing to just annotate a plain C code with *pragmas*, is particularly appealing [5].

These new programming possibilities, in conjunction with the high intrinsic hardware parallelism, and an increased amount of computing resources and DSPs for floating-point operations [6,7], are increasing the interest towards their usage as accelerators in HPC installations [6]. A great interest in FPGAs, in the HPC context, is also towards their possible high energy-efficiency – thanks to their intrinsic parallelism and low clock frequencies – wrt ordinary processors, and also GPUs for some applications [8].

Before embracing any code porting activity to target FPGA devices in the HPC context, one would like to assess in advance which is the expected performance and energy-efficiency of such devices. Theoretical estimations exist, but empirical benchmarks are preferred, since actual codes rarely reach theoretical estimations, in particular when high level languages are used. In this work we try to address this problem, looking for a method to experimentally estimate the maximum achievable performance on an FPGA device, programmed using an high level programming approach. The method we follow is based on the theoretical foundations of the Roofline Model [9], but is strongly experimental and has been already adopted to study the obtainable performance on CPUs, GPUs and many-core processors, such as the different Xeon Phi models [10]. In particular, we have implemented a synthetic benchmark, named FPGA Emprirical Roofline (FER). Our tool is based on the same priciples of the Emprical Roofline Toolkit [10], which empirically determines the peak memory bandwidth and peak computing performance, that are needed to measure the machine characteristics for the Roofline Model.

FER has been developed using the OmpSs programming model [11], a high level language based on directives, allowing the same code to target FPGAs as accelerators, but also other devices, such as GPUs or multi-core CPUs [12]. This choice is due to the will to estimate a realistic maximum performance for an actual HPC code, developed using high level programming tools such the ones commonly used in the scientific HPC community.

Moreover, in this work, we use FER also to assess the energy-efficiency of the tested FPGA device, using an external power meter and thus obtaining also the maximum reachable FLOP/Watt.

1.1. The EuroEXA Project

This work has been performed in the context of the EuroEXA project *Co-designed innovation and system for resilient exascale computing in Europe: from application to silicon*, which is a H2020 FET HPC project funded by the EU commission with a budget of ≈ 20M€. The aim of the project is to develop a prototype of an *exascale* level computing architecture suitable for both compute- and data-intensive applications, delivering world-leading energy-efficiency. To reach this goal this project proposes to adopt a cost-efficient, modular integration approach enabled by: novel inter-die links; FPGAs to leverage data-flow acceleration for compute, networking and storage; an intelligent memory compression technology; a unique geographically-addressed switching interconnect and novel Arm based compute units. As main computing elements, multi-core Arm processors combined with Xilinx UltraScale+ FPGAs are going to be adopted, to be used both as compute accelerators and to implement a high bandwidth and low-latency interconnect between computing elements.

Form the software platform point of view, EuroEXA provides five high-level programming frameworks that enable FPGA-accelerated computing: Maxeler MaxCompilerMPT [2], OmpSs@FPGA [11], OpenStream [13], SDSoC or SDAccel [3] with OpenCL, and Vivado High Level Syntesis [4]. These frameworks are used to implement several key HPC applications across climate/weather, physics/energy and life-science/bioinformatics scientific domains. More details about the EuroEXA project can be obtained from its website: https://euroexa.eu.

The aim of this paper is to develop a tool able to estimate the maximum computing and energy-efficiency performance obtainable by FPGAs devices, in order to use such performance upper bounds in the evaluation task of the EuroEXA project, in order to assess the optimization level of applications ported to such architecture.

1.2. The OmpSs Programming Model

In this work we adopted the directive based language, named OmpSs [14], developed by the Barcelona Supercomputing Center (BSC). OmpSs is very similar to the widely known OpenMP, and in fact it can be considered a forerunner of OpenMP, where new features get introduced and developed before getting pushed in the OpenMP standard.

One of such extensions is in fact the possibility to offload a function to an FPGA device, using OmpSs@FPGA [11]. This is consequently one of the tools selected to be used in the framework of the EuroEXA project to exploit the embedded FPGAs as accelerators and allow to define task functions to be offloaded to such accelerators, providing the automatic generation of a wrapper code handling data copies and dependencies. The code to be actually offloaded to the FPGA get transformed into a bitstream by the VivadoHLS toolchain, allowing the programmer also to use HLS directives in the source code.

Thanks to the OmpSs directives, simply changing the offload target (which directly affect the final compiler to be used), the same source code can be compiled for several architectures, possibly targeting different accelerators. This approach is very interesting

[2] https://www.maxeler.com/solutions/low-latency/maxcompilermpt/
[3] https://www.xilinx.com/products/design-tools/all-programmable-abstractions.html
[4] https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

to allow HPC scientific software developers to target FPGA devices using a language which is much more close to the common programming paradigms adopted in their communities.

2. FPGA Empirical Roofline

Following the same approach used by the Empirical Roofline Toolkit (ERT) [10] developed at the Berkeley Lab, we have developed a tool named FPGA Empirical Roofline (FER), using OmpSs@FPGA [11]. This custom benchmarking code is able to extract most of the floating-point throughput from an FPGA, using its DSPs to perform floating-point computations. FER has the capability of tuning the computational intensity (i.e., the FLOP/Byte ratio) of a kernel function to be run on the FPGA device, allowing to find experimentally the maximum floating-point throughput and at the same time to measure the maximum memory bandwidth, allowing to obtain all the characteristics required to produce an empirical Roofline plot of such device.

This tool is meant to assess an FPGA maximum compute and bandwidth performance, in order to use these figures as an experimental upper bound for the performance obtainable by a generic floating-point intensive application.

We report in Listing 1 the main kernel of the FER tool. The compilation and bitstream synthesis processes are actually carried out in several steps, but from a high level point of view, the two initial *omp* directives instruct OmpSs@FPGA to: generate an FPGA bitstream containing a single instance of the *kernel* function, to be loaded into the FPGA; avoid implicit data transfers, in order to allow to directly access the DRAM from the FPGA; treat the *kernel* function, from the host side (i.e., the Arm CPU), as a task, reading from the *input* array and writing to the *output* array.

In practice, during this code execution, one element of the input array is read for each FPGA clock cycle and moved into the FPGA itself, where it is entered into a pipeline computing a chain of dependent FMA operations. In particular *FLOP_ELEM / 2* Fused Multiply Accumulate (FMA) operations are performed for each element, which accounts for a computational intensity of *FLOP_ELEM / sizeof(element_data_type)*. Multiple elements can be concurrently into the pipeline accordingly to the pipeline depth. Once an element exits the pipeline it is written back to the DRAM into the *output* array.

Therefore, once the pipeline is filled, for each FPGA clock cycle one element is read from *input*, and one element is written to *output*, thus we can infer that at each clock cycle the FPGA is performing *FLOP_ELEM/2* FMA, which translate to *FLOP_ELEM* FLOP, allowing us to compute the FLOP/s performance.

To find which is the maximum obtainable performance it is enough to increase the *FLOP_ELEM* value up to the point at which the FPGA resources are not enough to synthesize a working bitstream. To additionally estimate the energy consumption, it is enough to measure the average power drain while executing the FER tool and then divide the FLOP/s by the average power drain to obtain the GFLOP/s per Watt metric.

3. Results

Running the FER benchmark, compiled and synthesized using OmpSs@FPGA v1.3.2 and Vivado 2017.3, we have been able to evaluate the maximum single- and double-

Listing 1: The main *kernel* function of the FER tool annotated with OmpSs and HLS directives in order to compute *FLOP_ELEM / 2* Fused Multiply Accumulate (FMA) operations for each *input* array elements. This accounts for a computational intensity of *FLOP_ELEM / sizeof(element_data_type)*.

```
#pragma omp target no_copy_deps num_instances(1) device(fpga)
#pragma omp task in([C_DIM]input) out([C_DIM]output)
void kernel( const data_t * input, data_t * output) {

 size_t i;

 for (i = 0; i < DIM; i++){

    #pragma HLS pipeline II=1

    const data_t alpha = 0.5;
    const data_t elem = input[i];

    data_t beta = 0.8;

#if (FLOP_ELEM & 2) == 2           // add 2 FLOPs
     FMA(beta,elem,alpha);
#endif

#if (FLOP_ELEM & 4) == 4           // add 4 FLOPs
     REP2(FMA(beta,elem,alpha));
#endif

    ...

    output[i] = beta;

 }

}
```

precision floating-point performance of the 16nm FinFET+ FPGA embedded in the Xilinx Zynq UltraScale+ XCZU9EG MPSoC hosted on a Trenz TE0808 board. Using the same software tool we have measured also the maximum bandwidth between the FPGA and the host DRAM, which is the main host system memory (i.e., the same memory used by the Arm CPU).

Moreover, powering the XCZU9EG MPSoC using an external power supply and a custom developed current monitoring tool, similar to the one used in [15], we have also measured the computing module power drain, being able to estimate the floating-point energy-efficiency of such device.

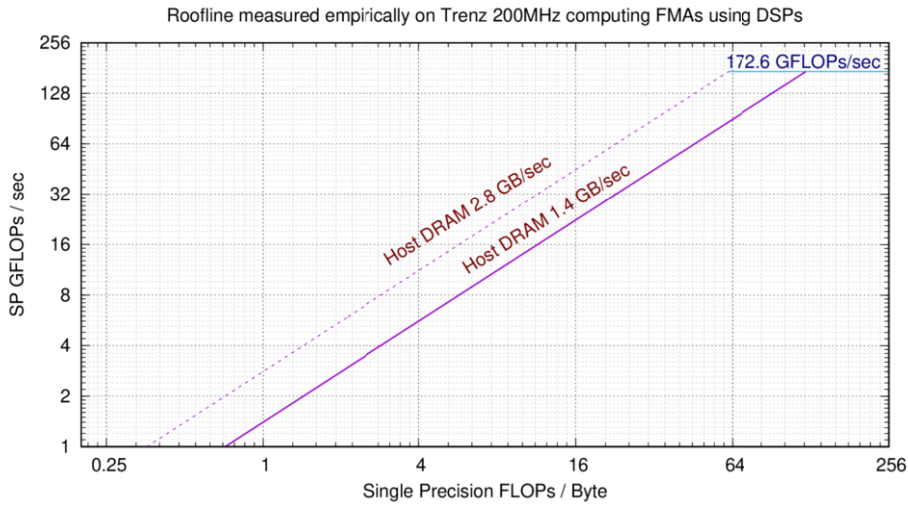


Figure 1. Experimental Roofline plot of the FPGA embedded on a Xilinx Zynq UltraScale+ MPSoC clocked at 200 MHz. The maximum performance has been reached using 99.4% of the available DSPs, computing 487 FMAs (i.e., 974 FLOP) per element, in single-precision.

3.1. Performance

Thanks to the measurements obtained with FER, we could produce the experimental Roofline plots shown in Fig. 1 for single-precision elements and Fig. 2 for double-precision elements.

As shown in Fig. 1, using 99.4% of the available DSPs on this FPGA, we have been able to reach 172.6 GFLOP/s in single-precision, computing 487 FMA operations on each element fed to the FPGA, processing one element per FPGA clock cycle. Concerning the bandwidth, when using single-precision elements, for each FPGA clock cycle we move one 32-bit element in and one 32-bit element out, reaching 1.4 GB/s of bi-directional bandwidth between the DRAM and the FPGA. Anyhow, a wider bus is available [16] and in fact a factor 2 improvement in the bandwidth can be reached using 64-bit elements, as demonstrated empirically using double-precision elements. This is the reason why in Fig. 1 we plot the measured bandwidth using the single-precision version of FER, but also the maximum measurable bandwidth, which is 2.8 GB/s. In an actual code, to reach the maximum bandwidth, would be enough to pack elements in 64-bit structures.

Concerning the double-precision run, as shown in Fig. 2, Using 96.6% of the available DSPs on this FPGA, we have been able to reach 63.5 GFLOP/s, computing 179 FMA operations on each element fed to the FPGA, processing one element per FPGA clock cycle. From the bandwidth point of view, reading and writing a double-precision element per FPGA clock cycle, this translate to a bandwidth of 2.8 GB/s.

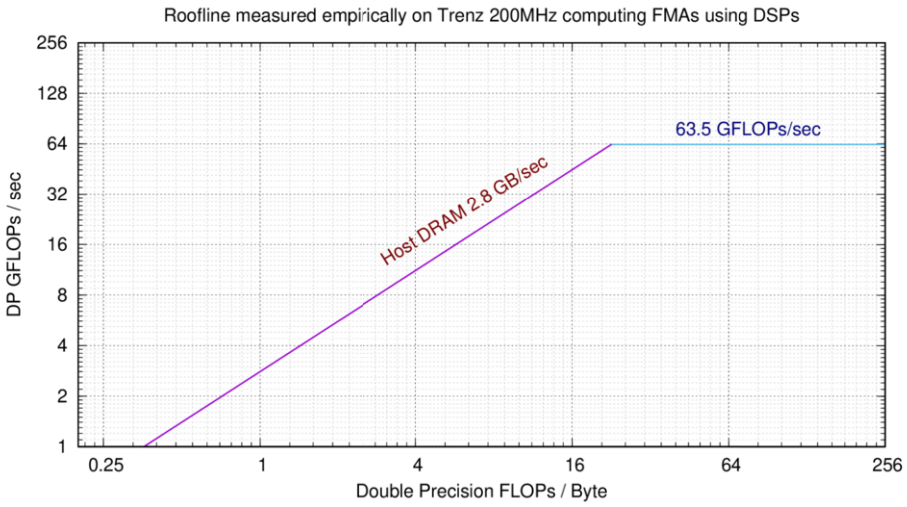


Figure 2. Experimental Roofline plot of the FPGA embedded on a Xilinx Zynq UltraScale+ MPSoC clocked at 200 MHz. The maximum performance has been reached using 96.6% of the available DSPs, computing 179 FMAs (i.e., 358 FLOP) per element, in double-precision.

3.2. Energy

The Trenz TE0808 development board used in this work, does not embed usable power meters, thus it is not possible to measure the power drain of just the hosted UltraScale+ MPSoC. Without hardware modifications, one may measure the whole board power drain, but this would take into account a lot of ancillary electronics which is not actually used to perform the computations required by the FER synthetic benchmark.

To measure only the compute module power drain, we have disabled the on-board voltage regulators which provide to the MPSoC the required 3.3V power supply. Then we provided this voltage from an external bench top power supply, monitoring the drained current with a custom DAQ system, shown in Fig. 3 and sampling at 20KHz. Running the FER benchmark for several iterations, while monitoring the current drain, we have been able to compute the average power drain in Watt at a sustained GFLOP/s rate, allowing us to compute the GFLOPs/Watt metric.

In particular, we measure 21.0 GFLOPs/Watt for single-precision floating-point operations, and 4.7 GFLOPs/Watt for double-precision ones. This highlight the fact that this hardware is not only more efficient from the performance point of view in computing single-precision operations ($\approx 2.7\times$ faster), but is also much more energy-efficient in this condition ($\approx 4.5\times$ more energy-efficient). On the other side, the double-precision performance is less attractive, probably due to the DSP design, which has been optimized for 32-bit operations. Moreover, given the available DSPs, the implementation of double-precision operations can be obtained in different ways [17,18], with different tradeoff

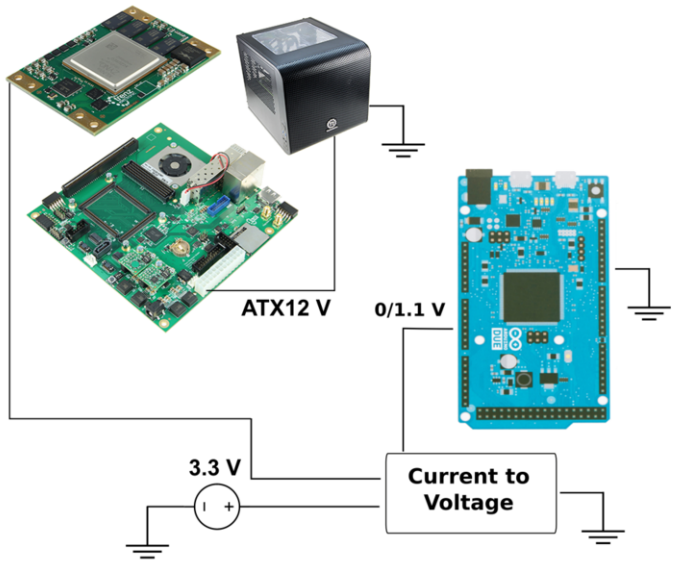


Figure 3. Schema of the custom current monitoring system used to measure the power drain of the MPSoC compute module, without taking into account the whole development board power drain. The power supplying lane from the development board to the compute module has been interrupted and an external power supply is used to drive the required 3.3V. This external lane pass through an Hall effect current-voltage transducer, which can be sampled at 20KHz by the ADC of an Arduino DUE board.

between resources, area and latency, thus a further investigation would be needed to understand if the default one is the optimal one form the energy-efficiency point of view.

4. Conclusions and future works

We can summarize in Tab. 1 all the information obtained running FER on an actual FPGA hardware (i.e., the 16nm FinFET+ XCZU9EG MPSoC), including the energy-efficiency metrics collected thank to the use of the custom DAQ system described in Sec. 3.2.

From these results, we can predict that adopting such FPGA as an accelerator, scientific applications requiring single-precision floating-point computations, could reach a reasonable performance, but more interestingly, a high energy-efficiency.

As a comparison, Intel Broadwell CPUs released in the same period, built around 14nm technology, can reach a theoretical double-precision peak performance of $\approx 100-400$ GFLOP/s, with a Thermal Design Power (TDP) of $\approx 80-160$ Watt, according to the models (number of cores and core frequency), using AVX2 vector instructions [5]. This translate to a performance in the same order of magnitude as the one empirically reached with the tested FPGA, but requiring a power consumption of one order of magnitude higher. Using FMA3 instructions Broadwell CPUs could actually reach a higher theoretical performance, but the used synthesis tool do not produce any special handling

[5] https://www.microway.com/knowledge-center-articles/detailed-specifications-of-the-intel-xeon-e5-2600v4-broadwell-ep-processors/

Table 1. Results concerning the execution of the FER synthetic benchmark, compiled using OmpSs@FPGA and Vivado 2017.3 and run on the FPGA embedded in the XCZU9EG MPSoC.

	DSPs usage [%]	FLOP per Elem.	Performance [GFLOP/s]	Avg Power [Watt]	Energy-efficiency [GFLOPs/W]
Single Precision	96.63	974	172.6	8.2	21.0
Double Precision	99.44	358	63.5	13.6	4.7

of FMAs [6]. In the FPGA in fact each FMA is implemented as a multiplier plus an adder and given the fact that actual applications do not perform just FMAs, it seems more fair to compare to CPUs not using special FMA instructions.

Concerning scientific applications requiring double-precision floating-point computations, these would enjoy a lower energy-efficiency using such FPGA as a target accelerator. This could be caused by the hardware architecture of the DSPs contained in the FPGA, which were not optimized for double-precision operations, but may also be related to a floating-point IP Core, used by the synthesis tools, which could be improved [18].

As future works we plan to further investigate the performance and energy-efficiency of this device, producing different empirical Roofline plots for different FPGA clock frequencies. We will also try to increase the double-precision performance, initially trying newer versions of the Xilinx Vivado tools, since the floating-point IP Core seems to be improved in the last months and also specialized implementations for FMA operations seems now to be available [7]. We also plan to experiment software techniques, such as the use of extended-precision [19], evaluating its performance, energy-efficiency and usability for actual HPC scientific applications, wrt regular double-precision.

Moreover, we will further investigate the performance and energy-efficiency of the Arm cores [20] embedded in the same MPSoC, in order to have a full characterization of this device, assessing the possibility to share the computations between the Arm CPU and the FPGA. Eventually we plan also to provide a more comprehensive comparison with other architectures commonly adopted in the HPC field.

Acknowledgments: E.C. has been supported by the European Union's H2020 research and innovation programme under EuroEXA grant agreement No. 754337. This work has been done in the framework of the EuroEXA EU project and of the COKA, and COSA, INFN projects. We thank Angelo Cotta Ramusino and Stefano Chiozzi, from Electronic Services of INFN Ferrara, for the help in setting up the power measuring system.

References

[1] Baity-Jesi, M., et al.: Janus II: A new generation application-driven computer for spin-system simulations. Computer Physics Comm. 185(2), 550 – 559 (2014), doi:10.1016/j.cpc.2013.10.019

[2] Güneysu, T., Kasper, T., Novotný, M., Paar, C., Rupp, A.: Cryptanalysis with copacobana. IEEE Transactions on Computers 57(11), 1498–1513 (Nov 2008), doi:10.1109/TC.2008.80

[3] Vanderbauwhede, W., Benkrid, K.: High-performance computing using FPGAs, vol. 3. Springer (2013), doi:10.1007/978-1-4614-1791-0

[6] We would measure the same performance executing a generic sequence of additions and multiplications.

[7] https://www.xilinx.com/support/documentation/ip_documentation/ru/floating-point.html#zynquplus

[4] Zohouri, H.R., Maruyama, N., Smith, A., Matsuda, M., Matsuoka, S.: Evaluating and optimizing opencl kernels for high performance computing with fpgas. In: SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 409–420 (Nov 2016), doi:10.1109/SC.2016.34

[5] Bonati, C., Calore, E., Coscetti, S., D'Elia, M., Mesiti, M., Negro, F., Schifano, S.F., Tripiccione, R.: Development of scientific software for HPC architectures using OpenACC: the case of LQCD. In: The 2015 International Workshop on Software Engineering for High Performance Computing in Science (SE4HPCS). pp. 9–15. ICSE Companion Proceedings (2015), doi:10.1109/SE4HPCS.2015.9

[6] Véstias, M., Neto, H.: Trends of cpu, gpu and fpga for high-performance computing. In: 2014 24th International Conference on Field Programmable Logic and Applications (FPL). pp. 1–6 (Sep 2014), doi:10.1109/FPL.2014.6927483

[7] Jin, Z., Finkel, H., Yoshii, K., Cappello, F.: Evaluation of a floating-point intensive kernel on fpga. In: Heras, D.B., Bougé, L., Mencagli, G., Jeannot, E., Sakellariou, R., Badia, R.M., Barbosa, J.G., Ricci, L., Scott, S.L., Lankes, S., Weidendorfer, J. (eds.) Euro-Par 2017: Parallel Processing Workshops. pp. 664–675 (2018)

[8] Nurvitadhi, E., Venkatesh, G., Sim, J., Marr, D., Huang, R., Ong Gee Hock, J., Liew, Y.T., Srivatsan, K., Moss, D., Subhaschandra, S., Boudoukh, G.: Can fpgas beat gpus in accelerating next-generation deep neural networks? In: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. pp. 5–14. FPGA '17, ACM (2017), doi:10.1145/3020078.3021740

[9] Williams, S., Waterman, A., Patterson, D.: Roofline: An insightful visual performance model for multicore architectures. Commun. ACM 52(4), 65–76 (Apr 2009), doi:10.1145/1498765.1498785

[10] Lo, Y.J., Williams, S., Van Straalen, B., Ligocki, T.J., Cordery, M.J., Wright, N.J., Hall, M.W., Oliker, L.: Roofline model toolkit: A practical tool for architectural and program analysis. In: Jarvis, S.A., Wright, S.A., Hammond, S.D. (eds.) High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation. pp. 129–148. Springer International Publishing, Cham (2015)

[11] Filgueras, A., Gil, E., Alvarez, C., Jimenez, D., Martorell, X., Langer, J., Noguera, J.: Heterogeneous tasking on SMP/FPGA SoCs: The case of OmpSs and the Zynq. In: 2013 IFIP/IEEE 21st International Conference on Very Large Scale Integration (VLSI-SoC). pp. 290–291 (Oct 2013), doi:10.1109/VLSI-SoC.2013.6673293

[12] Bosch, J., Filgueras, A., Vidal, M., Jimenez-Gonzalez, D., Alvarez, C., Martorell, X.: Exploiting parallelism on GPUs and FPGAs with OmpSs. In: Proceedings of the 1st Workshop on Autotuning and adaptivity approaches for energy efficient HPC Systems. p. 4. ACM (2017), doi:10.1145/3152821.3152880

[13] Pop, A., Cohen, A.: Openstream: Expressiveness and data-flow compilation of openmp streaming programs. ACM Transactions on Architecture and Code Optimization (TACO) 9(4), 53 (2013), doi:10.1145/2400682.2400712

[14] Duran, A., Ayguadé, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: OmpSs: a proposal for programming heterogeneous multi-core architectures. Parallel processing letters 21(02), 173–193 (2011)

[15] Calore, E., Schifano, S.F., Tripiccione, R.: Energy-performance tradeoffs for HPC applications on low power processors. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 9523, 737–748 (2015), doi:10.1007/978-3-319-27308-2_59

[16] Xilinx: Zynq ultrascale+ mpsoc: Embedded design tutorial (2018), uG1209

[17] Jaiswal, M.K., Cheung, R.C.: Area-efficient architectures for double precision multiplier on FPGA, with run-time-reconfigurable dual single precision support. Microelectronics Journal 44(5), 421 – 430 (2013), doi:https://doi.org/10.1016/j.mejo.2013.02.021

[18] Jaiswal, M.K., So, H.K.: DSP48E efficient floating point multiplier architectures on FPGA. In: 2017 30th International Conference on VLSI Design and 2017 16th International Conference on Embedded Systems (VLSID). pp. 1–6 (Jan 2017), doi:10.1109/ICVD.2017.7913322

[19] Thall, A.: Extended-precision Floating-point Numbers for GPU Computation. In: ACM SIGGRAPH 2006 Research Posters. SIGGRAPH '06 (2006), doi:10.1145/1179622.1179682

[20] Laurenzano, M., Tiwari, A., Jundt, A., Peraza, J., Ward, W.J., Campbell, R., Carrington, L.: Characterizing the Performance-Energy Tradeoff of Small ARM Cores in HPC Computation. In: Euro-Par 2014 Parallel Processing, LNCS, vol. 8632, pp. 124–137 (2014), doi:10.1007/978-3-319-09873-9_11

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200086

GPU Acceleration of Four-Site Water Models in LAMMPS

Vsevolod NIKOLSKIY [a,b,1], Vladimir STEGAILOV [a,b]
[a] *National Research University Higher School of Economics Moscow, Russia*
[b] *Joint Institute for High Temperatures of Russian Academy of Sciences Moscow, Russia*

Abstract. In this work, a new algorithm was developed for calculating the four-point water model TIP4P on graphics accelerators. It was designed as a part of the flexible molecular dynamics modeling package LAMMPS in the library module "GPU". In this paper we describe two approaches to implement the TIP4P model for GPU: 1) to divide the related computations between CPU and GPU; 2) to compute the interaction fully on the GPU. We verify the program, benchmark and profile it. The achieved speedup of interaction computation is about x7, acceleration of the entire calculation of about 55%.

Keywords. TIP4P, LAMMPS, atomistic modeling, accelerator, empirical potential

1. Introduction

Molecular dynamics is an extremely powerful tool in modern science. It is used in a wide variety of fields, including materials science, biology, theoretical physics, and many others. Engaged in the multiscale approach, molecular dynamics is necessary for parameterization of next-order models.

In the development of the method, two main directions can now be distinguished. First, the development of new physical models to expand the boundaries of the applicability of the method or to obtain more accurate results. Modern MD packages already include a huge number of implemented models and calculation methods, assembling which, as a constructor, and correctly configuring, one can make discoveries in the subject area.

Secondly, this is the development of the computational capabilities of the already described physical models. The fact is that molecular dynamics is an extremely resource-intensive method that requires tremendous computing power to build complex and large models. From the very first works, MD relies on the development of the computer industry.

Nowadays, the period of extensive development of supercomputer technologies has exhausted itself, and increasingly sophisticated technologies are applied to further in-

[1]Corresponding Author: Vsevolod Nikolskiy, International Laboratory for Supercomputer Atomistic Modelling and Multi-scale Analysis NRU HSE, 34 Tallinskaya Ulitsa, 123458, Moscow, Russia; E-mail: vnikolskiy@hse.ru

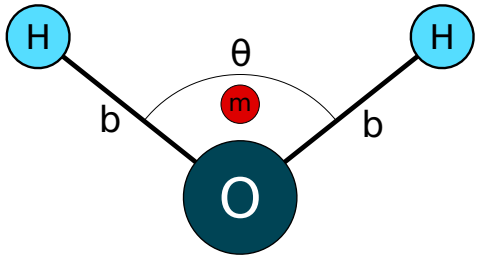


Figure 1. Rigid four-point water model TIP4P. H - hydrogen, O - oxygen, b - bonds and Theta is the angle. The particle "m" represents the virtual massless charge.

crease computing power, requiring efforts from software developers and users for the most efficient use. In addition, an increase in the universality of solutions and the possibility of code reuse is in demand, since otherwise the frequent change of the most relevant hardware inevitably leads to the need for routine support of an increasingly large code base of scientific packages.

In this work, we implement the well-known TIP4P water model for use on GPU accelerators as part of the popular LAMMPS package. The code can be compiled with CUDA and OpenCL backends due to the use of library. This approach allows us to increase the efficiency of use of computer resources because heterogeneous architectures are ubiquitous on modern supercomputers. On the other hand, our code does not duplicate, but effectively uses the huge number of features of the LAMMPS package for implementing molecular dynamics methods and their parallelization.

The further text will be organized as follows: in Section 2 we consider several related works, some of which we rely on during the development of this project. In Section 3, we describe two approaches that were created in the process of solving the problem posed in the project. Section 4 includes verification and performance testing of the developed code. The last Section 5 is the conclusion.

2. Related Work

It may seem that the most natural way to model water is to specify all or some of the three atoms that make up the water molecule as van der Waals and Coulomb interaction points [1]. In some cases, such a simple model is enough, but it is shown that the use of the fourth virtual massless charge point on the bisector of the H-O-H angle (Figure 1) significantly improves the electrostatic properties of the model. With the correct parameterization, such a model has wide applicability limits [2,3]. TIP4P water model in LAMMPS can be used as a basis of centroid molecular dynamics (CMD) quantum simulations to consider the effects of zero point energy and tunnelling [4].

There are GPU implementations of water models (TIP4P including) in GROMACS [5] and OpenMM [6], but these software tools are focused on biomolecular and soft matter models. The LAMMPS package has the greatest flexibility; it includes the largest number of potentials and possibilities for combining and extending methods. It includes the TIP4P water model for computing on the CPU in several versions — short ranged "cut" version and long-ranged with KSPACE computation [7] (Particle-Particle — Particle Mesh method). They are labeled as lj/cut/tip4p/cut and lj/cut/tip4p/long with

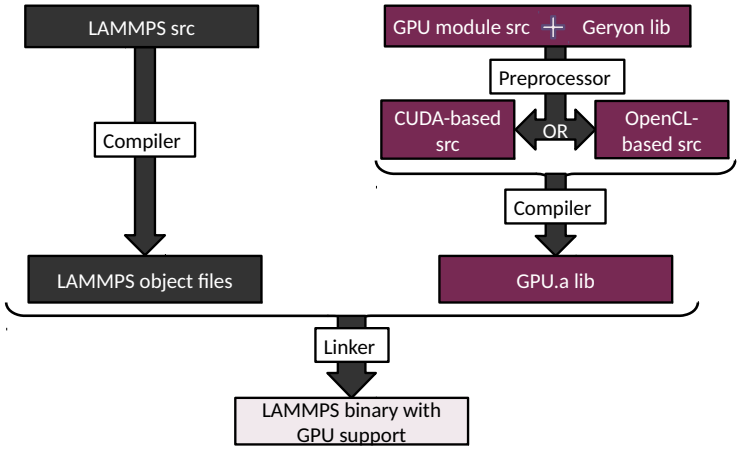


Figure 2. Geryon library used in LAMMPS allows one to compile the same code with both CUDA or OpenCL backends by preprocessing.

pppm/tip4p, respectively. The LAMMPS package implements an impressive set of potentials with GPU acceleration [8,9,10], but TIP4P is not among them. Nevertheless, we rely on the developments from the LAMMPS package on the implementation of accelerated potentials. Our project uses the Geryon library, which at the preprocessing level allows you to compile single code for CUDA and OpenCL backends (Figure 2), and the library includes the class hierarchy for molecular dynamics programming. Adapting the algorithm to use the accelerators raises some issues critical to performance [11,12], such as organizing data access [13]. Some of them are solved by the library.

We also rely on the KSPACE module for calculating long-ranged interaction [7]. This allows us to calculate only the short-range part of the Coulomb interaction, and to get the full value running the PPPM/TIP4P solver from the model script.

3. Implementation

To determine the coordinates of the virtual charge, it is necessary at each step to know the position of all three particles of the molecule. In this project, we relied on the source code of the potential lj/cut/tip4p/long to implement a "suffix-accurate-compatible" accelerated version. This code works by presume that in the input, the atoms that make up the water molecules are ordered, and each oxygen is followed by two corresponding hydrogen:

```
O - H - H - O - H - H - ... - O - H - H
```

Such ordering is stored in particle identifiers, but during the program's operation this data is reordered in memory unpredictably, and another numbering is used when traversing the local arrays in LAMMPS. To find information about all the particles that

make up the molecule, a separate code is intended. This code essentially uses the internal methods LAMMPS `Atom->map` and `Domain->closest_image` to correctly compose the molecular structure based on global particle identifiers. Porting these methods to the GPU is not entirely natural, so the first idea was to keep for execution on the CPU part of the code that accesses these functions.

Another feature of the lj/cut/tip4p/long is that it relies heavily on the use of Newtons third law, i.e. when calculating the interaction between **i** and **j** particles, the calculation result is saved for both participants in the interaction. It is unacceptable for GPU kernels, since an arbitrary thread cannot change data related to any arbitrary particle without expensive synchronization. It is necessary to organize the calculations so that only a certain group of threads is working on calculation of the force acting on the **i**-th particle. This requires a change in the algorithm and careful handling of some extreme cases.

3.1. Redundant Computation Approach

The information about the molecules is collected using the methods of the LAMMPS classes `Atom->map` and `Domain->closest_image`. It is possible to prepare molecular structure on the CPU and transfer it to the GPU. The problem is that information on virtual charges is needed not only for local, but also for **j** atoms. Therefore, the CPU has to make not so few calculations. Data on the molecular structure is stored as follows: atom numbers and a flag are stored in the hneingh array, extended to 4 number alignment if necessary. The coordinates of the virtual charge are calculated immediately and transferred to the m array, although this part of the algorithm can also be separated and transferred to the GPU.

```
hneigh[iO] = {iH1, iH2, 0, flag}
hneigh[iH1] = hneigh[iH2] = {iO, 0, 0, flag}
m[iO] = m[iH1] = m[iH2] = {x, y, z, flag}
```

When the data is prepared and transferred, the GPU kernel starts. Here we use the Redundant Computation Approach (RCA) [10]. It is expected that in our case this is not the fastest approach for naive implementation, but it is useful as a basis for further complication. Each molecule contains a single virtual charge **m**, but the force acting on it is distributed between all three real atoms [14]. Thus, in order to obtain the total sum of forces without writing data to the memory of the **j**-th atoms, it is possible to calculate two components of the electrostatic force for each ith atom: direct interaction and distributed (Figure 3). Thus, the effect on the virtual charge **m** of each **j**-th atom have to be calculated not once, but three times, but the purpose of this is that there will be no need for any additional synchronization.

The code for this approach was not as simple as originally expected. The correct calculation of interactions required introducing a rather large number of additional checks and conditions into the kernel. Overall performance is not worth the effort of porting to the GPU. But this code can be easily improved.

3.2. Reduced Redundant Computation Approach

It was decided to port all the calculations to the GPU and use additional memory order of $\mathcal{O}(n)$ to reduce the number of redundant calculations. All particles and their neighbors

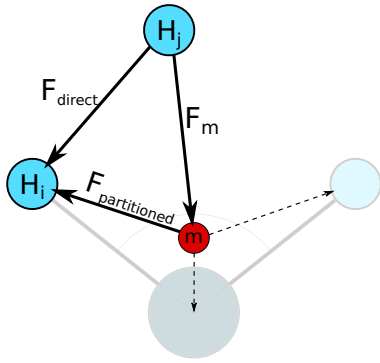


Figure 3. We consider the electrostatic force of hydrogen **j** on hydrogen **i** as part of a water molecule in the TIP4P model. It consists of two components: direct action, calculated as the usual Coulomb interaction, and indirect, through a virtual charge. Indirect action is calculated as the Coulomb interaction of hydrogen **j** with a virtual charge **m**, and then the distribution of this force.

are considered on the GPU. In the first step, the atoms that make up the molecule are detected, and then the coordinate of the virtual charge is calculated and stored. These procedures are carried out only once for each molecule, two other atoms read the information that is already stored. Then the force calculation is made: for hydrogen, a direct effect on it, and for oxygen, the effect on the displaced charge is calculated. For oxygen, this value is stored in a separate array.

The final part of the calculation is placed in a separate GPU-kernel, since these calculations should not begin before the first kernel is done for all particles and all molecular structures and preliminary force values are determined. In this part of the code, the forces acting on a displaced charge, calculated and stored in the first kernel, are distributed between the three atoms of the molecule.

Special cases that require separate processing complicate the calculation. For some local hydrogens, "ghost" oxygen may be related (Figure 4). So the contribution values will not be naturally calculated and saved for the second kernel for them since the use of Newtons third law is turned off in the GPU calculation and reverse synchronization (especially between the steps of calculating pairwise interaction) is not possible. These particles are processed according to the principle of redundant calculations, which was described in the Section 3.1: in the first kernel, a molecular structure is compiled for such hydrogens and the force acting on the displaced charge is calculated. It is necessary to increase the radius of consideration of electrostatic interactions (the radius of cutting of the force itself remains the same) to make it work correctly, and the effect of hydrogen on the oxygen of its own molecule also requires accurate accounting.

4. Results

4.1. Verification

For verification, we used the tried approach based on three criteria [15]. Conservation of full energy helps to find the first errors — it is a basic, but sensitive criterion. Since we have a reference calculation based on LAMMPS, it is convenient to use the congruence

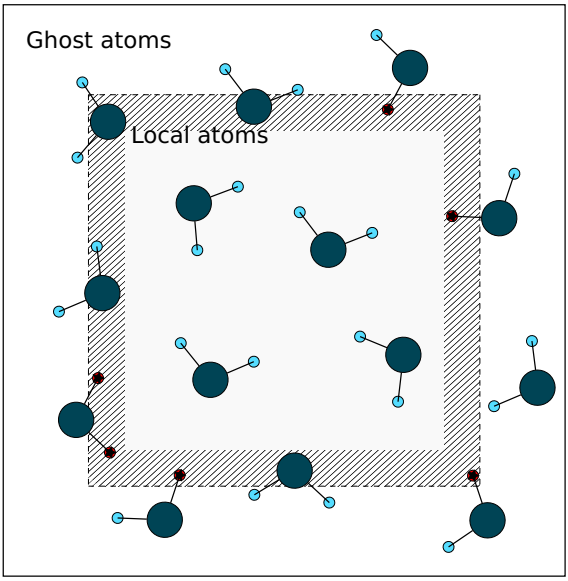


Figure 4. Light-gray area outlines the domain where the local atoms are found. It includes the shadowed area where oxygens can be found as "ghost atoms" for the corresponding hydrogens since they are not local. But that area is a small fraction of the total for typical cases.

of potential energy (Figure 6) as the second criterion, which is a function of the coordinates of all particles in the system and is also calculated simultaneously with forces in our code. The third criterion comes from the stochastic properties of the MD calculation [16]. The average displacement of the coordinates and particle velocities (Figure 5) in our simulation compared to the reference one should be equal to the machine accuracy in the first steps, then an exponential increase of the error is observed, followed up to reaching a plateau. Hopping coordinate differences on Figure 5 are likely to be artifacts of processing periodic boundary conditions.

4.2. Performance

We used two platforms for testing:

1. 8 core Intel Xeon E5-2620v4 with GPU Nvidia GeForce GTX 1070 (Pascal)
2. 8 core AMD Epyc 7251 with GPU Nvidia Titan V (Volta)

The code for the GPU can be compiled in any of three modes: single, double and mixed precision, while the calculations on the CPU are always performed in double precision. We use mixed precision for the GTX 1070, as the Nvidia GeForce GPUs are significantly slower with double-precision arithmetics [17]. Mixed precision is acceptable for many molecular dynamics calculations. At the same time, on a newer or server-level GPUs that fully supports double precision, our algorithm shows good acceleration in double precision, as it can be seen in the example of the Titan V (Volta generation).

Figure 7 shows the time profile for different parts of the task when executing the program on the CPU and on the GPU on our hardware. The number of atoms in the simulation is 32000. In the tests, all processor cores were loaded using MPI parallelization.

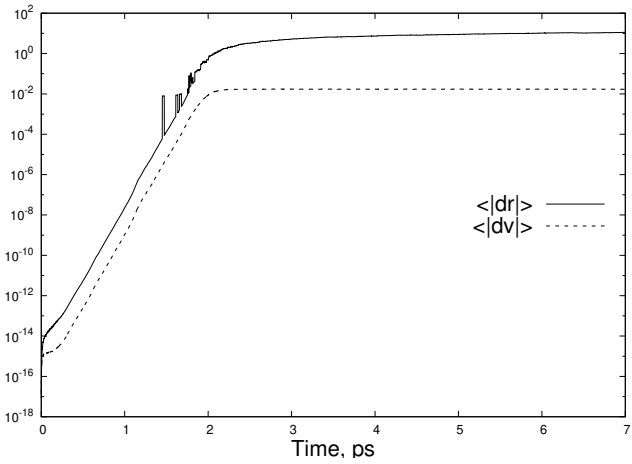


Figure 5. The normalized averaged deviations of coordinates and velocities on two trajectories calculated from identical initial conditions with LAMMPS lj/cut/tip4p/long and with our GPU-accelerated code. The exponential dependence with a further saturation regime is in agreement with the stochastic theory of molecular dynamics.

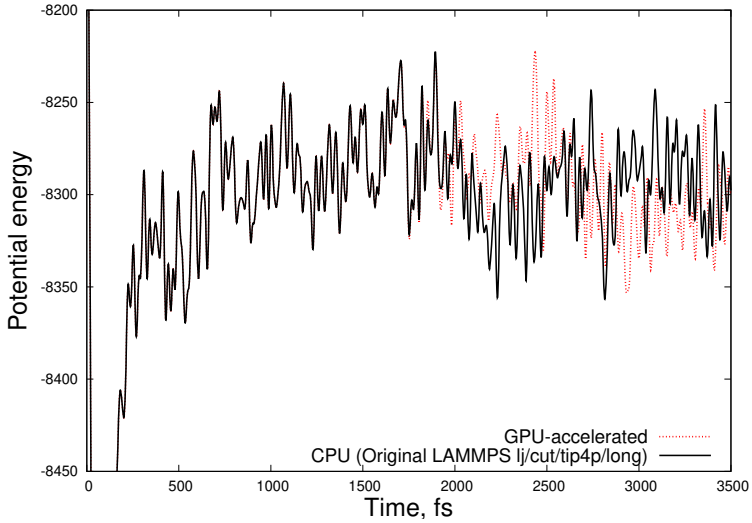


Figure 6. The potential energy calculated by our GPU code is equal to the potential energy calculated by the original LAMMPS lj/cut/tip4p/long at the beginning of the calculation, but the difference grows rapidly after passing the time of dynamic memory of the system.

It can be clearly seen that the time for calculating pairwise interactions is significantly reduced: almost seven times on the GTX 1070 and almost six times on the Titan V. It worth noting, that when one turn on the GPU module in the program, Neigh (the time it takes to build neighbor lists) approximately doubles. This is due to the disabling of the use of Newtons third law - thus the neighbors lists are doubled in size, considering each particle as **i** and as **j**, which explains the increase in time. The increased communication time at the stand with Titan V using double precision remains unclear. We conducted a

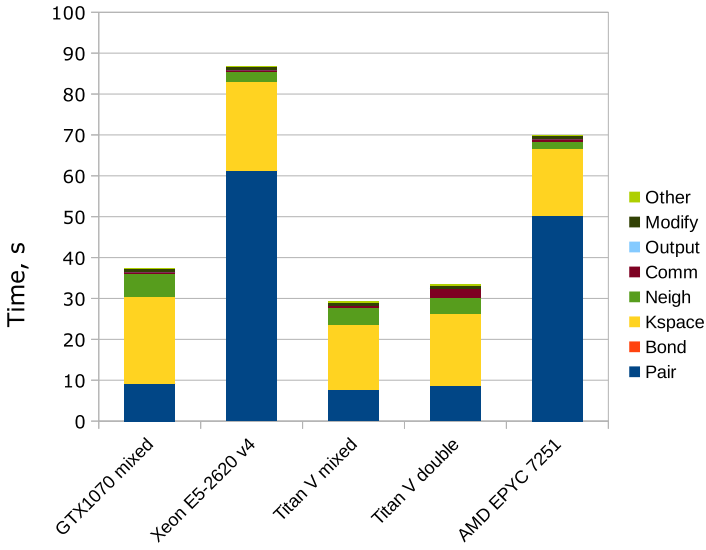


Figure 7. Time profile for 5 setups: 1) GPU algorithm in mixed precision on Nidia GTX 1070; 2) LAMMPS CPU algorithm on 8 core CPU Intel Xeon E5-2620v4; 3) GPU algorithm in mixed precision on Nidia Titan V; 4) GPU algorithm in double precision on Nidia Titan V; 5) LAMMPS CPU algorithm on 8 core CPU AMD Epyc 7251. Lower time is better. The code was implemented as "Pair" part of timestep breakdown. The number of atoms is 32000.

smaller number of experiments with Titan V, and, perhaps, it's work can be improved by proper tuning for the new architecture. We expect that the use of GPUs leads to better energy-efficiency [18]. Energy consumption is also affected by tuning.

5. Conclusion

An algorithm was developed and the corresponding code (available on GitHub [19]) was written for GPU-acceleration of the TIP4P water model as a part of the popular LAMMPS package. In this work, two solutions to this problem are described: with the execution of part of the algorithm on the CPU and the completely GPU-computed kernel. Verification of calculations is performed with both CUDA and OpenCL backends, it proves that we implemented the desired model with the machine precision. The second approach shows the overall acceleration about 55% compared to a fully loaded server processor, and the calculation of interactions is accelerated by almost six times. Future plans include refinement of the code and comprehensive testing of the stability.

Acknowledgment

The study was funded by RFBR according to the research project No. 18-37-00487 and supported within the framework of the Basic Research Program at the National Research University Higher School of Economics (HSE) and within the framework of a subsidy by the Russian Academic Excellence Project 5-100. Work has been partially supported by

the grant of the President of Russian Federation for support of leading scientific schools grant NSh-5922.2018.8.

References

[1] W. L. Jorgensen, J. Chandrasekhar, J. D. Madura, R. W. Impey, and M. L. Klein, "Comparison of simple potential functions for simulating liquid water," *The Journal of Chemical Physics*, vol. 79, no. 2, pp. 926–935, 1983.

[2] J. L. F. Abascal and C. Vega, "A general purpose model for the condensed phases of water: Tip4p/2005," *The Journal of Chemical Physics*, vol. 123, no. 23, p. 234505, 2005.

[3] J. L. F. Abascal, E. Sanz, R. Garca Fernndez, and C. Vega, "A potential model for the study of ices and amorphous water: Tip4p/ice," *The Journal of Chemical Physics*, vol. 122, no. 23, p. 234511, 2005.

[4] N. D. Kondratyuk, G. E. Norman, and V. V. Stegailov, "Quantum nuclear effects in water using centroid molecular dynamics," *Journal of Physics: Conference Series*, vol. 946, p. 012109, jan 2018.

[5] M. J. Abraham, T. Murtola, R. Schulz, S. Pll, J. C. Smith, B. Hess, and E. Lindahl, "GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers," *SoftwareX*, vol. 1-2, pp. 19 – 25, 2015.

[6] P. Eastman, J. Swails, J. D. Chodera, R. T. McGibbon, Y. Zhao, K. A. Beauchamp, L.-P. Wang, A. C. Simmonett, M. P. Harrigan, C. D. Stern, R. P. Wiewiora, B. R. Brooks, and V. S. Pande, "OpenMM 7: Rapid development of high performance algorithms for molecular dynamics," *PLOS Computational Biology*, vol. 13, pp. 1–17, 07 2017.

[7] S. J. Plimpton, R. Pollock, and M. Stevens, "Particle-mesh Ewald and rRESPA for parallel molecular dynamics simulations," in *PPSC*, 1997.

[8] W. M. Brown, P. Wang, S. J. Plimpton, and A. N. Tharrington, "Implementing molecular dynamics on hybrid high performance computers short range forces," *Computer Physics Communications*, vol. 182, no. 4, pp. 898 – 911, 2011.

[9] W. M. Brown, A. Kohlmeyer, S. J. Plimpton, and A. N. Tharrington, "Implementing molecular dynamics on hybrid high performance computers particleparticle particle-mesh," *Computer Physics Communications*, vol. 183, no. 3, pp. 449 – 459, 2012.

[10] W. M. Brown and M. Yamada, "Implementing molecular dynamics on hybrid high performance computersthree-body potentials," *Computer Physics Communications*, vol. 184, no. 12, pp. 2785 – 2793, 2013.

[11] K. Halbiniak, R. Wyrzykowski, L. Szustak, and T. Olas, "Assessment of offload-based programming environments for hybrid cpumic platforms in numerical modeling of solidification," *Simulation Modelling Practice and Theory*, vol. 87, pp. 48 – 72, 2018.

[12] B. Glinsky, I. Kulikov, I. Chernykh, D. Weins, A. Snytnikov, V. Nenashev, A. Andreev, V. Egunov, and E. Kharkov, *The Co-design of Astrophysical Code for Massively Parallel Supercomputers*, pp. 342–353. Cham: Springer International Publishing, 2016.

[13] K. Rojek and R. Wyrzykowski, "Performance modeling of 3D MPDATA simulations on GPU cluster," *The Journal of Supercomputing*, vol. 73, pp. 664–675, Feb 2017.

[14] K. A. Feenstra, B. Hess, and H. J. C. Berendsen, "Improving efficiency of large time-scale molecular dynamics simulations of hydrogen-rich systems," *Journal of Computational Chemistry*, vol. 20, no. 8, pp. 786–798, 1999.

[15] V. Nikolskii and V. Stegailov, "Domain-decomposition parallelization for molecular dynamics algorithm with short-ranged potentials on Epiphany architecture," *Lobachevskii Journal of Mathematics*, vol. 39, pp. 1228–1238, Nov 2018.

[16] G. E. Norman and V. V. Stegailov, "Stochastic theory of the classical molecular dynamics method," *Mathematical Models and Computer Simulations*, vol. 5, no. 4, pp. 305–333, 2013.

[17] V. P. Nikolskiy, V. V. Stegailov, and V. S. Vecher, "Efficiency of the Tegra K1 and X1 systems-on-chip for classical molecular dynamics," in *2016 International Conference on High Performance Computing Simulation (HPCS)*, pp. 682–689, July 2016.

[18] F. Mantovani and E. Calore, "Performance and power analysis of hpc workloads on heterogeneous multi-node clusters," *Journal of Low Power Electronics and Applications*, vol. 8, no. 2, 2018.

[19] https://github.com/Vsevak/lammps

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200087

Energy Consumption of MD Calculations on Hybrid and CPU-Only Supercomputers with Air and Immersion Cooling

Ekaterina DLINNOVA [a,b,1], Sergey BIRYUKOV [c] and Vladimir STEGAILOV [a,b]
[a] *National Research University Higher School of Economics, Moscow, Russia*
[b] *Joint Institute for High Temperatures of RAS, Moscow, Russia*
[c] *JSC NICEVT, Russia*

Abstract. The article presents the energy consumption and efficiency analysis based on the data from three small-size supercomputers installed in JIHT RAS. One system is the air-cooled hybrid supercomputer Desmos with AMD FirePro GPUs and two others are the air-cooled and liquid-cooled segments of the supercomputer Fisher based on AMD Epyc Naples CPUs. To collect data, we implement the same real-time analytics infrastructure on all three supercomputers. We consider classical molecular-dynamics problem as a benchmarking tool. Our results quantify the energy savings that are provided by the GPU-based calculations in comparison with CPU-only calculations and by liquid cooling in comparison with air-cooling. During strong scaling benchmarks, we detect an interesting minimum of energy consumption in the CPU-only case.

Keywords. supercomputers, monitoring systems, statistics analysis, energy profiles, energy consumption

1. Introduction

The effective use of supercomputer resources is an extremely important task in the field of high-performance computing [1]. However, currently there are no standard generally accepted methods that allow to collect, to analyze and to evaluate the optimal use of supercomputer resources. Energy efficiency is becoming an increasingly decisive requirement for supercomputers, as race in industrial production involves the use of the next generation of powerful computing systems. Technology has come a long way, but it is clear that there are still some difficult but crucial work that industry needs to do in the area of energy efficiency. Enlarging computing power without increasing energy consumption will undoubtedly require a deep transformation that extends across all aspects of HPC systems design.

Graphics processing units (GPUs) became widely used as accelerators for scientific and HPC applications due to their energy efficiency and high memory bandwidth. And

[1]Corresponding Author: Ekaterina Dlinnova, International Laboratory for Supercomputer Atomistic Modelling and Multi-scale Analysis NRU HSE, 34 Tallinskaya Ulitsa, 123458, Moscow, Russia; E-mail: edlinnova@hse.ru

in this work we are giving real-life values for comparison of GPU-accelerated and CPU-only computations.

Cooling technologe is another avenue for improving energy efficiency of HPC systems. And in this work we compare one systems based on a de facto standard air-cooling with a similar system that uses immersion oil cooling technology.

2. Related works

In [2], the authors raise the problem that large-scale distributed systems consume a huge amount of energy. To solve this problem, it is proposed to use job shutdown policies that can dynamically adapt the amount of resources to the actual workload. The sheer amount of energy consumed by large-scale computing and network systems, such as data centers and supercomputers, is causing serious concern in a society increasingly dependent on information technology. Trying to solve this problem, the research community and industry have proposed many methods to curb the energy consumed by IT systems.

The article [3] discusses methods and solutions aimed at improving the energy efficiency of computing and network resources. It discusses methods for estimating and modeling the energy consumed by these resources, and describes methods that work at the distributed system level, trying to improve aspects such as resource allocation, planning, and managing network traffic. This work is aimed at reviewing the state of technology in the field of energy efficiency in order to further facilitate research on the creation of networks and computing resources more efficient. Several indicators have been suggested that are most commonly used for infrastructure, such as data centers.

In order to assess how optimally the supercomputer complex consumes electricity, it is necessary to enter certain indicators and characteristics. For example, the authors of [4] characterize the energy efficiency of the Cray XC30 supercomputer system in three metrics: time to solution for a given workload; workload power consumption and energy efficiency (PUE) of the data center where the system resides. The decision time and energy consumption are closely related. For example, choosing a processor can reduce your solution time by increasing power consumption.

In [5] and [6], new methods for distributing resources were presented that take into account the topology of the machine, the patterns of interaction of tasks, and the characteristics of the application to select the best node among those available on the platform.

The article [7] discusses the current state of energy-efficient methods of parallel computing in order to achieve optimal resource consumption in conditions of limited energy consumption. The authors of [7] believe that the power consumption of modern high-performance computing systems should be reduced by at least one order of magnitude before they can be increased to ExaFLOP performance. Although new hardware technologies and architectures can be expected to contribute to this goal, software technology such as proactive and energy-efficient planning, resource allocation, and fault-tolerant computing should also bring significant success.

The paper [8] presents the energy consumption model for a hybrid supercomputer (which ranked first in Green500 in June 2013), which combines CPU, GPU, and MIC technologies to achieve high levels of energy efficiency. This model takes into account both the characteristics of the workload, the amount and location of resources that are used by each task at a certain time, and it also calculates the predicted energy consumption at the system level.

Table 1. Specifications of supercomputers considered

Supercomputer	Major components
Desmos	32 nodes Intel Xeon E5-1650v3, 6 cores, 3.5 GHz AMD FirePro S9150 GPU, Angara interconnect (4D torus), Air cooling
Fisher Air segment	18 nodes 2 x AMD Epyc7301, 16 cores, 2.7 GHz (8 DIMMs per socket) InfinibandFDR interconnect (switch) , Air cooling
Fisher Immersion segment	24 nodes 2 x AMD Epyc7301, 16 cores, 2.7 GHz (4 DIMMs per socket) Angara interconnect (switch), Immersion oil cooling

Article [9] provides a detailed analysis of the problems and possibilities of supercomputer computing in various fields of human activity: in machine learning, astronomy, medicine, materials science and energy efficiency. The authors discuss the scalability problems of both technical equipment and software and algorithms. In this regard, the discussion of the problems of efficiency from the point of view of the future of exascale-computing and analysis of large data is extremely relevant and important.

3. Hardware

For this study, we analyze the statistics of three supercomputers, all of which were installed at the Joint Institute for High Temperatures of RAS.

The first supercomputer Desmos consists of 32 nodes with AMD FirePro S9150 graphics accelerators, interconnected with a low-latency high bandwidth Angara interconnect [10]. The supercomputer is aimed at carrying out calculations by the classical molecular dynamics method, and can also effectively accelerate the calculations of the electronic structure of materials.

The second supercomputer is the Fisher supercomputer that consists of air-cooled and oil-cooled segments with AMD Epyc 7301 CPUs (see Table 1). The air-cooled segment consists of 18 dual-socket nodes connected by Infiniband FDR. The oil-cooled segment consists of 24 dual-socket nodes connected by Angara network (its switch-based fat-tree variant). The immersion cooling system was designed by the Immers company.

4. Model used for benchmarks

We analyse the power consumption of the calculation for the same resource-intensive scientific code, namely, the large-scale molecular dynamics problem in the LAMMPS package, running on the three above-mentioned supercomputers. For the benchmarking, we use a typical molecular-dynamics (MD) problem of 4 millions Lennard-Jones atoms.

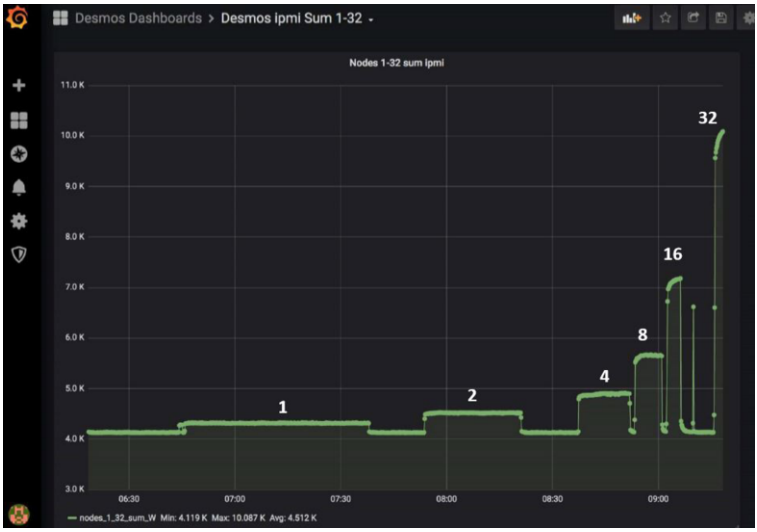


Figure 1. Real-time Desmos visualization in Grafana. We can see the dependence between energy consumption and time on Desmos while LAMMPS is running on different number of nodes (1,2,4,8,16,32).

5. Monitoring system

To collect, analyze and visualize statistics on the use of supercomputers, we used a set of applications:

- Telegraf is a utility for collecting time series measurements.
- InfluxDB is a clustered database specifically designed for storing time series.
- Grafana is a time series visualization tool. Web application for setting up charts and dashboards.

Telegraf is an agent written in Go for collecting performance metrics from the system it is running on and the services running on that system. Data aggregation infrastructure is based on InfluxDB. Grafana is an open source platform for visualizing, monitoring and analyzing data. Grafana allows users to create dashboards with panels, each of which displays certain indicators for a set period of time. Each dashboard is universal, so it can be customized for a specific project or taking into account any needs. Grafana builds visualization of the cluster state in real-time. For example, we can verify the energy consumption of the Desmos supercomputer (Figure 1).

Another possible option for building a cluster monitoring system is to use a software stack consisting of Elasticsearch, Logstash, and Kibana (the so-called ELK stack). ELK is specially designed to solve the problems of collecting, storing and processing system logs.

However, the ELK stack is designed for highly loaded web-projects, which are based on the products of companies that contain hundreds of servers of the same type. It is advisable to use ELK if you intend to analyze hundreds of megabytes of logs every day, hundreds of production servers on which you want to catch events, and also have your own highly loaded application and it needs to be monitored. In addition, logstash consumes server resources for each rule, since before processing data, it first processes

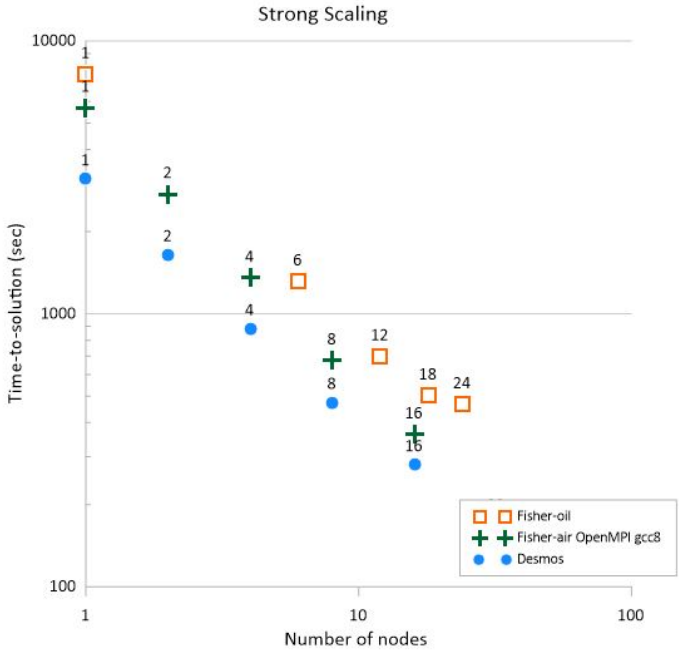


Figure 2. Strong scaling of the LAMMPS test problem: the dependence between the time-to-solution and the number of nodes for three supercomputers considered.

them. For the above reasons, we have decided that using the ELK stack is less suitable for building an HPC monitoring system.

6. Possible levels of energy consumption analysis

To collect data, we implement the following three-tier infrastructure on all three supercomputers:

- Level 1: RAPL-like protocols for CPU/DIMM energy consumption,
- Level 2: IPMI protocol at the node level (with limitations for FirePro GPUs),
- Level 3: SNMP protocol for collecting data from UPS smart-cards.

Recent Intel processors support the Running Average Power Level (RAPL) interface, which among other things provides estimated energy measurements for the CPUs, integrated GPU, and DRAM. AMD Epyc CPUs have compatible interface [2]. These measurements are easily accessible by the user, and can be gathered by a variety of tools, including the Linux event interface. This allows an easy access to energy information when designing and optimizing an energy-aware code.

[2]This interface, however, demonstrates some problems, see https://community.amd.com/thread/243717

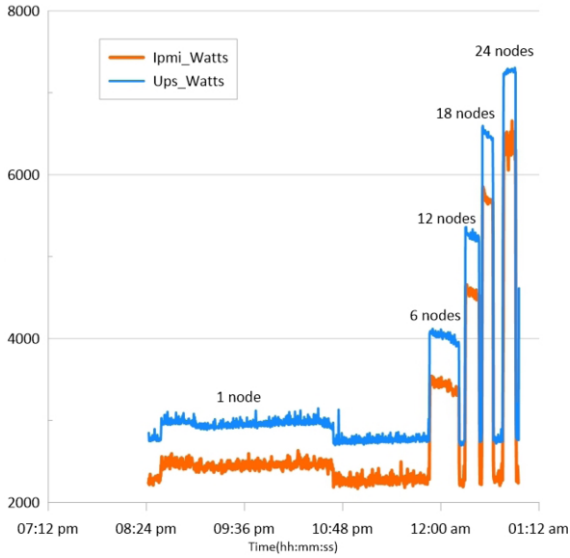


Figure 3. The Fisher immersion segment energy profile. We can see the time profile of the consumed power when LAMMPS is running on a different number of node. The blue line shows the data collected from the power supply. The orange line shows the data collected using IPMI.

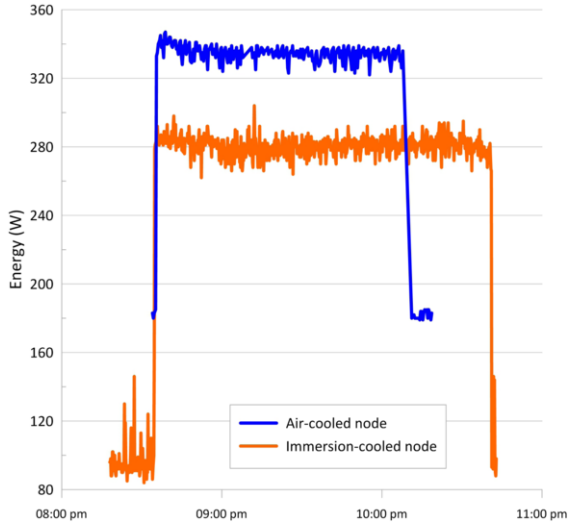


Figure 4. Comparison of the power consumption profiles (collected via IPMI) for the single node runs on the air-cooled and immersion segments of the Fisher supercomputer.

While greatly useful, on most systems these RAPL measurements are estimated values, generated on the fly by an on-chip energy model. The values are not documented well, and the results (especially the DRAM results) have limited validation.

Through the Intelligent Platform Management Interface (IPMI), it is possible to connect remotely to the server and manage its operation:

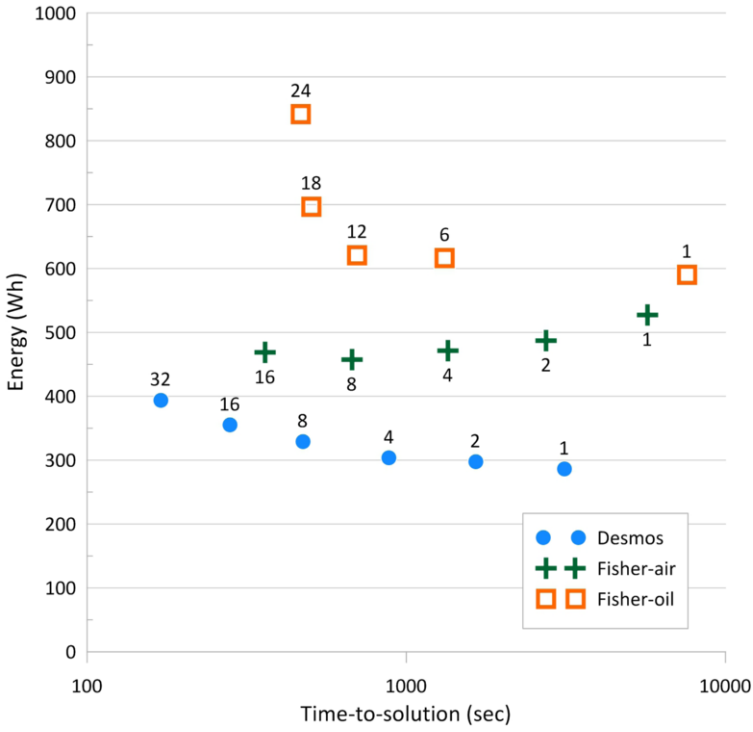


Figure 5. The dependence between the energy-to-solution and the time-to-solution for the test LAMMPS runs on three supercomputers considered.

- Monitor the physical condition of the equipment, for example, check the temperature of the individual components of the system, voltage levels, fan speed, energy consumption.
- Restore the server in automatic or manual mode (remote system reboot, power on / off, loading ISO images and updating software).
- Manage peripheral devices.
- Keep an event log.
- Store information about the equipment used.

Using the Simple Network Management Protocol (SNMP) allows to monitor almost any server, workstation or network equipment. SNMP monitoring is a standard way to obtain the characteristics of network resource utilization by routers and other network equipment. Many other parameters, such as disk space or CPU utilization, can also be obtained from the target device via SNMP. In this paper we present the results gathered at the second level only (via IPMI).

7. Benchmarking procedure and results

In this section the benchmarking procedure is described. First of all, we freeze the tasks queue on three supercomputer considered, waite for all already started jobs finish and start our measurements. We execute sequentially LAMMPS on different number of nodes

on Desmos, Fisher Air and Fisher Immers. The energy consumption information is collected using special Telegraf-exec plugin, after parsing it was inserted into InfluxDB database. We analyze the real-time energy consumption visualization using Grafana and its dashboards.

Figure 2 depicts the strong scaling results for three supercomputers. We can see in the log-log scale the dependence between the time-to-solution and the number of nodes.

Figure 3 depicts the results of the benchmarks for the Fisher supercomputer segment with immersion oil cooling. We see instantaneous values of energy consumption (W) when we execute the LAMMPS code with different number of nodes.

Figure 4 depicts the profiles of power consumption for the single node runs on the air-cooled and immersion-cooled segments of Fisher supercomputer.

To calculate total energy consumption during the running time period in kWh, we used the InfluxDB function *integral*:

```
SELECT
FROM "ipmitool_raw"
WHERE time >= '2019-09-29T20:19:00Z'
AND time <= '2019-09-29T20:25:00Z'
AND host='10.2.1.101'
```

A subtle question is how to choose the start and finish time points. In this work we choose them manually looking at the power profile in Grafana. Changing these times even by a few seconds can result in significant changes of the integral value. So this selection of time periods is very important, and in the future works we plan to use task manager synchronisation to determine start and finis time points.

The benchmark results of three supercomputers are presented on Figure 5. We can see the dependence between the total consumed energy-to-solution and the time-to-solution. We see that hybrid computations are more energy efficient despite we compare the novel CPUs (Zen microarchitecture uses 14 nm FinFET) and slightly old Haswell CPUs (22 nm FinFET) and FirePro GPUs (28 nm CMOS). Immersion cooling does not demonstrate evident benefits (despite the fact that at this stage we have not taken into account the energy consumption of the liquid transfer subsystem and the heat exchanger). The immersion segment demonstrates longer values of time-to-solution due to the reduced memory bandwidth of its nodes (see Table 1), their lower power consumption does not compensate this fact (see Figure 4) and the energy integrals is larger than for the air-cooled segment.

The results for the air-cooled segment show an interesting feature: the energy consumption decreases when we increase the number of nodes and there is a minimum energy consumption at 8 nodes. The origin of this effect is presumably connected with the dynamic fan speed control in the nodes during the benchmarks and deserves a separate study in the future.

8. Conclusions

We have implemented identical energy monitoring systems for real-time analytics of power and energy consumption on three supercomputers: the hybrid air-cooled Desmos

supercomputer and the air-cooled and the liquid-cooled CPU-only segments of the Fisher supercomputer.

Benchmarking results based on a single MD model calculation example show the following:

- The excess consumption of an air-cooled system compared to an immersion-cooled system is on average 30% or 1.4 kW.
- The energy efficiency gain of a hybrid air-cooled system is 200 Wh (46%) for 1 node and decreases with an increase in the number of nodes used for the test calculation.
- We detected a minimum of total energy consumption for the test problem on CPU-only systems.

Acknowledgment

The study has been partially supported by the grant of the President of Russian Federation for support of leading scientific schools grant NSh-5922.2018.8 and supported within the framework of the Basic Research Program at the National Research University Higher School of Economics (HSE) and within the framework of a subsidy by the Russian Academic Excellence Project 5-100.

References

[1] F. Mantovani and E. Calore, "Performance and power analysis of HPC workloads on heterogeneous multi-node clusters," *Journal of Low Power Electronics and Applications*, vol. 8, no. 2, 2018.

[2] A. Benoit, L. Lefevre, A.-C. Orgerie, and I. Raïs, "Reducing the energy consumption of large-scale computing systems through combined shutdown policies with multiple constraints," *The International Journal of High Performance Computing Applications*, vol. 32, no. 1, pp. 176–188, 2018.

[3] E. Jaureguialzo, "Pue: The green grid metric for evaluating the energy efficiency in dc (data center). measurement method using the power demand," in *2011 IEEE 33rd International Telecommunications Energy Conference (INTELEC)*, pp. 1–8, IEEE, 2011.

[4] G. Pautsch, D. Roweth, and S. Schroeder, "The Cray® XC supercomputer series: Energy-efficient computing," tech. rep., Technical Report, 2013.

[5] Y. Georgiou, E. Jeannot, G. Mercier, and A. Villiermet, "Topology-aware job mapping," *The International Journal of High Performance Computing Applications*, vol. 32, no. 1, pp. 14–27, 2018.

[6] C. Gómez-Martín, M. A. Vega-Rodríguez, and J.-L. González-Sánchez, "Performance and energy aware scheduling simulator for HPC: evaluating different resource selection methods," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 17, pp. 5436–5459, 2015.

[7] A.-C. Orgerie, M. D. d. Assuncao, and L. Lefevre, "A survey on techniques for improving the energy efficiency of large-scale distributed systems," *ACM Computing Surveys (CSUR)*, vol. 46, no. 4, p. 47, 2014.

[8] A. Sîrbu and O. Babaoglu, "A data-driven approach to modeling power consumption for a hybrid supercomputer," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 9, p. e4410, 2018.

[9] C. Jin, B. R. de Supinski, D. Abramson, H. Poxon, L. DeRose, M. N. Dinh, M. Endrei, and E. R. Jessup, "A survey on software methods to improve the energy efficiency of parallel computing," *The International Journal of High Performance Computing Applications*, vol. 31, no. 6, pp. 517–549, 2017.

[10] V. Stegailov, E. Dlinnova, T. Ismagilov, M. Khalilov, N. Kondratyuk, D. Makagon, A. Semenov, A. Simonov, G. Smirnov, and A. Timofeev, "Angara interconnect makes GPU-based Desmos supercomputer an efficient tool for molecular dynamics calculations," *The International Journal of High Performance Computing Applications*, vol. 33, no. 3, pp. 507–521, 2019.

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200088

Direct *N*-Body Application on Low-Power and Energy-Efficient Parallel Architectures

David GOZ [a,1], Georgios IERONYMAKIS [b], Vassilis PAPAEFSTATHIOU [b], Nikolaos DIMOU [b], Sara BERTOCCO [a], Antonio RAGAGNIN [a], Luca TORNATORE [a], Giuliano TAFFONI [a] and Igor CORETTI [a]

[a] *INAF-Osservatorio Astronomico di Trieste, Italy*
[b] *FORTH-ICS, Heraklion, Crete, Greece*

Abstract.
The aim of this work is to quantitatively evaluate the impact of computation on the energy consumption on ARM MPSoC platforms, exploiting CPUs, embedded GPUs and FPGAs. One of them possibly represents the future of High Performance Computing systems: a prototype of an Exascale supercomputer. Performance and energy measurements are made using a state-of-the-art direct *N*-body code from the astrophysical domain. We provide a comparison of the time-to-solution and energy delay product metrics, for different software configurations. We have shown that FPGA technologies can be used for application kernel acceleration and are emerging as a promising alternative to "traditional" technologies for HPC, which purely focus on peak-performance than on power-efficiency.

Keywords.
Astrophysics, HPC, *N*-body, ARM MPSoC, GPUs, FPGAs, exascale, energy-delay-product

1. Introduction and motivation

Energy efficiency is one of the main problems for exascale computing systems, since simply re-scaling the current petascale systems would require an unfeasible amount of power consumption. A re-design of the underlying technologies (i.e., processors, interconnect, storage, and accelerators) is needed to reduce energy requirements by about one order of magnitude [1]. To exploit the upcoming new architectures, software developers are forced to face the challenge of re-designing algorithms.

Commodity single board platforms are an interesting case of heterogeneous systems for performance and energy-efficiency studies (e.g. [2,3,4,5]). They are based on low-power System-on-Chip (SoC) architectures with embedded CPUs, GPUs, FPGAs, memory, storage and general purpose I/O ports. Many companies are delivering single-board computers equipped with different hardware components and utilize Multi-processing System-on-Chip (MPSoC) where the energy efficiency is the main concern.

[1]Corresponding Author: David Goz, ORCID: 0000-0001-9808-2283, INAF-Osservatorio Astronomico Di Trieste, Via G.B. Tiepolo 11, 34131 Trieste, Italy; E-mail: david.goz@inaf.it

This work arises in the framework of the ExaNeSt and EuroExa European funded projects aiming at the design and development of a prototype of an exascale HPC facility based on ARM SoC and FPGA technology [9,13]. Our goal is to study the trade-off between time-to-solution and energy-to-solution using real code, a direct N-body solver for astrophysical simulations, instead of benchmarking these machines by means of standard suites (e.g. HPL [14], DGEMM, STREAM [15]). We assess the performance and the associated power-efficiency across different platforms, namely, the MPSoC Firefly-RK3399 produced by Rockchip, and the Zynq-7000 SoC and Zynq UltraScale+ MPSoCs both produced by Xilinx. We further compare these results with a commodity architecture based on an x86 Intel desktop equipped with a high-end gaming GPU.

To the best of our knowledge, this work provides the first comprehensive evaluation of a real application, coming from the astrophysical domain, on low-cost and low-power boards hosting ARM (64 bit) mobile-class cores, embedded GPUs and FPGAs.

The paper is organized as follows. In Section 2 we describe the code and discuss strategies in order to optimize algorithms on heterogeneous platforms. In Section 3 we present the computing platforms used in the test. Section 4 is devoted for the discussion of the methodology adopted to make the performance and energy tests. In Section 5 we present the performance measurements for all platforms along with the power consumption. The last section is dedicated to the conclusions and future work.

2. N-body astrophysical code

In astrophysics, the N-body problem is the problem of predicting the individual motions in a group of celestial objects interacting with each other gravitationally. The main drawback related to the direct N-body problem relies on the fact that the algorithm requires $O(N^2)$ computations. Our application, called HY-NBODY, a modified version of a GPU N-body code [6,7], is based on high order Hermite integration schema [8] and has been developed in the framework of the ExaNeSt project [9]. HY-NBODY has been designed to fully exploit the compute capabilities of heterogeneous platforms. Three versions of the code are available: one written in Standard C, cache-aware designed for CPUs, one that is implemented and optimized using OpenCL kernels, allowing us to exploit any OpenCL-compliant device (e.g. GPUs), and one that is written also in Standard C using Xilinx Vivado High Level Synthesis (HLS) tool and is implemented for FPGAs.

Code profiling shows that, during a single time step of the simulation, more than 90% of time is spent on a single kernel with an arithmetic intensity $I \simeq 10^4$ [FLOPs/byte] (ratio of FLOPs to the memory traffic), using 32^3 particles. In the following, time and energy measurements on a given device refer to this compute-bound kernel.

2.1. Floating point arithmetic considerations

The Hermite 6^{th} order integration schema requires double precision (DP) floating-point arithmetic in the evaluation of inter-particle distance and acceleration in order to minimize the round-off error, so as to preserve the total energy and the angular momentum of the N-body system during the simulation.

Full IEEE-compliant DP floating-point arithmetic is efficient in contemporary CPUs, but it is still extremely resource-eager and performance-poor in other accelerators

like gaming or embedded GPUs. As an alternative, the extended-precision (EX) (or emulated double precision) numeric type [11] can represent a trade-off in porting HY-NBODY on devices not specifically designed for scientific calculations. An EX-number provides approximately 48 bits of mantissa at single-precision exponent ranges.

3. Computing platforms

In this section we describe the four computing platforms used in our tests. Table 1 lists the devices present in each computing platform, and we highlight in bold the devices used in our tests. The platforms are:

- **Firefly-RK3399 board**: it is equipped with the ARM big.LITTLE architecture, four Cortex-A53 cores with 32KB L1 cache and 512KB L2 cache, and a cluster of two Cortex-A72 high performance cores with 32KB L1 cache and 1MB L2 cache. Each cluster operates at independent frequencies, ranging from 200MHz up to 1.4GHz for the LITTLE, and up to 1.8GHz for the big. The board contains 4GB DDR3 - 1333MHz RAM. The board contains also the OpenCL-compliant Mali-T864 embedded GPU;
- **x86 desktop**: it is equipped with four Intel i7-3770 cores running at 3.4 GHz with 32KB L1 cache, 256KB L2 cache and 8192KB L3 cache. The board contains 16GB DDR3 - 1866 MHz RAM and the NVIDIA GeForce-GTX-1080 GPU in the PCI Express (16X) bus;
- **ZedBoard**: it is equipped with the Xilinx Zynq 7000 MPSoC, with dual-core ARM Cortex-A9 processors integrated with 28nm Artix-7 based programmable logic (FPGA). The board contains 512 MB DDR3 RAM;
- **QFDB**: the Quad-FPGA DaughterBoard (QFDB) is the compute-unit of the prototype developed within the framework of the ExaNeSt project [9,10]. The compute board, whose block diagram is shown in Figure 1, contains four Xilinx Zynq UltraScale+ MPSoC devices (ZU9EG), each featuring four Cortex-A53 and two Cortex-R5 cores, along with a rich set of hard IPs and Reconfigurable Logic. A 16GB DDR4 RAM is connected to each Zynq device. The maximum sustained power of the board is 120 Watts. Targeting a compact design, the dimension of the board is 120-130mm while no component on top or below the printed circuit board (PCB) is taller than 10mm. The PCB stackup consists of 16 layers, with Megtron-6 dielectric. Within the board, multiple high-speed serial links (HSSLs) connect the four Zynq devices, each operating at a line rate of 16.375Gbps. One of the Zynq devices is connected to the outside world through 10 high-speed serial links (HSSLs) using GTH transceivers. On each QFDB, the measurement of the power consumption is accomplished by using a set of TI INA226 current/power sensor, coupled with high-power shunt resistors. The INA226 sensor's minimal capture time is $140\mu s$. However, the Linux INA drivers (and the power-on set-up) set the capture time to 1.1ms by default. The Linux driver also enables averaging from 16 samples and captures both the shunt and the bus voltages. To collect data from the sensors, each board includes 15 I2C power sensors, which allow measuring the power consumption of the primary sub-systems of the board.

Table 2 shows the clock, the theoretical peak performance in FP64/FP32 and the achieved performance of the devices using DP/EX arithmetic. Since FPGAs do not have a fixed architecture, a generic way to calculate their peak performance does not exist for the following reasons:

(i) each type of calculation needs a different amount of resources to be implemented;
(ii) a single type of calculation can be implemented in various ways;
(iii) the FPGA can operate with various clock frequencies;
(iv) usually an accelerator takes a part of the FPGA and not the entire FPGA (even a design of 90% utilization is very difficult to be placed and routed and it becomes even more difficult in higher clock frequencies).

Thus, in Table 2, regarding the FPGAs, we present the theoretical performance of the implemented kernels versus the actual performance obtained including the latency of the memory I/O and the time needed to handle the kernels from the software application.

Table 1. The platforms and the associated devices. The devices exploited in the test are highlighted in bold.

Platform	CPU	GPU	FPGA
Firefly-RK3399	**ARM A53x4 + A72x2**	**ARM Mali-T864**	None
Desktop	**Intel i7-3770x4**	**NVIDIA GeForce-GTX-1080**	None
ZedBoard	ARM A9x2	None	**Zynq-7000**
QFDB	4x(ARM A53x4 + R5x2)	4x(ARM Mali-400)	4x(**Zynq-US+**)

Table 2. The clock, the theoretical peak performance and the achieved performance of the devices tested in this work. The actual performance has been obtained using 65536 particles.

Device	i7x4	A53x4	A72x2	GTX-1080	T864	Z-7000	Z-US+
Clock (GHz)	3.40	1.42	1.80	1.733	0.800	0.100	0.300
Peak FP64/FP32 (GFLOPS)	108/217	11.36/45.44	7.2/28.8	277.3/8873	32/109	8.5/39.1	102/351.9
Actual DP/EX (GFLOPS)	8.0/4.1	2.6/2.6	3.0/2.6	55.2/4984	1.5/14.4	1.2/4.9	95.3/333.1

4. Methodology

In this section we discuss how power measurements were made. On the first three platforms, namely, the Firefly-RK3399, the desktop and the ZedBoard, the electric power draw is measured by means of a power meter (Yokogawa WT310E), while on the QFDB it relies on the on-board sensors.

After booting up the platform, we measure the watt-hours consumed in idle during a period of three minutes (ΔT_3), giving us the $E_{baseline}$ of the system. Then, $E^{device}_{baseline}$ is the electric power drawn by the system running a given code implementation using a particular device (CPU, GPU or FPGA) over ΔT_3.

The energy-to-solution of the specific device is:

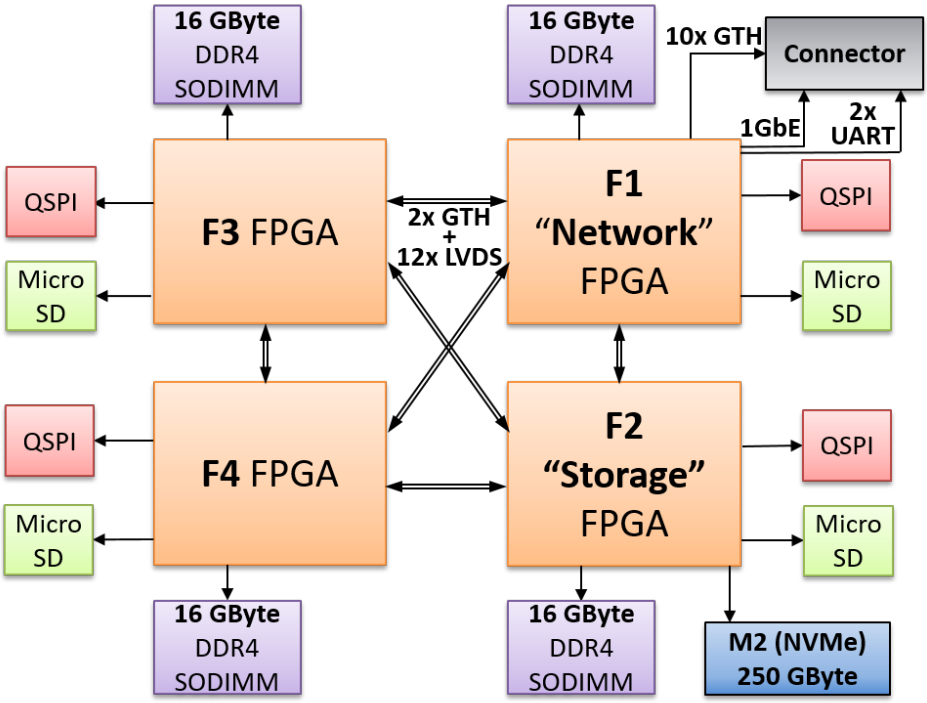


Figure 1. The Quad-FPGA daughterboard block diagram and interconnects.

$$E_{impl}^{device} = \left(E_{baseline}^{device} - E_{baseline}\right) \cdot \left(T^{device}/\Delta T_3\right), \tag{1}$$

where T^{device} is the kernel running time (time-to-solution averaged over ten runs). We point out that the benchmark runs have been done taking into account the output to the main memory, as happens in real production runs.

We also estimate the total energy impact of the application in terms of Energy Delay Product (EDP), as suggested by Cameron [16], and defined as:

$$EDP = E_{baseline}^{device} \cdot \left(T^{device}/\Delta T_3\right)^{w}, \tag{2}$$

where w is a parameter to weight performance versus power (usually $w = 1, 2, 3$). The EDP is a "fused" metric to evaluate the trade-off between time-to-solution and energy-to-solution.

5. Computational performances and energy consumption

First we investigate the time-to-solution running the code varying the number of particles. In the case of CPUs, we exploit all the available cores by means of OpenMP threads. On the Firefly-RK3399 board equipped with the big.LITTLE ARM architecture, we pinned the processes first to the four cores of the Cortex-A53 and then to the two cores of the Cortex-A72. Kernel execution times on GPUs, both Nvidia-GeForce-GTX-

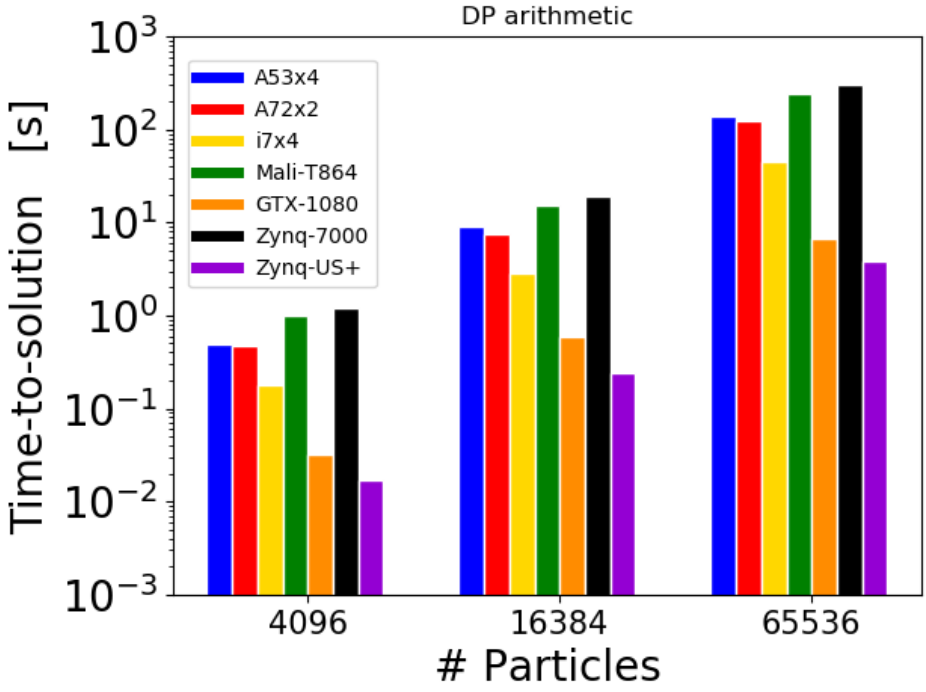


Figure 2. Time-to-solution in seconds for different devices as a function of the number of particles for double-precision arithmetic.

1080 and ARM Mali-T864, have been obtained using the OpenCL's built-in profiling functionality, which allows the host to collect runtime information, while in the case of QFDB the kernel was executed only on a single FPGA out of the four it comprises.

In Figure 2, we compare the time-to-solution for the devices reported on Table 1 for DP arithmetic. From a pure performance point of view, regarding the DP arithmetic, the Zynq UltraScale+ FPGA and the Nvidia GPU are the most powerful devices, while the FireFly MPSoC and the Zynq-7000 FPGA performances are almost two orders of magnitude lower.

To better study the effect of the extended-precision arithmetic, in Figure 3 we show the ratio of time-to-solution between DP and EX arithmetic. The performance improvement is a factor of ~ 2 for the Mali-T864 GPU and ~ 20 for the GTX-1080, while CPUs suffer a significant performance degradation. Regarding the QFDB, the EX kernel shows a 32% degradation in performance compared to the DP implementation. Although single precision arithmetic requires less FPGA resources overall, the extra calculations (in particular accumulations) needed to minimize the roundoff and overflow errors in the intermediate results of EX precision introduce an additional overhead which impacts the size of the kernel that can be implemented inside the FPGA. Thus, although the EX kernel was designed to execute on $\sim 25\%$ less particles per cycle, because of these extra calculations it occupies 10% and 8% more in terms of DSPs and LUTs accordingly compared to the DP implementation.

Figure 4 shows the total energy-to-solution (E^{device}) for all devices and for both DP and EX arithmetic using 65536 particles. For CPUs and GPUs, the instantaneous power

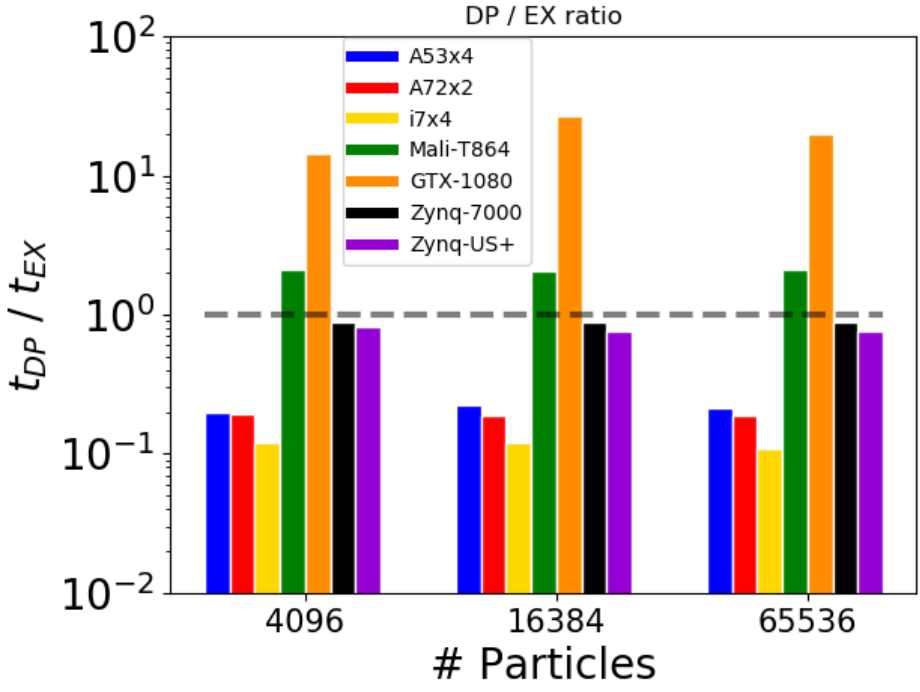


Figure 3. The ratio of the time-to-solution between DP arithmetic and EX arithmetic as a function of the number of particles for different devices.

consumption is pretty much the same running the DP or EX kernel implementation, however using EX arithmetic on GPUs leads to better energy-efficiency because of the reduced time-to-solution. Our findings show that the Zynq UltraScale+ FPGA is more energy-efficient than the GTX-1080 by a factor of 15 when using DP arithmetic.

Moreover, we obtain EDP, for $w = 1, 3$, when running the application using 65536 particles and with the methodology discussed in Section 4. In Figure 5 we plot the EDP comparing the devices (top panel for $w = 1$, and bottom panel for $w = 3$). When performances are highly valued (i.e. $w = 3$), the GTX-1080 is the device with the best trade-off between time-to-solution and energy-to-solution using EX arithmetic, while the Zynq UltraScale+ FPGA is favorable in terms of energy and execution time when DP floating-point arithmetic is used.

6. Conclusions and future work

The energy footprint of scientific applications will become one of the main concerns in the HPC sector. In this work we employ a real scientific application, coming from Astrophysical domain, to explore the impact of software design on time-to-solution and energy-to-solution using low-cost MPSoC-based platforms and FPGA-based technologies that can be potentially used in future HPC systems.

Due to the computational intensive nature of our application, accelerators, like GPU and FPGA, outperform CPU peak performance, as expected. In particular, the introduc-

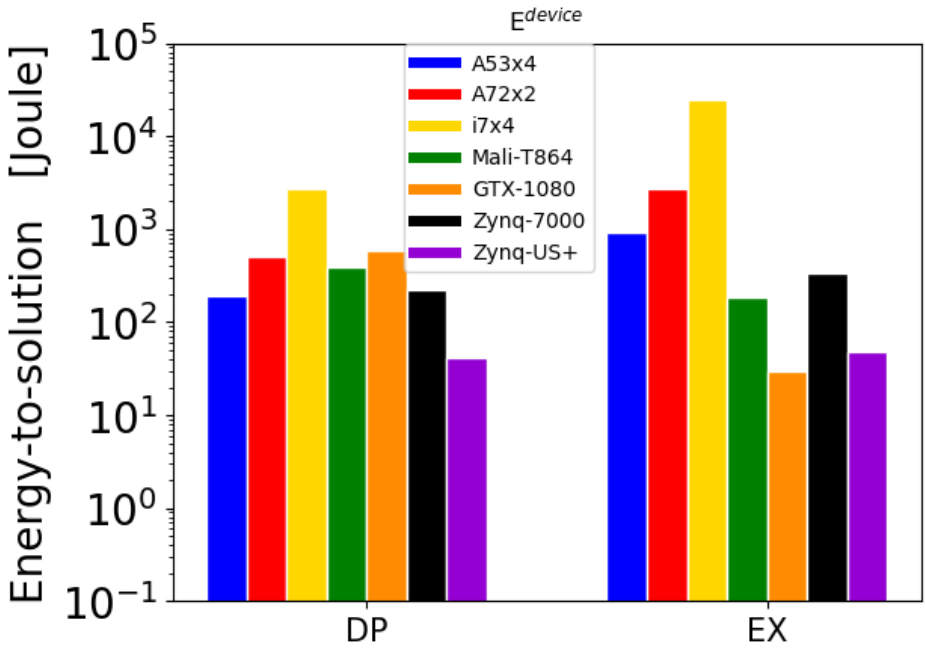


Figure 4. The energy-to-solution of the specific device is shown.

tion of the emulated-double precision improves the application performance on SoCs with embedded GPUs, like the ARM Mali, opening the path for successful and cost-effective use of such devices in HPC.

The crucial findings of this work are the achieved performances, both in terms of time-to-solution and energy-to-solution, exploiting the Zynq UltraScale+ FPGA on the ExaNeSt QFDB prototype. Kernel development for FPGAs is slightly different than traditional GPU development in that the hardware is created for the specific functions being implemented. Understanding the difference between SIMD parallelism and pipeline parallelism employed on FPGAs, and taking advantage of FPGA features, such as heterogeneous memory support, channels, loop pipelining and unrolling, are key factors to unlock high performance-per-watt solutions.

In conclusion, we have shown that SoC technology is emerging as a promising alternative to "traditional" technologies for HPC, which purely focus on peak-performance rather than power-efficiency. The main drawback is that programmers of scientific applications will have to re-engineer their code in order to fully exploit new computing facilities based on heterogeneous hardware.

Our future plan is to assess the energy footprint of the HY-NBODY application on a cluster of MPSoCs, hosting CPUs, GPUs and FPGAs, and to compare it with HPC resources, where multi-node communication becomes an important aspect of the application.

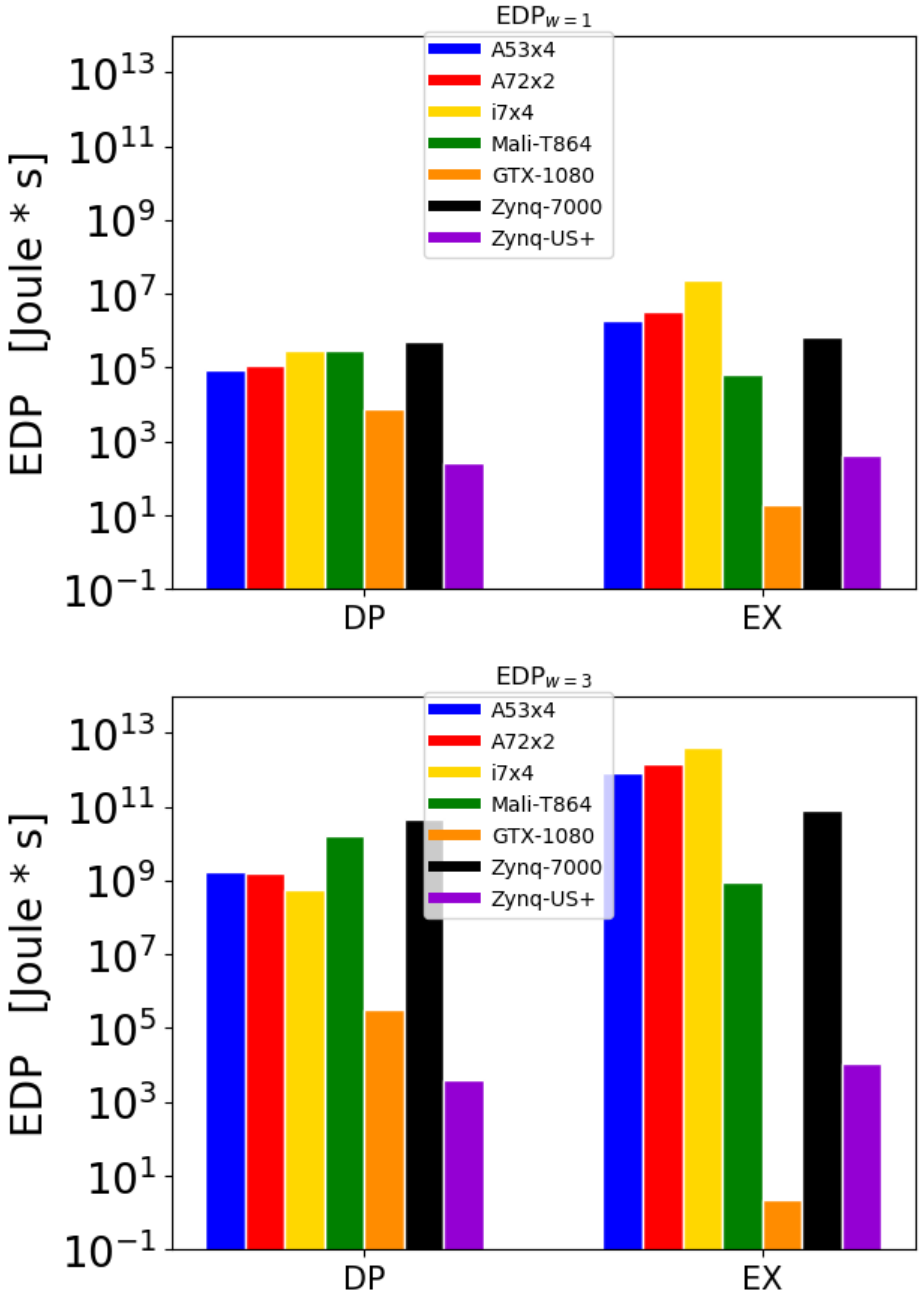


Figure 5. EDP for DP arithmetic and for EX arithmetic varying the device. Top panel for $w = 1$, and bottom panel for $w = 3$.

7. Acknowledgments

This work was carried out within the EuroEXA and ExaNeSt FET-HPC projects (grant no. 754337 and no. 671553), funded by the European Union's Horizon 2020 research and innovation program.

References

[1] Hammer, N., Jamitzky, F., Satzger, H., et al. "Extreme Scale-out SuperMUC Phase 2 - lessons learned," In ParCo 2015, Advances in Parallel Computing Vol. 27, 2015

[2] Calore E., Schifano S. F., Tripiccione R.: Energy-Performance Tradeoffs for HPC Applications on Low Power Processors. Euro-Par 2015: Parallel Processing Workshops. Springer International Publishing (2015). doi: 10.1007/978-3-319-27308-2_59

[3] V. P. Nikolskiy, V. V. Stegailov and V. S. Vecher: Efficiency of the Tegra K1 and X1 systems-on-chip for classical molecular dynamics. (2016) International Conference on High Performance Computing & Simulation (HPCS), Innsbruck, 2016, pp. 682-689. doi: 10.1109/HPCSim.2016.7568401

[4] Nikolskii V., and Stegailov, V.: Domain-Decomposition Parallelization for Molecular Dynamics Algorithm with Short-Ranged Potentials on Epiphany Architecture. Lobachevskii Journal of Mathematics (2018). doi:10.1134/S1995080218090159

[5] Morganti L, Cesini D, Ferraro A: Evaluating Systems on Chip through HPC Bioinformatics and Astrophysics Applications. 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP) 2016, 541-544, 2016. doi: 10.1109/PDP.2016.82

[6] Capuzzo-Dolcetta R., Spera M.: A performance comparison of different graphics processing units running direct N-body simulations. Computer Physics Communications 184:25282539 (2013)

[7] Spera M.: Using Graphics Processing Units to solve the classical N-body problem in physics and astrophysics. ArXiv e-prints 1411.5234 (2014)

[8] Nitadori K., Makino J.: Sixth- and eighth-order Hermite integrator for N-body simulations. New Astronomy 13:498507, (2008)

[9] Katevenis M., Chrysos N., Marazakis M., Mavroidis I., Chaix F., Kallimanis N., et al.: The ExaNeSt Project: Interconnects, Storage, and Packaging for Exascale Systems, 2016 Euromicro Conference on Digital System Design (DSD), Limassol, pp. 60-67 (2016)

[10] F. Chaix, A.D. Ioannou, N. Kossifidis, N. Dimou, G. Ieronymakis, M. Marazakis, V. Papaefstathiou, V. Flouris, M. Ligerakis, G. Ailamakis, T.C. Vavouris, A. Damianakis, M. G.H. Katevenis and I. Mavroidis, "Implementation and impact of an ultra-compact multi-FPGA board for large system prototyping", 5th International Workshop on Heterogeneous High-performance Reconfigurable Computing ($H^2RC'19$), held in conjunction with SC'19, 2019

[11] Thall A.: Extended-precision floating-point numbers for gpu computation. p 52, (2006) DOI 10.1145/1179622.1179682

[12] Terpstra D., Jagode H., You H., Dongarra J.: Collecting performance data with papi-c. In: M. S. Muller, M. M. Resch, A. Schulz, W. E. Nagel (Eds.),Tools for High Performance Computing 2009, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 157173 (2009)

[13] Ammendola R., Biagioni A. , Cretaro P., Frezza O., Cicero FL et al.: The Next Generation of Exascale-Class Systems: The ExaNeSt Project. In Euromicro Conference on Digital System Design (DSD), Vienna, pp. 510-515 (2017) http://dx.doi.org/10.1109/DSD.2017.20

[14] JJ. Dongarra1, P. Luszczek1, and A. Petitet, "The LINPACK Benchmark: past, present and future," Concurrency Computat.: Pract. Exper. 2003; 15, pp 803–820

[15] J. McCalpin, "STREAM: Sustainable Memory Bandwidth in High Performance Computers," http://www.cs.virginia.edu/stream/

[16] Cameron K.W., Ge R., Feng X., Varner D., Jones C.: High-performance, power-aware distributed computing framework. In Proceedings of the International Conference on High Performance Computing, Networking, Storage, and Analysis (SC), ACM/IEEE, (2004)

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200089

Performance and Energy Efficiency of CUDA and OpenCL for GPU Computing Using Python

Håvard H. HOLM [a,c,1] André R. BRODTKORB [a,d], and Martin L. SÆTRA [b,d]
[a] *SINTEF Digital, Department of Mathematics and Cybernetics.*
[b] *Norwegian Meteorological Institute, Information Technology Department.*
[c] *Norwegian University of Science and Technology, Department of Mathematics.*
[d] *Oslo Metropolitan University, Department of Computer Science.*

Abstract. In this work, we examine the performance and energy efficiency when using Python for developing HPC codes running on the GPU. We investigate the portability of performance and energy efficiency between CUDA and OpenCL; between GPU generations; and between low-end, mid-range and high-end GPUs. Our findings show that for some combinations of GPU and GPU code, there is a significant speedup for CUDA over OpenCL, but that this does not hold in general. Our experiments show that performance in general varies more between different GPUs, than between using CUDA and OpenCL. Finally, we show that tuning for performance is a good way of tuning for energy efficiency.

Keywords. GPU Computing, CUDA, OpenCL, High Performance Computing, Shallow-Water Simulation, Power Efficiency

1. Introduction

GPU computing was introduced in the early 2000s, and has since become a popular concept. The first examples were acceleration of simple algorithms such as matrix-matrix multiplication by rephrasing the algorithm as operations on graphical primitives (see e.g., [1]). This was cumbersome and there existed no development tools for general-purpose computing. However, many algorithms were implemented on the GPU as proof-of-concepts, showing large speedups over the CPU [2]. Today, the development environment for GPU computing has evolved tremendously and is both mature and stable: Advanced debuggers and profilers are available, making debugging, profile-driven development, and performance optimization easier than ever.

The GPU has traditionally been accessed using compiled languages such as C/C++ or Fortran for the CPU code, and a specialized programming language for the GPU. The rationale is often that performance is paramount, and that compiled languages are therefore required. However, for many GPU codes, most of the time is spent in the numerical code running on the GPU. In these cases, we can possibly use a higher-level

[1] Corresponding Author

language such as Python for the program flow without significantly affecting the performance.[2] The benefit is that using higher-level languages can significantly increase productivity [3].

We study PyCUDA and PyOpenCL [4] in this work, which offer access to CUDA and OpenCL from Python. They have become mature and popular packages since their initial release nearly ten years ago. We compare the performance and energy efficiency of PyCUDA and PyOpenCL for three different explicit numerical stencils for simulating shallow-water flow. This represents a class of problems that are particularly well suited for GPU computing [5,6]. We demonstrate how profile-driven development in Python is essential for increasing performance (up to 5 times) for GPU code, and show that the energy efficiency increases proportionally with performance tuning. Finally, we investigate the portability of the improvements and power efficiency both between CUDA and OpenCL and between different GPUs.

2. Related work

There are several high-level programming languages and libraries that offer access to the GPU for certain sets of problems and algorithms. OpenACC [7] is one example which is pragma-based and offers a set of directives to execute code in parallel on the GPU. However, such high-level abstractions are typically only efficient for certain classes of problems and are often unsuitable for non-trivial parallelization or data movement. CUDA [8] and OpenCL [9] are two programming languages that offer full access to the GPU hardware, including the whole memory subsystem. This is an especially important point, since memory movement and data transfers are often key bottlenecks in today's numerical algorithms [10] and therefore have significant impact on energy consumption.

The performance of GPUs has been reported extensively [11], and several authors have previously shown that GPUs are efficient in terms of energy-to-solution. Huang et al. [12] demonstrated early on that GPUs could not only speed up computational performance, but also increase power efficiency dramatically using Nvidia CUDA. Qi et al. [13] show how OpenCL on a mobile GPU can increase performance of the discrete Fourier transform by 1.4 times and decrease the energy by 37%. Dong et al. [14] analyze the energy efficiency of GPU BLAST which simulates compressible hydrodynamics using finite elements with CUDA and report a 2.5 times speedup and a 42% increase in energy efficiency. Klôh [15] report that there is a wide spread in terms of energy efficiency and performance when comparing 3D wave propagation and full waveform inversion on two different architectures. They compare an Intel Xeon coupled with an ARM-based Nvidia Jetson TX2 GPU module, and find that the Xeon platform performs best in terms of computational speed, whilst the Jetson platform is most energy efficient. Memeti et al. [16] compare the programming productivity, performance, and energy use of CUDA, OpenACC, OpenCL and OpenMP for programming a system consisting of a CPU and GPU or a CPU and an Intel Xeon Phi coprocessor. They report that CUDA, OpenCL and OpenMP have similar performance and energy consumption in one benchmark, and that OpenCL performs better than OpenACC for another benchmark. In terms of productivity, the actual person writing the code is important, but OpenACC and OpenMP require

[2] Kernel launch overhead is on the same order of magnitude as in C++ in our experiments.

less effort than CUDA and OpenCL, and CUDA can require significantly less effort than OpenCL.

Previous studies have also shown that CUDA and OpenCL can compete in terms of performance as long as the comparison is fair [17,18,19,20] and there have also been proposed automatic source to source compilers that report similar results [21,22].

3. GPU Computing in Python

In this work we focus on using Python to access the GPU through CUDA and OpenCL. These two GPU programming models are conceptually very similar, and both offer the same kind of parallelism primitives. The main idea is that the computational domain is partitioned into equally sized subdomains that are executed independently and in parallel. Even though the programming models are similar, their terminology differs slightly, and in this paper we will use that of CUDA. A full review is outside the scope of this work, but can be found in [23,24]. The following sections give an overview of important parts of CUDA and OpenCL, and discuss their respective Python wrappers.

3.1. CUDA

CUDA [8] (Compute Unified Device Architecture) was first released in 2007, and is available on all Nvidia GPUs as Nvidia's proprietary GPU computing platform. It includes third-party libraries and integrations, the directive-based OpenACC [7] compiler, and the CUDA C/C++ programming language. Today, five of the ten fastest supercomputers (including number one) use Nvidia GPUs, as well as nine out of the ten most energy-efficient [25].

CUDA is implemented in the Nvidia device driver, but the compiler (nvcc) and libraries are packaged in the CUDA toolkit and SDK.[3] The toolkit and SDK contain a plethora of examples and libraries. In addition, the toolkit contains Nvidia Nsight, which is an extension for Microsoft Visual Studio (Windows) and Eclipse (Linux) for interactive GPU debugging and profiling. Nsight offers code highlighting, unified CPU and GPU trace of the application, and automatic identification of GPU bottlenecks. The Nvidia Visual Profiler is a stand-alone cross-platform application for profiling of CUDA programs, and CUDA versions for debugging (cuda-gdb) and memory checking (cuda-memcheck) also exist.

3.2. OpenCL

OpenCL [9] (Open Compute Language) is a free and open heterogeneous computing platform that was initiated by Apple in 2009, and today the OpenCL standard is maintained and developed by the Khronos group. Whilst CUDA is made specifically for Nvidia GPUs, OpenCL can run on a number of heterogeneous computing architectures including GPUs, CPUs, FPGAs, and DSPs. Contrary to CUDA, there is no official toolkit for OpenCL, but there are several third-party libraries.

[3] Available at `https://developer.nvidia.com/cuda-zone`.

Profiling an OpenCL application can be challenging, and the available tools vary depending on your operating system and hardware vendor.[4] It is possible to get timing information on kernel and memory transfer operations by adding counters and enabling event profiling information explicitly in your source code. This requires extra work and makes the code more complex. Visual Studio can measure the amount of run time spent on the GPU, and CodeXL [26] can be used to get more information on AMD GPUs. CodeXL is a successor to gDebugger which offers features similar to those found in Nsight in addition to power profiling, and is available both as a stand-alone cross-platform application and as a Visual Studio extension. While it is possible to use Visual Profiler for OpenCL, this requires the use of the command-line profiling functionality in the Nvidia driver, which needs to be enabled through environment variables and a configuration file. After running the program with the profiling functionality in effect, the profiling data can be imported into Visual Profiler.

One disadvantage of OpenCL is that there are large differences between the OpenCL implementations from different vendors, and good performance might rely on vendor-specific extensions. One example is that OpenCL 2.2 is required for using C++ templates in the GPU code, but vendors such as Nvidia only support OpenCL version 1.2. It should also be mentioned that OpenCL has been deprecated in favour of Metal [27] by Apple in their most recent versions of Mac OS X.

3.3. GPU computing from Python

Researchers spend a large portion of their time writing computer programs [28], and compiled languages such as C/C++ and Fortran have been the de facto standard within scientific computing for decades. These languages are well established, well documented, and give access to a plethora of native and third-party libraries. C++ is the standard way of accessing CUDA and OpenCL today, but developing code with these languages is time consuming and requires great care. Using higher-level languages such as Python can significantly increase development productivity [3,29]. However, it should be mentioned that the OpenCL and CUDA *kernels* themselves are not necessarily made neither simpler nor shorter by using Python: The productivity gain comes instead from Python's less verbose code style for the CPU part of the code. This influences every part of the host code, from boilerplate initialization code and data pre-processing, to CUDA/OpenCL API calls, post-processing, and visualization of results.

PyCUDA and PyOpenCL [4] are Python packages that offer access to CUDA and OpenCL, respectively. Both libraries expose the API of the underlying programming models, and try to minimize the performance impact. The GPU kernels, which are crucial for the inner loop performance, are written in native low-level CUDA or OpenCL, and memory transfers and kernel launches are made explicit through Python functions. The result is an environment suitable for rapid prototyping of high-performing GPU code in a Python environment.

[4] An extensive list of OpenCL debugging and profiling tools can be found at https://www.khronos.org/opengl/wiki/Debugging_Tools.

Table 1. A list of the GPUs used in this work. The profile-driven development was carried out on the 840M and GTX780, and all three GPUs were used to benchmark computational performance and power efficiency. Note that the performance in gigaFLOPS is for single precision.

Model	Class	Architecture (year)	Memory	GigaFLOPS	Bandwidth	Power device
GeForce 840M	Laptop	Maxwell (2014)	4 GiB	790	16 GB/s	Watt meter
GeForce GTX780	Desktop	Kepler (2013)	3 GiB	3977	288 GB/s	Watt meter
Tesla V100	Server	Volta (2017)	16 GiB	14899	900 GB/s	nvidia-smi

4. Benchmarking performance and power efficiency

Throughout this section, we consider simulation of the shallow-water equations using three different numerical schemes:

- a linear finite difference scheme,
- a nonlinear finite difference scheme, and
- a high-resolution finite volume scheme.

We benchmark different versions for each of the three codes with respect to computational performance and power efficiency on three different GPUs. The schemes are used for simulating real-world ocean currents, and two of them have been used operationally. All three schemes are essentially stencil operations with an increasing level of complexity, and their details are summarized in Holm et al. [30].

4.1. Profile-driven optimization and porting

Our starting point is OpenCL implementations of the three numerical schemes [30,31], and even though the code is algorithmically well suited for the GPU, little effort has been made to thoroughly optimize its performance on a specific GPU. It is well known in the GPU computing community that performance is not portable between GPUs, neither for CUDA nor OpenCL, and automatically generating good kernel configurations is an active research area (see e.g., [32,33,34]). Using the available profiling tools for CUDA, we perform profile-driven optimization. This is an iterative process that starts by profiling the code to identify the main performance bottleneck. The next step is then to optimize this bottleneck, before running tests that ensure that the optimization did not introduce bugs to the code. Thereafter the code is analysed in the profiler again in search for a new bottleneck [35]. The profiling and tuning is carried out mainly on a laptop with a dedicated GeForce 840M GPU, representing the low-end part of the GPU performance scale, and some final tuning was performed on a desktop with a GeForce GTX780 GPU, representing a typical mid-range GPU. We compare the performance of the original and optimized implementations with PyCUDA and PyOpenCL using these two GPUs, as well as on a recent high-end Tesla V100 GPU commonly found in servers and super computers. Information on these three GPUs is listed in Table 1. Together they represent the different types of GPUs a typical researcher might have access to in a research environment: a laptop and/or a desktop with local access and an expensive powerful server GPU in a remote location.

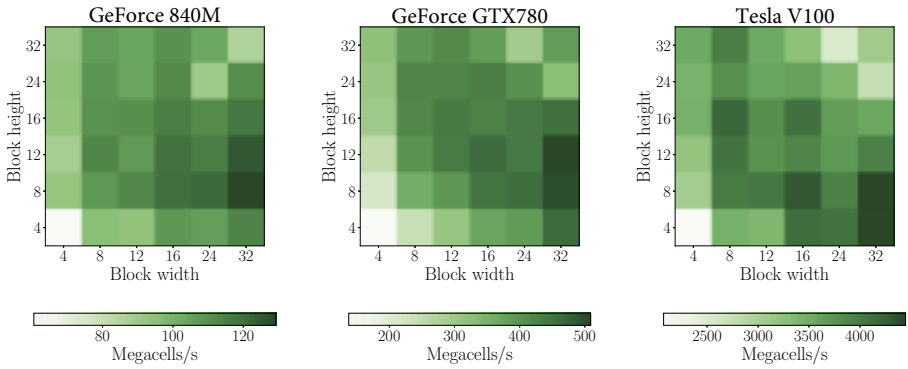


Figure 1. Heat map of performance as a function of block width and block height for selected sizes for the high-resolution scheme on three different GPUs. Notice that even though the performance patterns have similarities, the performance on the V100 would be suboptimal if the optimal configuration from the GTX780 is used. The performance increase is $2-5\times$ for all three GPUs from the slowest to the fastest block size.

An important performance parameter for GPUs is the domain decomposition determined by the block size. CUDA decomposes the work into a grid with equally sized blocks, and all blocks are executed independently. At runtime, the GPU takes the set of blocks and schedules them to the different cores within the GPU. Using a too small block size will under-utilize the GPU, and using a too large block size will similarly exhaust the GPU's resources. Figure 1 shows how the block size has a major impact on performance, and also illustrates that finding the best block size can be difficult. Because of this, we experimentally obtain the optimal configuration for each scheme before starting profiling, and again after the code has been optimized.

The porting process between PyOpenCL and PyCUDA requires changes in both the kernel code that runs on the GPU, and the API calls in the CPU code. Most of the changes in the kernels are straight-forward and consist of changing keywords due to different terminologies [21,22]. The CPU APIs are however quite different between OpenCL and CUDA, but their Python wrappers reduce the differences somewhat. It still requires correct handling of devices, contexts and streams, compiling and linking kernels, setting kernel arguments, grid and block dimensions, and memory transfers. Furthermore, there is also a large difference in availability of native and third-party libraries, built-in and specialized functions, and data types that need to be handled.

In our case, the linear and nonlinear schemes were found to be very memory-bound. Both schemes were originally implemented with three separate kernels (one for each of the three variables in the shallow-water equations), with a large overlap in the data dependency. The first tuning step consisted of fusing the three kernels into one, and thus reduced the overall memory bandwidth requirement of both schemes.

Profiling the high-resolution scheme revealed that the occupancy was very low due to excessive use of shared memory. The optimizations consisted of largely reducing the amount of shared memory used, and instead relying more on re-computations. When shared memory usage was no longer the performance bottleneck, we reduced the number of registers used per thread. All three schemes were further optimized through the use of compiler flags.

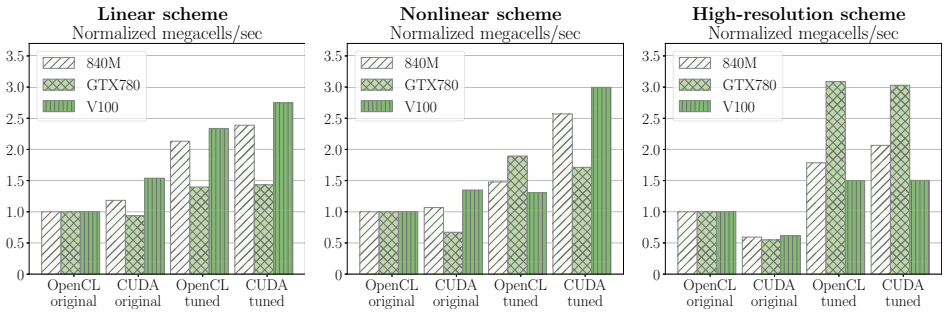


Figure 2. Performance of original, ported, and optimized codes measured in megacells per second, normalized with respect to performance of the original OpenCL implementation. Notice that there is relatively little difference between CUDA and OpenCL, except for the nonlinear scheme on the 840M and V100, whilst there is a significant difference in how effective the tuning is for the different architectures. Furthermore, there is a significant loss of performance when porting from OpenCL to CUDA in our original approach for the high-resolution scheme. From our experience, this relates to how the two languages optimize mathematical expressions with and without the fast-math compilation flags.

4.2. Comparing performance

The overall performance gain of our optimization is shown in Figure 2, where all results are given in megacells per second normalized with respect to the original PyOpenCL implementation. The original porting from PyOpenCL to PyCUDA gave a noticeable reduction in performance for the high-resolution scheme on all GPUs. After careful examination, we attribute this to different default compilation flags in PyCUDA and PyOpenCL: In PyCUDA, the fast-math flag was shown to double the performance for the high-resolution scheme, while we found that it gave less than 5% performance gain with PyOpenCL. Note that this affects the linear and nonlinear schemes to less extent, as these schemes contain fewer complex mathematical operations, and we instead observe a varying effect on performance of porting the original OpenCL code to CUDA. When examining the numerical schemes one by one, we see that the optimizations performed for the high-resolution scheme appears to be highly portable when back-ported to PyOpenCL for all GPUs. For the tuned nonlinear scheme, however, we see that the 840M and V100 GPUs give significantly higher performance using CUDA than OpenCL, but similar performance on the GTX780. Finally, for the linear scheme, the tuning gives a similar increase in performance both on CUDA and OpenCL, relative to the original versions. In total, we see that certain scheme and GPU combinations result in a significant speedup for CUDA over OpenCL, but we cannot conclude whether this is caused by differences in driver versions or from other factors. We are therefore not able to claim that CUDA performs better than OpenCL in general.

4.3. Measuring power consumption

We measure power consumption in two ways. The first method is by using the nvidia-smi application, which can be used to monitor GPU state parameters such as utilization, temperature, power draw, etc. By programmatically running nvidia-smi in the background during benchmark experiments, we can obtain a log containing a high-resolution power

draw profile for the runtime of the benchmark. The downside of using nvidia-smi is that information about power draw is only supported on recent high-performance GPUs. From our selection of GPUs, this applies only to the Tesla V100 GPU. Further, nvidia-smi monitors the energy consumption of the GPU only, meaning that we do not have any information about energy consumed by the CPU. For each experiment, nvidia-smi is started in the background exactly 3 s before the benchmark, and is configured to log the power draw every 20 ms. This background process is stopped again exactly 3 s after the end of the simulation. This approach allows us to measure the energy consumption of the idle GPU both before and after each benchmark, and we ignore the idle sections when computing the mean and total power consumption for each experiment. All results presented here are with the idle load subtracted from the experimental results.

The second method is to measure the total amount of energy used by the entire computer through a watt meter. The use of the watt meter requires physical access to the computer, and we are therefore restricted to do measurements on the laptop and desktop, containing the GeForce 840M and GeForce GTX780 GPUs, respectively. The watt meter offers no automatic logging or reading, but displays the total power used with an accuracy of 1 Wh. To get sufficiently accurate readings we need to run each benchmark long enough to keep the GPU busy for approximately one hour, after which we read the total and mean power consumption for each experiment. Before and after each benchmark, we also record the background power of the idle system, and the maximum recorded power during the experiment, to monitor whether the operating system is putting any non-related background load on the computer. It should also be noted that the battery was removed from the laptop during these experiments. Similarly to the first method, we subtract the idle loads from the result of each experiment, but we will here have the increased power load on the CPU also included.

4.4. Comparing power consumption

Figure 3 shows the results from the power efficiency experiments using the watt meter on the laptop (840M) and desktop (GTX780). The top row repeats the results for computational performance also shown in Figure 2 for the relevant GPUs, whereas the second row show the normalized mean power consumption with respect to the original OpenCL versions. The first thing we notice is that CUDA seems to require less power on the 840M compared to OpenCL for all versions of the three schemes. On the GTX780, however, there are no differences between the two programming models for equivalent versions. In fact, only the tuned high-resolution scheme seems to be different from the others (using about 30% more power), and this behavior can also be seen on the 840M. The power efficiency of the three schemes is shown in the bottom row in Figure 3, and we see that on the 840M the tuned CUDA versions are the most power efficient. This is because CUDA is both more efficient and uses less power on this system. On the GTX780, CUDA and OpenCL have equivalent power efficiency for all tuned schemes.

The results for the Tesla V100 are shown in Figure 4. The top row shows the computational performance in megacells per second, repeating the results from Figure 2. The second row shows the mean power used by each version of the three schemes. Note here that both CUDA versions for the nonlinear scheme use 60-90% more power than the OpenCL versions, which is the opposite result compared to the 840M results. For both the linear and high-resolution schemes, the results do not differ significantly in favor of

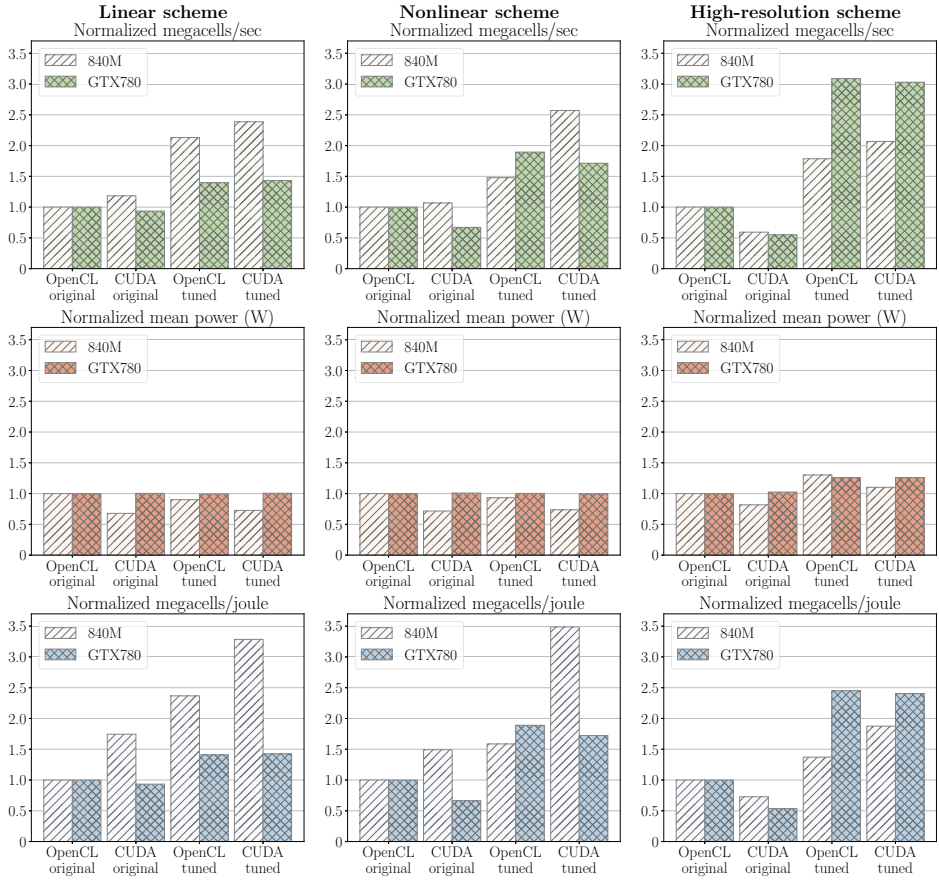


Figure 3. Comparison of original, ported and optimized codes measured in megacells per second (top row), mean power usage (mid row), and megacells per joule (bottom row), normalized with respect to the original OpenCL implementation, for the laptop (840M) and desktop (GTX780) GPUs. The power is measured through a watt meter, and represents the power consumed by the entire computer. Note that the CUDA versions requires less power than the OpenCL versions on the 840M, whereas there are no differences between equivalent versions on the GTX780. In terms of power efficiency, CUDA is more efficient than OpenCL on the M840, whereas the GTX780 gives the same power efficiency.

either CUDA or OpenCL, but the tuned OpenCL version uses slightly more power for the high-resolution scheme. When we consider the power efficiency in the bottom row, we see that the tuned CUDA versions are the best versions for all schemes. In particular for the nonlinear scheme, this is mostly due to the large difference in computational efficiency between CUDA and OpenCL for this particular scheme on this particular GPU.

In general, we observe that all experiments show a mean power usage within about 30% of the original OpenCL versions, with the exception of the nonlinear scheme on the V100. On the other side, the computational performance increases up to 5 times (the high-resolution scheme on the GTX780). This shows that the most important factor for improving power efficiency is to increase computational performance.

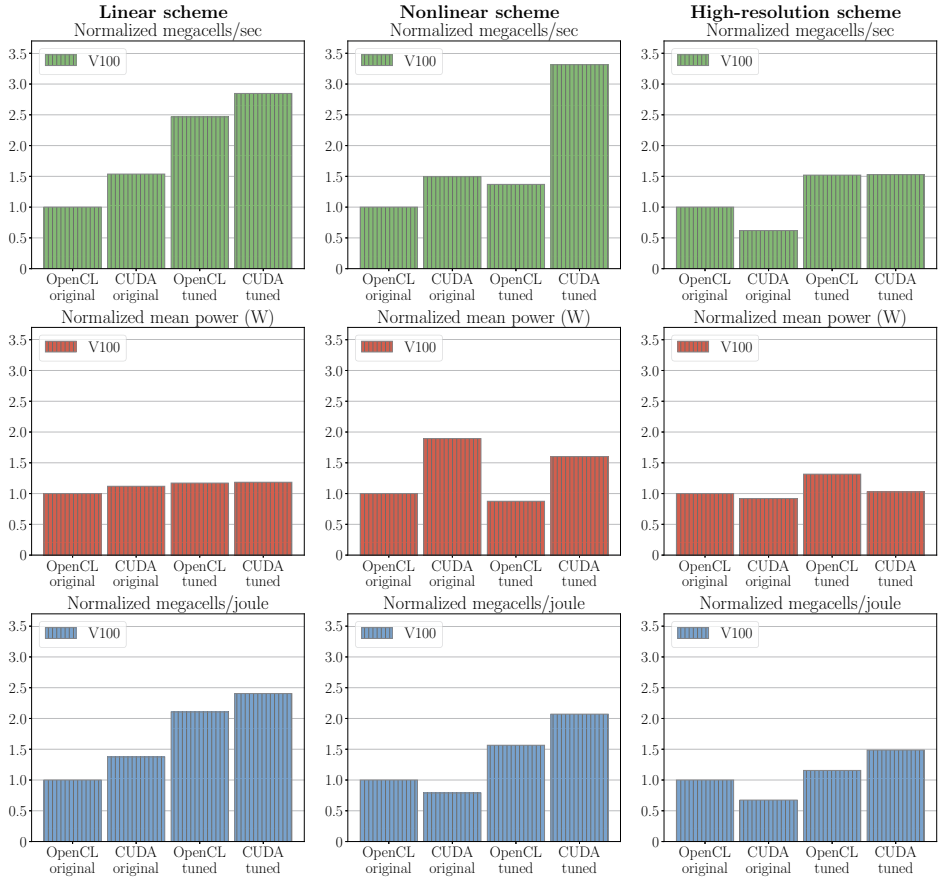


Figure 4. Comparison of original, ported and optimized codes measured in megacells per second (top row), mean power usage (mid row), and megacells per joule (bottom row), normalized with respect to the original OpenCL implementation, for the Tesla V100 GPU. The power is measured through nvidia-smi, and represents the power consumed by the GPU only. There are only minor differences in mean power consumption between different versions of the linear and high-resolution scheme, but CUDA uses more power than OpenCL for the nonlinear scheme. CUDA is however more power efficient than OpenCL for all three tuned schemes.

5. Summary

In this work, we have benchmarked three different OpenCL codes, our ported code in CUDA, our optimized CUDA code, and finally our OpenCL code with optimizations found using the CUDA tools. Our results are shown for three different GPUs, thus representing many of the GPU architectures in use today. Finally, we have looked at the power consumption for all versions of the code.

The original motivation for using OpenCL was to support GPUs and similar architectures from multiple vendors. Our motivation for changing from OpenCL to CUDA was because of the better software ecosystem for CUDA, and we have been very happy with our choice. CUDA appears to be a much more stable and mature development ecosystem with better tools for development, debugging and profiling for our hardware.

We found it interesting that our initial port from OpenCL to CUDA imposed a performance penalty, due to different default compiler optimizations. Even though some authors have reported OpenCL to be slower than CUDA, we find no conclusive results that support this in general. The performance gain varied much more with the GPU being used than whether we used CUDA or OpenCL. However, we do observe that for certain combinations of scheme and GPU, we get a significant speedup for CUDA over OpenCL. Additionally, we found that the performance gain of tuning the numerical schemes have vastly different effects on the run time for different GPUs. Even though we profiled and optimized mainly using a laptop GPU, the highest relative performance gain was for a server class and a desktop class GPU.

There does not seem to be any clear relations between the power consumption when comparing different schemes, optimization levels, GPUs, and programming models. When we look at power efficiency, we see that CUDA performs better than OpenCL for all tuned schemes on the Tesla V100 and GeForce 840M GPUs, whereas there are no differences on the GeForce GTX780. When we examine the impact of performance tuning on power efficiency, there appears to be a systematic and clear relationship: A fast code is a power efficient code.

6. Acknowledgments

This work is supported by the Research Council of Norway through grant number 250935 (GPU Ocean). The GPU Ocean project acknowledges the support from UNINETT Sigma2 - the National Infrastructure for High Performance Computing and Data Storage in Norway under project number nn9550k.

References

[1] E. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, SC '01, pages 55–55. ACM, 2001.

[2] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, and T. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[3] S. Nanz and C. Furia. A comparative study of programming languages in rosetta code. In *IEEE International Conference on Software Engineering*, volume 1, pages 778–788, 2015.

[4] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Computing*, 38(3):157–174, 2012.

[5] T. Hagen, M. Henriksen, J. Hjelmervik, and K-A. Lie. *How to Solve Systems of Conservation Laws Numerically Using the Graphics Processor as a High-Performance Computational Engine*, pages 211–264. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[6] A. Brodtkorb, M. Sætra, and M. Altinakar. Efficient shallow water simulations on GPUs: Implementation, visualization, verification, and validation. *Computers & Fluids*, 55(0):1–12, 2012.

[7] OpenACC-Standard.org. The OpenACC application programming interface version 2.7, 2018.

[8] NVIDIA. NVIDIA CUDA C programming guide version 10.1, 2019.

[9] Khronos OpenCL Working Group. The OpenCL specification v. 2.2, 2018.

[10] K. Asanović, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical report, EECS Department, University of California, Berkeley, 2006.

[11] A. Brodtkorb, C. Dyken, T. Hagen, J. Hjelmervik, and O. Storaasli. State-of-the-art in heterogeneous computing. *Scientific Programming*, 18(1):1–33, 2010.

[12] S. Huang, S. Xiao, and W-C. Feng. On the energy efficiency of graphics processing units for scientific computing. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–8. IEEE, 2009.

[13] Z. Qi, W. Wen, W. Meng, Y. Zhang, and L. Shi. An energy efficient opencl implementation of a fingerprint verification system on heterogeneous mobile device. In *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 1–8. IEEE, 2014.

[14] T. Dong, V. Dobrev, T. Kolev, R. Rieben, S. Tomov, and J. Dongarra. A step towards energy efficient computing: Redesigning a hydrodynamic application on CPU-GPU. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 972–981. IEEE, 2014.

[15] V. Klôh, D. Yokoyama, A. Yokoyama, G. Silva, M. Ferro, and B. Schulze. Performance and energy efficiency evaluation for HPC applications in heterogeneous architectures. 2018.

[16] S. Memeti, L. Li, S. Pllana, J. Kołodziej, and C. Kessler. Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: programming productivity, performance, and energy consumption. In *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, pages 1–6. ACM, 2017.

[17] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing*, 38(8):391–407, 2012.

[18] J. Fang, A. Varbanescu, and H. Sips. A comprehensive performance comparison of CUDA and OpenCL. In *2011 International Conference on Parallel Processing*, pages 216–225, 2011.

[19] T. Gimenes, F. Pisani, and E. Borin. Evaluating the performance and cost of accelerating seismic processing with CUDA, OpenCL, OpenACC, and OpenMP. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 399–408, 2018.

[20] K. Karimi, N. Dickson, and F. Hamze. A performance comparison of CUDA and OpenCL. Technical report, D-Wave Systems Inc., 2011.

[21] G. Martinez, M. Gardner, and W c. Feng. CU2CL: A CUDA-to-OpenCL translator for multi- and many-core architectures. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, pages 300–307, 2011.

[22] J. Kim, T. Dao, J. Jung, J. Joo, and J. Lee. Bridging OpenCL and CUDA: a comparative analysis and translation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[23] D. Kaeli, P. Mistry, D. Schaa, and D. Zhang. *Heterogeneous Computing with OpenCL 2.0*. 2015.

[24] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.

[25] University of Mannheim, University of Tennessee, and NERSC/LBNL. Top 500 supercomputer sites. http://www.top500.org, June 2019.

[26] AMD Developer Tools Team. CodeXL quick start guide, 2018.

[27] Apple Inc. Metal programming guide, 2018.

[28] G. Wilson, D. Aruliah, C. Brown, N. Chue Hong, M Davis, R. Guy, S. Haddock, K. Huff, I. Mitchell, M. Plumbley, B. Waugh, E. White, and P. Wilson. Best practices for scientific computing. *PLOS Biology*, 12(1):1–7, 2014.

[29] L. Prechelt. An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program. Technical report, Karlsruhe Institute of Technology, 2000.

[30] H. Holm, A. Brodtkorb, K. Christensen, G. Broström, and M. Sætra. Evaluation of selected finite-difference and finite-volume approaches to rotational shallow-water flow. *Communications in Computational Physics*, 2019. [to appear].

[31] A. Brodtkorb. Simplified ocean models on the GPU. In *2018: Norsk Informatikkonferanse*, 2018.

[32] R. Singh, P. Wood, R. Gupta, S. Bagchi, and I. Laguna. Snowpack: Efficient parameter choice for GPU kernels via static analysis and statistical prediction. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, pages 8:1–8:8, New York, NY, USA, 2017. ACM.

[33] J Price and S. McIntosh-Smith. Analyzing and improving performance portability of OpenCL applications via auto-tuning. In *Proceedings of the 5th International Workshop on OpenCL*, pages 14:1–14:4, New York, NY, USA, 2017. ACM.

[34] T. Falch and A. Elster. Machine learning-based auto-tuning for enhanced performance portability of OpenCL applications. *Concurrency and Computation: Practice and Experience*, 29(8), 2017.

[35] A. Brodtkorb, T. Hagen, and M. Sætra. Graphics processing unit (GPU) programming strategies and trends in GPU computing. *Journal of Parallel and Distributed Computing*, 73(1):4–13, 2013.

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200090

Computational Performances and Energy Efficiency Assessment for a Lattice Boltzmann Method on Intel KNL

Ivan GIROTTO [a,c,d,1], Sebastiano Fabio SCHIFANO [b] Enrico CALORE [b]
Gianluca DI STASO [c] and Federico TOSCHI [c]

[a] *The Abdus Salam, International Centre for Theoretical Physics. Italy*
[b] *University of Ferrara and INFN Ferrara, Italy*
[c] *Eindhoven University of Technology, The Netherlands*
[d] *University of Modena and Reggio Emilia, Italy*

Abstract. In this paper we report results of the analysis of computational performances and energy efficiency of a Lattice Boltzmann method (LBM) based application on the Intel KNL family of processors. In particular we analyse the impact of the main memory (DRAM) while using optimised memory access patterns to accessing data on the on-chip memory (MCDRAM) configured as cache for the DRAM, even when the size of the data of the simulation fits the capacity of the on-chip memory available on socket.

Keywords. Lattice Boltzmann method, KNL, Cache mode, Flat mode, energy efficiency, computational performances

1. Introduction

The LBM [1] is widely used in computational fluid-dynamics to study the behaviour of fluid flows. Computational fluid-dynamics studies requires a huge amount of computational resources when simulating 3-dimensional systems at high-resolution. To study the behaviour of multicomponent emulsions, an amount of 40 millions core-hours was allocated, for a year, on the KNL partition of the MARCONI Tier-0 system, hosted at CINECA, within the PRACE 17th call, project named "TurEmu - The physics of (turbulent) emulsions". To successfully complete the project and perform the simulations on a large number of KNL processors, we developed a LBM based application which delivers good scaling performances on distributed systems while optimising the memory access through a data organisation that enables high computing efficiency. The overall code optimisation, as well as the optimised data layouts were introduced in [2], including a performance analysis in terms of both computing and energy consumption on the KNL processor. Other similar analysis were also previously studied on different computer architectures [3,4,5].

[1] Corresponding Author: Ivan Girotto; The Abdus Salam, International Centre for Theoretical Physics; Strada Costiera, 11 - 34151 Trieste - IT; E-mail: igirotto@ictp.it

In this paper we present a new analysis of computational performances and energy efficiency of our code on the Intel KNL family processors when using the on-chip memory as cache for the main memory even when the size of the data of the simulation fits the capacity of the on-chip memory available per socket. We show how optimized memory access patterns are efficient in using the on-chip memory. Indeed, accessing only the faster on-chip memory with generic data layouts, does not bring any benefit over using it as an additional cache level between the cores and the DRAM. On the other hand, we demonstrate that optimized data layouts allow our LBM application to fully exploit the on-chip memory, obtaining an higher performance and making it more efficient to use this memroy exlcusively, without involving the DRAM.

The rest of the paper is composed as follow: in section 2 we describe the main technical aspects of the KNL processor relevant to our analysis, in section 3 we introduce the scientific context we have been working on underlining the relevance of the performance analysis, in section 5 we present the performance results of the code on the KNL processors while, in section 6, an analysis on energy efficiency is reported.

2. The Marconi Tier-0 System and The KNL Processor

The KNL processor is equipped with 6 Double Data Rate fourth-generation (DDR4) channels at support of up to 384 GB, depending on the size of the memory modules, of synchronous Dynamic Random-Access Memory (DRAM) with a peak raw bandwidth of 115.2 GB/s. The processor also includes four high-speed memory banks based on the Multi-Channel DRAM (MCDRAM) that provides 16 GB of memory, capable to deliver an aggregate bandwidth of more than 450 GB/s when accessing the on-package high-bandwidth memory. Therefore, maximize the usage of the MCDRAM is a key aspect to achieve great performance for memory bounded applications. MCDRAM on a KNL can be configured at boot time in Flat, Cache or Hybrid mode. The Flat mode defines the whole MCDRAM as addressable memory allowing explicit data allocation, whereas Cache mode uses the MCDRAM as a last-level cache.

The Marconi KNL (A2 partition) was deployed at the end of 2016 and consists of 3600 Intel server nodes integrated by Lenovo. Each node contains 1 Intel Knights Landing processor with 68 cores. The entire system is connected via the Intel OmniPath network. CINECA provides to PRACE users, as well as to all other users the KNL nodes configured in Cache mode. Probably this is mainly due to the fact that performances of common applications are comparable on the KNL configured either in Cache mode or Flat mode when the size of the data of the application fits the 16 GB capacity of the MCDRAM, and the on-chip memory is used as cache of the DRAM memory. Indeed, as we show in this work, this is also confirmed by our analysis on LBM based applications when using common data layouts. On the other hand the Cache mode configuration is more flexible giving users applications the opportunity to exploit an higher capacity of memory per socket while relying on the MCDRAM as cache to maximise the memory throughput.

As we want to measure the impact of the DRAM when the KNL is configured in Cache mode and the dataset fits the memory available on the MCDRAM, we perform the performance analysis as well as the measures of energy efficiency on a 64-core Intel Xeon Phi CPU 7230 processor installed in our laboratory. The authors consider the sys-

tem equivalent to the one installed on Marconi as far as the present analysis is concerned. However, using a stand alone processor gives us the flexibility to make a number of tests by rapidly switching the configuration of the processor between Cache and Flat mode along with the opportunity to use the processor with root privileges that are required for measurements of energy efficiency. We only consider the Quadrant cluster configuration in which the 64-cores available are divided in four quadrants, each directly connected to one MCDRAM bank. The same configuration is used on Marconi for the PRACE projects production.

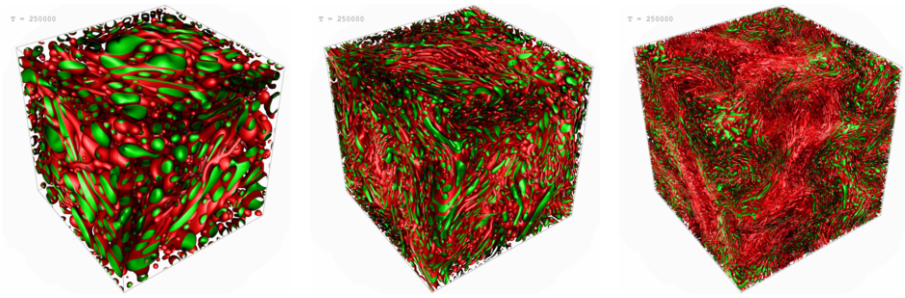
In this work, we used Intel library for Message Passing Interface (MPI) to compile the LBE3D application. Compiler auto-vectorization is activated at compile time using the -xMIC-AVX512 Intel compiler option. Multi-thread version of LBE3D is implemented using OpenMP and enabled at compile time. Threads affinity at run time is obtained with "KMP_AFFINITY=compact" and "I_MPI_PIN_DOMAIN=socket" environment variables. Memory allocation for all results related to the KNL configured in Flat mode is made by using the "`numactl -m 1 mpirun ./lbe3d`" command.

3. The Lattice Boltzmann Method

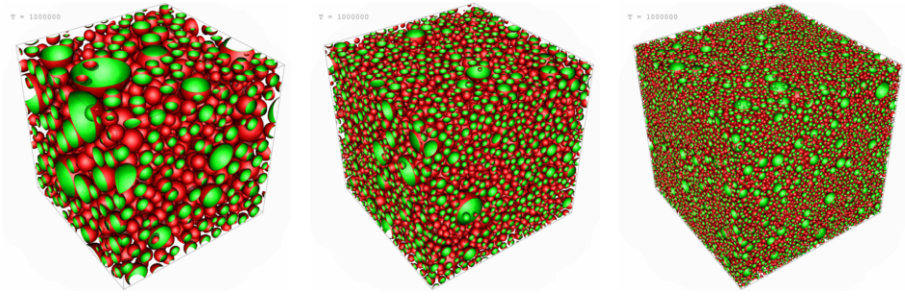
Multi-component fluids are extremely common in industrial as well as natural processes. In particular, multi-component fluids emulsions are an important ingredient of many foods and cosmetics, and at the same time fascinating systems from the point of view of fundamental science, due to the rich fluid dynamic phenomenology. To explore the physics of complex fluid emulsions we employ high-resolution and high-statistics simulations, via optimised computational codes based on the multicomponent LBM.

We implemented a largely-scalable and highly-optimised LBM based code to study high-resolution stabilised emulsions on a 3-dimensional domain. In particular the objective is to explore the physics of complex fluid emulsions using disjoining pressure at the surface between the two components [6]: from their production, via turbulent stirring, to their (statistical) behaviour under flowing, as well as, under resting jammed conditions (figure 1). We are interested in collecting a large statistics aimed at providing a detailed analysis of the emulsion morphology (e.g. droplet size distribution) of the various investigated systems by varying the turbulent forcing, the resolution, as well as the volume fraction of the two components.

LBM, recently emerged as a popular numerical method for computational fluid dynamics applications, aims to solve a discretized version of the Boltzmann Equation. Due to its ability to fulfil both symmetry and conservation constraints needed to reproduce hydrodynamics in the continuum limit, LB could be viewed as a kinetic solver for the Navier-Stokes equations. From an algorithmic point of view, the method is based on a splitting approach: a free flow streaming step (`propagate`) is followed by a collision step (`collide`). During the streaming step, the discrete probability distribution functions (frequently named as populations), arranged to form a lattice at each grid site, propagate along a predetermined number of molecular trajectories and with an assigned speed so that at each time step they can only move to next-nearest neighbor grid sites. The collision step, instead, features a relaxation process towards equilibrium determined according to the local flow properties. Such splitting approach guarantees clear advantages in practical implementations as a relevant number of parallelisms can be identified. This



(a) 256^3 (b) 512^3 (c) 1024^3



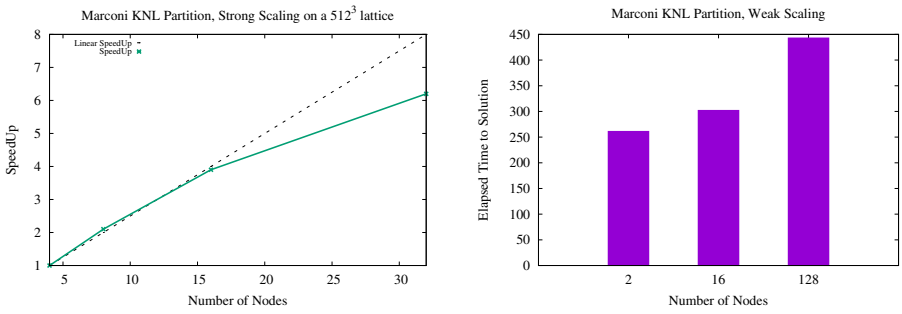
(d) 256^3 (e) 512^3 (f) 1024^3

Figure 1. Via turbulence stirring we could produce emulsions at different numerical resolutions, from top to bottom, 256^3, 512^3 and 1024^3. As it can be observed we obtain stretched droplets when turbulence is turned on (a-c) and jammed compositions of spherical droplets (d-f), with a volume fraction of approximately 40% of the droplets phase, in absence of turbulence.

makes LBM an ideal tool to investigate performances of modern high-performance computing systems [7,8,9]. In particular we use these two kernels to measure peak performance and memory bandwidth. Indeed, the `propagate` kernel is characterised by a data movement in memory with no floating point operations while the `collide` kernel includes high number of floating point operations. By counting the floating points operation used to implement the (`collide`) kernel we can estimate the peak performance at runtime. However, the measurement presented in this paper is only a reference for a generic LBM based application as the number of floating point operations implemented within the `collide` kernel changes accordingly to the physics of the numerical model. We adopt the D3Q19 LBM stencil, a 3-dimensional model with a set of 19 population elements corresponding to (pseudo)-particles moving one lattice point away along all 19 directions. Therefore, due to the complexity of the model and the number of simulations we plan to investigate the physical (statistical) behaviour of the system by varying the initial configuration on a parameter space as well as the resolutions, we are strongly motivated to optimise and analyse in details the performance of the main kernel of our 3d-dimensional implementation of the LBM. Indeed, this requires a large amount of computational resources, especially considering the two lattices needed to describe multicomponent emulsions.

4. Code Optimisations

Legacy data structure such as Array of Structures (AoS) or Structure of Arrays (SoA) are commonly used to implement stencil based applications, including LBM. In [2] we have described two additional data structure that can be used for LBM based application but delivering better performances if compared with canonical AoS or SoA data layouts. The two new layouts called CSoA and CAoSoA are seen as an extension of the SoA where VL lattice-site data at distance L/VL (L dimension of major order store) are clustered in consecutive elements for each population array, with VL equal to the number of double precision elements that can be stored in the vector register available on the given architecture. The newly designed layouts for high-performance LBM based codes allow to store data properly aligned in memory and aiding the compiler in the process of auto-vectorization of the steps required to compute the LBM main loop. The CAoSoA structure is a mix between CSoA and AoS, and allows to exploit the benefit of the VL clusterization of lattice sites element as introduced by the CSoA schema but with the benefit of higher locality in regards to the populations. In [2] we have also described the optimisation implemented in our LBM code by fusing the `collide` and `propagate` kernels, avoiding to store intermediate hydrodynamic quantities when not needed. In the following analysis we refer with CF to the classical schema implemented in high-performance LBM based applications where the `propagate` and the `collide` kernel are separated and the density stored on a separate structure, and with FF the fully fused version where the two kernels are compact and the density only temporary computed.



(a) Strong Scaling (b) Weak Scaling

Figure 2. Strong and weak scaling of the LBE3D code using the CSoA data layout on the KNL partition of Marconi. The strong scaling chart shows the scaling achieved on a 512^3 dataset by increasing the number of nodes with a reference initial point of 4 KNL nodes. The weak scaling chart reports how the code scales in function on the number of node by fixing the amount of work per process. In this case a 256^3 lattice is for the benchmark on 2 nodes, a 512^3 lattice on 16 nodes and a 1024^3 lattice on 128 nodes

5. Performance Measurements

In [2] we have been reporting performance measurements mainly related to the KNL when configured in flat mode and analysing the scaling as function of the number of thread used per SMT core. As that work demonstrated how the best configuration was

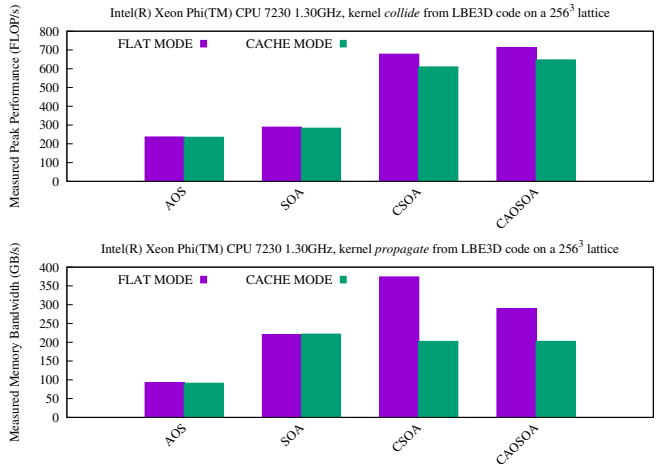


Figure 3. For the KNL processor configured in both FLAT mode and CACHE mode, we report in the upper chart the measured peak performance for the `collide` kernel while in the bottom chart the measured memory bandwidth for the `propagate` kernel. The performance measurement is reported using a single MPI process and 64-threads, one per each SMT KNL core. The performance measurement is performed on a lattice size of 256^3 which represent a real workload but at the same time fits the 16 GB capacity of the MCDRAM available on a single KNL socket.

achieved when using a single thread per SMT core, in this new analysis we only refer to configuration where the used number of thread is equivalent to the number of available cores on socket. Only a brief reference to the cache mode configuration was reported to show the performance impact when considering larger data-set that would not fit the 16 GB capacity of the MCDRAM memory. In other words, we were considering the KNL configured in cache mode only as the unique alternative when having to deal with dataset larger than 16 GB. However, as the Marconi KNL partition is configured in cache mode and we were allocated a significant amount of computer resources on the system for the PRACE project production we analysed performances also when using smaller data structures. Indeed, the LBE3D code demonstrated to efficiently scale on a large number of nodes and while scaling up we can reduce the amount of data per single compute node up to the point that we can always fit the 16 GB capacity of the MCDRAM memory, figure 2.

The first analysis was performed to measure the performance of the memory bandwidth using the `propagate` kernel in both configurations while using the same data-set and varying the presented data layouts. In figure 3 the result are reported. It is interesting to see how, only when using highly optimised data layouts such as CSoA and CAoSoA, there is a significant boost in terms of memory bandwidth moving from the Cache mode to the Flat mode. In particular for the CSoA data layout we could achieve a 50% of performance improvement in memory bandwidth for the same data set and the same code, simply switching the KNL configuration. Much lower impact is instead measured if considering the peak performance (FLOP/s) as reported for the `collide` kernel in figure 3. In this case we still register a significant boost of the highly-optimised data layouts if compared with the more common layouts but there is not the same performance lost when comparing the KNL processor configured in Flat or in Cache mode. In this case

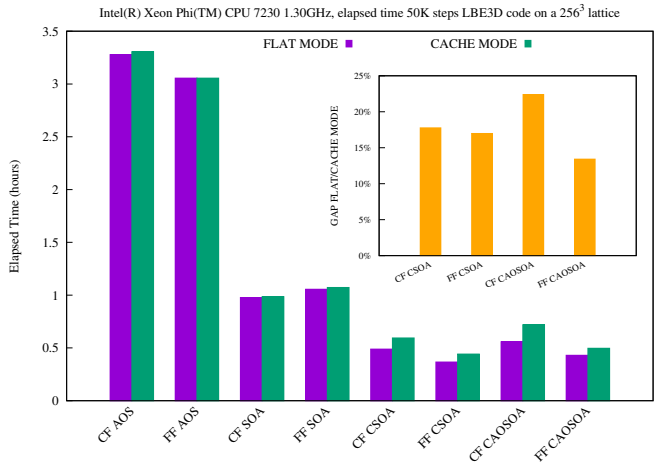


Figure 4. Measured time to solution for the LB main loop for all considered data layout in the fused fashion. Data are reported for KNL configured in cache mode as well as in flat mode. The inset reports the ratio between Flat and Cache mode of both CF and FF versions of the code for the optimised data layouts. The same setting and lattice size as for figure 3 is used also here.

only a performance benefit of about 10% is registered if comparing the two configurations.

Other than comparing the single kernels of `propagate` and `collide` we have also compared the performance gap between two configuration while considering the entire LB loop which includes boundary exchange and other minor kernels, figure 4. Again the results confirm the trend reported by measuring the single kernels. First it is possible to notice how the common layout AoS and SoA do not show any performance difference between the two KNL configurations. On the other hand, as reported in the inset of figure 4 the highly-optimized layouts provide a performance benefit between 15% to 23% when considering the KNL configured in Flat mode rather then in Cache mode.

6. Energy Efficiency

We now consider energy efficiency for the LBE3D across the multiple data layouts presented in both the CF and the FF version of the code. Again we want to analyse the energy consumption in both Flat and Cache mode using a data-set that fits the on-chip memory capacity and try to understand what is the impact of the DRAM memory utilisation. We use data from the Running Average Power Limit (RAPL) register counters available in the KNL read through the custom library developed in [10]. In Figure 5, we show the measured values of energy consumption (million Joule) for the LBE3D application, respectively, for the processor and for the off-chip DRAM memory. The DRAM energy consumption is generally lower as the KNL is configured in Flat mode and during the simulation data are all stored into the MCDRAM. Indeed, energy measurements for the DRAM memory takes into account just the idle consumption, while the MCDRAM energy consumption is accounted in the CPU Package value, since MCDRAM is an on-package memory, such as caches.

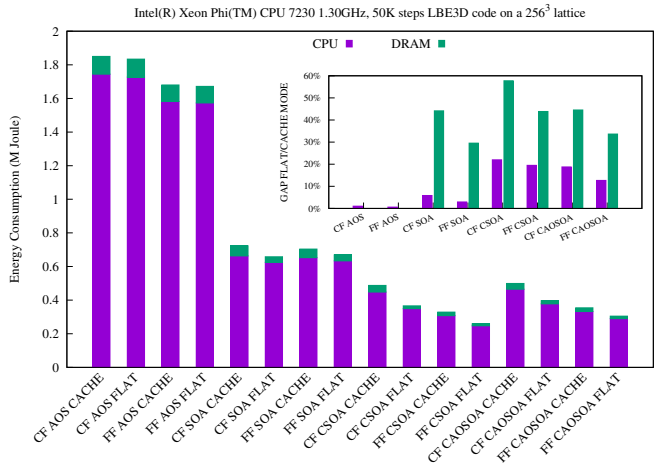


Figure 5. Measure of energy consumption for the LB main loop for all considered data layout in the fused fashion. Data are reported for KNL configured in cache mode as wall as in flat mode. The inset reports the ration between the fused versions in term of both CPU and DRAM. The same setting and lattice size as for figure 3 is used also here.

It is relevant to notice that, despite the CSoA and CAoSoA data layouts are expected to stress the CPU system more than the AoS and SoA (higher utilisation of the VPU), we can assume that the absorbed power remains approximately constant when considering different data layouts. Indeed, it is evident how the energy consumption remains mostly proportional to the time to solution, figure 4. On the other end it is impressive to see how we measure a much higher energy consumption as far as the DRAM is concerned. Our analysis reports that when using the highly optimized data layouts on the KNL configured in Cache mode, more energy is required due to DRAM if compared to the Flat mode.

7. Conclusions

Our LBM based application implements highly optimised data structures capable to deliver high-performance on modern many-cores processors systems such as the Intel KNL processor. In this work we have demonstrated that the introduced data layouts CSoA and CAoSoA are also extremely efficient in exploiting memory on-chip such as the MCDRAM. Indeed, while canonical structure such as AoS and SoA do not show any benefit when switching from Flat mode to Cache mode, the clustered data structure are much more efficient in Flat mode than in Cache mode. Highly scalable optimised code based on LBM can enable computer simulations at the frontier of science by reducing amount of memory required per core up to the capacity of the on-chip memory and excluding completely the main memory (DRAM). While this effort would be useless if considering common data layouts for stencil codes, our analysis shows that, when the clustered data layouts are used, a benefit between 15% to 25% is achieved by excluding the main memory DRAM. Moreover, results have confirmed the efficiency of highly-optimized data layouts also in term of energy consumption. The unnecessary access to the DRAM, in case the size of the data of the simulation fits the on-chip memory, translates in a significant overhead in energy consumption when considering the clustered data layouts.

8. Acknowledgements

We would like to thank PRACE for the granted project (ID 2018184340) "TurEmu - The physics of (turbulent) emulsions" along with CINECA, INFN and The University of Ferrara for access to their HPC systems.

References

[1] S. Succi, *The Lattice Boltzmann Equation: For Fluid Dynamics and Beyond*. Clarendon Press, 2001, ISBN: 978-0-19-850398-9.

[2] I. Girotto, S. F. Schifano, E. Calore, G. D. Staso, and F. Toschi, "Performance Optimization of D3Q19 Lattice Boltzmann Kernels on Intel KNL," in *INFOCOMP 2018: The Eighth International Conference on Advanced Communications and Computation*, 2018, pp. 31–36.

[3] E. Calore, N. Demo, S. F. Schifano, and R. Tripiccione, "Experience on Vectorizing Lattice Boltzmann Kernels for Multi- and Many-Core Architectures," in *Parallel Processing and Applied Mathematics: 11th International Conference, PPAM 2015, Krakow, Poland, September 6-9, 2015. Revised Selected Papers, Part I*, ser. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016, pp. 53–62. doi: 10.1007/978-3-319-32149-3_6

[4] E. Calore, A. Gabbana, S. F. Schifano, and R. Tripiccione, "Early experience on using knights landing processors for lattice boltzmann applications," in *Parallel Processing and Applied Mathematics: 12th International Conference, PPAM 2017, Lublin, Poland, September 10-13, 2017*, ser. Lecture Notes in Computer Science, vol. 1077, 2018, pp. 1–12. doi: 10.1007/978-3-319-78024-5_45

[5] E. Calore, A. Gabbana, J. Kraus, S. F. Schifano, and R. Tripiccione, "Performance and portability of accelerated lattice Boltzmann applications with OpenACC," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 12, pp. 3485–3502, 2016. doi: 10.1002/cpe.3862

[6] M. Sbragaglia, R. Benzi, M. Bernaschi, and S. Succi, "The emergence of supramolecular forces from lattice kinetic models of non-ideal fluids: applications to the rheology of soft glassy materials," *Soft Matter*, vol. 8, pp. 10 773–10 782, 2012. doi: 10.1039/C2SM26167G

[7] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick, "Optimization of a Lattice Boltzmann computation on stateof-the-art multicore platforms," *Journal of Parallel and Distributed Computing*, vol. 69, pp. 762–777, 2009. doi: 10.1016/j.jpdc.2009.04.002

[8] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. A. Yelick, "Lattice Boltzmann simulation optimization on leading multicore platforms," *2008 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–14, 2008. doi: 10.1109/IPDPS.2008.4536295

[9] M. Bernaschi, M. Fatica, S. Melchionna, S. Succi, and E. Kaxiras, "A flexible high-performance lattice Boltzmann gpu code for the simulations of fluid flows in complex geometries," *Concurrency Computat.: Pract. Exper.*, pp. 22: 1–14, 2009. doi: 10.1002/cpe.1466

[10] E. Calore, A. Gabbana, S. F. Schifano, and R. Tripiccione, "Evaluation of dvfs techniques on modern hpc processors and accelerators for energy-aware applications," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 12, pp. 1–19, 2017. doi: 10.1002/cpe.4143

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200091

Performance, Power Consumption and Thermal Behavioral Evaluation of the DGX-2 Platform

Matej SPETKO [a,1], Lubomir RIHA [a] and Branislav JANSIK [a]
[a] *IT4Innovations National Supercomputing Center, VŠB – Technical University of Ostrava, Ostrava, Czech Republic*

Abstract. In this paper, we evaluate the performance, power consumption and its variation and also thermal behavior of the DGX-2 server from Nvidia. We present a development of specialized synthetic benchmarks to measure raw performance of GPUs for single, double, half precision and also Tensor Core units. With these benchmarks, we were able to reach peak performance and verify the specification provided by Nvidia. We achieved 130.79 TFLOPS peak performance in half-precision on Tensor Cores. We also measured the thermal stability of the DGX-2 system. It can hold its peak performance when all 16 GPUs are fully loaded except Tensor Core workload, when thermal throttling occurred with with up to 1 % performance penalty. During single-precision workload we observed 23 % variation of the power consummation of individual GPUs installed in the system. Finally, we have evaluated the behavior of the Tesla V100-SXM3 chip under the DVFS tuning. Running at optimal frequency, the compute bound workload can save up to 39% energy while the run-time increases by 51 %. More importantly, memory bound workload can save up to 31 % with 2 % throughput penalty and during the communication over NVLink one can save up to 26 % energy with no penalty.

Keywords. DGX-2, tensor core, performance analysis, energy efficiency, dynamic voltage and frequency scaling (DVFS)

1. Introduction

In this paper, we evaluate the performance of the Nvidia DGX-2 system using a new synthetic benchmark, designed to achieve and measure the peak performance of both CPUs as well as Nvidia GPUs. For this paper, we have developed a new version of this benchmark with support for Tensor Cores [1]. With our benchmark, we were able to match V100-SXM3's peak performance stated by Nvidia. In adition, we measured GPU memory and NVLink throughput.

Research in related work is focused on different aspects of DGX-2 system. For instance, *Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking* [2] is more focused on the V100 GPU architecture. This work explores deeply the whole V100 memory hierarchy, including throughput and latency measurements. It also inspects native

[1]Corresponding Author: IT4Innovations National Supercomputing Center, VŠB – Technical University of Ostrava, 17. listopadu 15/2172, 708 33 Ostrava – Poruba, Czech Republic; E-mail: matej.spetko@vsb.cz.

Volta instructions with issue latency measurements. Furthermore, *Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect* [3] focuses on GPU communication technologies. It analyses aspects like throughput, latency and topology of different GPU interconnects that are used on several GPU servers, including the DGX-2.

The goal of this paper is to evaluate the thermal stability and GPU power consumption. Moreover we performed dynamic frequency and voltage scaling (DVFS) for compute bound, memory bound and communication workloads and stated the most efficient configuration for these workload types. In the end we also evaluate power consumption of the whole node.

1.1. DGX-2 platform description

The main focus of the DGX-2 server is to accelerate tasks in artificial intelligence. However, it is well-suited to run any GPU or multi-GPU application. It contains 16 Tesla V100-SXM3 GPUs interconnected with high speed NVLink interconnect [4]. It also features a pair of Intel Xeon Platinum 8168 CPUs, 1.5 TB of memory and 30 TB of fast NVMe SSD storage. The server can be equipped with either eight EDR Infiniband or 100 Gb Ethernet network cards. [5] The GPUs are spread across two trays, each containing 8 GPUs in two rows. Cooling fans are located at the start of the tray as shown in Figure 1.

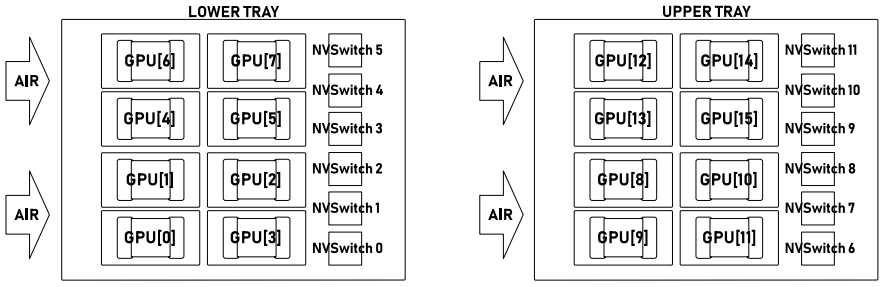


Figure 1. Physical GPU layout of the DGX-2 server.

The V100-SXM3 GPU is equipped with 80 streaming multiprocessors (SMs) and 32 GB of HBM2 memory. Each SM consists of these processing units: 64 FP32 (float), 64 INT32 (32 bit integer), 32 FP64 (double precision) and 8 Tensor Cores (16 bit floating point – half precision). [4]. The basic operation with 16 bit floating point data type – half is performed on floating point units. It can also perform half2 vector operations and reach double the performance of float.

GPUs on DGX-2 system are interconnected with hi-speed bus called NVLink in version 2. Single NVLink-V2 link can provide 25 GBps throughput in single direction and 50 GBps in both directions. The system is also equipped with 12 NVSwitches. Each GPU is connected to six of these switches with single NVLink-V2 link, providing 300 GBps bidirectional peer-to-peer (P2P) throughput. [3]

The Volta architecture introduce Tensor Cores – processing units designed to perform fused multiply-add operation with 16×16 half precision matrices. The result accumulation can be done either in half-precision or in single-precision. The programmer can access `mma_sync()` function which performs a warp level operation: every thread of warp is participating in the matrix multiplication. [1]

2. Measurement Methodology Description

2.1. Benchmarks

The Mandelbrot benchmark is designed to measure pure floating point performance of the processor at very high arithmetic intensity. It executes the Mandelbrot iterations $z_{k+1,i} = z_{ki}^2 + c_i$ where $z_{0i} = 0$ and the constants c_i are from the Mandelbrot set of complex numbers. The Mandelbrot iterations may be repeated indefinitely and remain bounded. For simplicity and efficiency, we select the constants c_i only from the numbers on the real axis. The benchmark is implemented in CUDA PTX assembly code [6]. Each thread on the GPU device is initialized with eight unique constants c_i. We use 32 threads per block and 12 blocks per streaming multiprocessor. After the initialization, all computation runs in the registers only, avoiding any references to memory. The loop over k updates all values of z_{ki} using FMA instructions. Further, the loop is unrolled 100 times, counting 800 consecutive fused mulitply-add instructions, in order to out-weight the loop overhead of three instructions. The loop counter runs one million times to vastly outrun the clock granularity and provide reliable performance measurements. The measurement may be repeated number of times. The arithmetic intensity of the benchmark is 12.5×10^6 FLOP per byte in double precision and up to quadruple of that in single and half precision.

The Mandelbrot benchmark may be naturally extended to matrix domain. In matrix form, the square matrix Z is updated as $Z_{k+1} = Z_k * Z_k + C$, where the $*$ refers to matrix-matrix multiplication, the matrix $Z_0 = \mathbf{0}$ and the square matrix C has eigenvalues from the Mandelbrot set. Such matrix iterations may be repeated indefinitely and the matrix Z will remain bounded. It would be natural to use the matrix Mandelbrot iterations as a load to benchmark the Tensor Cores. However the WMMA interface does not allow to insert the output of the WMMA instruction as an input into the next WMMA instruction directly due to the fact that the matrix fragments held by individual thread registers are not identical for input and output matrices. Reusing the output registers as input registers into the WMMA instruction introduces permutations into the matrix, in addition some matrix elements are repeated and some are lost. Nevertheless, the l2 norm of the matrices created in this way remains approximately correct. Recognizing that the reuse of the output registers as input registers to WMMA instructions approximately conserves the l2 norm and using the property of sub-additivity and sub-multiplicativity of the l2 norm, we are able to select the C as real valued, random matrices, taken such that their eigenvalues lie well within the bounds of the Mandelbrot set and the matrix iterations remain bounded indefinitely. Utilizing this result, we have implemented the matrix Mandelbrot benchmark for the Tensor Cores, using the PTX WMMA instructions API [6]. The data are kept in the registers only, the Z and C being 16 bit floating point 16×16 matrices. Each block is initialized to unique C matrix. The C matrices are pre-computed off the benchmark code, by shifting and scaling a randomly generated square matrices. The block count, loop unrolling and loop count remains the same as for the scalar version. The arithmetic intensity exceeds millions of floating point operations per byte. The arithmetic intensity of the matrix benchmark for the Tensor Cores is 1.6×10^9 FLOP per byte. [7]

The throughput of the memory subsystem was measured by STREAM [8] benchmark, modified for GPUs, also available at the GIT repository [7]. All functions of the STREAM benchmark were measured: copy, scale, add, triadd. The throughput of NVLink interconnect was measured by performing peer-to-peer (P2P) data transfer between two

GPUs with `cudaMemcpyPeerAsync()` call. [9] The throughput was measured in single direction as well as bidirectionally.

2.2. Frequency Scaling and Energy Measurement

To simulate compute bound workload, we took our Mandelbrot benchmark. On the other hand, memory bound workload is represented by the STREAM benchmark. Furthermore, measurement of P2P data transfer over NVLink was also performed. Energy measurement and frequency scaling was performed using tools provided by Nvidia. To measure energy consumption, the Nvidia Management Library – `NVML` was used. For the frequency scaling and taking samples with power, temperature and frequency the `nvidia-smi` utility was used. In this paper, we use the same methodology to measure energy efficiency as described in the Green500 tutorial [10] with the exception that we use our Mandelbrot benchmark to determine peak performance instead of Linpack benchmark.

`NVML` provides C-based programmatic interface for monitoring and managing Nvidia GPUs. It is intended to be a platform for building third party applications. During the experiments, `NVML` was used to access a total energy consumption counter for the GPU. This counter can be accessed with `nvmlDeviceGetTotalEnergyConsumption()` function call [11]. To measure energy consumed by certain workload the value of this counter was read two times: right before launch and right after it finishes. Subtracting these values yield energy consumed by the workload in mJ. Application initialization and cleanup is not included in this measurement, only the main loop with the measured workload.

The Nvidia System Management Interface: `nvidia-smi` was used to collect power, temperature and frequency samples to analyze power and thermal properties. These samples were captured at approximately 100 Hz sampling rate. Furthermore this utility was used to change frequency of GPUs. The frequency was decreased from 1597 MHz to 675 MHz in approximately 7 MHz predefined steps. HBM2 memory frequency cannot be tuned, thus staying at 958 MHz even when the card is idle.

3. Results

3.1. Performance

We have not found any peak performance numbers published for V100-SXM3 GPU used in DGX-2 server. However, we were able to retrieve these numbers from Nvidia Profiler. The performance of Tensor Cores is not stated for this version of V100 GPU. V100-SXM2 revision has Tensor Core peak performance of 125 TFLOPS in the half-precision, running at 1530 MHz [4]. If we scale up this number to match SXM3's 1597 MHz, we should be getting 130.484 TFLOPS in half-precision. Global memory bandwidth is according to Nvidia Profiler 980.992 GBps. NVLink's unidirectional P2P throughput is 150 GBps and 300 GBps in both directions. Results of Mandelbrot benchmark, STREAM benchmark and NVLink P2P transfer benchmark are shown in Table 1.

3.2. Power and Thermal Properties

The physical layout of the DGX-2 server causes that cold air is not distributed equally among all the GPUs. The GPUs are placed in two trays, where each tray contains 8 GPUs

Mandelbrot benchmark		
	Specification [TFLOPS]	Measurement [TFLOPS]
double	8.177	8.1765
float	16.353	16.3530
half2	32.707	32.7038
tensor	130.484	130.7928

STREAM benchmark	
	Throughput [GBps]
copy	825.473
scale	826.518
add	873.631
triadd	872.368

NVLink P2P transfer	
latency	2.45 us
unidirectional	145.16 GBps
bidirectional	266.46 GBps

Table 1. Table on the left compares performance measured by Mandelbrot benchmark to the performance specified by Nvidia. Table in the middle shows memory throughput measured by STREAM benchmark. Table on the right shows the result of P2P data transfer over NVLink interconnect.

in two rows. High-RPM cooling fans are located at the beginning of these trays. GPUs placed in the first row are facing these fans directly and receive cold air, while GPUs in the second row receive air that has been already heated by the GPUs in the first row.

In general, this causes that GPUs in the second row run at higher temperature than the ones in the first row. This also means that they can reach their TDP of 350 W when they are under the full load and thermal throttling must be performed, which results in performance imbalance among the GPUs. Figure 2 shows how GPUs in the first row influence GPUs in the second row by running Mandelbrot benchmark on Tensor Cores for 4 minutes on all 16 GPUs one by one.

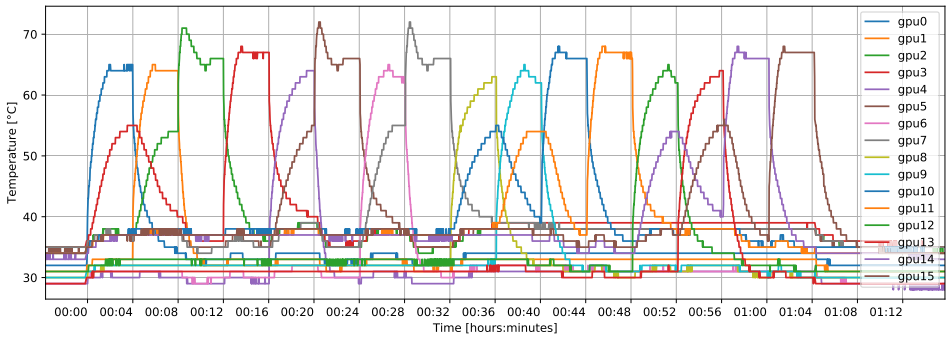


Figure 2. Temperature of all GPUs when fully loaded with Tensor core benchmark one by one. Each GPU was loaded for approximately 4 minutes. GPU in the first row (0, 1, 4, 6, 8, 9, 12, 13) increases also temperature of the GPU located directly behind it in the second row (2, 3, 5, 7, 10, 11, 14, 15).

When running Tensor Core Mandelbrot benchmark on all 16 GPUs at once, GPUs in the front row reach a maximum temperature of 57 °C while GPUs in the second row peak is 72 °C. During this benchmark, certain GPUs from the second row tend to throttle down their frequencies to as low as 1575 MHz (from the maximum 1597 MHz) causing approximately 1 % performance loss, see Figure 3. It shows that running the same Mandelbrot benchmark on all the GPUs results in significant variation in power consumption of individual GPUs, reaching up to 23 % for single precision version. This is caused by both their location in the server as well as their manufacturing variations. We can also observe that for single precision, double precision and Tensor Core version, when some GPUs reach the TDP, they under-clock their frequencies.

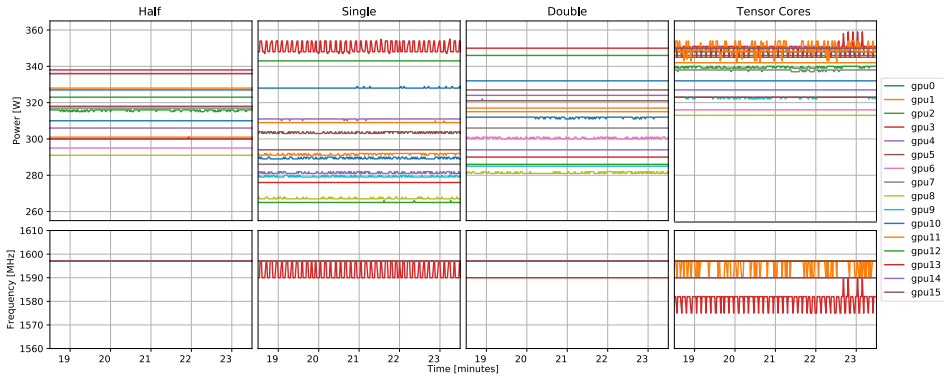


Figure 3. Power consumption variation of all the GPUs in the DGX-2 when under full load using compute bound workload with four different data-types. The variation for the single-precision workload is up to 23%.

3.3. Frequency Scaling

To determine whether we can get better energy efficiency of the Tesla V100-SXM3, we have performed the DVFS tuning test for compute bound, memory bound and NVLink workloads. The first frequency scaling test was performed for all the data types of the Mandelbrot benchmark (float, double, half2, tensor). Benchmark in each data type performed the same amount of the floating-point operations: 81920×10^9. The frequency was scaled from 1597 MHz to 675 MHz in approximately 7 MHz steps. Each frequency step was measured 6 times and average value is reported. Heat up runs were performed before the actual measurement.

Figure 4 shows the result of the frequency scaling. The two bottom plots display the same data in logarithmic scale as the top two plots in linear scale. Table 2 compares runs at base frequency 1597 MHz with the runs at the most energy efficient frequency for each workload type.

In general, the most efficient frequency for Mandelbrot benchmark is 1057 MHz. Running at this frequency we can save up to 39 % of the energy while the run-time will increase by 51 %. Interesting number to point out is the energy efficiency of double data type. Running at base frequency it achieves 24.8 GFLOPS/W whereas running at 1050 MHz the efficiency reaches 40.67 GFLOPS/W. This efficiency number is getting close to 50 GFLOPS/W, which is the limit for building exascale system with 20 MW power consumption [12]. On the other hand, the peak performance at this frequency is only 5.37 TFLOPS which is 66 % of the original 8.17 TFLOPS.

The second frequency scaling test was done using the STREAM benchmark. During this experiment each workload transferred the same amount of data: 7.924 TB. Each frequency step was measured 6 times and average value is reported. Before the measurement started heat up runs were performed.

The results of the frequency scaling of the STREAM benchmark are shown in the Figure 5. The peak throughput achieved during this experiment is lower than in the subsection 3.1, because we were measuring the average throughput and not the best case like the original STREAM does. Furthermore, the Figure 5 also shows a staircase shape when the frequency is lower than 1 GHz. This is probably caused by the GPU having certain memory operation modes. These modes do not match the granularity of which the

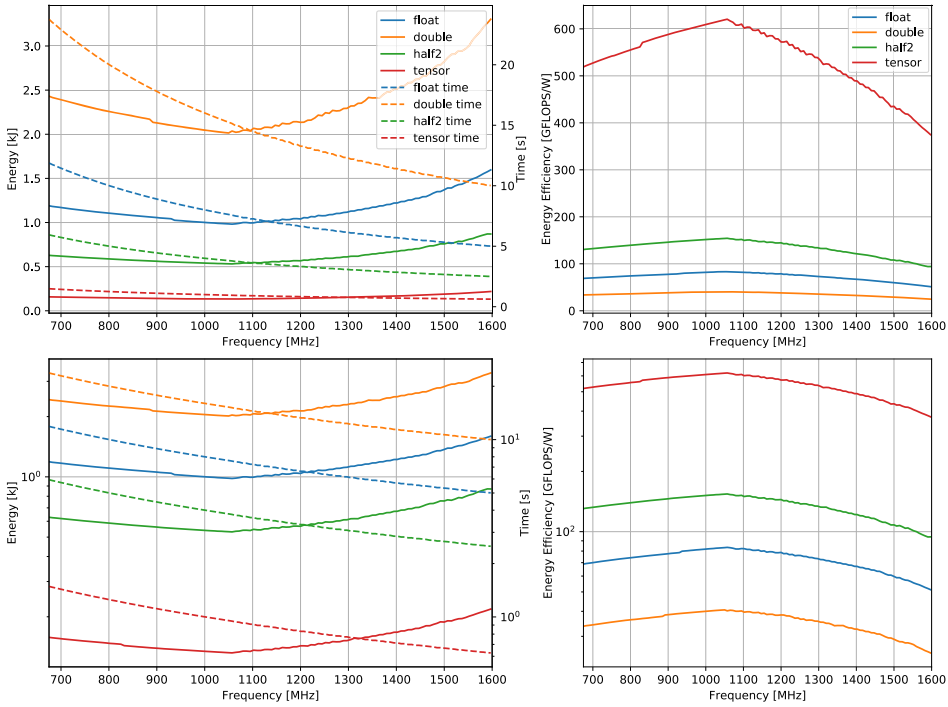


Figure 4. Frequency scaling of Mandelbrot benchmark. Plot in the top left corner shows consumed energy and run-time of workload. Plot in the top right corner shows energy efficiency. The two plots at the bottom display the same data in logarithmic scale.

	Frequency [MHz]	Time [s]	Time difference	Energy [J]	Energy savings	Performance [TFLOPS]	Energy efficiency [GFLOPS/W]
double	1597	10.02		3303		8.17	24.80
	1050	15.25	152.16%	2015	39.01%	5.37	40.67
float	1597	5.01		1596		16.34	51.33
	1057	7.57	150.99%	982	38.50%	10.82	83.46
half2	1597	2.51		870		32.69	94.18
	1057	3.78	151.05%	531	38.97%	21.64	154.30
tensor	1597	0.63		219		130.65	374.90
	1057	0.95	151.04%	132	39.58%	86.50	620.51

Table 2. Mandelbrot benchmark running at base frequency compared to the most efficient frequency for each workload.

streaming multiprocessor can change its frequency. The result of the energy consumption for base frequency and the most efficient frequency is shown in Table 3. On average, up to 31 % of the energy can be saved by scaling down to 1005 MHz. By doing that, the transfer time increased by 2 % which is almost identical in compare to the data transfer at the base frequency.

The last frequency scaling experiment was done for unidirectional P2P data transfer over NVLink. The amount of transferred data was 859 GB. One frequency step was

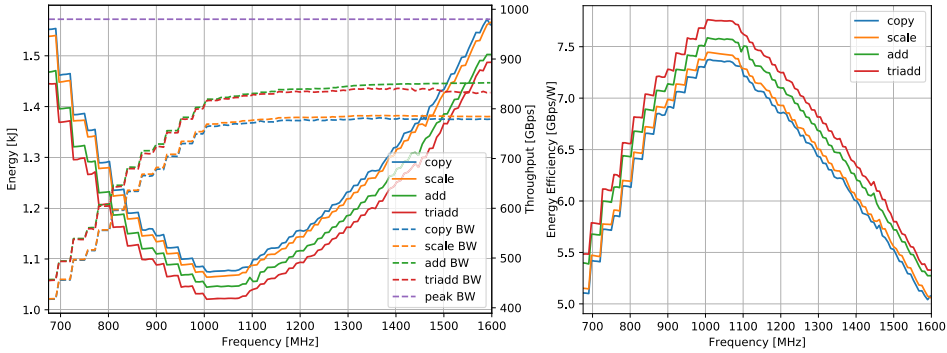


Figure 5. Frequency scaling of the STREAM benchmark. Plot on the left shows consumed energy and throughput. Plot on the right shows the energy efficiency.

	Frequency [MHz]	Time [s]	Time difference	Energy [J]	Energy savings	Throughput [GBps]	Energy efficiency [GBps/W]
copy	1597	10.17		1561		779.12	5.08
	1012	10.35	101.78%	1074	31.18%	765.47	7.38
scale	1597	10.10		1566		784.43	5.06
	1005	10.30	101.99%	1064	32.02%	769.14	7.45
add	1597	9.30		1503		852.27	5.27
	1005	9.67	104.02%	1044	30.49%	819.31	7.59
triadd	1597	9.52		1487		832.54	5.33
	1005	9.71	101.98%	1021	31.36%	816.40	7.76

Table 3. STREAM benchmark running at base frequency compared to the most efficient frequency for each workload.

measured 10 times. Figure 6 shows the result of this experiment. Running at 1140 MHz can save up to 26 % energy without any throughput penalty. The throughput starts to drop when the frequency decreases from 1140 MHz. In addition, the staircase shape similar to Figure 5 can be seen. Table 4 shows the most efficient frequency for source and receive device and compares it to the base frequency.

	Frequency [MHz]	Time [s]	Time difference	Energy [J]	Energy savings	Throughput [GBps]	Energy efficiency [GBps/W]
SRC DEV	1597	5.93		563		144.90	1.51
	1110	6.19	104.34%	421	25.22%	138.88	2.04
RCV DEV	1597	5.93		569		144.90	1.53
	1140	5.94	100.15%	417	26.71%	144.69	2.08

Table 4. NVLink P2P transfer benchmark running at base frequency compared to the most efficient frequency for source device and receive device.

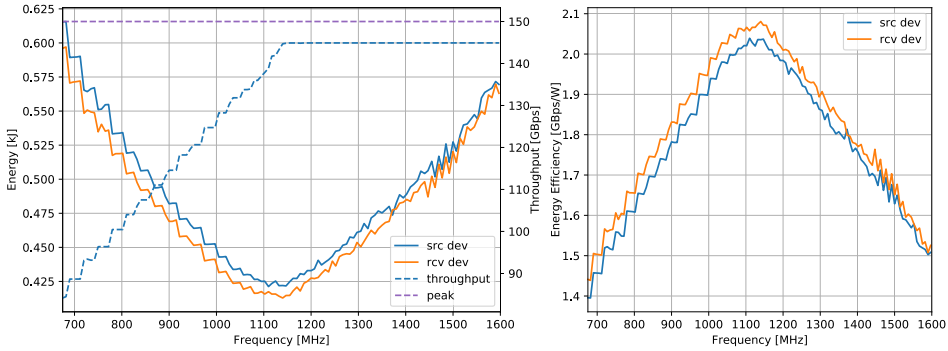


Figure 6. Frequency scaling of the NVLink P2P transfer benchmark. Plot on the left shows consumed energy and throughput. Plot on the right shows the energy efficiency.

3.4. Overall power consumption of DGX-2 server

To supplement the overall picture of DGX-2 efficiency, we also need to look at the energy consumption of the server as a whole. Unfortunately, we measured these numbers only with one power sample because the administrative privileges are needed to retrieve them. Nevertheless, they can give some idea about the efficiency and power consumption of the whole node including all peripherals and cooling. These power consumption numbers were retrieved using `ipmitool` utility.

When idle, the DGX-2 consumes 2340 W, GPUs altogether consumes 768 W. When loaded with Tensor Core Mandelbrot benchmark at 1597 MHz, the consumption rises to 7254 W whereas GPUs alone consume 5340 W. When running the same benchmark at 1057 MHz, the whole node consumption drops to 4056 W and GPUs consume 2248 W.

When running double precision Mandelbrot benchmark at base frequency GPUs consume 4936 W and the whole node 6708 W. At this frequency the server reaches 130.8 TFLOPS, meaning the performance per watt reaches 19.52 GFLOPS/W. When we scale down the frequency to 1057 MHz, GPUs alone consume 2116 W. Consumption of the whole node drops to 3666 W. As a result, the DGX-2 can achieve efficiency of 23.60 GFLOPS/W at this frequency but the performance drops to 86.4 TFLOPS.

4. Conclusion

We have developed a set of benchmarks to determine the raw performance of GPUs in the Nvidia DGX-2 server. We verified that performance numbers of V100-SXM3 GPU are according to specification. We were able to reach 130.79 TFLOPS in half precision using Tensor Cores on a single GPU. When running full load on all 16 GPUs at the same time, some of the GPUs may thermal throttle by 1 % due to an uneven cooling solution and manufacturing variations. We observed 23 % variation in power consumption fo GPUs when running float Mandelbrot benchmark. To get the best performance per watt out of V100-SXM3 GPU, it makes sense to scale down the frequency. For compute bound workload the most efficient frequency is 1057 MHz making 39 % energy savings while run-time is increased by 51 %. The energy efficiency achievable for double precision workload is 40.67 GFLOPS/W whereas running at base frequency is

only 24.8 GFLOPS/W. Memory bound workload has its sweet spot at 1005 MHz. At this frequency, the throughput penalty is only 2 % while energy savings can reach 31 %. Peer-to-peer transfer achieves the best energy efficiency at 1140 MHz frequency, being able to save 26 % energy without any throughput penalty. The whole DGX-2 node in the idle mode consumes 2.3 kW of power. When all 16 GPUs are loaded with double precision workload, the consumption increases to 6.7 kW with 19.52 GFLOPS/W energy efficiency. However, running the same workload at 1057 MHz it consumes 3.6 kW, having the energy efficiency 23.60 GFLOPS/W.

5. Acknowledgments

This work was supported by The Ministry of Education, Youth and Sports from the Large Infrastructures for Research, Experimental Development and Innovations project IT4Innovations National Supercomputing Center LM2015070.

This work was also partially supported by the SGC grant No. SP2019/59 "Infrastructure research and development of HPC libraries and tools", VŠB – Technical University of Ostrava, Czech Republic.

References

[1] M. A. Raihan, N. Goli, and T. M. Aamodt, "Modeling deep learning accelerator enabled gpus," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 79–92, March 2019.

[2] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the nvidia volta gpu architecture via microbenchmarking," *ArXiv*, vol. abs/1804.06826, 2018.

[3] A. Li, S. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker, "Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect," *ArXiv*, vol. abs/1903.04611, 2019.

[4] NVIDIA Corp., "NVIDIA TESLA V100 GPU ARCHITECTURE." http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, August 2017. "[Online; accessed 2019-07-12]".

[5] NVIDIA Corp., "DGX-2/2H SYSTEM User Guide." https://docs.nvidia.com/dgx/pdf/dgx2-user-guide.pdf, May 2019. "[Online; accessed 2019-07-15]".

[6] NVIDIA Corp., "PARALLEL THREAD EXECUTION ISA." https://docs.nvidia.com/cuda/pdf/ptx_isa_6.4.pdf, May 2019. "[Online; accessed 2019-07-12]".

[7] IT4Innovations, "Mandelbrot CPU benchmark." https://code.it4i.cz/jansik/mandelbrot. "[Online; accessed 2019-07-12]".

[8] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[9] NVIDIA Corp., "CUDA RUNTIME API." docs.nvidia.com/pdf/CUDA_Runtime_API.pdf, July 2019. "[Online; accessed 2019-09-24]".

[10] R. Ge, X. Feng, H. Pyla, K. Cameron, and W. Feng, "Power measurement tutorial for the green500 list." https://www.top500.org/files/green500/tutorial.pdf, June 2007. "[Online; accessed 2019-09-24]".

[11] NVIDIA Corp., "NVML Reference Manual." https://docs.nvidia.com/pdf/NVML_API_Reference_Guide.pdf, August 2019. "[Online; accessed 2019-09-16]".

[12] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, K. Yelick, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, P. Kogge, R. S. Williams, and K. Yelick, "Exascale computing study: Technology challenges in achieving exascale systems," 2008.

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200092

On the Performance and Energy Efficiency of Sparse Matrix-Vector Multiplication on FPGAs

Panagiotis MPAKOS [a], Nikela PAPADOPOULOU [a], Chloe ALVERTI [a], Georgios GOUMAS [a], and Nectarios KOZIRIS [a]
[a] *National Technical University of Athens*
{*pmpakos, nikela, xalverti, goumas, nkoziris*}*@cslab.ece.ntua.gr*

Abstract. The Sparse Matrix-Vector Multiplication kernel (SpMV) has been one of the most popular kernels in high-performance computing, as the building block of many iterative solvers. At the same time, it has been one of the most notorious kernels, due to its low flop per byte ratio, which leads to under-utilization of modern processing system resources and a huge gap between the peak system performance and the observed performance of the kernel. However, moving forward to exascale, performance by itself is no longer the holy grail; the requirement for energy efficient high-performance computing systems is driving a trend towards processing units with better performance per watt ratios. Following this trend, FPGAs have emerged as an alternative, low-power accelerator for high-end systems. In this paper, we implement the SpMV kernel on FPGAs, towards an accelerated library for sparse matrix computations, for single-precision floating point values. Our implementation focuses on optimizing access to the data for the SpMV kernel and applies common optimizations to improve the parallelism and the performance of the SpMV kernel on FPGAs. We evaluate the performance and energy efficiency of our implementation, in comparison to modern CPUs and GPUs, for a diverse set of sparse matrices and demonstrate that FPGAs can be an energy-efficient solution for the SpMV kernel.

Keywords. sparse matrix vector multiplication, FPGA, performance, energy efficiency

1. Introduction

Sparse matrices appear in multiple scientific problems, putting sparse linear algebra at the core of high-performance scientific computing. The Sparse Matrix-Vector Multiplication kernel (*SpMV*) has been one of the most popular kernels of this category, as the building block of many iterative solvers. At the same time, it has been one of the most notorious kernels, due to its low flop per byte ratio, which leads to under-utilization of modern processing system resources and a huge gap between the peak system performance and the observed performance of the kernel. A plethora of sparse matrix formats [1] and a variety of optimizations for multi-core processors [2], many-core processors [3] and GPUs [4] have been proposed and applied, to improve the performance of the SpMV kernel.

However, moving forward to exascale, performance by itself is no longer the holy grail; the requirement for energy efficient high-performance computing systems is driving a trend towards processing units with better performance per watt ratios. Following this trend, FPGAs have emerged as an alternative, low-power accelerator for high-end systems. This trend has been further supported by the development of high-level synthesis tools, which significantly reduce the programming effort required to port applications to FPGAs. FPGAs are already used as accelerators in production in datacenters, and several efforts focus on bringing FPGAs to the HPC world. Such an effort is EuroEXA [5], the EU-funded project that aims to implement and prototype a petascale-level system, embracing FPGA acceleration.

In this paper, in the context of the EuroEXA project, we implement the SpMV kernel on FPGAs, towards an accelerated library for sparse matrix computations, for single-precision floating point values. Our implementation focuses on optimizing access to the data for the SpMV kernel and applies common optimizations to improve the parallelism and the performance of the SpMV kernel on FPGAs. We evaluate the performance and energy efficiency of our implementation, in comparison to modern CPUs and GPUs, for a diverse set of sparse matrices, and demonstrate that FPGAs can be an energy-efficient solution for the SpMV kernel.

2. An efficient implementation of SpMV on FPGAs

2.1. Experimental platform

Our experimental platform is a Xilinx Zynq UltraScale+ MPSoC ZCU102 board. The MPSoC of the board consists of a quad-core ARM Cortex A53 processor, operating at (up to) 1.5 GHz and a Zynq UltraScale ZU9EG FPGA. The FPGA contains around 600K logic cells, 32 Mbs of BlockRAM and about 2500 DSP slices. The MPSoC contains 4GB of DDR4 DRAM (referred to as main memory from now on), which is accessible from both the ARM processor and the FPGA. We use the Xilinx SDSoC environment (version 2018.1) which utilizes the Xilinx Vivado-HLS compiler and Vivado Design Suite tools to compile synthesizable C/C++ functions into programmable logic.

```
void SpMV(int nnz, int nrows, float *values, float *col_ind,
                          float *row_ptr, float *x, float *y)
{
  for (int i = 0 ; i < nrows ; i++)
    for (int j = row_ptr[i] ; j < row_ptr[i+1] ; j++)
        y[i] += values[j] * x[col_ind[i]];
  return;
}
```

Algorithm 1. The CSR-SpMV kernel

2.2. CSR-SpMV: Properties and challenges

The Compressed Sparse Row (CSR) representation is the most commonly used sparse matrix representation, since it is generic, agnostic to the sparsity pattern and leads to fair performance on CPUs with no preprocessing cost. In the CSR format, a sparse matrix is represented with three vectors: the *values* vector contains the values of all non-zero elements of the matrix, the *col_ind* vector contains the column index for each non-zero element and the *row_ptr* vector stores the row pointers. The $y = A \times x$ CSR-SpMV kernel, for the sparse matrix A with *nrows* rows and *nnz* non-zero elements is presented in Algorithm 1.

There are two key observations regarding CSR-SpMV; first, the x vector is accessed randomly, and second, there is no reuse for the elements of the A matrix, i.e., the kernel is memory-bound. This is particularly important for our FPGA implementation: accessing data on the main memory of the MPSoC is costly, due to the limited memory bandwidth, and it is preferable to move data to the FPGA BRAM. However, BRAM capacity is limited. Thus, an efficient implementation requires careful data movement and placement.

Another challenge for the FPGA implementation of CSR-SpMV using HLS tools is that the boundaries of the inner loop are unknown at compile time. This impedes efficient loop unrolling by the compiler and limits the parallelism of the implementation.

2.3. pCSR: A packed, CSR-based representation format for sparse matrices

Using our key observations about the CSR-SpMV code, we opt for an implementation where the sparse matrix A is efficiently streamed from the main memory to the FPGA. The x vector is stored locally on the FPGA BRAM, to ensure fast access. The y vector is accessed sequentially, therefore we stream it from the FPGA back to the main memory, and do not store it locally on the BRAM. To efficiently stream the sparse matrix to the FPGA, we need to exploit the four available High Performance (HP) memory ports of our MPSoC. These ports allow for the highest throughput of data transfer from the main memory to the FPGA. Extending the MCSR representation proposed in [6], we first transform the *row_ptr* vector to a *row_length* vector, where each element refers to the number of non-zero values per row. We then pack the *row_length*, *col_ind* and *values* vector into a single stream of data, as following: for a single row of the sparse matrix, we use a zero element to denote a new line, followed by the number of non-zero elements in this row. For every non-zero element in the row, the *col_ind* and the value of the element follow in pairs. In order to fully exploit the available bandwidth of HP ports, the stream is then split into 128-bit wide parts. We also use the `hls::stream` objects, which are FIFO queues, to stream the data to the FPGA through each available port. We note that, in our SpMV implementation, the x vector is copied and stored locally on the FPGA BRAM, to ensure fast access. Therefore, the largest problem size that we can solve on our FPGA depends on the number of columns of the sparse matrix, i.e. the length of the x vector. On the other hand, the length of the y vector does not constrain our design: since the y vector is accessed sequentially, it is streamed from the FPGA back to main memory.

2.4. Optimizations

2.4.1. Increasing parallelism with vectorization

To increase the parallelism of our design, we implement SIMD parallelism by partially unrolling the inner loop of the SpMV code, by a factor of II. We note that, in order to implement vectorization, zero-padding is required, so that the elements of each row are a multiple of the vectorization factor. We test our design with vectorization factors of 2, 4 and 8.

2.4.2. Increasing parallelism with 1D-blocking

To further increase parallelism in our design, as well as the resource utilization of the FPGA, we employ multiple compute units (CU). The multiple compute units implement multiple instances of the SpMV kernel. This is equivalent to row-wise partitioning or 1D-blocking: each compute unit works on a contiguous subset of rows of the sparse matrix. We test our design with 2, 4 and 8 compute units. We note that using more than four compute units leads to full utilization of the available HP ports. However, compute units cannot share data and therefore, each compute unit requires a separate copy of the vector x. Therefore, increasing the number of compute units limits the maximum size of the x vector that can fit in the FPGA BRAM.

2.4.3. Increasing problem size with 2D-blocking

To overcome the limitation of the limited BRAM capacity that arises when using multiple compute units, we employ 2D-blocking, i.e. row-wise and column-wise partitioning, of the sparse matrix. In this way, each compute unit only needs to store part of the x vector, i.e. the part that is used by the columns of the 2D-block. Intermediate results are stored in $y\_partial$ vectors, which are then accumulated on the host side. Blocking on this second dimension allows us to split the matrix to as many blocks needed to fit the multiple copies of the partial x vectors on the FPGA BRAM. However, to implement 2D-blocking, alongside vectorization, the number of elements of each row of each block needs to be a multiple of the vectorization factor, which results to additional zero padding. Compression of the zero-padded elements can be employed to alleviate the memory overhead that occurs in this case.

2.4.4. Load balancing

Depending on the sparsity pattern of the matrix, 1D-blocking, i.e. row-wise partitioning, commonly produces load imbalance among the multiple compute units. We easily mitigate this problem by equally distributing non-zero elements across compute units. In this case, each compute unit solves the SpMV kernel on variable numbers of contiguous rows, but with better load distribution.

2.5. Increasing performance with clock frequency configuration

The FPGA of our experimental platform can be configured to operate under clock frequencies ranging from 100 to 600 MHz. Our implementation meets the timing requirements for frequencies up to 300 MHz. Figure 1 shows how execution time and power

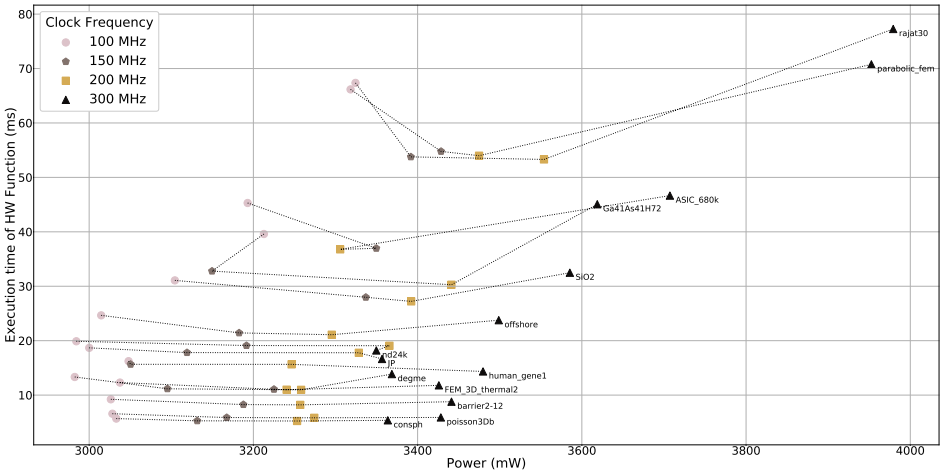


Figure 1. Comparison of execution time and power consumption of FPGA-SpMV for different clock frequencies. Each line represents a different sparse matrix.

consumption vary while we increase the clock frequency. Connected points represent measurements for the same matrix under different clock frequencies. As expected, higher clock frequency leads to higher power consumption. For all the matrices, execution time decreases as we increase the clock frequency up to 200MHz. The decrease is more significant for the larger matrices. However, when the frequency is set at 300MHz, the synthesized bitstream consumes more of the BRAM resources, limiting the available BRAM to store the x vector. This causes large size matrices to be split in more blocks, hence we observe an increase in execution time. Due to the algorithmic nature of SpMV, increasing the clock frequency of the FPGA does not proportionally improve the performance of our implementation. Therefore, we conclude that the optimal frequency, considering both performance and power consumption, is 150 MHz.

3. Evaluation

To evaluate our FPGA implementation, we use a diverse set of 19 sparse matrices from the University of Florida Sparse Matrix Collection [7], with a variety of sparsity patterns and sizes. The number of floating point operations per non-zero element is 2 (multiplication and addition). For the pCSR representation,we use 64 bits to store each row of the matrix, and 64 bits for each non-zero element of the matrix (value and col_ind index). Therefore, the flops:byte ratio is calculated by the formula $nnz/(4*nrows+4*nnz)$. We compare the performance and energy efficiency of SpMV on our FPGA against CSR-SpMV using the Intel MKL library on an Intel Xeon E5-2630V4 (Broadwell) CPU with 10 cores, 25MB LLC and 256GB of memory, and against CSR-SpMV using the cuSPARSE library on an NVIDIA Tesla K40 GPU, with 2880 cores and 12GB GDDR5 memory. We use RAPL performance counters to measure energy on the CPU. For the FPGA, we modified the power monitoring application proposed in [8], in order to execute it on our board. For the GPU, we use the GPU power sensors and compute energy according to the runtime.

Table 1. Matrix suite used for experimental evaluation

Matrix	Dimension	Non-zeros	Size(MB)	f:b ratio
human_gene1	22283	12345963	94.2772	0.2495
nd24k	72000	14393817	110.091	0.2488
JP	87616	13734559	105.121	0.2484
consph	83334	3046907	23.564	0.2433
poisson3Db	85623	2374949	18.4461	0.2413
barrier2-12	115625	3897557	30.1771	0.2428
FEM_3D_thermal2	147900	3489300	27.1854	0.2398
SiO2	155331	5719417	44.2282	0.2434
degme	185501	8127528	62.7158	0.2444
offshore	259789	2251231	18.1666	0.2241
Ga41As41H72	268096	9378286	72.5734	0.2431
parabolic_fem	525825	2100225	18.0293	0.1999
rajat30	643994	6175377	49.5711	0.2264
ASIC_680k	682862	3871773	32.1442	0.2125
Hardesty2	929901	4020731	34.2231	0.203
boneS10	914898	28191660	218.575	0.2421
audikw_1	943695	39297771	303.418	0.2441
webbase-1M	1000005	3105536	27.5081	0.1891
thermal2	1228045	4904179	42.1006	0.1999

Table 2. Hardware platforms

Device	Operating Frequency	Memory	Memory Bandwidth
Intel Xeon E5-2630V4	2.2 GHz	256 GB	40 GB/s
NVIDIA Tesla K40	745 MHz	12 GB	6 GB/s
Xilinx MPSoC ZCU102	150 MHz	4 GB	9.6 GB/s

Figure 2 shows the execution time for SpMV on the 19 matrices, for the CPU, the GPU and the FPGA. For our FPGA implementation, we showcase the results for a vectorization factor of 4, and 4 compute units. In addition, the frequency of the FPGA is set to 150MHz. We note that the execution time for the FPGA implementation includes transfers from the main memory to the FPGA and vice versa. For a fair comparison against the GPU, we compare against the performance with and without transfers over the PCIe (6GB/s). In comparison to the CPU, our FPGA implementation is slower from 7 to 62 times, with an average slowdown of 26x. We consider this performance gap to be reasonable, given the 15x difference in frequency and the 2.5x difference in cores (compute units), between the CPU and the FPGA. GPU performance is close to that of the CPU, apart from three matrices in our suite (*ASIC_680K*, *rajat30*, *degme*), which suffer from imbalance [3]. However, if we include the transfers from the host to the GPU, the average slowdown for SpMV on the FPGA is 3x.

Figure 3 shows the energy consumption for SpMV, for the CPU, the GPU and the FPGA. Despite the large difference in execution times, the CPU consumes up to 5 times more energy than the FPGA for the SpMV kernel, with the exception of the largest matrices in our dataset (*webbase_1M*, *thermal2*). The GPU consumes about the same energy with the FPGA for the computational part of the SpMV kernel, with the exception of the three imbalanced matrices. However, if we take into account the data transfers

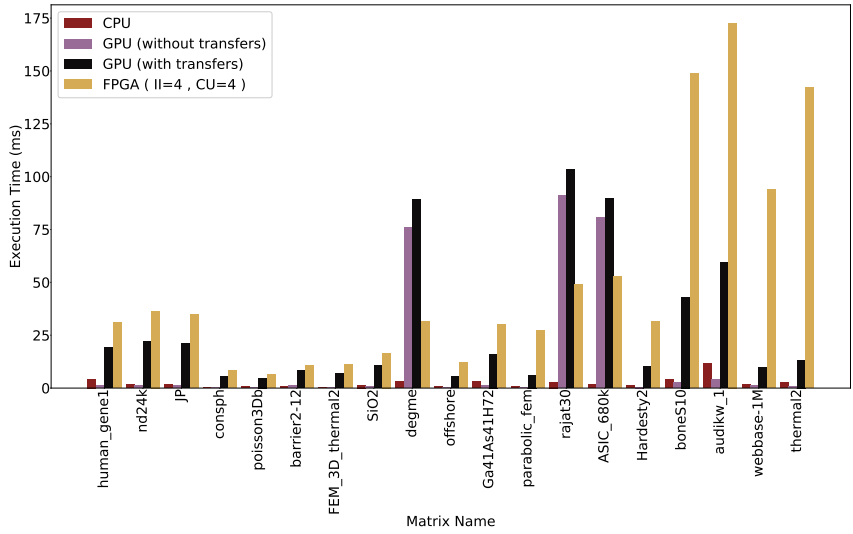


Figure 2. Comparison of performance of the CSR-SpMV kernel among different architectures.

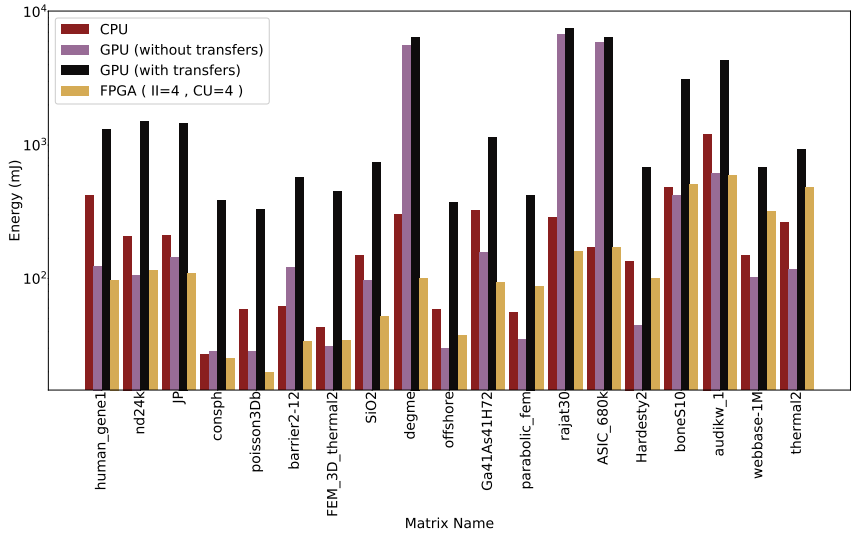


Figure 3. Comparison of the energy consumption (in Joules) of the CSR-SpMV kernel among different architectures. The y axis is in logarithmic scale.

from the host to the GPU, the GPU becomes the least energy-efficient option among the three architectures.

Another metric that can be used to measure the energy efficiency of each architecture is the performance per Watt, i.e. FLOPs per Watt [9]. In SpMV, two floating-point operations occur for each non-zero element; multiplication with the respective element of the x vector and accumulation of the result in the y vector. Thus, SpMV FLOPs are calculated by dividing the doubled number of non-zeros of the matrix with the execution time. Figure 4 shows how each architecture performs in GFLOPs/W. For smaller

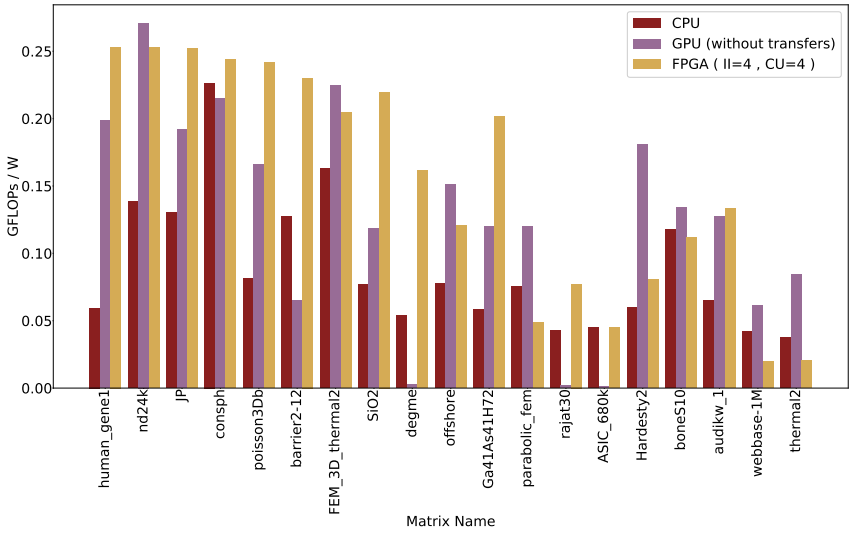


Figure 4. Comparison of the energy efficiency (in GFLOPs/W) of the CSR-SpMV kernel among the different architectures.

matrices (leftmost part of the figure), the FPGA significantly outperforms the other architectures in terms of energy efficiency. For larger matrices, although the performance of our FPGA implementation is degraded due to extensive zero-padding, the FPGA still performs well in the GFLOPs/W metric, being a viable, energy-efficient option for the SpMV kernel.

4. Related work

Multiple works present implementations of the SpMV kernel on FPGAs, using hardware design tools. Zhuo et al. [10] provide an efficient design for CSR SpMV on FPGAs, using a number of subtrees of multipliers and a specialized reduction unit. Their implementation also stores the x vector on the FPGA. Sun et al. [11] describe a design using multiple processing elements which include a deep pipeline with a multiplier, an accumulation circuit and FIFO queues. Their implementation uses both CSR and Row Blocked CSR. Kestur et al. [12] design a library for SpMV and propose the CVBV format, to reduce memory capacity requirements and memory bandwidth requirements. Dorrance et al. [13] implement the SpMV kernel using CSC, to make memory accesses to the x vector, stored in the main memory, sequential, reducing memory bandwidth requirements. Grigoras et al. [14] propose a dictionary-based compression format to improve the effective memory bandwidth of SpMV designs. Their implementation uses Maxeler tools. Several implementations of SpMV with OpenCL appear in recent work [15,9,16]. Our work extends the implementation of CSR SpMV proposed by Hosseinabady et al. in [6], which uses HLS tools to synthesize SpMV as a streaming dataflow engine.

In addition, a number of works examine the performance and energy efficiency of various algorithms on CPUs, GPUs and FPGAs. Vestias et al. [17] explore general trends in peak performance and power for CPUs, GPUs and FPGAs. Betkaoui et al. [18] compare the performance and energy efficiency of GPUs and FPGAs for four commonly used

benchmarks. Fowers et al. [19] focus their analysis of performance and energy efficiency on sliding-window applications. Rucci et al. [20] focus on state-of-the-art implementations of the Smith-Waterman protein database search, on CPUs, co-processors, GPUs and FPGAs. Finally, Giefers et al. [9] perform a performance and energy-efficiency analysis for sparse matrix-vector and sparse matrix-matrix multiplication on co-processors, GPUs and FPGAs.

5. Conclusions

In this work, we examine the performance and energy efficiency of the sparse matrix-vector multiplication on FPGAs. We design and optimize the CSR-SpMV kernel for a Xilinx Zynq UltraScale+ board with a ZU9EG FPGA, using HLS tools. We evaluate the performance and energy efficiency of this kernel with the equivalent CSR-SpMV implementations on an Intel Broadwell CPU and an NVIDIA Tesla K40 GPU. Our experimental results show that the CPU and GPU outperform the FPGA in terms of performance for the SpMV kernel, however, the energy consumption of the FPGA is lower for most of the matrices in our dataset. In addition, comparing the achieved FLOPs/W for the three platforms, the FPGA is a particularly energy-efficient option for the SpMV kernel, and a further optimized design can bring in additional gains for energy consumption and energy efficiency. Future directions of this work focus on solving the SpMV on FPGAs and include further optimizations of the CSR-SpMV kernel for the FPGA, the exploration of alternative storage formats for the sparse matrices and their impact on performance and energy efficiency, as well as the evaluation of our CSR-SpMV kernel on FPGAs with higher BRAM capacity, higher bandwidth and more DSP units.

Acknowledgment

This research has received funding from the European Unions Horizon 2020 research and innovation programme under Grant Agreement no 754337 (EuroEXA project).

References

[1] D. Langr and P. Tvrdik, "Evaluation criteria for sparse matrix storage formats," *IEEE Transactions on parallel and distributed systems*, vol. 27, no. 2, pp. 428–440, 2015.

[2] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pp. 1–12, IEEE, 2007.

[3] A. Elafrou, G. Goumas, and N. Koziris, "Performance analysis and optimization of sparse matrix-vector multiplication on intel xeon phi," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 1389–1398, IEEE, 2017.

[4] M. M. Baskaran and R. Bordawekar, "Optimizing sparse matrix-vector multiplication on gpus," *IBM Research Report RC24704*, no. W0812-047, 2009.

[5] G. Goumas, N. Koziris, N. Papadopoulou, K. Nikas, V. Karakostas, C. Alverti, J. Goodacre, M. Lujan, O. Palomar, M. Ashworth, *et al.*, "Co-designed innovation and system for resilient exascale computing in europe: From applications to silicon (euroexa)," 2017.

[6] M. Hosseinabady and J. L. Nunez-Yanez, "A streaming dataflow engine for sparse matrix-vector multiplication using high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[7] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

[8] "Zynq-7000 ap soc low power techniques part 4 - measuring zc702 power with a linux application tech tip." https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18841995. Accessed: 2019-09-30.

[9] H. Giefers, P. Staar, C. Bekas, and C. Hagleitner, "Analyzing the energy-efficiency of sparse matrix multiplication on heterogeneous systems: A comparative study of gpu, xeon phi and fpga," pp. 46–56, IEEE, 2016.

[10] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on fpgas," in *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pp. 63–74, ACM, 2005.

[11] J. Sun, G. Peterson, and O. Storaasli, "Sparse matrix-vector multiplication design on fpgas," in *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*, pp. 349–352, IEEE, 2007.

[12] S. Kestur, J. D. Davis, and E. S. Chung, "Towards a universal fpga matrix-vector multiplication architecture," in *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pp. 9–16, IEEE, 2012.

[13] R. Dorrance, F. Ren, and D. Marković, "A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on fpgas," in *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pp. 161–170, ACM, 2014.

[14] P. Grigoras, P. Burovskiy, E. Hung, and W. Luk, "Accelerating spmv on fpgas by compressing nonzero values," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 64–67, IEEE, 2015.

[15] Q. Gautier, A. Althoff, P. Meng, and R. Kastner, "Spector: An opencl fpga benchmark suite," in *2016 International Conference on Field-Programmable Technology (FPT)*, pp. 141–148, IEEE, 2016.

[16] Y. Shan, T. Wu, Y. Wang, B. Wang, Z. Wang, N. Xu, and H. Yang, "Fpga and gpu implementation of large scale spmv," in *2010 IEEE 8th Symposium on Application Specific Processors (SASP)*, pp. 64–70, IEEE, 2010.

[17] M. Vestias and H. Neto, "Trends of cpu, gpu and fpga for high-performance computing," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–6, IEEE, 2014.

[18] B. Betkaoui, D. B. Thomas, and W. Luk, "Comparing performance and energy efficiency of fpgas and gpus for high productivity computing," in *2010 International Conference on Field-Programmable Technology*, pp. 94–101, IEEE, 2010.

[19] J. Fowers, G. Brown, P. Cooke, and G. Stitt, "A performance and energy comparison of fpgas, gpus, and multicores for sliding-window applications," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pp. 47–56, ACM, 2012.

[20] E. Rucci, C. García, G. Botella, A. De Giusti, M. Naiouf, and M. Prieto-Matías, "State-of-the-art in smith–waterman protein database search on hpc platforms," in *Big Data Analytics in Genomics*, pp. 197–223, Springer, 2016.

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200093

Evaluation of DVFS and Uncore Frequency Tuning Under Power Capping on Intel Broadwell Architecture

Lubomir RIHA [a,1], Ondrej VYSOCKY [a] and Andrea BARTOLINI [b]
[a] *IT4Innovations national supercomputing center, VŠB – Technical University of Ostrava, Ostrava, Czech Republic*
[b] *University of Bologna, DEI, via Risorgimento 2, 40136 Bologna, Italy*

Abstract. In this paper we present an evaluation of the Intel Xeon Broadwell platform in the CINECA Galileo supercomputer when DVFS and UnCore Frequency (UCF) tuning is performed under the active power capping using RAPL powercap registers. This work is an extension of our previous work done under the H2020 READEX project which focused on a dynamic tuning of DVFS and UCF for complex HPC applications, but with no powercap limit enforced. Power capping is an essential technique that allows system administrators to maintain the power budget of an entire system or data center using either out-of-band management system or runtime systems such as GEOPM.

In this paper we use two boundary workloads, Compute Bound Workload (CBW) and Memory Bound Workload (MBW) to show the behavior of the platform under power capping and potential for both energy and runtime savings when compared to the default CPU behavior. We show that DVFS and UCF tuning behave differently under the limited power budget. Our results show that if CPU has a limited power budget the proper tuning can provide both improved energy consumption as well as reduced runtime and that it is important to tune both DVFS and UCF.

For MBW we can save up 22 % for both runtime and energy when compared to default behavior under powercap. For CBW we can improve both performance, up to 9.4 %, and energy consumption, up to 14.9 %.

1. Introduction

Energy and power consumption become limiting parameters of new peta- and exa-scale HPC clusters. Due to that accelerators are more common hardware used to provide the performance of the system [1]. Nevertheless it is not only hardware but also software and runtime systems that must be improved to reduce energy and power clusters' hungriness to stay below the 20 MW limit that is being considered as a peak power for an HPC system [2,3].

Energy savings given by software tuning come from better utilization of the hardware resources. It is up to the developers to improve performance of their application,

[1]Corresponding Author: IT4Innovations National Supercomputing Center, VSB-Technical University of Ostrava, 17. listopadu 15/2172, 708 00 Ostrava - Poruba, Czech Republic; E-mail: lubomir.riha@vsb.cz

or apply one of many approaches that limit the resources to the level, that the application does not waste the resources. Typically CPU core frequency is being reduced (also known as Dynamic Voltage and Frequency Scaling, DVFS) for this purpose. In several researches the DVFS is usually set to one specific frequency. Fraternali et al. [4] study the impact of DVFS and HW/SW variability in heterogeneous workloads. Bonati et al. [5] focuses on evaluation this trade-off in a multi-node multi-accelerator context. Calore et al. [6] also evaluate the effect of DVFS on modern HPC processors and accelerators. This approach is efficient in case of single-purpose kernels, however it does not work well when a complex application is tuned.

This work is an extension of our previous effort done under the H2020 READEX project [7,8] which was focused not only on a tuning of CPU core frequency but also its uncore frequency. CPU uncore frequency (UCF) refers to frequency of subsystems in the physical processor package that are shared by multiple processor cores e.g., L3 cache or on-chip ring interconnect. READEX has developed an open-source runtime system called READEX Runtime Library (RRL) that performs dynamic tuning of hardware parameters during a complex parallel applications run, based on Score-P [9] regions instrumentation. RRL uses a configuration file created during the analysis of an application and applies the optimal settings for different parts of the code. RRL supports both DVFS and UCF tuning and also a concurrency throttling, however with no power cap limit enforced.

Power capping is an essential technique that allows system administrators to maintain the power budget of an entire system or data center using either out-of-band management system or runtime systems such as GEOPM [10]. This runtime system in addition to capping CPU's package power consumption also may tune the CPU's core frequencies, but it does not control uncore frequency of the chip. This paper shows that adding a support for UCF tuning will have significant impact on both performance and energy consumption. In [11] Zhang et.al. presents an approach for maximizing the performance under powercap by tuning the DVFS, number of cores, hyper-threads and potentially number of sockets, however also in this research the UCF tuning is not presented.

The proposed method is implemented into our open-source library MERIC [12], that has been also developed under the READEX project.

2. Methodology

2.1. Experiments description

We have conducted a set of experiments that is defined in Table 1. This set covers all the possibilities: (1) pure CPU firmware automatic tuning of all parameters - EXP0; (2) combination of user and firmware tuning - EXP1 to EXP6 and (3) pure user tuning of all three parameters - EXP7. The goal is to find out in which cases user tuning can help and when it can harm the performance or energy consumption.

For each of the experiments we have run a compute bound workload (CBW) and memory bound workload (MBW) to evaluate the behavior of the Intel RAPL power capping system [13] in two situations. When high core frequency is more important the uncore frequency can be reduced without major performance penalty, which is the case of the CBW. The exactly opposite situation is for the MBW. As a compute bound

Experiment number	Powercapping	DVFS (core freq. tuning)	Uncore freq. tuning	Description
0	-	-	-	default CPU behavior (powersave scaling governor)
1	x	-	-	default CPU behavior under powercap
2	-	x	-	default CPU behavior under DVFS tuning
3	-	-	x	default CPU behavior under uncore freq. tuning
4	-	x	x	READEX tuning approach - DVFS & uncore freq.
5	x	x	-	DVFS tuning under powercap; uncore freq. unset
6	x	-	x	uncore freq. under powercap; DVFS unset
7	x	x	x	DVFS and uncore freq. tuning under powercap

Table 1. A set of experiments performed on the platform to determine its behavior.

region we have selected a loop of tangents (TAN) operation and memory bound region is represented by a loop with a matrix vector multiplication (DGEMV).

2.2. Hardware Platform Description and Measurement Setup

The evaluation was done on the Broadwell partition of the Galileo supercomputer installed in CINECA [14]. The servers in this partition are dual socket machines equipped with two 18-core Intel Xeon E5-2697v4 processor [15] running at 2.3 GHz nominal frequency. The turbo frequency when all 18 cores are utilized is 2.7 GHz. This was verified by our measurements. The TDP of the processor is 145 W. Further details are shown in Table 2 including the ranges of all tunable parameters and their granularity.

	nominal value	minimal value	maximal value	minimal step
CPU core frequency (DVFS)	2.3 GHz	1.2 GHz	2.8 GHz turbo	100 MHz
CPU uncore frequency	-	1.2 GHz	2.8 GHz	100 MHz
Power capping	145 W[2]	33 W	145 W	0.125 W

Table 2. Key tunable parameters of the 18-core Intel Xeon E5-2697v4 processor and their respective ranges and steps.

All test were performed in a way that the workload was executed on socket 1, while socket 0 was not utilized. This way we were able significantly reduce the effect of the system noise on the measurements. Also all measurements were repeated ten times and outliers were eliminated using interquartile range rule[3].

3. Results

3.1. DVFS and UCF Tuning without Powercap

It this section we will evaluate the behavior of the platform without enforcing the power cap.

[2]TDP value of the E5-2697v4 processor.

[3]see: https://www.mathwords.com/o/outlier.htm

The key behavior of the platform when running CBW is: the DVFS has key impact on performance/runtime; the uncore frequency has no effect on performance/runtime, it can only affect the power and therefore energy consumption.

On the other hand the key behavior of the MBW is: the uncore frequency has major impact on performance/runtime; the CPU core frequency has no effect on performance/runtime. It can only affect the power and therefore energy consumption.

The Figure 1 presents runtime and energy consumption for both CBW and MBW in EXP2 and EXP4 configuration. The key observations for the compute bound workload are:

- The energy consumption (full red line) significantly increases from 305 J to 358 J (by 17.4 %) when core frequency increases from 2.2 GHz to 2.3 GHz (the nominal frequency) while runtime decreases by only 4.2 %.
- At this point the CPU switches to the highest available uncore frequency, which is confirmed by the test that runs at maximum uncore frequency (red dashed line).
- One can further reduce the energy consumption by reducing the UCF to minimum value, see the red doted line. In this case the energy consumption is reduced from 358 J to 284 J (by 26 %) for nominal frequency (2.3 GHz).
- The same behavior remains when CPU runs at turbo frequency (2.7 GHz), in this case the energy consumption is reduced from 354 J to 299 J (by 18.4 %) when UCF is set minimum. For CBW this is the optimal point from both energy and performance point of view.

The key observations from Figure 1 for memory bound workload are:

- By default CPU runs at high uncore frequency in the entire range of CPU core frequencies.
- Reducing the uncore frequency to low values increases both runtime and energy consumption.

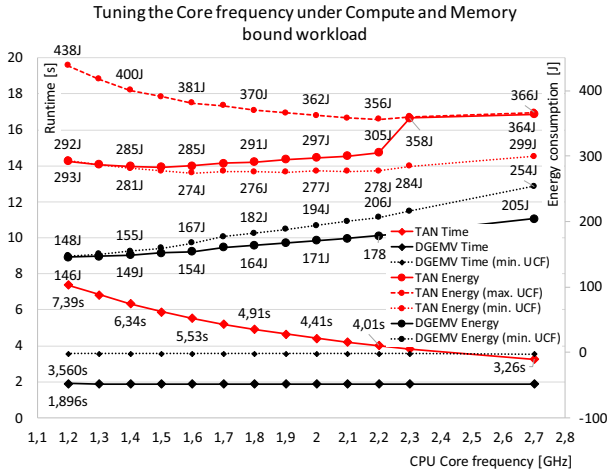


Figure 1. The behavior of the platform for the DVFS tuning for compute bound and memory bound workloads. The solid lines show default behavior without UCF tuning, the dashed lines show the behavior for maximum UCF and the doted lines show the behavior for minimum UCF.

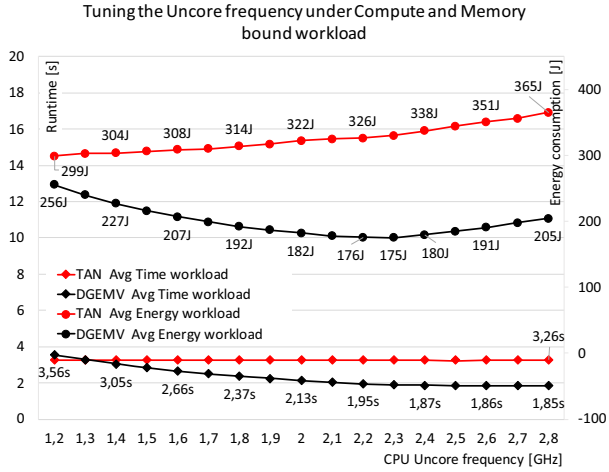


Figure 2. The behavior of the platform under the uncore frequency tuning for compute bound and memory bound workload.

Figure 2 shows the behavior of the platform for UCF tuning, the EXP3. We can see that, as expected, for CBW the uncore frequency has no effect on performance (runtime remains the same in the entire range, while energy consumption grows with higher UCF.

On the other hand, for the MBW the optimal performance requires high UCF. From energy point of view the optimal frequency is 2.3 GHz. If one increase the UCF to 2.8 GHz the gain is only 2.1 % higher performance at a cost of additional 14.6 % of energy.

3.2. DVFS and UCF Tuning under Powercap for Memory Bound Workload

Figure 3 shows the behavior of the platform when running memory bound workload under three different power cap levels: 100 W, 80 W and 60 W.

The default behavior of the CPU without powercap is represented by the EXP0 results: 1.88 sec runtime; 197 J energy consumption. In terms of runtime, this represents the maximum achievable performance.

For all three powercap levels EXP1 results presents the default behavior of the CPU under the powercap. These values are the baselines for all further experiments and are as follows: for 100 W it is 1.88 sec and 188.2 J; for 80 W it is 1.92 sec and 153.2 J; and for 60 W it is 2.47 sec and 147.8 J.

In the previous section where no power limit was set we have observed that for memory bound workload, tuning the DVFS does not affect the performance, but has a significant impact on energy consumption. The results of EXP5 for 100 W powercap level still hold this behavior. The runtime remains 1.88 sec while energy consumption is reduced from 188.2 J to 148.6 J when CPU core frequency is reduced from turbo frequency (2.7 GHz) to its minimal value 1.2 GHz. However, the expected behavior is no longer true for the 80 W powercap level. In this case the energy consumption remains similar and it is only slightly reduced from 153.2 J to 146.0 J. The runtime is also only slightly reduced from 1.92 sec to 1.89 sec. The most visible effect on both performance and energy consumption has the DVFS tuning under the 60 W powercap. In this case

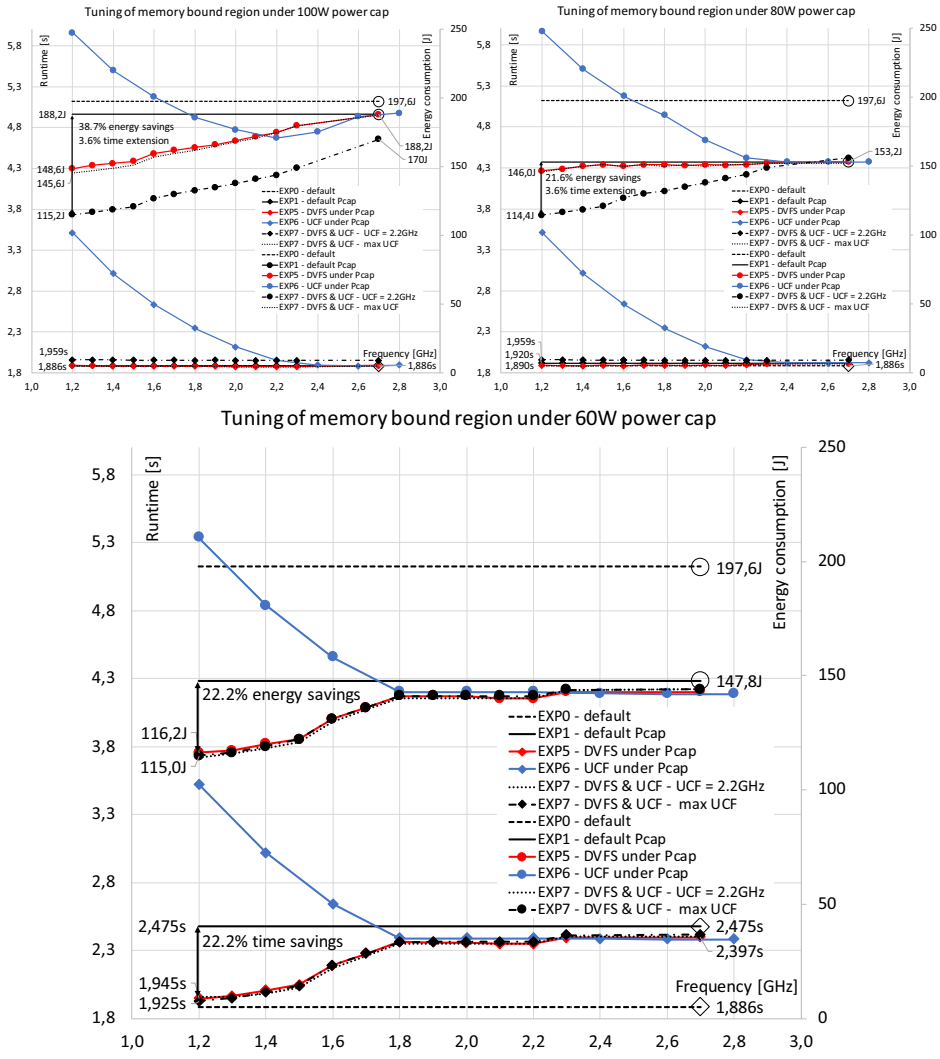


Figure 3. The behavior of the platform running memory bound workload (GEMV) for (EXP1) default CPU behavior under powercap, (EXP5) DVFS tuning under powercap, (EXP6) UCF tuning under powercap and (EXP7) DVFS and UCF tuning under powercap. All tests are done for 60, 80 and 100 W powercap levels.

both runtime and energy are reduced by 22.2 % when core frequency is set to its minimal value (1.2 GHz). We explain this behavior as follows: by limiting the performance and as a consequence the power consumption of CPU cores, the uncore part of the chip responsible for communication with memory gets higher power budget and it can run on higher frequency and achieve higher performance. Therefore CPU executes the MBW more efficiently. Under such very limited power budget (60 W) this makes the significant difference against the default CPU behavior.

Results of EXP6 shows that tuning the UCF has a significant impact on performance and it should be kept as high as possible. In terms of energy consumption the optimal setting is 2.2 GHz (it is the most visible in EXP6 results for 100 W powercap). However

the key observation is that tuning *ONLY* the UCF for MBW has small impact as by default CPU keeps it high enough.

Finally, the results of EXP7 show that adding the UCF tuning to DVFS tuning has a significant impact on energy consumption for higher power cap (100 W and 80 W). For 100 W powercap CPU saves up to 38.7 % of energy while increases the runtime by 3.6 % only. For 80 W powercap CPU saves 21.6 % of energy with the same time penalty (3.6 %). For 60 W powercap both energy consumption and runtime are almost identical to the DVFS tuning only (EXP5).

3.3. DVFS and UCF Tuning under Powercap for Compute Bound Workload

Figure 4 shows the behavior of the platform when running CBW under three different power cap levels: 100 W, 80 W and 60 W. The default behavior of the CPU without powercap is represented by the EXP0 results: 3.27 sec runtime; 363.4 J energy consumption. In terms of runtime, this represents the maximum achievable performance.

For all three powercap levels EXP1 results presents the default behavior of the CPU under the powercap. These values are the baselines for all further experiments and are as follows: for 100 W it is 3.45 sec and 344.4 J; for 80 W it is 3.90 sec and 311.8 J; and for 60 W it is 4.94 sec and 296, 0 J.

When compared to MBW results we can see that CPU requires more power to execute CBW. By reducing the powercap from 140 W (TDP level) to 100 W the performance is reduced by 5.2 %. The 80 W powercap reduces performance by 16.2 % and the 60 W power reduce performance by 33.8 %.

For a compute bound workload DVFS tuning is a key knob to control the performance for all powercap levels. Any energy savings gained by the DVFS tuning are paid by significant performance penalty. However if energy savings are needed this knob has the highest impact for higher power budgets. If power budget gets lower the UCF tuning gains on importance.

The key findings comes from EXP6 for tuning the UCF frequency. For 100 W powercap level by reducing the UCF to 2.2 GHz or bellow we improve the performance by 4.5 % over the default level, from 3.45 s to 3.29 s. If one further reduces the the UCF to its minimum value, 1.2 GHz the performance remains the same but energy consumption is improved by 14.9 % against the default powercap behavior (EXP1). For 80 W powercap level since the CPU is already struggling with the limited power budget the performance increase is visible in the entire range of UCF going from max. to min. value. The same holds for energy consumption. Both the best performance and the lowest energy consumption is achieved at 1.2 GHz (minimal) UCF frequency. In this case the performance is improved by 8.4 % and energy consumption by 8.5 % against the default behavior under powercap (EXP1). Also, the performance is only 8.6 % lower against non powercapped CPU (EXP0), without UCF tuning this penalty was 16.2 %. For 60 W powercap the CPU behavior is similar to 80 W powercap. The best performance is achieved at minimal uncore frequency, and for this case the performance is increased by 9.4 % and energy consumption is reduced by 9.1 %. Against the default CPU behavior without powercap (EXP0) the performance penalty is reduced from 33.8 % to 27.0 %.

Finally, the results of EXP7 show again that adding the UCF tuning to DVFS tuning has a significant impact on energy consumption. It is the most visible for the 100 W and 80 W powercap experiments. But under all powercap levels the minimum energy con-

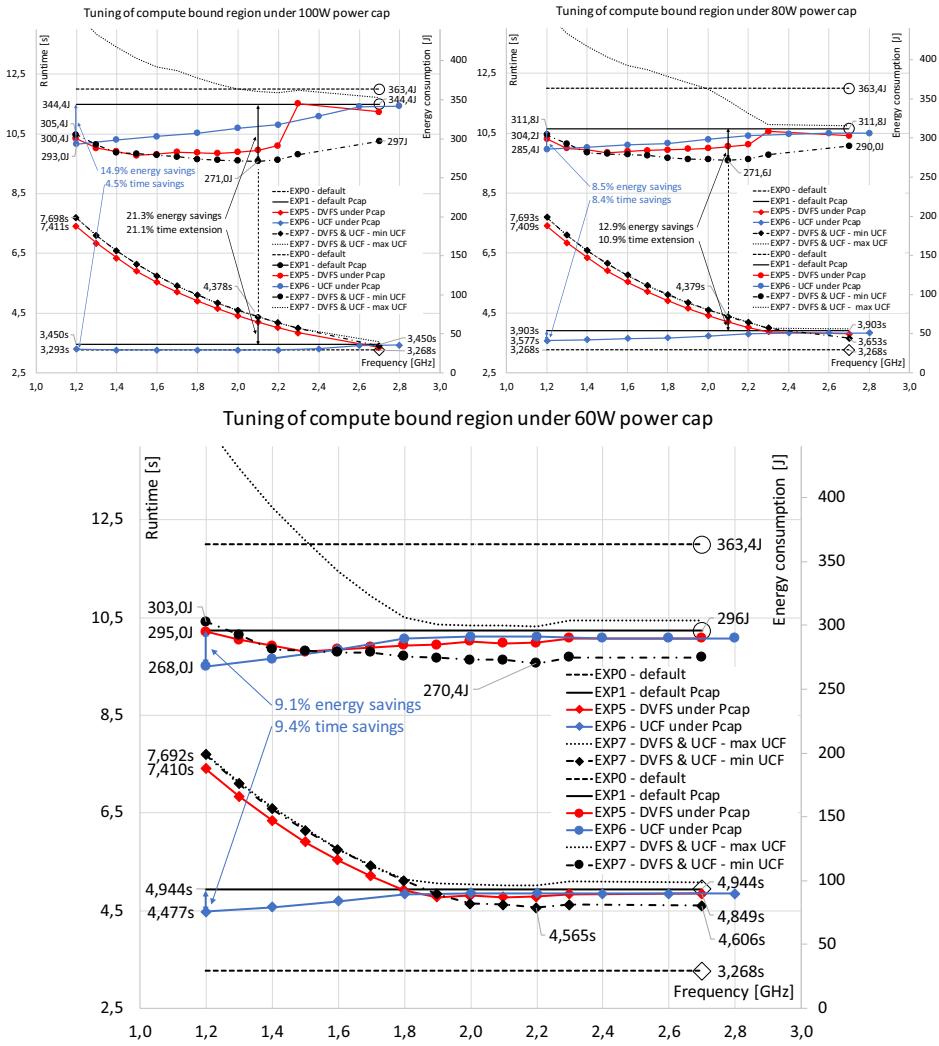


Figure 4. The behavior of the platform running compute bound workload (TAN) for (EXP1) default CPU behavior under powercap, (EXP5) DVFS tuning under powercap, (EXP6) UCF tuning under powercap and (EXP7) DVFS and UCF tuning under powercap. All tests are done for 60, 80 and 100 W powercap levels.

sumption, within a very small margin, approximately 271 J is achieved. This is achieved for minimal uncore frequency and 2.1 GHz core frequency.

However it is important to note, that by tuning both core and uncore CPU frequencies the performance gained by tuning the UCF only was not met. For 100 W UCF only tuning is 3.8 % faster, for 80 W it is 2.0 % faster, and for 60 W it is 1.9 % faster.

To summarize the numbers for EXP7: for 100 W by DVFS we can save 21.3 % energy at 21.1 % runtime penalty; for 80 W by DVFS we can save 12.9 % energy at 10.9 % runtime penalty; and for 60 W by DVFS we can save 8.6 % and at 7.7 % runtime penalty. All against the default CPU behavior under the same level of powercap (EXP1).

4. Conclusion - Summary of Observations and Best Practises

The Intel RAPL power capping system guarantees that the CPU keeps its energy consumption in a specified time window under a power boundary. We present how the system reduces both CPU core and uncore frequencies to reach this constraint. Since the system does not identify the kind of the workload running on the chip, it leads to the situation that core frequency is reduced while uncore frequency is still inefficiently too high for the given workload running or vice versa. Manually forcing a CPU configuration, DVFS or UCF, does not mean that the configuration will be applied if it infringes the power cap limit given to RAPL. However, manual reduction of one of the frequencies opens the availability to tune the other one to higher frequencies as it enables power bugdet shifting from one part of chip to the other one.

We have identified the optimal configuration of the CPU frequencies to reach the minimal energy consumption of the two workloads. When the powercap is applied, the CPU frequencies are reduced accordingly but not efficiently, due to that our manual frequency tuning leads to both time and energy savings.

To conclude, the results show that for MBW the proposed tuning can achieve:

- Under the power budget lower that 80 W settings the DVFS to minimum value boost the performance of the uncore part by 22 %.
- In addition to DVFS tuning the uncore frequency has low effect on the performance but a major one on energy consumption (between 21 % to 38 %).

The results show that for CBW the proposed tuning can achieve:

- To achieve the best possible performance it is key to reduce the UCF to minimum level. This way *BOTH* performance (up to 9.4 %) and energy consumption (up to 14.9 %) are improved.
- If further energy savings are required (up to 21 %) it can be achieved by DVFS tuning by lowering the core frequency. This comes at penalty in runtime (up to 21 %). This effect is more visible for higher powercap levels.

In the future work we would like to extend our measurements with benchmark, that can set vary arithmetic intensity on a fine grain (instruction) level and evaluate new CPU architectures code-named Skylake and Cascade Lake.

Acknowledgement

This work was supported by The Ministry of Education, Youth and Sports from the Large Infrastructures for Research, Experimental Development and Innovations project IT4Innovations National Supercomputing Center LM2015070.

The work has been performed under the Project HPC-EUROPA3 (INFRAIA-2016-1-730897), with the support of the EC Research Innovation Action under the H2020 Programme.

This work was supported by the Moravian-Silesian Region from the programme "Support of science and research in the Moravian-Silesian Region 2017" (RRC/10-/2017).

This work was also partially supported by the SGC grant No. SP2019/59 "Infrastructure research and development of HPC libraries and tools", VŠB - Technical University of Ostrava, Czech Republic.

References

[1] E. Strohmaier, "Highlights of the 51st top500 list," 2018.

[2] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, K. Yelick, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, P. Kogge, R. S. Williams, and K. Yelick, "Exascale computing study: Technology challenges in achieving exascale systems," 2008.

[3] V. Sarkar, W. Harrod, and A. E. Snavely, "Software challenges in extreme scale systems," *Journal of Physics: Conference Series*, vol. 180, p. 012045, jul 2009.

[4] F. Fraternali, A. Bartolini, C. Cavazzoni, and L. Benini, "Quantifying the Impact of Variability and Heterogeneity on the Energy Efficiency for a Next-Generation Ultra-Green Supercomputer," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, pp. 1575–1588, July 2018.

[5] C. Bonati, E. Calore, M. DElia, M. Mesiti, F. Negro, S. F. Schifano, G. Silvi, and R. Tripiccione, "Early Experience on Running OpenStaPLE on DAVIDE," in *High Performance Computing* (R. Yokota, M. Weiland, J. Shalf, and S. Alam, eds.), Lecture Notes in Computer Science, pp. 387–401, Springer International Publishing, 2018.

[6] E. Calore, A. Gabbana, S. F. Schifano, and R. Tripiccione, "Evaluation of dvfs techniques on modern hpc processors and accelerators for energy-aware applications," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 12, p. e4143, 2017. e4143 cpe.4143.

[7] READEX, "Horizon 2020 READEX project." https://www.readex.eu, 2018.

[8] J. Schuchart, M. Gerndt, P. G. Kjeldsberg, M. Lysaght, D. Horák, L. Říha, A. Gocht, M. Sourouri, M. Kumaraswamy, A. Chowdhury, M. Jahre, K. Diethelm, O. Bouizi, U. S. Mian, J. Kružík, R. Sojka, M. Beseda, V. Kannan, Z. Bendifallah, D. Hackenberg, and W. E. Nagel, "The readex formalism for automatic tuning for energy efficiency," *Computing*, vol. 99, pp. 727–745, Aug 2017.

[9] A. Knüpfer, C. Rössel, D. a. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, "Score-p: A joint performance measurement run-time infrastructure for periscope,scalasca, tau, and vampir," in *Tools for High Performance Computing 2011* (H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch, eds.), (Berlin, Heidelberg), pp. 79–91, Springer Berlin Heidelberg, 2012.

[10] J. Eastep, S. Sylvester, C. Cantalupo, B. Geltz, F. Ardanaz, A. Al-Rawi, K. Livingston, F. Keceli, M. Maiterth, and S. Jana, "Global extensible open power manager: A vehicle for hpc community collaboration on co-designed energy management solutions," in *ISC*, (Cham), pp. 394–412, Springer International Publishing, 2017.

[11] H. Zhang and H. Hoffmann, "Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, (New York, NY, USA), pp. 545–559, ACM, 2016.

[12] O. Vysocky, M. Beseda, L. Riha, J. Zapletal, V. Nikl, M. Lysaght, and V. Kannan, "Evaluation of the HPC applications dynamic behavior in terms of energy consumption," in *Proceedings of the Fifth International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering*, pp. 1–19, 2017. Paper 3, 2017. doi:10.4203/ccp.111.3.

[13] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "Rapl: Memory power estimation and capping," in *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, pp. 189–194, Aug 2010.

[14] CINECA, "GALILEO User Guide." https://wiki.u-gov.it/confluence/display/SCAIUS/UG3.3%3A+GALILEO+UserGuide, 2018.

[15] Intel corp., "Intel Xeon Processor E5-2697v4 Product Specification." https://ark.intel.com/content/www/us/en/ark/products/91755/intel-xeon-processor-e5-2697-v4-45m-cache-2-30-ghz.html, 2019.

This page intentionally left blank

ELPA – A Parallel Dense Eigensolver for Symmetric Matrices with Applications in Computational Chemistry

This page intentionally left blank

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200095

ELPA: A Parallel Solver for the Generalized Eigenvalue Problem[1]

Hans-Joachim BUNGARTZ [a], Christian CARBOGNO [b], Martin GALGON [c], Thomas HUCKLE [a,2], Simone KÖCHER [a], Hagen-Henrik KOWALSKI [b], Pavel KUS [d], Bruno LANG [c], Hermann LEDERER [d], Valeriy MANIN [c], Andreas MAREK [d], Karsten REUTER [a], Michael RIPPL [a], Matthias SCHEFFLER [b] and Christoph SCHEURER [a]

[a] *Technical University of Munich*
[b] *Fritz Haber Institute, MPG*
[c] *University of Wuppertal*
[d] *Max Planck Computing and Data Facility*

Abstract. For symmetric (hermitian) (dense or banded) matrices the computation of eigenvalues and eigenvectors $Ax = \lambda Bx$ is an important task, e.g. in electronic structure calculations. If a larger number of eigenvectors are needed, often direct solvers are applied. On parallel architectures the ELPA implementation has proven to be very efficient, also compared to other parallel solvers like EigenExa or MAGMA. The main improvement that allows better parallel efficiency in ELPA is the two-step transformation of dense to band to tridiagonal form. This was the achievement of the ELPA project. The continuation of this project has been targeting at additional improvements like allowing monitoring and autotuning of the ELPA code, optimizing the code for different architectures, developing curtailed algorithms for banded A and B, and applying the improved code to solve typical examples in electronic structure calculations. In this paper we will present the outcome of this project.

Keywords. ELPA-AEO, eigensolver, parallel, electronic structure calculations

1. Introduction

The ELPA-AEO project is a continuation of the ELPA project [1,2] where mathematicians, computer scientists, and users collaborate in order to develop parallel software for the Generalized Symmetric Eigenvalue Problem (GSEP) $AX = BX\Lambda$. The project partners are the Max-Planck Computing and Data facility in Garching, the University of Wuppertal, the Departments of Computer Sci-

[1]This work was supported by the Federal Ministry of Education and Research within the project "Eigenvalue soLvers for Petaflop Applications – Algorithmic Extensions and Optimizations" under Grant No. 01IH15001.
[2]Corresponding author; E-mail: huckle@in.tum.de

ence and of Chemistry of the Technical University of Munich, and the Fritz Haber Institute in Berlin. The former ELPA project developed a basic parallel GSEP solver and provided the software library `https://elpa.mpcdf.mpg.de/about`. Objective of the follow-up ELPA-AEO project is to include useful tools like monitoring and automatic performance tuning, to optimize the software for certain architectures, and to develop a special solver for banded GSEP.

The ELPA GSEP solver works in the following way:

- compute the Cholesky factorization $B = U^H U$ and the related standard eigenvalue problem (SEP) $\tilde{A}\tilde{X} = \tilde{X}\Lambda$ with $\tilde{A} = U^{-H} A U^{-1}$;
- reduce the SEP in $\tilde{A}$ directly (ELPA1) or with an intermediate banded matrix (ELPA2) to tridiagonal form;
- apply a divide-and-conquer solver to the tridiagonal SEP;
- transform the derived eigenvectors back to the original GSEP according to the previous steps.

In electronic structure computations a whole sequence of eigenproblems has to be solved with changing A.

In the following sections we will present the performance improvements included in ELPA-AEO, namely

- monitoring, autotuning, and optimization;
- improved matrix multiplication in the transformations via Cannon's algorithm;
- taking advantage of banded structure in A and B via Crawford's method;
- solving huge important GSEPs in electronic structure computations.

2. Optimization, Monitoring, and Autotuning

Clearly the most obvious change in the recent ELPA releases from the user perspective is the complete redesign of the library API. The new API requires the user to first create the ELPA object, then allows various manipulations with it in order to influence the library performance and finally to call one of the solution routines. Examples (shortened) of a program using ELPA in Fortran and C can be seen in Figures 1 and 2, respectively. The new API brought many benefits for the library users, whilst keeping the user-code changes on very reasonable level. Not only are the calling commands more elegant, but many new options and functionalities have been implemented. One of the most important is the introduction of autotuning. An example code, showing a possible use of this functionality is shown in Figure 3.

The autotuning works as follows. First of all, a set of parameters that should be tuned is selected (either by choosing the level of autotuning or manually). Each of the parameters can attain a limited number of values (e.g. all the different kernel implementations, or different values of certain block sizes, etc.). To alleviate the user from the need to wait too long and to avoid the necessity of wasting the valuable computer time, the autotuning can be performed during the production run with repeating calls (e.g. during the SCF cycle) to the solution routine, each time with one of the possible parameter combinations with the possibility to

```
use elpa
class(elpa_t), pointer :: e
integer :: success
e => elpa_allocate(success)
if (success \= ELPA_OK) ... !handle error
! set the matrix size
call e%set("na", na, success)
if (success \= ELPA_OK) ... !checks further omitted
! set in the same way all the required parameters
! describing the matrix and its MPI distribution.
call e%set("nev", nev, success)
call e%set("local_nrows", na_rows, success)
call e%set("local_ncols", na_cols, success)
call e%set("nblk", nblk, success)
call e%set("mpi_comm_parent", mpi_comm_world, success)
call e%set("process_row", my_prow, success)
call e%set("process_col", my_pcol, success)
success = e%setup()
! if desired, set other run-time options
call e%set("solver", elpa_solver_2stage, success)
! values of parameters can be retrieved
call e%get("stripewidth_real", stripewidth, success)
! call one of the solution methods
! the data types of a, ev, and z determine whether
! it is single/double precision and real/complex
call e%eigenvectors(a, ev, z, success)
! or, in the case of generalized EVP
call e%generalized_eigenvectors(a, b, ev, z, ...
    is_already_decomposed, success)
! cleanup
call elpa_deallocate(e)
call elpa_uninit()
```

Figure 1. Example use of the ELPA object. In the old API, all parameters were passed in one function call, which, with increasing number of customization parameters and options, became too inflexible and error prone since the signature of the function became too long and each newly introduced parameter would change the library API. With the new API, arbitrary large number of parameters can be added in the future. A new API for generalized EVP has been added, allowing the user to specify, whether he or she has already called the function with the same matrix B (using the is_already_decomposed parameter) and wants to re-use its factorizations, which is useful during the SCF cycle.

interrupt and resume the process and finally to store the optimal setting for future use, as it is suggested in Figure 3.

Apart from the previously mentioned changes, a lot of effort has been put into classical HPC optimizations of the code with respect to different architectures. This includes optimizations for the new CPU architectures, GPUs and interconnects. One of the recent HPC architectures, where ELPA has been successfully deployed is the supercomputer cobra at MPCDF, which comprises of skylake-based compute nodes, partially equipped with NVIDIA Volta V100 GPUs and the OmniPath interconnect. The performed optimizations included writing hand-tuned AVX-512 kernels (using compiler intrinsics), addressing MPI performance issues (finally solved by using Intel MPI 2019.3 or higher) and various GPU-related optimizations.

```
#include <elpa/elpa.h>
elpa_t handle;
handle = elpa_allocate(&error);
elpa_set(handle, "na", na, &error);
elpa_get(handle, "solver", &value, &error);
printf("Solver is set to %d \n", value);
elpa_eigenvectors(handle, a, ev, z, &error);
elpa_deallocate(handle);
elpa_uninit();
```

Figure 2. Example use of the C interface. The object-oriented approach is implemented using the handle pointer. Apart from this, the library use is very similar as through the Fortran interface (as presented in Figure 1), and the C example is thus kept very short for brevity.

As ELPA originated as a replacement for the ScaLAPACK routines P?SYEVR and P?SYEVD, it is natural to compare its performance with the best available implementation of this widely used and de-facto standard library for a given architecture, as it has been done in the past ([1], [3]) on Intel-based machines and also recently by independent authors in [4] using the Cray system. Such comparison can be seen in Figure 4, comparing the performance of the ELPA library with Intel MKL 2019.5 for a matrix of the size 20000. Scaling curves for larger matrices including a cross-island run can be seen in Figure 5. It is obvious, that the performance of the ELPA library, especially its implementation of the two-stage algorithm, exceeds the performance of the MKL routines significantly, as it is consistent with other reports.

A lot of effort has been put into GPU related optimizations of ELPA, since the number of GPU-equipped HPC systems is on the rise. We have already reported this effort and the obtained results in the previous papers [5] and [3], so let us here only present a typical performance output (see Table 1) and reiterate some conclusions:

- ELPA 1-stage can run significantly faster using GPUs, which is not the case for ELPA 2-stage, where the speed-up is moderate to none at the moment.
- In order to benefit from the GPUs, there has to be enough data to saturate them. It is thus beneficial to use them for setups, where there are large local matrices (possibly up to the memory limits), thus for large matrices and/or moderate number of GPU equipped nodes.
- To fully utilize both the CPUs and GPUs, ELPA is run as a purely MPI application with one MPI rank per core and the efficient use of the GPU cards is achieved through the NVIDIA MPS daemon.

We can thus conclude (see Table 1), that the GPU implementation of ELPA 1-stage is utilizing the GPUs well and given a suitable problem setup, it can be very efficiently used to reduce the total application runtime.

3. Reduction of Full Generalized Eigenvalue Problems

The solution of a GSEP $AX = BX\Lambda$ with A hermitian and B hermitian positive definite typically proceeds in four steps.

```
use elpa
class(elpa_t),           pointer :: e
class(elpa_autotune_t), pointer :: tune_state
e => elpa_allocate()
! set all the required fields, omitting others
call e%set("na", na, error)
! alternatively exclude some parameters from autotuning by ...
    setting them
call e%set("gpu", 0)
!set up the ELPA object and create the autotuning object
success = e%setup()
tune_state => e%autotune_setup(level, domain, error)

if(done_with_autotuning) then
  call e%load_all_parameters("autotuned_pars.txt")
elseif(autotuning_in_progress) then
  call e%autotune_load_state(tune_state, "atch.txt")
endif
iter=0
! application-specific cycle, where multiple similar
! EVP problems are solved, e.g. the SCF cycle
do while (continue_calculation)
  if(.not. done_with_autotuning) &
    finished = .not. e%autotune_step(tune_state)
  if(finished) then
    ! set and print the autotuned-settings
    call e%autotune_set_best(tune_state)
    ! the current values of the parameters can be saved
    call e%save_all_parameters("autotuned_pars.txt")
    done_with_autotuning = .true.
  endif
  ! do the actual calculation
  call e%eigenvectors(a, ev, z, error)
  ! do whatever needed with the result
end do
if (.not. done_with_autotuning) then
  ! the status of the autotuning can be saved
  call e%autotune_save_state(tune_state, "atch.txt")
endif
! de-allocate autotune object
call elpa_autotune_deallocate(tune_state)
```

Figure 3. A sketch of a code, which performs autotuning during a production run of a program which calls the ELPA library repeatedly. It also shows how to split the autotuning process into multiple calls of the program by saving the autotuning state into a checkpoint file atch.txt. Each actual library call is performed with slightly different settings. After all combinations have been exhausted, the optimal settings are saved to the autotuned_pars.txt file, the autotuning is not performed any more and the optimal setting is used ever since.

i) Compute Cholesky decomposition $B = U^H U$.
ii) Reduce the GSEP to an equivalent SEP $\tilde{A}\tilde{X} = \tilde{X}\Lambda$, where $\tilde{A} = U^{-H} A U^{-1}$.
iii) Solve the SEP.
iv) Back-transform the eigenvectors via $X = U^{-1}\tilde{X}$.

Since one key application of ELPA is electronic structure theory, where often a

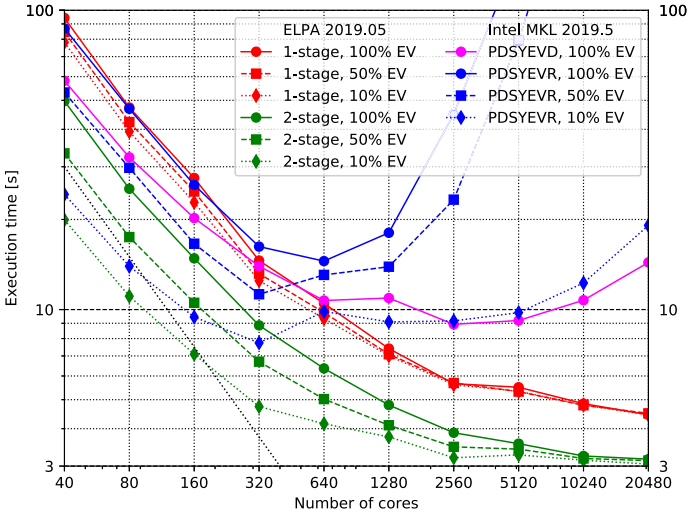


Figure 4. Scaling results from the most recent skylake-based supercomputer at MPCDF using real double precision matrix of the size 20000. Compared is the performance of the two relevant MKL 2019.5 routines and ELPA one and two stage. Where possible, also results for 10% and 50% of eigenvectors are shown. The MKL routines offer comparable or superior performance to the ELPA one-stage algorithm for small and moderate number of cores, but do not scale for larger core counts. ELPA scaling is generally much better in the investigated region. For the very large core counts and thus very small local matrices, the speed-up with growing number of cores is slowing down, but the performance is not deteriorating, which can be very beneficial when coupled with a well scaling application. The ELPA two-stage solver clearly outperforms all the other routines in this setup.

sequence of GSEPs $A^{(k)}X^{(k)} = BX^{(k)}\Lambda^{(k)}$ *with the same matrix* B have to be solved during a *self consistent field* (SCF) cycle, ELPA's approach for the above step ii) is to explicitly compute B^{-1} and then to do (triangular) matrix multiplications to obtain $\tilde{A}$. Alternative approaches use the inverse only implicitly; cf. the routines `PDSYNGST` and `TwoSidedTrsm` in the ScaLAPACK [6] and ELEMENTAL [7] libraries, resp.

With the inverse U^{-1} available explicitly (again upper triangular, denoted as $\hat{U}$ in the following), a computationally efficient way to implement the above step ii), $\tilde{A} = \hat{U}^H A \hat{U}$, is as follows [8].

ii.a) Compute the upper triangle M_u of $M := A\hat{U}$.
ii.b) Transpose M_u to obtain the lower triangle M_l of $M^H = \hat{U}^H A^H = \hat{U}^H A$.
ii.c) Compute the lower triangle of $\tilde{A} = M_l \hat{U}$.
ii.d) If the whole matrix $\tilde{A}$ is needed then reflect its lower triangle along the diagonal.

During the ELPA-AEO project, new algorithms have been developed for the multiplications in steps ii.a) (*Multiplication 1*: compute upper triangle of "hermitian × upper triangular") and ii.c) (*Multiplication 2*: compute lower triangle of "lower triangular × upper triangular"). Compared to these multiplications, the transpositions in steps ii.b) and d) are inexpensive [9].

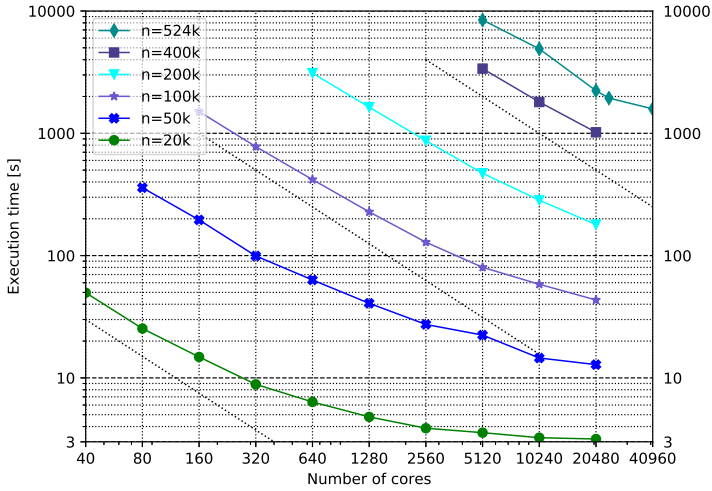


Figure 5. Strong scaling graphs for ELPA 2-stage computing all eigenvectors with different matrix sizes n. The line for n=20k corresponds to the line of the same color from Figure 4. The results shown in both figures were obtained on the supercomputer cobra at MPCDF, comprising of compute nodes containing two Intel Xeon Gold 6148 processors (Skylake with 20 cores (each) at 2.4 GHz) connected through a 100 Gb/s OmniPath interconnect. Most of the calculations shown were performed within a single island with a non-blocking, full fat tree network topology. The blocking factor among islands is 1:8. The only cross-island run is for the largest matrix n=524k and 40960 cores showing reasonable performance despite the weaker network between the islands.

Table 1. ELPA runtimes (s) on a full Skylake node (40 cores in total) equipped with two NVIDIA Volta V100 GPUs. As it is usually the case, ELPA is running as purely MPI application (thus using 40 MPI ranks). In the GPU case, each of the MPI ranks is offloading compute intensive kernels to one of the GPUs (through the NVIDIA MPS for efficiency). As it can be seen from the results, even using one particular architecture, it is not possible to determine the generally best option. In this particular case, ELPA 1-stage CPU is the best option for very small matrices, ELPA 2-stage CPU for larger and ELPA 1-stage GPU for the largest. The ELPA 2-stage GPU is not listed, since its performance is almost never the best possible and is thus currently not recommended.

	CPU		**GPU**
matrix size	ELPA 1	ELPA 2	ELPA 1
1024	**0.11**	0.13	0.93
8192	10.7	**5.57**	8.45
20000	110	52.7	**37.0**
65536	5795	2551	**733**

Our algorithms are based on Cannon's method [10]; they exploit the triangular structure to save on arithmetic operations and communication, and they have been extended to work on non-square $p_r \times p_c$ grids with integer aspect ratio $p_c : p_r$. In this case, they take p_r phases, which improves over the p_c phases of the approach described in [11] for full matrices.

Here we only point out the main ideas for Multiplication 1. Assume that

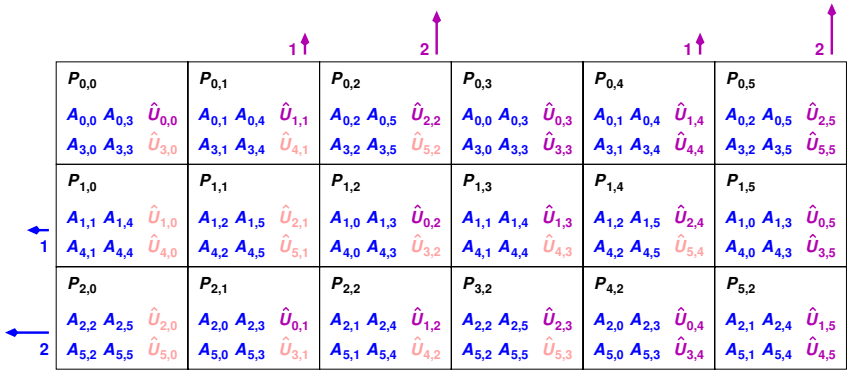


Figure 6. Distribution of the matrices for Multiplication 1 *after* the initial skewing and the sharing of A blocks. The numbers next to the arrows indicate the distance of the skewing shifts within rows/columns of the process grid. See the main text for a description.

$A \in \mathbb{C}^{n\times n}$ and $\hat{U} \in \mathbb{C}^{n\times n}$ have been partitioned into $N \times N$ blocks $A_{i,j}$ ($\hat{U}_{i,j}$, resp.) of size $n_b \times n_b$, where $N = \lceil n/n_b \rceil$, and that they are distributed over the process grid in a *block torus wrapped* manner, i.e., process $P_{k,\ell}$ holds exactly those blocks $A_{i,j}$ and $\hat{U}_{i,j}$ such that $i \equiv k \mod p_r$ and $j \equiv \ell \mod p_c$. This is also the default distribution in ScaLAPACK and ELPA. Considering the case $N = 6$, $p_r = 3$, $p_c = 6$ as an example (cf. also Figure 6), process $P_{1,5}$ would hold the blocks $A_{1,5}$, $A_{4,5}$, $\hat{U}_{1,5}$, and $\hat{U}_{4,5}$. Next we do a Cannon-type *initial skewing*: In row k of the process grid, $k = 0, \ldots, p_r - 1$, the local portions of A are shifted by k positions to the left, and in column ℓ, $\ell = 0, \ldots, p_c - 1$, the local portions of $\hat{U}$ are shifted by $\ell \mod p_r$ positions upwards (with cyclic connections along rows and columns). Therefore, $P_{1,5}$ now has $P_{1,5+1} \equiv P_{1,0}$'s original blocks from A (i.e., $A_{1,0}$ and $A_{4,0}$) and $P_{1+2,5} \equiv P_{0,5}$'s original blocks from $\hat{U}$ (i.e., $\hat{U}_{0,5}$ and $\hat{U}_{3,5}$). Finally, groups of p_c/p_r processes that are p_r positions apart in the same row, share their portion of A. In our example, $P_{1,5}$ shares the A blocks with $P_{1,2}$, such that both hold the same blocks $A_{1,0}$, $A_{4,0}$, $A_{1,3}$, $A_{4,3}$ from A, but different blocks from $\hat{U}$, cf. Figure 6. Note that the blocks $\hat{U}_{i,j}$ in the strict lower triangle of $\hat{U}$ are zero and therefore need not be stored and sent; they bear a light color in Figure 6.

After these preparations, the computation proceeds in p_r phases. In each phase, every process multiplies its current local A with the current local $\hat{U}$. In our example, taking into account the structure of $\hat{U}$ and the fact that we compute only the lower triangle of the product $M = A\hat{U}$, in the first phase $P_{1,5}$ would update $\begin{bmatrix} M_{1,5} \\ M_{4,5} \end{bmatrix} = \begin{bmatrix} M_{1,5} \\ M_{4,5} \end{bmatrix} + \begin{bmatrix} A_{1,0} & A_{1,3} \\ A_{4,0} & A_{4,3} \end{bmatrix} \cdot \begin{bmatrix} \hat{U}_{0,5} \\ \hat{U}_{3,5} \end{bmatrix}$, whereas the update in $P_{1,3}$ reads $\begin{bmatrix} M_{1,3} \\ M_{4,3} \end{bmatrix} = \begin{bmatrix} M_{1,3} \\ M_{4,3} \end{bmatrix} + \begin{bmatrix} A_{1,1} \\ A_{4,1} \end{bmatrix} \cdot \hat{U}_{1,3}$, and $P_{1,1}$ performs no computation at all in this phase. At the end of each phase, the local A blocks are shifted by one position to the left in the process grid, and the $\hat{U}$ blocks are shifted by one position up. It is not hard to verify that, after p_r such phases, $P_{1,5}$ has computed "its" blocks

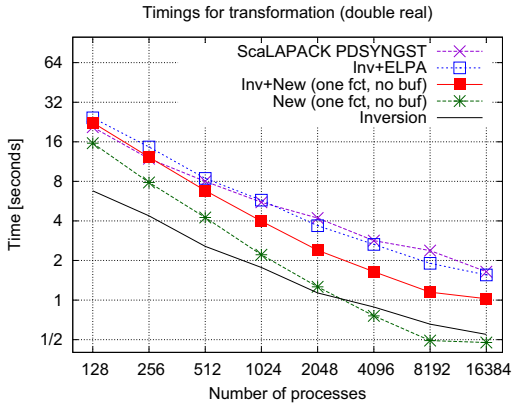


Figure 7. Timings on HYDRA for the complete transformation $A \to \tilde{A}$ with the ScaLAPACK routine `PDSYNGST` (one step, inverse of the Cholesky factor used only implicitly) and with the "invert and multiply" approach (multiplication routines from ELPA or the new Cannon-based implementations). $n = 30{,}000$, $n_b = 64$, 16 single-threaded processes per node.

$M_{1,5}$ and $M_{4,5}$ of the block torus wrapped-distributed product M, and similarly for the other $P_{k,\ell}$.

For a description of Multiplication 2 and a discussion of possible savings from combining the two multiplications in one function and buffering some data the reader is referred to [12].

In Figure 7 we present timings obtained on the HYDRA system at the Max Planck Computing and Data Facility in Garching. Each HYDRA node contains two 10-core Intel Ivy Bridge processors running at 2.8 GHz. All matrices were double precision real of size $n = 30{,}000$, and the block size was $n_b = 64$. We observe that explicit inversion, combined with our Cannon-based matrix multiplications, can be highly competitive even for solving a single generalized eigenproblem (red curve, including the time for inverting U). For sequences of GSEPs with the same B, where the inversion can be skipped in most cases, the new reduction according to steps ii.a) to d) is significantly faster (green curve).

In [12] we have considered only MPI parallelization, using p processes for utilizing a total of p cores. Alternatively, one can reduce the number of processes and enable multithreaded execution. This may or may not be beneficial, depending on several factors. In particular, while multithreading reduces the size of the process grid and therefore leads to savings in communication, it also can cause a loss of computational performance if running a process' computations with φ threads does not speed them up by a factor of φ.

In the left picture of Figure 8 we see that using multiple threads per process may extend the range of scalability. More details are exposed in the right picture, which shows the relative timings for the same runs. We see that for numbers of cores p that are not squares and therefore would lead to a non-square grid when using single-threaded processes, using 2 threads per process (leading to a square process grid) reduced the time for multiplication 1 by roughly 10%, whereas it increased the time for those p that are square and not very large. This indicates

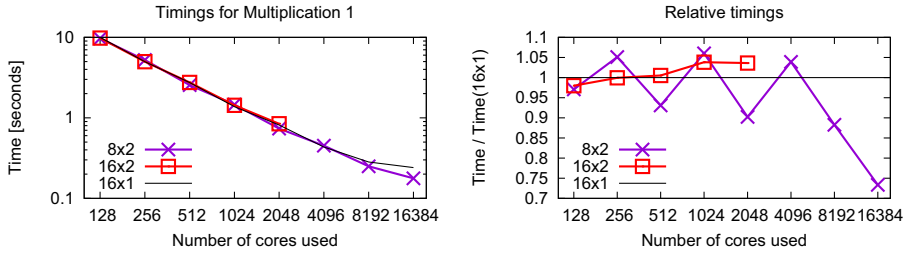


Figure 8. (Left) Time of Multiplication 1 on HYDRA for $n = 30{,}000$, $n_b = 64$, if 16 cores per node are used for 16 single-threaded processes per node ("16×1"), 8 processes with 2 threads each ("8×2"), and 16 processes with 2 threads each (hyperthreading, "16×2"). (Right) Times relative to the baseline of 16 single-threaded processes per node.

a slight preference of that routine for square process grids. If the number of cores is large enough that communication contributes significantly to overall time then multithreading pays independently of the grid's shape because it leads to a smaller process grid and therefore reduces communication.

The effect of multithreading also differs between the routines in the ELPA library; see Table 2. Some of them contain explicit OpenMP directives or pragmas for controlling thread parallelism. Others rely exclusively on multithreaded BLAS, and the efficiency of the latter depends on the size and shape of the involved matrices, which may be rather different; this is the case, e.g., with the `GEMM` calls in Multiplication 1 vs. Multiplication 2 [12].

A detailed discussion of these issues is not within the scope of this work, but note that even the decision whether to use multithreading for the complete solution of an eigenproblem (considering all routines involved) may depend on whether it is part of a whole sequence of eigenproblems, as common in SCF cycles, or just a single eigenproblem; cf. the last two lines in Table 2. See Section 2 on support in ELPA for taking such decisions in a partially or fully automated way.

Table 2. Timings (in seconds) on HYDRA for $n = 30{,}000$, $n_b = 64$, with different setups of 4096 cores (256 nodes with 16 cores each): 16 single-threaded processes per node, 8 processes with 2 threads, and 4 processes with 4 threads.

	16×1	8×2	4×4
Cholesky decomposition	1.804	1.078	0.878
Invert	0.885	0.804	0.781
New transformation	0.759	0.709	0.705
Solution of standard eigenproblem	8.189	8.668	8.758
Back-transformation 33%	0.247	0.235	0.203
Overall without Chol. & Invert	9.195	9.612	9.665
Overall including Chol. & Invert	11.883	11.494	11.324

4. Eigenvalue Solver for Banded Matrices

If, additionally, the two matrices A, $B = U^H U$ in the GSEP $AX = BX\Lambda$ are banded, the procedure described in Section 3 is not optimal as it leads to a full matrix $\tilde{A}$ (the Cholesky factor of B is still banded, but the inverse of the Cholesky factor is in general a full matrix and hence $\tilde{A}$ becomes a full matrix).
The two-stage solver in ELPA however, first transfers a full matrix C of a SEP $CY = Y\Lambda$ to a banded matrix $\hat{C}$ and then further transforms it to a tridiagonal matrix $\bar{C}$ which is solved for the eigenvalues and eigenvectors. Subsequently, the eigenvectors undergo two backtransformation steps to obtain the eigenvectors of the SEP. In this framework, by maintaining the band while transforming the GSEP to a SEP, the first step (transformation of the full matrix to the banded matrix) can be omitted as well as the second step of the backtransformation.

Crawford proposed an algorithm for maintaining the band in [13]. His algorithm stepwise applies the Cholesky factorization of B and removes the occurring fill-in outside the band by a series of QR factorizations. Lang extended the algorithm in [14]. His version offers more flexibility for blocksizes and bandwidth and utilizes a twisted factorization for B instead of a standard Cholesky factorization. The latter allows to reduce computational work when removing the occurring fill-in drastically. In the following we will briefly describe our parallel implementation of Lang's algorithm including the backtransformation of the eigenvectors. A more detailed description can be found in [15].

For the parallel implementation we use a unified blocksize $n_b = \max(b_A, b_B)$ (as in the original Crawford algorithm) to get an efficient pipelining algorithm. The matrices A and B can therefore be subdivided into $N \times N$ blocks with $N = \lceil \frac{n}{n_b} \rceil$. The case when n is not a multiple of n_b can be covered by adding an incomplete block at the end.

The matrix U originates from the twisted factorization of B with twist position p and twist block P (the twist position p is chosen such that it is the end of a block; this block is referred to as twist block). U can itself be factorized as

$$U = U_P \cdot U_{P-1} \cdots U_1 \cdot U_{P+1} \cdots U_{N-1} \cdot U_N.$$

Each of the factors U_i has the shape of an identity matrix besides one block row between the rows $(i-1)n_b + 1$ and in_b. These rows contain the same values as in the matrix U at the same place. Figure 9 gives an illustration of the matrix shapes and the block structure.
The transformation $\tilde{A} = U^{-H} A U^{-1}$ can therefore be reformulated to the stepwise application

$$\begin{aligned}\tilde{A} = U_P^{-H} \cdot U_{P-1}^{-H} \cdots U_1^{-H} \cdot U_{P+1}^{-H} \cdots U_{N-1}^{-H} \cdot U_N^{-H} \cdot \\ A \cdot U_N^{-1} \cdot U_{N-1}^{-1} \cdots U_{P+1}^{-1} \cdot U_1^{-1} \cdots U_{P-1}^{-1} \cdot U_P^{-1}.\end{aligned}$$

As it can be seen from Figure 9 every U_i consists of one block row that differs from the identity matrix. In this block row, we will denote the diagonal block as $U_{i,i}$ and the other non-zero block as $U_{i,i-1}$ or $U_{i,i+1}$, depending on the position in the lower or upper matrix half. When inspecting the inverse of the matrix,

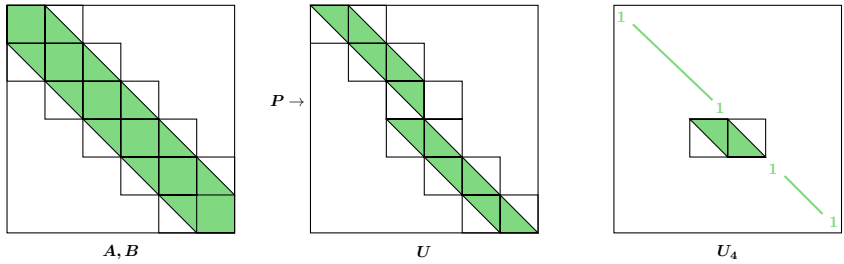


Figure 9. Block structure of the matrices A and B (left), the twisted factorization of B, U with its twist block P (middle), and one of its factors, U_4 (right).

U_i^{-1}, it can be seen that it has the same structure as U_i. The diagonal block of the inverse matrix turns out to be the inverse of the diagonal block of U_i. In the further text we use $D_i := U_{i,i}^{-1}$ as abbreviation for this block. The other block of the inverse matrix is denoted as E_i and can be described by $E_i := -D_i U_{i,i-1}$ in the lower matrix half or $E_i := -D_i U_{i,i+1}$ in the upper matrix half, respectively.

Applying one factorization step U_i in the lower matrix half hence takes the i-th block column of A, multiplies it from the right with E_i and adds it to block column $i-1$ (upper matrix half: block column $i+1$). Afterwards, block column i is multiplied from the right with D_i. The same procedure is rolled out for the multiplication from the left with U_i^{-H}. Block row i times E_i^H is added to block row $i-1$ (upper matrix half: block row $i+1$) and subsequently block row i is multiplied by D_i^H.

Figure 10 shows in the left picture the application of a factorization step in the lower matrix half and the occurring fill-in (left two block columns). The fill-in is created in the blocks $A_{i,i-1}$, $A_{i+1,i-1}$ and $A_{i+1,i}$. Due to symmetry we restrict the description to the lower triangle of the matrix A. On the blocks $A_{i,i-1}$ and $A_{i+1,i-1}$ a QR decomposition is computed and the block rows i and $i+1$ are multiplied with the obtained Q from the left as well as the block columns i and $i+1$ from the right. The symmetric application of Q shifts the fill-in by one block row and one block column towards the lower end. By repeating the QR step, the fill-in can be completely evicted from the matrix and the next factorization step can be applied. The procedure for the upper matrix half is the same, only the QR factorization is replaced by a QL factorization and the fill-in moves stepwise towards the top left of the matrix.

Denoting the Qs following the application of U_i with $Q_i^{(k)}$, the series $U_i, Q_i^{(1)}, Q_i^{(2)}, \ldots, Q_i^{(\nu_i)}$ applies one step of the factorization and restores the band. Hence, using

$$\tilde{U}^{-1} = U_N^{-1} \cdot Q_N^{(1)} \cdots Q_N^{(\nu_N)} \cdots U_{P+1}^{-1} \cdot Q_{P+1}^{(1)} \cdots$$
$$Q_{P+1}^{(\nu_{P+1})} \cdot U_1^{-1} \cdot Q_1^{(1)} \cdots Q_1^{(\nu_1)} \cdots U_P^{-1} \cdot Q_P^{(1)} \cdots Q_P^{(\nu_P)},$$

the overall transformation with restoring the band can be described as $\hat{A} = \tilde{U}^{-H} A \tilde{U}^{-1}$.

The eigenvalues of the SEP $\hat{A}\hat{X} = \hat{X}\Lambda$ are the same as the eigenvalues of the GSEP $AX = BX\Lambda$, but for the eigenvectors a backtransformation step has to be

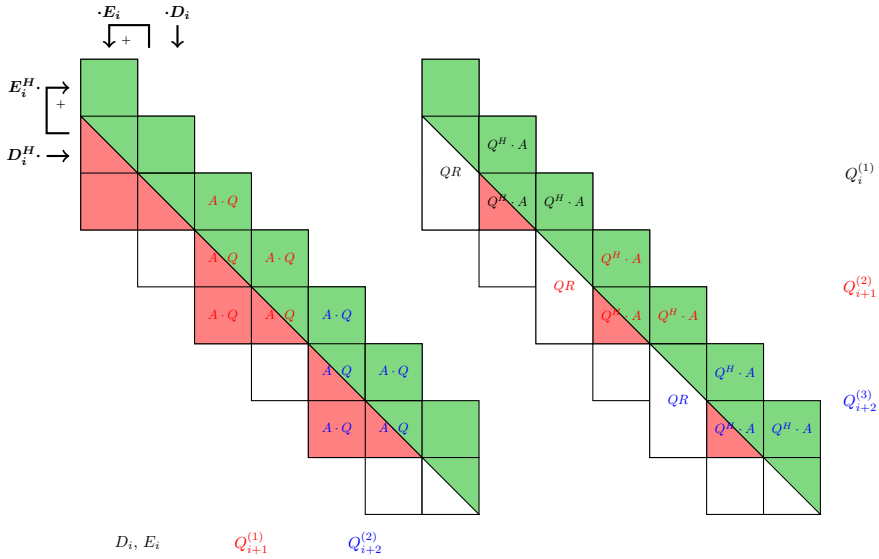


Figure 10. The two steps of the algorithm (in the lower matrix half, upper matrix half similar but mirrored): 1. (left) Applying the factorization (D_i, E_i) and performing the right sided update of the two-sided Q-application. Since operating on different block columns, Q of different factorization steps can be applied in parallel. Red indicates the bulge of newly created non-zeros outside the band (in green). 2. (right) Eliminating the fill-in by generating the QR decomposition and applying it from the left to the matrix. Since operating on different block rows, the QR decomposition and the left sided application of Q of different factorization steps can be applied in parallel.

applied. These eigenvectors of the GSEP can be found by applying $\tilde{U}^{-1}$ to the eigenvectors of the SEP: $X = \tilde{U}^{-1}\hat{X}$.

Contrary to the computation of $\hat{A}$ the eigenvectors are multiplied from the left with $\tilde{U}^{-1}$ and not with the hermitian of it. Therefore the order of the operations is reverse: $Q_i^{(\nu_i)}, \ldots, Q_i^{(2)}, Q_i^{(1)}, U_i$. Figure 11 gives an illustration of the updating scheme. In the lower matrix half the applications of $Q_i^{(k)}$ update the block rows $i+k-1$ and $i+k$. After having applied the Qs, D_i multiplies the block row i from the left and to this block row the $i-1$st block row multiplied from the left with E_i is added. The block rows to update in the upper matrix half are slightly different. $Q_i^{(k)}$ update the block rows $i-k+1$ and $i-k$ and instead of multiplying the $i-1$st block row with E_i, the $i+1$st block row is added to block row i (which has been multiplied by D_i).

Having a closer look on the application of the factorization and the generation and application of the Q, it can be seen that by splitting the two-sided application of Q and by interchanging the order of the $Q_i^{(k)}$ a pipelining structure can be obtained. It is based on the fact that a left sided update with $Q_i^{(k)}$ updates only two consecutive block rows and a right sided update and the application of the factorization only update two consecutive block columns. The order of execution has to be kept within a factorization step, meaning $Q_i^{(k)}$ has to be executed before $Q_i^{(k+1)}$, but $Q_i^{(k)}$ can be executed at the same time as $Q_{i+1}^{(k+1)}$. Details on the

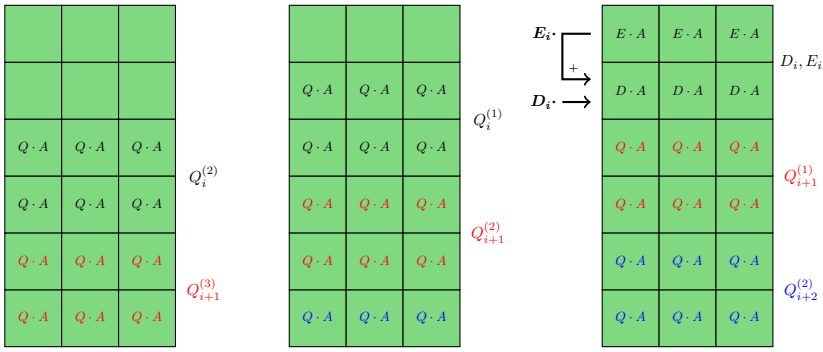


Figure 11. Three consecutive steps in the backtransformation: The applications of different Q can be done in parallel in the same way the Q have been created (see Figure 10, right picture). Finally, the D_i and E_i are applied.

interchangeability can be found in [15]. Figure 10 shows the pipelining structure in the lower matrix half in the computation of $\hat{A}$.

A similar pipelining scheme can be obtained for the backtransformation step: All $Q_{i+j}^{(k+j)}$ can be applied to the eigenvectors simultaneously. Additionally, the application of D_i and E_i can be decoupled from the application of the $Q_i^{(k)}$ and can be executed afterwards. Figure 11 shows the application of different $Q_i^{(k)}$ which can be executed concurrently.

Besides the pipelining scheme, $\tilde{U}^{-1}$ offers another parallelization layer which comes by the twisted factorization. The operations in $\tilde{U}^{-1}$ first process the lower matrix half and afterwards the upper matrix half. These operations, however, do not overlap besides the twist block P that is updated by the upper and the lower matrix half of the factorization. Therefore they can be run in parallel with a synchronization point at the twist block. Concluding, the algorithm provides three parallelization layers: parallel execution of the upper and lower matrix half, parallel execution of the independent steps in the pipeline and parallelization of the operations in the single blocks. Additionally, the use of a threaded BLAS library can provide a fourth layer of parallelization.

The process setup is hence chosen in a way to exploit the parallelization layers. The available processes are separated in processes for the upper and the lower matrix half. Processes of a matrix half are further subdivided into groups which compute the operations of a block. These groups are ordered in a grid and if not enough groups are available to fill all blocks, repeated cyclically. All operations involve communication between the processes of two groups. Due to the constant neighbourhood, local communicators are used to perform these operations efficiently. In the backtransformation step the process setup is used in the same way, exploiting to have the Householder vectors already in place.

Figures 12 and 13 show the strong scaling behaviour of the algorithm for matrix sizes of 51200 and 204800. The overall runtime as well as the two main steps are plotted: the backtransformation of the eigenvectors and the application of $\tilde{U}^{-1}$. The bandwidth of the matrices was in both cases 1% of the matrix size. Both matrix sizes show good scaling for a selected number of processes per group. If all groups only hold one block further speedup can be achieved by using more

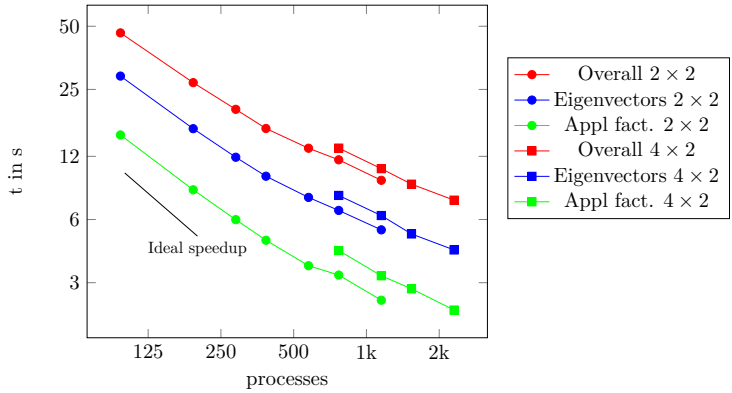


Figure 12. Strong scaling for a matrix of size 51200. The bandwidth (and hence the blocksize) is 512, the twist index is at 25600. The backtransformation is done for 12800 eigenvectors (25%). Per group 2×2 and 4×2 processes have been used. The runs have been carried out on the Cobra Supercomputer.

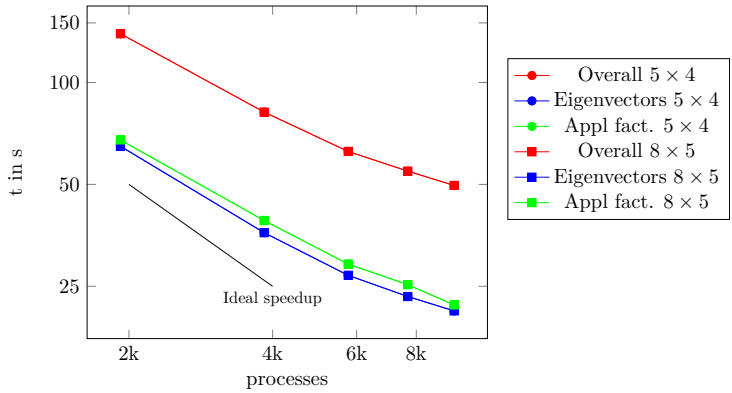


Figure 13. Strong scaling for a matrix of size 204800. The bandwidth (and hence the blocksize) is 2048, the twist index is at 102400. The backtransformation is done for 25600 eigenvectors (12.5%). Per group 5×4 and 8×5 processes have been used. The runs have been carried out on the Cobra Supercomputer.

processes per group. Using more processes per group, however, comes along with a loss in performance compared to the same number of processes with less processes per group. Not shown here is the additional bandreduction step which is necessary for bandwidth of size 512 or 2048. The savings compared to computing the dense eigenvalue problem however, will still be significant.

For solving generalized eigenvalue problems with banded matrices this procedure allows to compute eigenpairs at matrix sizes where the standard procedure with factorizing B, applying B to A and using a standard dense solver for the full resulting matrix C would consume to much memory or result in way more computation. When considering sparse eigenvalue solvers, the computation of

higher percentages of the eigenpairs becomes expensive. This approach, however, provides the possibility to overcome this issue.

5. Applications

First-principles simulations in computational chemistry, solid state physics, and materials science typically involve to determine the interactions between the M nuclei described by $3M$ nuclear positions $\{\vec{R}\}$. Being able to compute the total energy of the system $E_0(\{\vec{R}\})$, i.e., the high dimensional potential energy surface (PES), as a function of $\{\vec{R}\}$ and, ideally, its derivatives such as the forces acting on the nuclei $\vec{F}_I(\{\vec{R}\}) = -\nabla_{\vec{R}_I} E_0(\{\vec{R}\})$, allows to investigate the properties of molecules and materials. For instance, one can systematically map out the PES $E_0(\{\vec{R}\})$ to search for (stable) minima and saddle points between them or explore it dynamically via molecular dynamics (MD) or statistical (e.g. Monte Carlo) sampling. Accordingly, a typical computational study often requires to determine $E_0(\{\vec{R}\})$ for thousands of nuclear configurations $\{\vec{R}\}$.

Computing $E_0(\{\vec{R}\})$ requires to solve the quantum-mechanical electronic-structure problem. In density-functional theory (DFT) [16], the most wide-spread electronic-structure formalism, this requires to identify the electronic density $n(\vec{r})$ that minimizes the convex total-energy functional $E_0 = \min E[n(\vec{r})])$ for a given number of electrons $N = \int d\vec{r} n(\vec{r})$. In Kohn-Sham (KS) DFT [17], this variational problem is mapped onto a series of eigenvalue problems (EVP), the so called self-consistent field (SCF) formalism. In each step of the SCF cycle, the EVP

$$H\left[n(\vec{r})\right]\Psi(\vec{r}) = \varepsilon\Psi(\vec{r}) \quad \text{with} \quad n(\vec{r}) = \sum_{s=1}^{N} |\Psi_s(\vec{r})|^2 \tag{1}$$

is solved to determine the eigenstates Ψ_s. The N eigenstates Ψ_s with the lowest eigenvalues ε_s allow to compute an updated and improved $n(\vec{r})$, for which Equation (1) is then solved again. This procedure is repeated until "self-consistency" is achieved at the end of the so called SCF cycle, i.e., until a stationary solution with minimal $E[n(\vec{r})]$ is found. In practice, a basis set expansion $\Psi_s = \sum_i x_{si}\varphi_i(\vec{r})$, e.g., in terms of Gaussians, plane waves, numerical functions, etc., is used to algebraize and solve Equation (1). By this means, one obtains the generalized EVP

$$A\left[n(\vec{r})\right]x = \lambda B x \,, \tag{2}$$

the size of which is determined by the number of basis functions $\varphi_i(\vec{r})$ employed in the expansion. Here, the Hamiltonian A and the overlap matrix B are given as:

$$A_{ij}[n(\vec{r})] = \int d\vec{r}\varphi_i^*(\vec{r})\, H\left[n(\vec{r})\right]\, \varphi_j(\vec{r}) \,, \quad B_{ij} = \int d\vec{r}\varphi_i^*(\vec{r})\varphi_j(\vec{r}) \,.$$

5.1. Autotuning: The Case of GPU Offloading

Due to the cubic scaling with system size, the generalized EVP (2) quickly becomes the numerical bottleneck in practical DFT calculations. It is thus more than desirable to use optimal ELPA settings (ELPA1 vs. ELPA2, architecture-specific kernels, etc.) to utilize the computational resources in the most efficient way so to obtain the optimal time-to-solution. As discussed above, this is of particular importance in first-principles simulations, which require solving many similar eigenvalue problems, e.g., the 10–100 individual SCF steps in one SCF cycle or the thousands if not millions of SCF steps performed in an iterative exploration of the PES $E_0(\{\vec{R}\})$. ELPA's autotuning feature allows to determine these optimal settings, which depend upon both the inspected physical problem and the used architecture, in an automated way [3].

This is particularly important for new and upcoming architectures featuring GPUs: This is exemplified in Figure 14, which shows calculations performed with the FHI-aims code [18] using ELSI [19] as interface to ELPA and the PBE exchange-correlation functional [20] for periodic Caesium Chloride crystals as function of the number of basis functions used. For this purpose, calculations with different system sizes, i.e., number of atoms, were performed. Since FHI-aims uses local atomic orbitals [18], the number of basis functions increases with the number of atoms: For example, the smallest investigated system contains 16 atoms and thus uses 496 basis functions, while the largest system contains 3,456 atoms and 107,136 basis functions. For all system sizes, we benchmarked ELPA1 and ELPA2 separately; in both cases, CPU only calculations as well as calculations using CPUs and full GPU acceleration (for the tridiagonalization, the solution of the eigenvalue problem, and the back transformation) were performed on four Intel Skylake (Xeon Gold 6138) + nVidia Tesla V100 nodes with two CPUs and GPUs each (20 cores/CPU @ 2.0 GHz).

As Figure 14 shows, the use of GPU acceleration offers a sizeable performance increase for large systems with respect to CPU-only calculations for both ELPA1 and ELPA2, whereby the gains are more pronounced for ELPA1. The threshold number of basis functions for which GPUs indeed accelerate the calculation is essentially determined by the workload on each CPU and GPU. For too small systems, the time spent transferring the data to the GPU is larger than the actual computational gains due to the GPU. In this particular case, GPUs are thus beneficial for ELPA1 for more than 10,000 basis functions and for ELPA2 for more than 20,000 basis functions. Overall, CPU-only ELPA1 is the fastest solver up to 4,000 basis functions, CPU-only ELPA2 for system between 4,000 up to roughly 20,000 basis functions, and CPU+GPU ELPA1 for all systems with even larger number of basis functions. Note that this might be quite surprising even for well-experienced ELPA users, given that ELPA2 is typically superior to ELPA1 for large system sizes in the CPU only case, as also shown in Figure 14. In practice, switching from CPU-only ELPA2 to CPU+GPU ELPA1 can thus lead to significant savings in computational time around 30%, as it is the case for a system size with 107,136 basis functions.

As shown above, optimal performance can only be achieved if different combinations of ELPA1 and ELPA2 with and without GPU acceleration are chosen

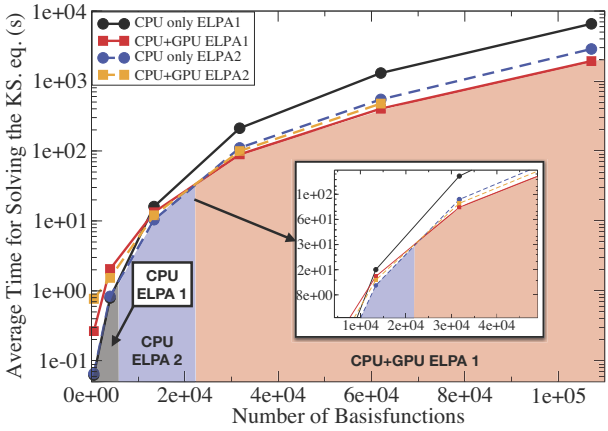


Figure 14. Computational time per SCF step (in seconds) as function of the numbers of basis functions employed. Solid lines denote ELPA1, dashed lines ELPA2 calculations. CPU-only and CPUs+GPU calculations were performed. The shaded areas denote which setup is fastest for different system sizes. The inlet shows the timings for system sizes at which CPU+GPU ELPA1 becomes the fastest solver (border between blue and red marked areas).

depending on the system size. Moreover, the actual threshold at which GPU acceleration becomes beneficial strongly depends on the number of nodes employed in the calculation: For smaller number of nodes, the workload on the individual nodes increases and GPU acceleration becomes beneficial earlier, i.e., for smaller system sizes. Eventually, let us note that for the calculations shown in Figure 14 the GPU acceleration was used for the tridiagonalization, the solution of the eigenvalue problem, and the back transformation. In practice, it can be beneficial to exploit GPUs only for a subset of these steps, as shown below. This particular application thus showcases the importance of ELPA's autotuning functionality, which saves the user from performing tedious benchmark calculations for all different settings and prevents him from choosing sub-optimal settings, e.g., by choosing ELPA2 for large systems based on previous CPU-only experience.

We have explicitly verified this by running calculations with autotuning enabled for two different system sizes with 13,392 and 31,744 basis functions, respectively. As shown in Table 3, the autotuning procedure is able to identify an optimal solution for both cases. In the smaller system with 13,392 basis functions, CPU-only ELPA2 is the optimal solution. Note that in this case the CPU kernel has been fixed to the AVX512-one in all calculations, otherwise also this parameter would have been optimized by the autotuning procedure [3]. For the larger system with 31,744 basis functions, ELPA1 with GPU acceleration is identified as the optimal setup. Compared to the earlier calculations shown in Figure 14, the autotuning procedure found out that it is beneficial to use GPU acceleration only for the tridiagonalization and the back transformation, whereas the solution of the eigenvalue problem is better performed only on the CPUs. The additional gain in computational saving of roughly 1% compared to the next-best solution is not earth-shattering in this case, but still noticeable, given that in actual simulations this 1% can be exploited for thousands if not millions of eigenvalue problems. As already discussed in [3], the cost of the autotuning procedure is well

Table 3. Computational time required for solving the KS equations in seconds for ELPA1 and ELPA2 (CPU-only and CPU+GPU calculations) as well as for the optimal settings found by ELPA's autotuning functionality.

Number of basis functions	ELPA1 CPU-only	ELPA1 CPU+GPU	ELPA2 CPU-only	ELPA2 CPU+GPU	Optimal
13,392	16.03s	13.29s	10.38s	12.05s	10.38s
31,744	211.40s	89.38s	110.90s	99.72s	88.73s

worth the gains in practical calculations. Although some sub-optimal setups such as CPU+GPU ELPA1 and CPU-only ELPA1 are tested during the autotuning for the small and large system, respectively, the benefits outrun these costs in the long term.

5.2. Performance Benefits by Reduced Precision

In DFT simulations, the individual SCF iterations leading up to self-consistency are of no particular interest. Only the final results of the converged SCF cycle have any physical relevance at all. Hence, it is worthwhile to study how a reduction in precision of the SCF procedure from double (DP) to single precision (SP) might accelerate the generalized EVP as the numerical bottleneck of DFT simulations, as long as the final converged result is not altered up to the precision required by the problem at hand. In this section, the precision is independently controlled for the following individual eigensolver steps: the Cholesky decomposition (i), the matrix multiplication in (ii) and (iv), and the solution of the eigenproblem via tridiagonalization (iii) (see Section 3). Since SP in the matrix inversion step $U \rightarrow U^{-1}$ destroys the convergence entirely [3], the inversion of U is always conducted in DP.

To demonstrate the gain in computational performance by both the algorithmic improvements and the readily available SP routines in the new version of ELPA, we have performed DFT calculations with FHI-aims [18] using ELSI [19] as interface to different ELPA versions. The model system chosen for performance comparisons is selected from a class of novel, self-organizing materials, called metal-organic frameworks (MOF). Their electric conductivity can be manipulated and tuned by doping with different metal ions [21]. Due to the low concentration of doping atoms, the theoretical description is challenging and requires the simulation of extensive supercells with a large number of atoms and hence basis functions. Therefore, the iron triazolate MOF doped with a single copper atom [22] is an ideal benchmark system to quantify the speed-up achieved by different precisions by evaluating five SCF cycles and the atomic forces for supercells ranging from 2,405 to 19,343 atoms and 30,167 to 244,529 basis functions, respectively. The calculations were conducted on Intel Xeon 'Skylake' (40 cores @ 2.4 GHz) and compared to the FHI-aims internal ELPA 2013 (only DP available).

As shown in Figure 15, replacing the FHI-aims internal ELPA 2013 by ELPA2018.11 (DP) provides a speed-up of about 1.6 for the solution of the Kohn-Sham eigenvalue problem. For high-level parallelization, where ELPA 2013 does not scale very well, speed-up factors over 2.0 can be achieved. The total computational time is reduced by an average speed-up factor of 1.3, which can go up to 1.7 for large runs. This speed-up comprises all improvements and developments in the

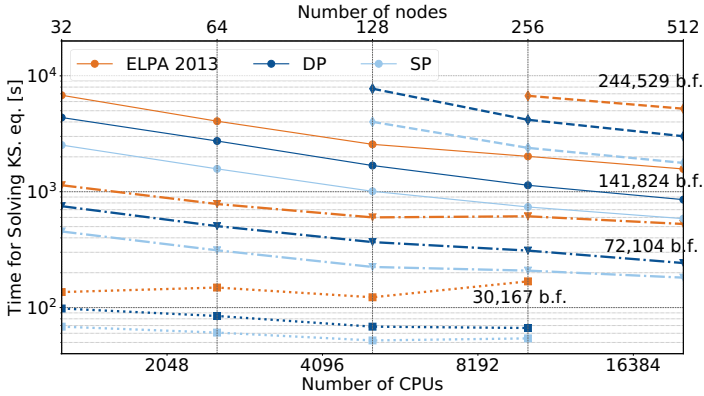


Figure 15. Computational time for solving the Kohn-Sham equation in five SCF iterations (in seconds) as function of the number of CPUs. Calculations conducted with the FHI-aims internal ELPA 2013 are denoted in orange. Application of ELPA2018.11 with DP and SP in steps (i) to (iv) are depicted in dark and light blue, respectively. The different line styles show the timings for different system sizes from 30,167 (dotted) via 72,104 (dash-dotted) and 141,824 (solid) to 244,529 (dashed) basis functions (b.f.).

ELPA library since the FHI-aims internal ELPA 2013 version, such as AVX-512 kernel optimization, autotuning etc. (see Section 2) but without the application of GPUs.

Table 4 summarizes the speed-up factors broken down into the individual steps of the GSEP. The Cholesky decomposition (i) is only conducted in the first SCF iteration of each SCF cycle, i.e., only once in each benchmark calculation. The gain by reduction to SP in the Cholesky step (i) is minimal with a speed-up factor of about 1.1. Whereas for strong-scaling situations with high parallelization ($<$ 20 basis functions / cpu), SP in the Cholesky decomposition can effectively increase the computational time of this step. In contrast, SP in the matrix multiplication of step (ii) and (iv) efficiently reduces the cpu time to 50% of the DP computational time (speed-up factor 2.0). Similarly, SP in the eigensolver (iii) achieves a speed-up of factor 1.9 for the computational time of step (iii). The combination of SP in steps (i), (ii), (iii), and (iv) provides a speed-up of about 1.7 for the solution of the Kohn-Sham eigenvalue problem and of about 1.3 for the total computational time, unless parallelization is high ($<$ 20 basis functions / cpu).

6. Conclusions

We have presented the recent advances in the ELPA eigenvalue solver project. Due to the API changes the autotuning functionality is now available for users. It allows also non-experts to find the best parameter setups for their runs. Especially in the setting of electronic structure theory where many similar eigenvalue problems have to be solved, autotuning is a very powerful instrument. Additional gain in computational time was demonstrated by a mixed-precision approach where certain steps to solve a generalized eigenvalue problem are done in single instead

Table 4. Speed-up factors for SP versus DP (ELPA2018.11) for five SCF iterations and increasing number of basis functions (b.f.) decomposed into each step of the GSEP solver: Cholesky decomposition (i), transformation of GSEP to SEP (ii), solution of the eigenproblem via tridiagonalization (iii), and back-transformation of the eigenvectors (iv). The last two columns summarize the speed-up factors for the computational time required for the solution of the Kohn-Sham equation and for the total computational time.

No. b.f.	(i)	(ii)	(iii)	(iv)	KS	total
30,167	0.9	1.8	1.4	1.7	1.3	1.1
72,104	1.0	1.9	1.7	1.9	1.5	1.2
141,824	1.1	2.0	1.8	2.0	1.6	1.3
244,529	1.2	2.3	2.1	2.2	1.8	1.4

of double precision. The computational kernels and routines have been further optimized and been ported for the newest GPU and CPU Hardware. This allows to accelerate the computation of eigenvalues and eigenvectors and compute even larger matrices. The new algorithmic developments improve the solution of the generalized eigenvalue problems. For the banded and the dense matrix case remarkable savings in computation time have been shown.

References

[1] T. Auckenthaler, V. Blum, H.-J. Bungartz, T. Huckle, R. Johanni, L. Kraemer, B. Lang, H. Lederer, and P. Willems, "Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations," *Parallel Comput.*, vol. 27, no. 12, pp. 783–794, 2011.

[2] A. Marek, T. Auckenthaler, V. Blum, H.-J. Bungartz, H. Ville, R. Johanni, A. Heinecke, B. Lang, and H. Lederer, "Parallel eigenvalue solutions for electronic structure theory and computational science," *J. Phys. Condens. Matter*, vol. 26, no. 21, p. 213201, 2014.

[3] P. Kůs, A. Marek, S. Koecher, H.-H. Kowalski, C. Carbogno, C. Scheurer, K. Reuter, M. Scheffler, and H. Lederer, "Optimizations of the eigensolvers in the ELPA library," *Parallel Comput.*, vol. 85, pp. 167 – 177, 2019.

[4] B. Cook, T. Kurth, J. Deslippe, P. Carrier, N. Hill, and N. Wichmann, "Eigensolver performance comparison on cray xc systems," *Concurr. Comp.-Pract. E.*, vol. 31, no. 16, p. e4997, 2019.

[5] P. Kůs, H. Lederer, and A. Marek, "GPU optimization of large-scale eigenvalue solver," in *Numerical Mathematics and Advanced Applications ENUMATH 2017* (F. A. Radu, K. Kumar, I. Berre, J. M. Nordbotten, and I. S. Pop, eds.), (Cham), pp. 123–131, Springer International Publishing, 2019.

[6] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*. Philadelphia, PA: SIAM, 1997.

[7] J. Poulson, B. Marker, R. A. van de Geijn, J. R. Hammond, and N. A. Romero, "Elemental: A new framework for distributed memory dense matrix computations," *ACM Trans. Math. Software*, vol. 39, pp. 13:1–13:24, Feb. 2013.

[8] J. J. Dongarra, L. Kaufman, and S. Hammarling, "Squeezing the most out of eigenvalue solvers on high-performance computers," *Linear Algebra Appl.*, vol. 77, pp. 113–136, 1986.

[9] J. Choi, J. J. Dongarra, and D. W. Walker, "Parallel matrix transpose algorithms on distributed memory concurrent computers," *Parallel Comput.*, vol. 21, pp. 1387–1405, Sept. 1995.

[10] L. E. Cannon, *A Cellular Computer to Implement the Kalman Filter Algorithm*. PhD thesis, Montana State University, Bozeman, MT, 1969.

[11] H.-J. Lee, J. P. Robertson, and J. A. B. Fortes, "Generalized Cannon's algorithm for parallel matrix multiplication," in *Proc. ICS '97, Intl. Conf. Supercomputing, July 7–11, 1997, Vienna, Austria*, pp. 44–51, ACM Press, 1997.

[12] V. Manin and B. Lang, "Cannon-type triangular matrix multiplication for the reduction of generalized HPD eigenproblems to standard form," 2018. Submitted.

[13] C. R. Crawford, "Reduction of a band-symmetric generalized eigenvalue problem," *Comm. ACM*, vol. 16, pp. 41–44, jan 1973.

[14] B. Lang, "Efficient reduction of banded hermitian positive definite generalized eigenvalue problems to banded standard eigenvalue problems," *SIAM J. Sci. Comput.*, vol. 41, no. 1, pp. C52–C72, 2019.

[15] M. Rippl, B. Lang, and T. Huckle, "Parallel eigenvalue computation for banded generalized eigenvalue problems," *Parallel Comput.*, vol. 88, p. 102542, 2019.

[16] P. Hohenberg and W. Kohn, "Inhomogeneous electron gas," *Phys. Rev.*, vol. 136, no. 3B, pp. B864–B871, 1964.

[17] W. Kohn and L. J. Sham, "Self-consistent equations including exchange and correlation effects," *Phys. Rev.*, vol. 140, no. 4A, pp. A1133–A1138, 1965.

[18] V. Blum, R. Gehrke, F. Hanke, P. Havu, V. Havu, X. Ren, K. Reuter, and M. Scheffler, "Ab initio molecular simulations with numeric atom-centered orbitals," *Comput. Phys. Commun.*, vol. 180, no. 11, pp. 2175 – 2196, 2009.

[19] V. W. Yu, F. Corsetti, A. Garcia, W. P. Huhn, M. Jacquelin, W. Jia, B. Lange, L. Lin, J. Lu, W. Mi, A. Seifitokaldani, Álvaro Vázquez-Mayagoitia, C. Yang, H. Yang, and V. Blum, "ELSI: A unified software interface for Kohn–Sham electronic structure solvers," *Comput. Phys. Commun.*, vol. 222, pp. 267 – 285, 2018.

[20] J. P. Perdew, K. Burke, and M. Ernzerhof, "Generalized gradient approximation made simple," *Phys. Rev. Lett.*, vol. 77, pp. 3865–3868, Oct 1996.

[21] F. Schröder and R. A. Fischer, "Doping of metal-organic frameworks with functional guest molecules and nanoparticles," *Top. Curr. Chem.*, vol. 293, pp. 77–113, 2010.

[22] L. S. Xie, L. Sun, R. Wan, S. S. Park, J. A. DeGayner, C. H. Hendon, and M. Dincă, "Tunable Mixed-Valence doping toward record electrical conductivity in a Three-Dimensional Metal–Organic framework," *J. Am. Chem. Soc.*, vol. 140, no. 24, pp. 7411–7414, 2018.

ParaFPGA 2019. Parallel Computing with FPGAs

This page intentionally left blank

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200097

Parallel Totally Induced Edge Sampling on FPGAs[1]

Akshit GOEL [a], Sanmukh R. KUPPANNAGARI [b,2], Yang YANG [b], Ajitesh SRIVASTAVA [b] and Viktor K. PRASANNA [b]

[a] *Department of Electrical and Electronics Engineering, Birla Institute of Technology and Science Pilani, India 333031*

[b] *Ming Hsieh Department of Electrical and Computer Engineering, University of Southern California, USA 90089*

Abstract. Graphs are a powerful tool for data representation in a wide range of domains like social, biological, informational, etc. But their extremely large sizes often makes it computationally infeasible to study the entire graphs. Graph sampling provides a solution by generating smaller subgraphs which are computationally feasible to analyze and can be used to infer the properties of the entire graph. In this work, we develop a high throughput parallel implementation of Totally Induced Edge Sampling (TIES) algorithm on FPGA. Prior research has shown that TIES performs better than other sampling techniques in terms of preserving the topological properties of the original graph, and thus generates better quality subgraphs. The algorithm randomly samples the edges and inserts the corresponding vertices into the sampled vertex set until the desired number of vertices are sampled. Then, the edges connecting the sampled vertices are included in the sampled subgraph. We use multiple parallel pipelines to achieve high throughput and faster graph sampling. The parallel pipelines need to access a global dynamic data structure which contains the vertices sampled thus far. To support this, we develop a novel dynamic hash table data structure which supports parallel accesses in each clock cycle. We vary the number of pipelines, the size of the sampled subgraph and analyze the performance of the design in terms of on-chip FPGA resource utilization, throughput and total execution time. Our design achieves a throughput as high as 2471 Million Edges Per Second (MEPS) and performs 3.6x better than the state-of-the-art multi-core design.

Keywords. Parallel Graph Sampling, Totally Induced Edge Sampling, FPGA, Dynamic Data Structure

1. Introduction

Graphs are being used to represent data in a wide range of applications including World Wide Web, social media, genomics, and machine learning [17,9]. Graph analysis is a key computational technology in such applications to understand, codify and derive hidden information. However, the large size of real world graphs prevents efficient processing

[1]This work has been supported by Xilinx and by the U.S. National Science Foundation (NSF) under grant OAC-1911229.

[2]Corresponding Author: Sanmukh R. Kuppannagari; E-mail:kuppanna@usc.edu

even with the help of distributed graph databases and highly optimized processing platforms [3,4,5]. Analyzing multiple smaller representative subgraphs is a possible solution as the results of their analyses can be used to infer the properties of the original graph [1,12,16]. In many data repositories, graphs are stored in form of subgraphs to make their analysis computationally feasible [7,6,8]. Graph sampling algorithms are widely used to generate such representative subgraphs from large graphs.

A popular graph sampling algorithm is Totally Induced Edge Sampling (TIES). It randomly samples edges and thus mitigates the downward degree bias of vertex sampling. Total induction over the sampled vertices selects all the edges between them, which further improves the connectivity in the sampled subgraph. According to [2], Induced Edge Sampling performs better than many sophisticated state-of-the-art node sampling and topology based algorithms, such as forest fire or snowball sampling in terms of preserving graph structure and thus has been our choice for hardware implementation in this work. Sequential sampling of very large graphs can be unacceptably slow, and thus we target high throughput parallel implementation of TIES on FPGA. Most significant challenge in a parallel implementation of TIES is to perform fast parallel access and updates to a data structure storing the sampled vertices.

In this work, we focus on sampling a small subgraph (thousands or tens of thousands of vertices) from a large graph. The sampled subgraph can then be used for downstream applications such as GCN [16]. Specifically, given a large graph $G = (V, E)$ in external memory, we need to generate a subgraph $Gs = (Vs, Es)$ and store it on the FPGA on-chip memory. Our design consists of multiple parallel pipelines which randomly select edges from E until a desired number $|Vs| \approx Ns$ distinct vertices in V have been sampled. This is followed by total induction phase that iterates over all the edges corresponding to the sampled vertices in E for induction in Vs. The major contributions of this work are as follows:

1. We develop a high throughput parallel implementation of the Totally Induced Edge Sampling (TIES) algorithm on FPGA. To the best of our knowledge, this is the first parallel FPGA implementation of the algorithm.
2. We design a novel parallel on-chip dynamic hash table data structure that enables multiple pipelines to read and insert data concurrently without stalling.
3. Our design achieves a throughput as high as 2471 Million Edges Per Second (MEPS) using 8 pipelines.

The rest of the paper is organized as follows: Section 2 presents the related work; Section 3 gives a brief overview of the algorithm; Section 4 discusses the architecture design for algorithm implementation and Section 5 reports the experimental results.

2. Related Work

In [2], the authors propose Induced Edge Sampling (or Totally Induced Edge Sampling), and discuss its merits as compared to other sampling techniques. According to [2], TIES offsets the downward degree bias of node sampling. It improves connectivity of the subgraph, and thus better preserves the topological properties of the input graph than many other sampling algorithms. In [7], the authors present a parallel implementation of the TIES algorithm on multi-core platform which samples a graph of size 1442M edges in <

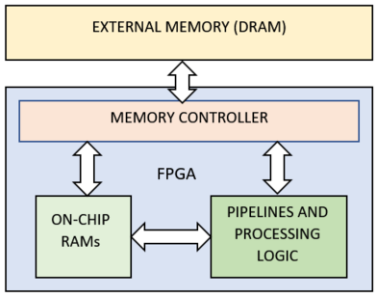


Figure 1. Overall Architecture Model

2.5 seconds. The implementation involves atomic operations to prevent concurrent updates by distinct threads. This synchronization steps introduces inefficiencies. Authors in [11] develop an FPGA implementation of a reduction based sampling algorithm which generates a subgraph by removing nodes and edges. This is highly inefficient for sampling small subgraphs from very large graphs. To the best of our knowledge, the work done in [11] is the only prior FPGA graph sampling implementation.

3. Overview of the TIES Algorithm

A detailed description of the TIES algorithm can be found in [2]. A brief description is as follows: The algorithm takes two inputs: 1) The graph to be sampled i.e. $G = (V, E)$ and 2) The number of vertices to be sampled Ns. The algorithm outputs a subgraph $Gs = (Vs, Es)$ from a graph $G = (V, E)$ such that $|Vs| \approx Ns$.

There are two major steps or phases of the algorithm which are described as following [2]:

1. Phase-1
   (a) Randomly select an edge $e_{i,j} = (i, j)$ from the set E.
   (b) Sample the vertices i and j into set Vs, if not already sampled.
   (c) Repeat (a) and (b) until $|Vs| = Ns$. This completes the set Vs.
2. Phase-2
   For each edge $e_{i,j}$ in set E, include it in set Es if i and $j \in Vs$. This completes the set Es

The sets Vs and Es represent the sampled subgraph $Gs = (Vs, Es)$.

4. Parallel Graph Sampling Architecture on FPGA

4.1. Architecture Model

Fig. 1 shows the high level view of the proposed architecture. The external memory stores the graph to be sampled. The FPGA design consists of multiple parallel pipelines which access the external memory containing the inputs graph through the memory controller.

The graph is sampled by the pipelines and is stored on the on-chip memory. Each pipeline is identical and operates independent of other pipelines.

We assume that each access to the external memory incurs a latency of l clock cycles. The value of l is typically 10-50 clock cycles. Memory latency does not impact our design since all the pipelines operate without stalling i.e. after an initial latency, the data is continuously streamed from the external memory without any delay. The external memory has a peak bandwidth bw. To achieve high throughput we increase the number of pipelines and saturate bw.

Phase-1 implementation consists of p pipelines which sample $\frac{p}{2}$ edges i.e. p vertices in parallel. Phase-2 implementation consists of p pipelines and operate on p vertices simultaneously. The design also consists of a dynamic hash table data structure that supports p inserts/searches in parallel.

Algorithm 1 Phase-1

rand() $\rightarrow$ random number generator; hash() $\rightarrow$ hash function
for $pipe = 0, 1, ..., p-1$ in parallel **do**
  while $|localVs[pipe]| \leq Ns/p$ **do**
    if pipe % 2 = 0 **then**
      $addr \leftarrow rand()$
      $(i, j) \leftarrow edge\_list(addr)$
      $v[pipe] \leftarrow i$
      $v[pipe+1] \leftarrow j$
    end if
    $v[pipe].hash\_value = hash(v[pipe])$
    if $v[pipe] \neq hash\_table[pipe](v[pipe].hash\_value)$ **then**
      $hash\_table[pipe](v[pipe].hash\_value) = v[pipe]$
      $localVs \leftarrow localVs \cup v[pipe]$
    end if
  end while
end for

Algorithm 2 Phase-2

$mv \rightarrow 0$; $n \rightarrow 0$
for $pipe = 0, 1, ..., 2p-1$ in parallel **do**
  while $mv \leq Ns$ **do**
    $i \leftarrow localVs[pipe].(mv)$
    while $n \leq |i.edges|$ **do**
      $j \leftarrow i.edges(n)$
      $j.hash\_value = hash(j)$
      if $j = hash\_table[pipe](j.hash\_value)$ **then**
        $localEs[pipe] \leftarrow localEs[pipe] \cup (i, j)$
      end if
    end while
  end while
end for

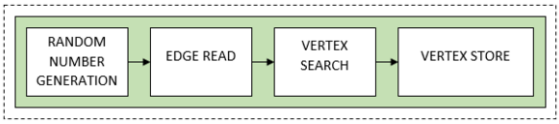


Figure 2. Pipeline Stages: Phase-1

4.2. Pipeline Design: Phase-1

The objective of Phase-1 is to sample N_s unique vertices. Phase-1 consists of p parallel pipelines. The various pipeline stages are shown in Fig. 2. These stages are – Random Number Generation, Edge Read, Vertex Search and Vertex Store. Each pipeline continues to operate in Phase-1 until it samples *Ns/p* unique vertices. Some pipelines may take more iterations than others because edges sampled by them might have common vertices. The various stages of each pipeline are described below:

1. *Random Number Generation*: Since each edge is represented by two vertices, the number of edges required to be sampled to generate p vertices simultaneously are *p/2*. Therefore, in every clock cycle, only *p/2* pipelines generate random numbers which serve as indices to read edges from the external memory. Thus, a total of *p/2* edge indices are given as input to the external memory in every clock cycle.
2. *Edge Read*: The architecture assumes that the graph is stored in the external memory in the form of a simple data structure where each edge is represented by its two end vertices. We assume the latency associated with fetching information from the external memory to be l clock cycles. The memory controller is given an input of *p/2* memory addresses in each clock cycle. Thus after an initial delay of l, the memory outputs *p/2* edges every clock cycle with each edge represented by its two vertices.
3. *Vertex Search*: Each pipeline is associated with an individual hash table and Local Vertex register. The details of the hash table design are discussed in Section 4.4. The vertices fetched by the pipelines in 'Edge Read' stage are searched in the corresponding hash table to check if they have already been sampled or not. Since each pipeline is associated with a separate hash table, therefore all p vertices fetched in previous stage are searched in O(1) time. The search returns whether the vertex is stored in the table or not.
4. *Vertex Store*: In this stage, each pipeline stores the fetched vertex if the vertex search stage returns negative value. The vertex is inserted in both the hash table and Local Vertex register of the corresponding pipeline. Thus, in a given clock cycle, a maximum of p vertices are inserted collectively by all the pipelines. Concatenation of all Local Vertex registers at the end of Phase-1 gives the sampled vertex set *Vs*.

4.3. Pipeline Design: Phase-2

Phase-2 consists of p parallel, identical five stage pipelines as shown in Fig. 3. These stages are – Vertex Read, Edge-Index Generation, Edge Read, Vertex Search, Edge Store. During Phase-2, all the edges either incident to or from the sampled vertices are checked for induction in *Es*. If the number of edges corresponding to their sampled vertices are less, some pipelines might finish Phase-2 before others. Each pipeline has a Lo-

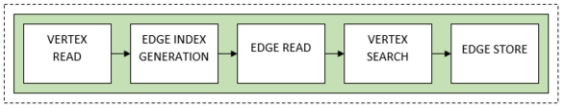


Figure 3. Pipeline Stages: Phase-2

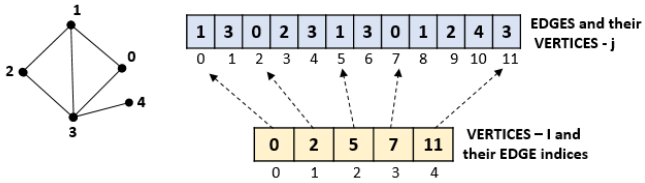


Figure 4. Input Graph Representation: Phase-2

cal Edge register made up of the on-chip memory to store the edges induced by it. The various stages of a pipeline are described below:

1. *Vertex Read*: The architecture assumes that the graph is stored in external memory in form of a CSR matrix as shown in the Fig. 4. To read an edge, we require two indices-one for reading the vertex and the other for reading the edge. Each pipeline reads a sampled vertex from its Local Vertex register and iterates over its edges. For each edge, the corresponding edge index is generated in the *Edge Index Generation* stage. Once all the edges of a vertex are processed, the next sampled vertex in the Local Vertex register is read.
2. *Edge Index Generation*: All the edges associated with a vertex are stored consecutively and a pointer points to the first edge of this set as shown in Fig. 4. Each pipeline inputs the vertex index (generated in previous stage) to the external memory, and fetches the edge index for the first edge of the vertex's edge set. The edge index fetched is incremented by $\log_2 m$ bits (assuming $|E| = m$) in every subsequent clock cycle to read the next edge until the edge index given by the next vertex of CSR matrix is reached. Each pipeline generates an edge index for the external memory and therefore a total of p edge indices are generated per clock cycle.
3. *Edge Read*: Each pipeline inputs the edge index generated in the previous stage to the external memory, and fetches the edge corresponding to it. Each fetched edge is represented by one of its vertex (other one being the vertex selected in Phase-1 itself). We assume the latency associated with fetching information from the external memory to be l clock cycles. Since for each edge we need two external memory accesses, thus after an initial delay of $2*l$ clock cycles, at the end of each clock cycle the memory gives a total of p edges for all pipelines.
4. *Vertex Search*: Each pipeline searches the vertex fetched in the previous stage in its corresponding hash table to check if it belongs to *Vs* or not. Since each pipeline has its separate hash table (Section 4.4), therefore all pipelines operate in parallel. The edges whose vertices belong to *Vs* are included in the subgraph in the next stage.
5. *Edge Store*: Each pipeline consists of a Local Edge register made up of on-chip memory. If in the previous stage, it is indicated that the edge vertex belongs to *Vs*,

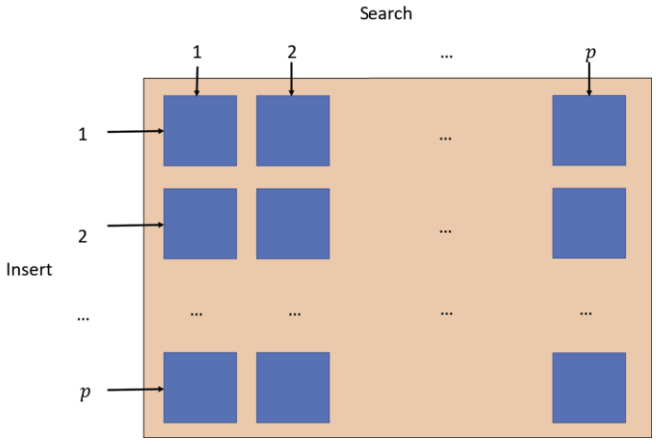


Figure 5. Overall hash table Architecture

then that edge is induced and stored in the corresponding pipeline's Local Edge register in form of its two end vertices. Concatenation of all Local Edge registers at the end of Phase-2 gives the sampled edge set *Es*.

Note that we assume different formats for both the phases. However, these formats can be generated just once in an offline manner and used to generate multiple subgraphs. Therefore, we do not consider format conversion time in our design and its evaluation.

4.4. Hash Table Architecture

In Phase-1, a vertex is inserted into the sampled vertex set only if it has not been already inserted before. Similarly, in Phase-2, an edge is inserted into the sampled edge set if both its vertices are sampled. As our design consists of p pipelines, we require a data structure that performs p parallel lookups of vertices without stalling. Thus, we propose a hash table to store the sampled vertices for quick parallel lookup. The hash table is implemented using on-chip FPGA RAMs (BRAM or URAM [14]). As a BRAM/URAM block supports a single read/write operation per clock cycle, implementing a hash table which supports p queries per cycle is challenging.

To address this challenge, we implement an on-chip hash table that can process p parallel queries in each clock cycle (in a pipelined manner) as shown in Figure 5. The hash table consists of p^2 hash blocks each of size $\frac{N_s}{p}$ vertices. Each hash block is implemented as two level hash tables. For insertion queries, pipeline p_l , $0 \leq l \leq p$ inserts the same value in each hash block along the row l. For search queries, each pipeline p_l reads all the hash blocks along the column l. Thus, the value inserted by a pipeline p_l will be available to a pipeline $p_{l'}$ via hash block ll' i.e. the block at the intersection of lth row and l'th column. Assuming it takes k cycles to insert into the pipeline, our design ensures that a vertex inserted by a pipeline is stored in at least one hash block of all the pipelines with a delay of k cycles.

5. Experiments and Results

5.1. Experimental Setup

The experiments were conducted using the Xilinx Alveo U200 Accelerator Card [13]. The target FPGA device has 1,182,240 LUTs, 591,840 LUTRAMs, 2,364,480 Flip-flops, and 35MB of on-chip SRAMs. We assume the external memory in the form of one DDR4 SDRAM chip, which has a peak data transfer rate of 19.2 GB/s. We synthesize, place-and-route, and simulate our designs using Xilinx Vivado Design Suite 2018.2 [15]. We evaluate the performance of our design using the the soclj dataset [10] which is a LiveJournal friendship social network. The dataset has 4.85 Million vertices and 68.46 Million edges. The vertices have an average degree of 14.1.

5.2. Evaluation Methodology and Performance Parameters

We analyze the performance of our design by varying the number of pipelines from 2 to 8, and varying the number of vertices to be sampled from 1,000 to 20,000. We target our design at 300 MHz FPGA frequency. Each design case is evaluated on the following performance parameters:

1. *Execution time*: The sum of execution times of Phase-1 and Phase-2, i.e. the total time taken to generate the subgraph from the input graph.
2. *Throughput*: Throughput of the design in terms of million edges sampled per second (MEPS). MEPS is calculated by dividing the number of edges in the sampled subgraph with total execution time.
3. *Resource Consumption*:Utilization of FPGA resources in terms of percent usage of LUTs, flip-flops, and on-chip RAMs (BRAM and URAM).

5.2.1. Results

Resource utilization of the design is reported in Figure 6. Figure 6 (a) shows the resource utilization by fixing the number of sampled vertices at 10,000, and changing the number of pipelines from 2 to 8. The Vivado tool chooses to use URAM as the default resource for large on-chip storage, however, in the event this leads to failure in meeting the target clock frequency, it falls back to using BRAM. We observe this behaviour for the cases of number of pipelines equal to 2 and 8. For the case of 2 pipelines, the large size of each hash block leads to this behaviour, while for the case of 8 pipelines the interconnection complexity of connecting each pipeline with all the hash blocks in its row/column leads to this behaviour. Therefore, the BRAM utilization is relatively higher in these two design points. Figure 6 (b) shows the resource utilization by fixing the number of pipelines to 8 and varying the size of sampled subgraph to 1000, 5000, 10000, and 50000 vertices. The URAM utilization increases almost linearly with the subgraph size, because the edge registers are mapped to URAM. We report the execution time and throughput performance in Table 1. The results clearly verify the scalability of our design with the number of pipelines, with each case supporting a maximum clock frequency of 300 MHz.

Table 1. Execution Time and Throughput

# Pipelines	Execution Time (μs)	Throughput (MEPS)
2	457.24	614.05
4	227.16	1236.01
6	151.49	1853.40
8	113.62	2471.14

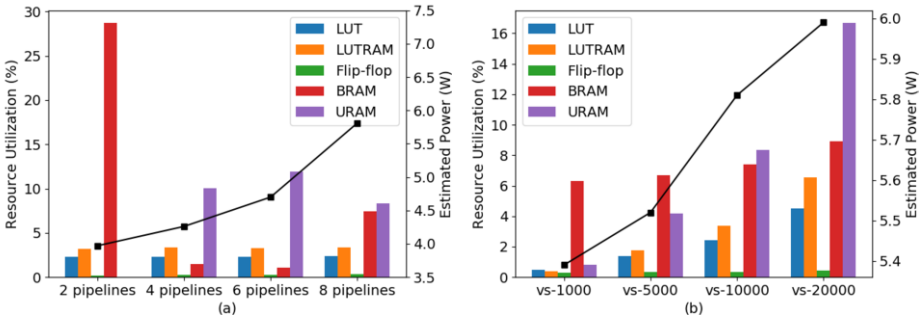


Figure 6. Resource Utilization and Estimated Power

5.3. Comparison with State-of-the-art Multi-core Design

We compare the performance of our 8 pipelines design with the highly optimized multi-core design of the same algorithm. In [7], the authors implement their design on a 16-core machine with 2 x 2.6GHz 8-core Intel Xeon E5-2650 processors (256KB L2 and 20MB L3 cache) and 128GB main memory with 2-way hyperthreading. We compare the performance in terms of throughput i.e. Million edges sampled per second (MEPS). As shown in Table 2, our design achieves a 3.63x improvement over the multi-core implementation. The majority of the speedup comes from the efficient and effective architecture of our design, as cost of the global synchronization, which is required on a multi-core design, is significantly reduced by the dynamic hash table. It is important to note that the design in [7] is tailored for sampling large size subgraphs (hundreds of thousands of vertices) unlike our design which is tailored for small size subgraphs.

Table 2. Comparison with multi-core design

Parameter	Multi-core Design	This Design (pipelines = 8)	Improvement
Throughput (MEPS)	680.87	2471.14	3.63

6. Conclusion

In this work, we presented an FPGA accelerated Totally Induced Edge Sampling (TIES) algorithm. We proposed a novel parallel hash table data structure that supports multiple

concurrent read and write requests. Our design achieves a high throughput of 2471 MEPS which is 3.6x times better than the state-of-the-art design. Our design consists of multiple pipelines operating in parallel without stalling. The design is scalable with number of pipelines and number of sampled vertices with a sustained clock frequency of 300 MHz.

References

[1] Nesreen K Ahmed, Nick Duffield, Jennifer Neville, and Ramana Kompella. Graph sample and hold: A framework for big-graph analytics. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1446–1455. ACM, 2014.

[2] Nesreen K Ahmed, Jennifer Neville, and Ramana Kompella. Network sampling: From static to streaming graphs. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 8(2):7, 2014.

[3] Vito Giovanni Castellana, Alessandro Morari, Jesse Weaver, Antonino Tumeo, David Haglin, Oreste Villa, and John Feo. In-memory graph databases for web-scale data. *Computer*, 48(3):24–35, 2015.

[4] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 17–30, 2012.

[5] Minyang Han and Khuzaima Daudjee. Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. *Proceedings of the VLDB Endowment*, 8(9):950–961, 2015.

[6] Jérôme Kunegis. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 1343–1350. ACM, 2013.

[7] Kartik Lakhotia, Rajgopal Kannan, Aditya Gaur, Ajitesh Srivastava, and Viktor Prasanna. Parallel edge-based sampling for static and dynamic graphs. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*, pages 125–134. ACM, 2019.

[8] Jure Leskovec and Andrej Krevl. {SNAP Datasets}:{Stanford} large network dataset collection. 2015.

[9] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[10] Stanford. Stanford large network datasets. https://snap.stanford.edu/data/socnets.

[11] Usman Tariq, Umer I Cheema, and Fahad Saeed. Power-efficient and highly scalable parallel graph sampling using fpgas. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6. IEEE, 2017.

[12] Bin Wu, Ke Yi, and Zhenguo Li. Counting triangles in large graphs by random sampling. *IEEE Transactions on Knowledge and Data Engineering*, 28(8):2013–2026, 2016.

[13] Xilinx. Alveo u200 data center accelerator card. https://www.xilinx.com/products/boards-and-kits/alveo/u200.html.

[14] Xilinx. Ultraram. https://www.xilinx.com/support/documentation/white_papers/wp477-ultraram.pdf.

[15] Xilinx. Vivado design suite. https://www.xilinx.com/products/design-tools/vivado.html.

[16] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. Accurate, efficient and scalable graph embedding. *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2019)*, 2019.

[17] Shijie Zhou, Rajgopal Kannan, Hanqing Zeng, and Viktor K Prasanna. An fpga framework for edge-centric graph processing. In *Proceedings of the 15th ACM International Conference on Computing Frontiers*, pages 69–77. ACM, 2018.

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200098

An Implementation of Non-Local Means Algorithm on FPGA

Hayato KOIZUMI and Tsutomu MARUYAMA
Systems and Information Engineering, University of Tsukuba, 1-1-1 Tennoudai, Tsukuba, 305-8573, Japan
Email: maruyama@darwin.esys.tsukuba.ac.jp

Abstract. Non-Local means (NL-means) algorithm is a robust image denoising algorithm. Its computational complexity is, however, higher than other algorithms, and its availability is limited. In this paper, we propose an implementation method of the NL-means algorithm on FPGA. In the NL-means, the cross correlations between the small windows are repeatedly calculated, and a large number of intermediate data have to be held temporarily to reduce the amount of its computation. In our approach, the scan direction of the image is changed in the zigzag way. This zigzag scan increases the computation time because of the recalculation on the scan borders, but the required memory size can be drastically reduced. We have implemented the circuit on a Xilinx FPGA, and showed that with a small size FPGA, its real-time processing is possible.

1. Introduction

Noise reduction is the process of removing noise from an image. Non-Local means (NL-means) algorithm is one of the powerful and robust noise reduction algorithms[1]. A higher noise reduction ratio can be expected than Gaussian filter, Bilateral filter and so on, and it is supported in a major library[2]. However, its computational complexity is much higher than other denoising algorithms.

In NL-means algorithm, a search window is defined centered at the target pixel, and for each pixel in the search window, a template window is considered. Then, using the template windows, the cross-correlations between the target pixel and all pixels in the search window are calculated. These cross-correlations are used to improve the value of the target pixel. These cross-correlations can be efficiently calculated based on the calculation method of the box filter, and high performance can be easily achieved on FPGAs. However, for this efficient calculation method, large size memory is required, which means a large FPGA with large on-chip memory is required, though only a small amount of its logic cells are used.

We have proposed a memory efficient computation method of box filters[3]. This work demonstrated that the cross-correlation of the windows in two images (left and right images in the stereo vision) can be efficiently calculated with much less memory by changing the scan direction. In this approach, the image is scanned in zigzag, not from top-left to bottom-right. In [3], the processing speed was almost half of the top-left to bottom-right scan (it can be controlled by changing the required memory size), but it was still fast enough for real-time processing.

In this paper, we show that this approach works well for the calculation of the cross-correlations in one image using NL-means algorithm as an example. In the NL-means algorithm, the cross-correlations of the pixels in the square search window are calculated, though in the stereo vision, those of the pixels on a line segment along the x axis are calculated. This difference requires more line buffers, and more memory to store the temporary data. However, as shown in this paper, our approach works well also in this case, and the required memory can be reduced to 14% when the processing speed is half of the top-left to bottom-right scan.

2. Non-Local Means Algorithm

In the Non-Local means algorithm, given an image I, the denoised image I^{dn} is given as follow.

$$p^{dn} = \sum_{q \in W_S(p)} w(p,q) \cdot q \tag{1}$$

Here, p is a pixel in the image I, and p^{dn} is the denoised pixel of p. $W_S(p)$ is a window centered at p, called a search window of p, and q is a pixel in $W_S(p)$.

Let $W_T(p)$ be a window centered at p, called a template window of p. Then, $d(p,q)$, the difference between the two template windows, $W_T(p)$ and $W_T(q)$, is defined as follows.

$$d(p,q) = ||W_T(p) - W_T(q)||^2 \tag{2}$$

Using $d(p,q)$, $w(p,q)$ is given as

$$w(p,q) = \frac{1}{N(p)} \exp(-\frac{d(p,q)}{\sigma^2}) \tag{3}$$

where σ is a constant, and

$$N(p) = \sum_{r \in W_S(p)} \exp(-\frac{d(p,r)}{\sigma^2}) \tag{4}$$

$N(p)$ is used to normalize $w(p,q)$.

Let $p = I_{(x,y)}$, $q = I_{(u,v)}$, the size of the search window $(2w_s+1)^2$, and that of the template window $(2w_t+1)^2$. Then, equation (2) can be rewritten as follows.

$$d(I_{(x,y)}, I_{(u,v)}) = \sum_{dx=-w_t}^{w_t} \sum_{dy=-w_t}^{w_t} ||I_{(x+dx,y+dy)} - I_{(u+dx,v+dy)}||^2 \tag{5}$$

And, equation (1) can also be rewritten as

$$I^{dn}_{(x,y)} = \sum_{dx=-w_s}^{w_s} \sum_{dy=-w_s}^{w_s} w(I_{(x,y)}, I_{(x+dx,y+dy)}) \cdot I_{(x+dx,y+dy)} \tag{6}$$

Fig.1 shows the relation of the search window and the template window. In Fig.1, p, the target pixel, is the center of the search window, and for each pixel q in the search window, $d(p,q)$ is calculated using the pixels in their template windows.

The computational complexity of non-local means algorithm is given as follows.

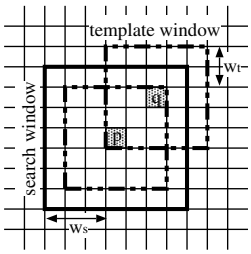


Figure 1. Search window and Template window

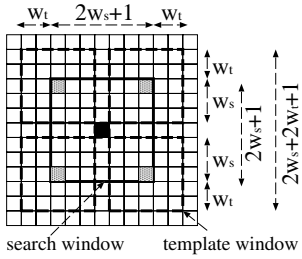


Figure 2. The Range of the Pixels Required for the Calculation

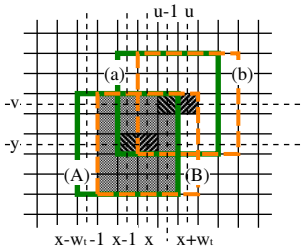


Figure 3. Overlapping of Template windows

1. The number of the pixels in the image is $W \times H$, where W and H are the width and height of the image.
2. For each pixel in the image, equation (6) is calculated. This means that equation (5) is calculated $(2w_s+1)^2$ times.
3. In each calculation of equation (5), $(2w_t+1)^2$ distances are calculated.

Thus, the computational complexity of this algorithm becomes $W \times H \times (2w_s+1)^2 \times (2w_t+1)^2$, which becomes very large for high resolution images.

3. Scan direction and the Performance

In this section, we compare the processing speed and the required memory size by the top-left to bottom-right scan and the zigzag scan[3]. Fig.2 shows how many pixels are necessary for calculating p^{dn} for pixel p. In Fig.2, the black pixel is the target pixel p, and the four gray pixels are the corner pixels of the search window. For calculating $d()$ for all pixels in the search window, $(2w_s+2w_t+1)^2$ pixels are required.

To simplify the discussion, we consider the calculation of only one $d(I_{(x,y)}, I_{(u,v)})$ though it is calculated for $(2w_s+1)^2$ pixels in the search window in the NL-means algorithm.

3.1. Top-left to bottom-right Scan

First, we describe an efficient calculation method of the NL-means algorithm when the image is scanned from top-left to bottom-right. This calculation method is widely used as the one for the box filter.

Suppose that $I^{dn}_{(x-1,y)}$ was calculated, and now $I^{dn}_{(x,y)}$ is going to be calculated. Here, we focus on the calculation of $d(I_{(x,y)}, I_{(u,v)})$ shown in Fig.3. In the calculation of $I^{dn}_{(x-1,y)}$, $d(I_{(x-1,y)}, I_{(u-1,v)})$ was calculated (Fig.3 (A) and (a)), and $d(I_{(x,y)}, I_{(u,v)})$ is going to be calculated for $I^{dn}_{(x,y)}$ (Fig.3 (B) and (b)). The differences of the gray pixels in (A) and (B) in Fig.3 can be shared in these calculations. Therefore, $d(I_{(x,y)}, I_{(u,v)})$ can be calculated from $d(I_{(x-1,y)}, I_{(u-1,v)})$ as follows.

$$\begin{aligned} d(I_{(x,y)}, I_{(u,v)}) =& d(I_{(x-1,y)}, I_{(u-1,v)}) + \\ & d_Y(I_{(x+w_t,y)}, I_{(u+w_t,v)}) - d_Y(I_{(x-w_t-1,y)}, I_{(u-w_t-1,v)}) \end{aligned} \quad (7)$$

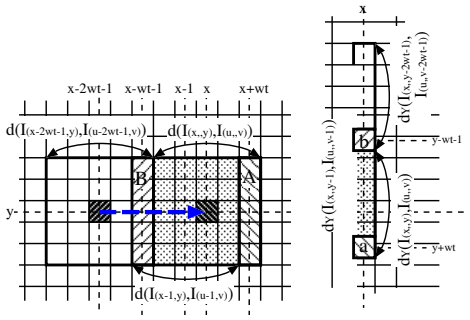


Figure 4. Calculation Method based on Box Filter

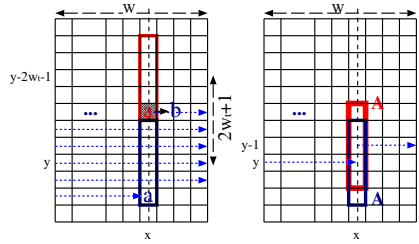


Figure 5. Memory usage in Top-left to Bottom-right Scan

where

$$d_Y(I_{(x,y)},I_{(u,v)}) = \sum_{dy=-w_t}^{w_t} ||I_{(x,y+dy)} - I_{(u,v+dy)}||^2 \tag{8}$$

Fig.4 illustrates this equation. By adding column A to $d(I_{(x-1,y)},I_{(u-1,v)})$, and subtracting column B, $d(I_{(x,y)},I_{(u,v)})$ can be obtained as shown in Fig.4-left. Column B was already calculated as column A of $d(I_{(x-2w_t-1,y)},I_{(u-2w_t-1,v)})$, and by keeping it for $2w_t+1$ steps (blue arrow in Fig.4-left, because the image is scanned from top-left to bottom-right), it can be reused as column B.

In the same way, $d_Y(I_{(x,y)},I_{(u,v)})$ can be calculated using the difference

$$\begin{aligned} d_Y(I_{(x,y)},I_{(u,v)}) =& d_Y(I_{(x,y-1)},I_{(u,v-1)}) + \\ & ||I_{(x,y+w_t)} - I_{(u,v+w_t)}||^2 - ||I_{(x,y-w_t-1)} - I_{(u,v-w_t-1)}||^2 \end{aligned} \tag{9}$$

Fig.4-right illustrates this calculation. By adding

$$a = ||I_{(x,y+w_t)} - I_{(u,v+w_t)}||^2,$$

to $d_Y(I_{(x,y-1)},I_{(u,v-1)})$, and subtracting

$$b = ||I_{(x,y-w_t-1)} - I_{(u,v-w_t-1)}||^2$$

$d_Y(I_{(x,y)},I_{(u,v)})$ can be obtained. Here, b was already calculated as a of $d_Y(I_{(x,y-2w_t-1)}, I_{(u,v-2w_t-1)})$, and by keeping it for $(2w_t+1)\times W$ steps, it is can be reused as b as shown in Fig.5-left. In Fig.5-left, a (the red one) was calculated for $d_Y(I_{(x,y-2w_t-1)},I_{(u,v-2w_t-1)})$, and by waiting $(2w_t+1)\times W$ steps, the focused pixel comes to $(x,y+w_t)$ by the top-left to bottom-right scan (the blue dotted arrows), and the a can be reused as b of $d(I_{(x,y)},I_{(u,v)})$. In this calculation method, $d_Y(I_{(x,y-1)},I_{(u,v-1)})$ (A in Fig.4) has to be also kept for W steps as shown in Fig.5-right.

With this calculation method, $d(I_{(x,y)},I_{(u,v)})$ can be obtained by calculating only a in Fig.4-right, if we can keep the following three values in the memory:

1. $d_Y(I_{(x+w_t,y)},I_{(u+w_t,v)})$ for $2w_t+1$ steps.
2. $b = ||I_{(x,y-2w_t-1)} - I_{(u,v-2w_t-1)}||^2$ for $(2w_t+1)\times W$ steps, and

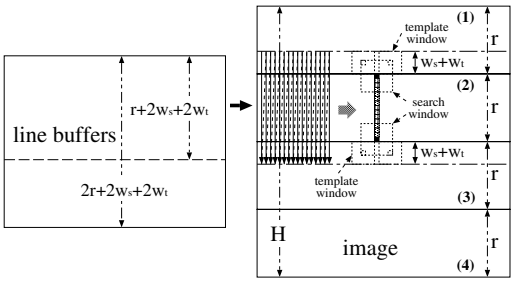


Figure 6. Zigzag scan

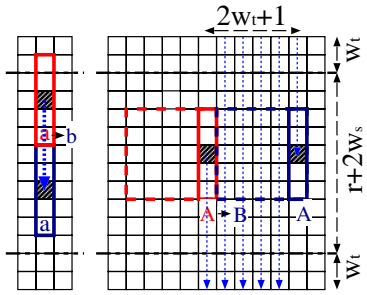


Figure 7. Memory usage in Zigzag scan

3. $d_Y(I_{(x,y-1)}, I_{(u,v-1)})$ for W steps.

In case of NL-means algorithm, these three values have to be kept for all pixels in the search window. Here, we ignore the first one, because its size is much smaller than the last two. The required memory size to keep these values becomes

$$(2w_s+1)^2 \times ((2w_t+1)+1) \times W. \tag{10}$$

In addition to this, $2w_s$ line buffers are necessary to supply $(2w_s+1)^2$ pixels in parallel. The size of these memory is proportional to the image width W, and larger search window (w_s) and template window (w_t) are required for higher resolution images. Therefore, this approach is not feasible for high resolution images, though it gives the minimum computational cost.

This computation method is widely used in many FPGA implementations, and has achieved very high performance, but it is difficult to process high resolution images even with the current largest FPGAs.

3.2. Zigzag scan

Fig.6 shows the outline of our zigzag scan. As shown in Fig.6, the image is divided vertically into blocks ((1),(2),(3) and (4) in Fig.6), the height of each is r. These blocks are processed sequentially from top to bottom. Each block is scanned in zigzag as shown in Fig.6 (in Fig.6, block (2) is being processed). The height of each block is r, but the scan width is $r+2w_s+2w_t$. This is to include all pixels that are necessary for the calculation of all template windows of the top and bottom pixels in the block (two black pixels in Fig.6).

This approach has one advantage, and two disadvantages. First, we describe the two disadvantages. As shown in Fig.6, the scan width for each block is $r+2w_s+2w_t$, but by this scan, $d()$ for only r pixels can be calculated. Thus, the computational efficiency becomes

$$\frac{r}{r+2w_s+2w_t} \tag{11}$$

which is apparently less than 1. Another disadvantage is the large memory size required for the line buffers as shown in Fig.6. To allow the zigzag scan of width $= r+2w_s+2w_t$, $r+2w_s+2w_t$ line buffers are required. In addition to this, r line buffers are necessary

to buffer the data that are necessary for the next block while processing the pixels in the current block. The total number of line buffers becomes $2r+2w_s+2w_t$.

The advantage of our approach is less memory size to hold the temporary data. In this scan method, $d_Y(I_{(x,y)}, I_{(u,v)})$ can also be calculated as shown in equation (9), but in this case, the pixels are scanned vertically, and b in Fig.4-right can be obtained by holding a only for $2w_t+1$ steps as shown in Fig.7-left, though it has to be held for $(2w_t+1)\times W$ steps in case of the top-left to bottom-right scan. To the contrary, for the computation of equation (7) shown in Fig.4-left, A has to be held for $(r+2w_s+2w_t)\times(2w_t+1)$ steps as shown in Fig.7-right, though it has to be held only $2w_t+1$ steps in case of the top-left to bottom-right scan. The main memory usage in this zigzag scan is given by (ignoring the ones used for $d_Y(I_{(x,y)}, I_{(u,v)})$)

$$(2w_s+1)^2\times(r+2w_s+2w_t)\times(2w_t+1). \tag{12}$$

This size is not proportional to W, which means that we can control the memory size by changing r, though the processing speed is also changed.

3.3. Comparison of the Processing Speed and Memory Size

The computational efficiently of the zigzag scan is given by equation (11). Its value can be controlled by changing r, though it also affect the required memory size. However, it is easy to keep this value larger than 0.5.

The ratio of the required memory size is given as follows.

$$\frac{(2w_s+1)^2\cdot(r+2w_s+2w_t)\cdot(2w_t+1)+(2r+2w_s+2w_s)\cdot W}{(2w_s+1)^2\cdot((2w_t+1)+1)\cdot W+2w_s\cdot W}$$

The numerator is the one for the zigzag scan; equation (12) and the line buffers, and denominator is the one for the top-left to bottom-right scan shown in equation (10) and its line buffers.

In our current implementation, $W=640$, $w_t=1$, $w_s=3$, and $r=8$. With these values of the parameters, the processing speed of our approach is 0.5 of the top-left to bottom-right scan, and the memory size ratio is 0.14. For larger W, consequently larger w_s and w_t, the processing speed is kept constant, but the memory size ratio becomes smaller, if r is $c\times w_s$ (c is a constant).

Table 1 compares the required memory size and the processing speed of the zigzag scan to those of the top-left to bottom-right scan when r is changed under the parameters given above. As shown in this table, by changing r, the required memory size can be changed in wide range. This makes it possible to choose the minimum size FPGA for the required processing speed, and also to achieve the maximum performance on the given FPGA by choosing proper r. To the contrary, in the top-left to bottom-right scan, the memory size cannot be reduced for any processing speed.

In both scan methods, by processing n pixels sequentially on the same unit, the logic cells can be reduced to $1/n$. With this sequential approach, the processing speed is also decreased to $1/n$, but the memory size cannot be reduced in both scan methods. However, in the zigzag scan, the required memory size is much smaller, and distributed RAMs can also be used, though only block RAMs can be used in the top-left to bottom-right scan because the memory depth must be W. This flexibility enables to achieve better performance on wide range of FPGAs.

Table 1. Memory usage and computational efficiency

r	4	8	16	32	64
memory size	0.09	0.14	0.23	0.40	0.75
processing speed	0.33	0.50	0.67	0.80	0.80

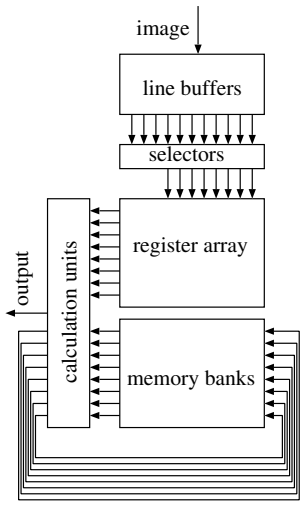


Figure 8. A Block Diagram

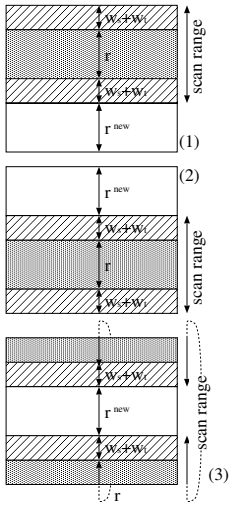


Figure 9. Line Buffers

4. An Implementation on FPGA

In the NL-means algorithm, equation (5) is calculated for all $(w_s+1)^2$ pixels in the search window. The processing speed can be controlled by changing the number of pixels that are processed in parallel. By processing all pixels in parallel, the maximum performance can be achieved. In our implementation, w_s+1 pixels are processed in parallel, and w_s+1 clock cycles are used to process $(w_s+1)^2$ pixels. This approach is taken to reduce the circuit size as much as possible while keeping the processing of 640×480 pixel images faster than 30 fps.

4.1. Block Diagram

Fig.8 shows a block diagram of our system. First, the data are sent from the host computer, and they are once stored in the line buffers for the zigzag scan. Then, the data in the line buffers are read onto the register array through selectors. The equations described in Section 3 are calculated using the values on the register array, and those from the memory banks. In the memory banks, the values discussed in Section 3.2 are stored.

4.2. Line buffers and Register Array

Fig.9 shows the usage of the line buffers in our approach. In Fig.9(1), $w_s+w_t+r+w_s+t+w_t$ line buffers from the top are used for the current zigzag scan, and for the pixels in r lines, NL-means algorithm is applied. This phase takes more than $W\times r$ clock cycles

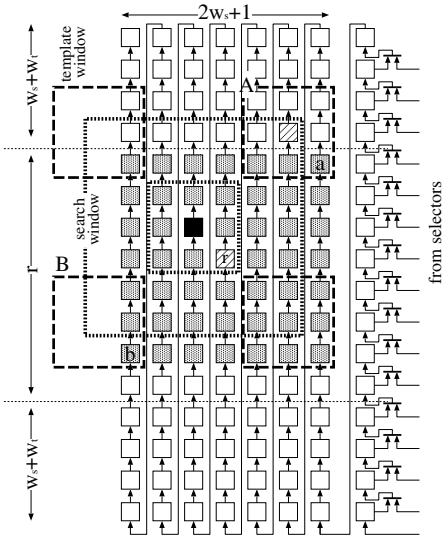


Figure 10. Register Array

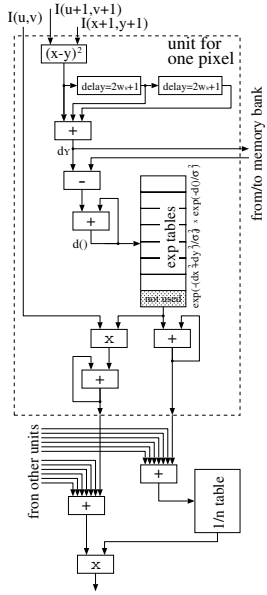


Figure 11. Calculation Units

even if all pixels in the search window are calculated in parallel. During this period, the pixels of the next r lines are read into the line buffers r^{new} for the next zigzag scan. Then, the next $w_s + w_t + r + w_s + t + w_t$ line buffers are used for the next zigzag scan as shown in Fig.9(2), and the next r lines are read into the next line buffers r^{new} . In our current implementation, $r = 8$, $w_s = 3$ and $w_t = 1$. Therefore, as shown in Fig.9, by repeating three phases ((1),(2) and (3)), all lines in the image can be processed.

Fig.10 shows the register array. The data in the line buffers are fetched onto the register array through selectors. As described above, only three phases are necessary for the assignment of the line buffers, which means that only 3-to-1 selectors are required. The register array consists of $(r + 2w_t + 2w_s) \times (2w_t + 1)$ registers. In Fig.10, the black pixel is the center of the search window. Then, the pixel r, the bottom-right corner pixel of its template window, is compared with $(2w_s + 1)^2$ pixels, the gray ones in the figure. As shown in Fig.10, only the values on gray registers are used for the calculations, and other registers are used only for keeping the data. The registers in the right-most column are used to get the data from the line buffers, and to give them to the register array.

4.3. Memory Banks

As described in Section 3.2, two kinds of memory banks are required. The first one is to store a in Fig.7, and the another is to store A. The required depth for the first one is $2w_t + 1$. In our current implementation, $w_t = 1$. Thus, a is held on the registers not in the memory.

The required memory size for the second one is $(2w_s + 1)^2 \times (r + 2w_s + 2w_t) \times (2w_t + 1)$. As described above, in our current implementation, $(2w_s + 1)^2$ pixels in the search window are processed in $2w_s + 1$ clock cycles by processing $2w_s + 1$ pixels in parallel. Thus, the required number of memory banks is $2w_s + 1$. In this case, for each bank, the temporary data for $2w_s + 1$ pixel can be stored. This means that the depth of

the memory banks is $(2w_s+1)\times(r+2w_s+2w_t)\times(2w_t+1)$. This depth becomes 336 under the current parameters. Block RAMs configured as 512×72 can be used as this memory.

4.4. Calculation Units

In our implementation, $2w_s+1$ units are used, and each unit processes $2w_s+1$ pixels sequentially using $2w_s+1$ clock cycles. Fig.11 shows a block diagram of one unit for processing one of $2w_s+1$ pixels. First, $I_{(x+1,y+1)}$ and $I_{(u+1,v+1)}$ are given ($w_t=1$), and their distance is calculated. In Fig.11, the circuit only for one channel is shown, though three channels for R, G and B are processed. Then, it is delayed for $2w_s+1$ clock cycles twice (distributed RAMs are used), and three values are added to calculate d_Y. In the discussion in Section 3, d_Y is calculated using the difference as shown in equation (9), but in the current implementation, $2w_t+1=3$, and the three values are directly added. It is sent to the memory banks (as A in Fig.7), and another one is read back from the memory banks (B in Fig.7). Their difference is added to the register that holds $d()$ to obtain its new value following equation (7). Then, one of the tables that hold $exp(-\frac{dx^2+dy^2}{\sigma_d^2})\cdot exp(-\frac{d()}{\sigma^2})$ is looked up using $d()$ (σ_d is a constant). The first term is a weight considering the distance from the center of the search window. This weight is not shown in equitation (3), but used in our current implementation. $7=2w_s+1$ tables are packed into one block RAM, and one of them is accessed according to the distance from the center pixel. The size of each table is 256, which is large enough to obtain our target PSNR. The output, and the product of the output and $I_{(u,v)}$ are accumulated respectively. Then, those values from $2w_s+1$ units are added. The reciprocal of the sum of the output is obtained by table look-up, and the final output obtained by multiplying them.

5. Experimental Results

We have implemented the circuit on Xilinx FPGA Kintex-7 XC7K160T. For this implementation, 24.6K LUTs (24.3%), 63 block RAMs (19.4%) and 87 DSP slices (14.5%) were used. The size of this circuit is small enough. Its operational frequency is 335.4MHz, and its processing speed is 78.0 fps for 640×480 pixel images. This processing speed is 408X of the software on Core i7-860 2.8GHz.

Fig.12 shows the input image, the image in which noise is added, the output by the original NL-means algorithm, and that by our system. The PSNR by the original algorithm is 31.7dB, and that by our system is 28.0 dB. The PSNR by ours is a bit worse, but it is higher than 25dB, and as reported in [6][7], visually, it is difficult to find the difference. The operation data width is reduced to keep the PSNR higher than 25.0dB, not to achieve the PSNR of the original algorithm. This is to reduce the circuit size as much as possible while keeping the enough quality for human recognition.

6. Conclusions

We have implemented a circuit for Non-Local means algorithm on FPGA. To reduce the memory size, the image is scanned in zigzag. With this scan method, the memory size can be reduced to 14% of the top-left to bottom-right scan. Its processing speed becomes

Figure 12. Comparison of the output images

half of the top-left to bottom-right scan, but it is still 78 fps for 640×480pixel images, which is fast enough for real-time processing.

The design based on this zigzag scan requires more effort than that for the top-left to bottom-right scan. To develop a library to make it easier is one of main future work.

Acknowledgment

This work was supported by JSPS KAKENHI Grant Number 18K11209, and the New Energy and Industrial Technology Development Organization (NEDO).

References

[1] Buades, Antoni, Bartomeu Coll, and J-M. Morel, A non-local algorithm for image denoising, CVPR, 2005.
[2] https://docs.opencv.org/3.2.0/d5/d69/tutorial_py_non_local_means.html.
[3] W. Sichao, and T. Maruyama, An implementation method of the box filter on FPGA, FPL, 2016.
[4] Jin Wang, Yanwen Guo, Yiting Ying, Yanli Liu and Qunsheng Peng, Fast non-local algorithm for image denoising, International Conference on Image Processing. 2006.
[5] L. L. Gambarra, J. C. Pessoa et. al., Fast non-local image denoising using a hardware implementation, Workshop on Circuits and System Design, 2012.
[6] Thomos, Nikolaos, Nikolaos V. Boulgouris, and Michael G. Strintzis, Optimized transmission of JPEG2000 streams over wireless channels, IEEE Transactions on image processing 15.1, 54-67. 2006.
[7] Li, Xiangjun, and Jianfei Cai, Robust transmission of JPEG2000 encoded images over packet loss channels, IEEE International Conference on Multimedia and Expo, 2007.

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200099

Accelerating Binarized Convolutional Neural Networks with Dynamic Partial Reconfiguration on Disaggregated FPGAs

Panagiotis SKRIMPONIS [a], Emmanouil PISSADAKIS [b], Nikolaos ALACHIOTIS [b,c] and Dionisios PNEVMATIKATOS [b,c]

[a] *NYU Tandon School of Engineering, Brooklyn, NY, USA*
[b] *Technical University of Crete, Chania, Greece*
[c] *FORTH-ICS, Greece*

Abstract.
Convolutional Neural Networks (CNNs) currently dominate the fields of artificial intelligence and machine learning due to their high accuracy. However, their computational and memory needs intensify with the complexity of the problems they are deployed to address, frequently requiring highly parallel and/or accelerated solutions. Recent advances in machine learning showcased the potential of CNNs with reduced precision, by relying on binarized weights and activations, thereby leading to Binarized Neural Networks (BNNs). Due to the embarassingly parallel and discrete arithmetic nature of the required operations, BNNs fit well to FPGA technology, thus allowing to considerably scale up problem complexity. However, the fixed amount of resources per chip introduces an upper bound on the dimensions of the problems that FPGA-accelerated BNNs can solve. To this end, we explore the potential of remote FPGAs operating in tandem within a disaggregated computing environment to accelerate BNN computations, and exploit dynamic partial reconfiguration (DPR) to boost aggregate system performance. We find that DPR alone boosts throughput performance of a fixed set of BNN accelerators deployed on a remote FPGA by up to 3x in comparison with a static design that deploys the same accelerator cores on a software-programmable FPGA locally. In addition, performance increases linearly with the number of remote devices when inter-FPGA communication is reduced. To exploit DPR on remote FPGAs and reduce communication, we adopt a versatile remote-accelerator deployment framework for disaggregated datacenters, thereby boosting BNN performance with negligible development effort.

Keywords. Binarized Neural Network, FPGA accelerator, Dynamic Partial Reconfiguration

1. Introduction

Considerable improvements in the development of high-performance systems for neural networks using multi-core technology have been proposed in recent years [1]. However, various challenges in power, cost, and performance scaling remain, due to the ever increasing model sizes (e.g., 50MB for GoogLeNet [2], 200MB for ResNet-101 [3], 250MB for AlexNet [4], or 500MB for VGG-Net [5]) that inevitably introduce pro-

hibitively high computational costs, steadily raising the need for accelerated solutions. The need for models with low memory and compute requirements is imperative.

Several works have been introduced to address the aforementioned challenges, and reduce the resource utilization requirements of CNNs, e.g., by exploiting the sparsity of the network connections [6], or by narrowing the data width [7, 8]. Another promising method is binarization, which relies on a considerably more compact data representation for the network weights and the neuron values than the one employed by regular CNNs. The underlying idea is to constrain each value to be either +1 or -1. Consequently, this reduces storage and memory bandwidth requirements and allows to replace floating-point operations with binary operations, thereby paving the way for efficient deep learning using FPGA technology.

Binarized Convolutional Neural Networks (BNNs) were first presented by Courbariaux et al. [9], who introduced a method to train BNNs with the permutation invariant MNIST, CIFAR-10, and SVHN [10] datasets, achieving state-of-art accuracy. Rastegari et al. [11] successfully trained a BNN with ImageNet models, reportedly improving accuracy, boosting performance, and reducing the model size, when compared with a full-precision AlexNet [4] implementation. Existing implementations of CNNs on FPGAs face several challenges due to their prohibitively high requirements for storage, memory bandwidth, and compute capacity. This problem exacerbates with more complex state-of-art models, such as the VGG model [7] that has 16 layers and 138×10^6 weights.

In this work, we investigate the potential of Dynamic Partial Reconfiguration (DPR) on modern FPGA-based multiprocessor system-on-chip (MPSoC) devices when deployed within a disaggregated-computing environment to boost BNN performance. Resource disaggregation addresses the problem of fixed resource proportionality in datacenters by creating and managing pools of different resource types, e.g., compute, memory, and accelerators. The immense parallel nature of BNNs suggests the eminent need for a disaggregated accelerator solution.

Devising a FPGA-based MPSoC disaggregated accelerator solution that exploits DPR beneficially to the performance of BNNs introduces additional challenges: 1. DPR brings flexibility in accelerator deployment, yet the high DPR overhead may diminish the expected performance gains. Thus, a beneficial computation-to-PR ratio is needed in order to justify the DPR overhead and improve performance, 2. The limited on-board memory resources set an upper bound on the maximum size of images that can be processed on a single node, and 3. The evident need for low-latency communication and synchronization in accelerator-rich environments becomes significantly more imperative in disaggregated computing platforms, where communication between remote nodes interconnected over a network is required [12, 13]. To address these challenges, we make the following contributions:

- We map BNN computations to ReFiRe [22], a remote-accelerator deployment framework for disaggregated computing. This allows to transparently exploit inter-FPGA parallelism and overcome the physical resource boundary per device for BNN computations by relying on the framework to stir computation to multiple disaggregated FPGA-based accelerator nodes. We find that throughput improves linearly with the number of accelerator devices, without requiring additional effort for communication or synchronization, neither on the host nor on the accelerator sides.

- We boost overall BNN throughput performance of a fixed set of three accelerator cores [14] per remote FPGA by transparently exploiting DPR and intra-layer parallelism through dedicated features of the employed accelerator deployment framework. We find that throughput improves up to 3x using DPR on a remote device than deploying the same set of accelerators locally through a software-programmable design flow [15]. Importantly, the proposed approach is highly generic and versatile, thus allowing to boost performance of existing CNN and/or BNN accelerators using DPR and parallelism, with negligible development effort.

2. Background

2.1. Convolutional Neural Networks

A typical CNN classifier consists of a parameterized pipelined multi-layer architecture. Layers require configuration of their parameters, often called weights, which must be determined by training the CNN offline on pre-classified data. Once the parameters are determined, the CNN can be deployed for the classification of new data points. The first layer takes as input a multi-channel input image and outputs a set of feature maps (fmaps). Each of the following layers read the fmaps, performs some computation on them, and produces a new set of fmaps to be fed into the next layer. Finally, a classifier produces the probability of that image belonging to each output class. The layer types are the following:

Convolutional layers realize a filter-like process, convolving the input fmaps with a $K \times K$ weight kernel. The results are summed, added with a bias, and passed through a non-linearity function to produce a single output fmap. This process is given in Eq. 1:

$$y_n = f(\sum_{m=1}^{M} x_m * w_{n,m} + b_n). \tag{1}$$

Pooling layers map each input fmap to an output fmap where every pixel is the max/mean of a $K \times K$ window of input pixels. They are inserted through a CNN to reduce the size of the intermediate fmaps.

Fully-Connected layers apply a linear transformation on the input 1-D vectors with a weight matrix. A bias is applied on the result, which is then passed through a non-linearity function to produce a single 1×1 output. This process is given in Eq. 2:

$$y_n = f(\sum_{m=1}^{M} x_m * w_{n,m} + b_n). \tag{2}$$

2.2. Binarized Convolutional Neural Networks

A BNN is essentially an extremely quantized, reduced-precision CNN model where weights and fmap pixels are binarized using the sign function. Positive weights are mapped to +1 and negative weights to -1, using a compact single-bit representation. Therefore, BNNs require significantly less storage than standard CNNs. The binarization of the neural networks can either be partial or full. In order to be considered full, it has to encompass the following aspects: binary input activations, binary synapse weights,

and binary output activations. Due to the quantization effect, there is no need for biasing since it does not compromise accuracy. However, in order to improve accuracy and scale down the error, a new layer type has to be introduced:

The **Batch normalization** [16] layer reduces the quantization error of the binarization by linearly shifting and scaling the input distribution to have zero mean and unit variance. The transformation is given in Eq. 3:

$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \varepsilon}}\gamma + \beta. \tag{3}$$

2.3. CIFAR-10 BNN Model

The CIFAR-10 dataset [17] contains sixty thousand 32×32 3-channel images consisting of photos taken of real world vehicles and animals. For the experiments, out of the 60,000 images, 50,000 images were chosen for training and 10,000 images for testing. Training of the CIFAR-10 BNN model was done using open-source Python code provided by Courbariaux et al. [9], which uses the Theano and Lasagne deep learning frameworks.

3. Related work

Multiple studies have explored the potential of FPGAs for implementing Artificial Neural Networks (ANNs). Zhang et al. [18] proposed an analytical CNN design scheme that is based on the roofline model [19] to explore various optimizations, such as loop tiling and transformation, to reduce resource underutilization and match the computation throughput to the available memory bandwidth on FPGAs. The study reports 61.62 GFLOPS peak performance at 100 MHz on a VC707 FPGA board.

Qiu et al. [7] presented a dynamic-precision data quantization method, as well as a convolver design for embedded FPGAs that performs well for all CNN layer types. The authors observed accuracy loss due to data quantization of as low as 0.4%, while the average performance of the convolutional layers and the full CNN is 187.8 GOP/s and 137.0 GOP/s at 150 MHz, respectively, when mapped to a Xilinx Zynq ZC706 board.

In 2015, Courbariaux et al. [9] introduced the idea of constraining weights to only two possible values, e.g., -1 and 1, in order to improve hardware performance of CNNs, since multiply-accumulate operations can be replaced by simple accumulations. Since then, multiple studies have presented FPGA-based accelerator architectures for BNNs. Umuroglu et al. [20] presented FINN, a framework to design efficient FPGA accelerators for BNNs by tailoring per-layer compute resources to user-provided throughput requirements. Employing a ZC706 board, the authors report up to 21,906 image classifications per second on the CIFAR-10 and SVHN datasets. Liang et al. [21] presented a BNN FPGA architecture that relies on bit-level XNOR and shifting operations, as well as data quantization and on-chip storage to achieve high performance. The authors report up to 9396.41 GOP/s for the CIFAR-10 dataset at 150MHz on a Stratix-V platform.

Zhao et al. [14] presented a novel design of a BNN accelerator for FPGAs, which is synthesized from a high-level language (C++) to Verilog using the Xilinx SDSoC [15] design flow. The overall accelerator, which operates at 143MHz and achieves 200 GOP/s for the CIFAR-10 dataset, consists of three computational cores, namely FP-Conv (first convolutional layer), Bin-Conv (binary convolutional layers), and Bin-FC (binary fully

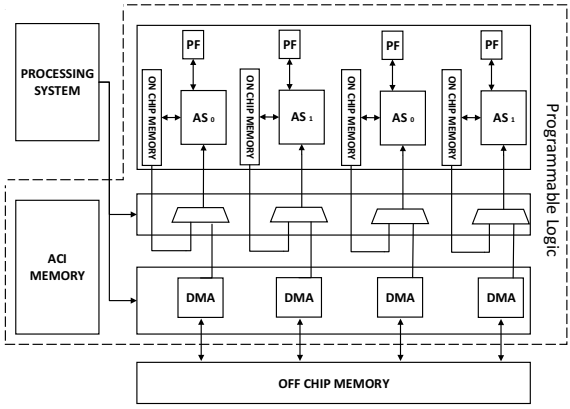


Figure 1. ReFiRe-based BNN accelerator architecture.

connected layers). Our work builds upon the work by Zhao et al. [14] and demonstrates how the proposed BNN accelerator can be mapped to the ReFiRe [22] remote-accelerator deployment framework, which allows to boost BNN performance without the need to redesign the aforementioned computational cores. ReFiRe [22] reduces communication and synchronization requirements between remote accelerator nodes (FPGA-based MPSoCs) in disaggregated datacenters by shifting control flow and partial reconfiguration decisions to the remote side through arbitrarily long instructions that encapsulate complex sequences of operations and their respective synchronization requirements. The framework abstracts away all the complexity of performing DPR on remote/disaggregated FPGAs, and allows to transparently exploit intra-FPGA parallelism per BNN layer, as well as inter-FPGA parallelism at image granularity. Note that, although DPR has been previously explored to boost CNN performance [23], this is the first work, to the best of the author's knowledge, that explores dynamic partial reconfiguration on disaggregated FPGAs to improve BNN performance.

4. Disaggregated Acceleration Framework

The ReFiRe [22] framework allows to efficiently deploy remote/disaggregated accelerators by improving the computation-to-communication ratio between a host processor and an arbitrary number of accelerator devices. This is achieved by relying on complex instructions of variable length, henceforth referred to as Advanced Coprocessor Instructions (ACIs), which describe partial reconfiguration events and the required flow of data among a set of remote partially reconfigurable accelerator cores.

4.1. Hardware architecture

The hardware architecture of the remote accelerator is illustrated in Figure 1. There are four accelerator slots (AS), with each AS being a partially reconfigurable region (PRR). Each AS has a private Parameter file (PF) to facilitate accelerator configuration. Furthermore, four Direct Memory Access (DMA) engines are responsible for transferring data between external memory and each AS. Depending on the BNN layer processed at each point in time, input data to each AS can arrive either from on-chip storage (output data

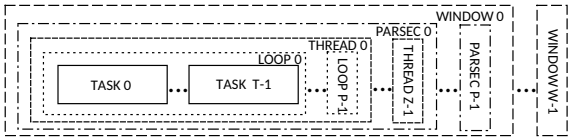


Figure 2. Hierarchy of ACI instruction classes (source: [22]).

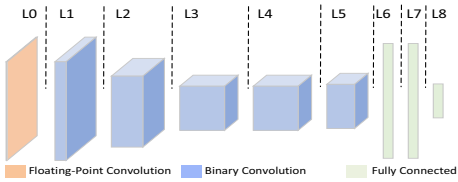


Figure 3. Binarized Neural Network architecture.

of a previous layer) or from external memory (input data of the very first layer). The ACI memory holds the active ACI at each juncture, and directs computation and PR events on each AS.

4.2. ACI architecture

The ACI memory consists of three main parts, namely SYNC, COMPUTE, and PARAM. The SYNC area facilitates host-accelerator synchronization. The PARAM area facilitates the parameter-based configuration of each accelerator per AS. The COMPUTE area holds a sequence of instructions that correspond to an application-specific execution scenario. There are five ACI instruction types, organized into a hierarchy of classes: WINDOW, PARSEC, THREAD, LOOP and TASK, as illustrated in Figure 2.

The TASK class contains information related to the input and output data per AS for a given operation. The multiplexer and the PF per AS are configured and initialized, respectively, based on data extracted from each TASK class. The LOOP class is a container class for TASK objects, which allows to reduce host-accelerator synchronization requirements by providing the required number of iterations per accelerator operation in an AS, as well as the desired stride for both the input and the output data. The THREAD class dictates an order of operations that are performed sequentially, whereas the PARSEC class indicates a parallel section with a number of THREAD classes that execute in parallel on different AS. Finally, the WINDOW class dictates PR requirements, as each WINDOW starts with one or more requests for partial reconfiguration.

4.3. Mapping the BNN to an ACI

The architecture of the BNN consists of nine layers, with the first six being convolutional layers while the next three are fully connected layers, as illustrated in Figure 3. The first layer (L0 in Fig. 3) receives fixed-point input data and binary weights, whereas the rest of the layers (L1 through L8 in Fig. 3) operate only on binary data. The convolutional layers rely on 3×3 filtering and edge padding, while the fully connected layers apply batch normalization prior to pooling, and binarization before writing data out to the buffers. The accelerator system presented by Zhao et al. [14] designed three accelerators, which

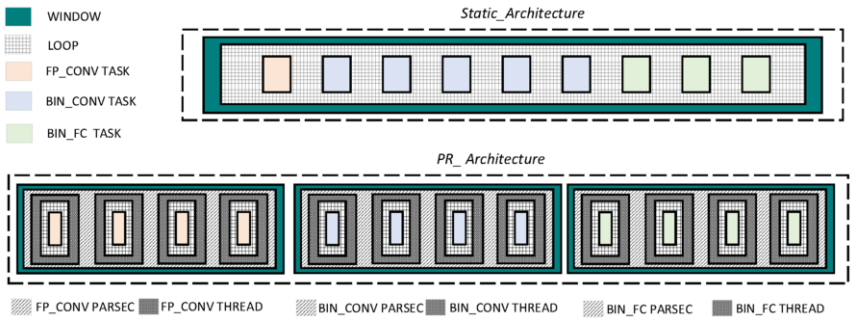


Figure 4. Illustration of the ACI format for the *Static_Architecture* and the *PR_Architecture* for FPGA-based BNN acceleration.

we employ as-is in our disaggregated accelerator systems. The *FP_CONV* core implements the L0 layer of the BNN. The *BIN_CONV* core is employed for the following five binary-only convolution layers (L1 through L5). Finally, the *BIN_FC* core accelerates the last three BNN layers (L6 through L8).

To map the required BNN computations to an ACI, we place each accelerator call in a dedicated TASK class, which also contains the respective core's configuration parameters and input/output address and sizes. The number of images that are processed in-between PR events is defined as the number of iterations of a LOOP class, with the stride being the image size. The THREAD and PARSEC classes allow to expose parallelism per layer by partitioning processing over multiple AS that host the same accelerator core. Finally, the WINDOW class performs one PR event per AS to deploy a different accelerator core to serve the needs of the next BNN layer. Due to the fact that there are three accelerator cores, the final ACI that implements the BNN consists of three WINDOW classes, one per accelerator core. Figure 4 illustrates alternative execution scenarios based on different ACI structures for the BNN. The *Static_Architecture* is identical to the reference execution scenario that is implemented on a software-programmable FPGA by Zhao et al. [14]. Due to the fact that ReFiRe is a native partially reconfigurable architecture, the *Static_Architecture* involves the initial deployment of the three accelerators in three AS. This is achieved by placing all nine TASK classes (one per BNN layer) in the same WINDOW class. The *PR_Architecture* exploits PR at run time and exposes intra-layer parallelism through the PARSEC/THREAD classes. Therefore, three WINDOW classes are required, one per accelerator core, and multiple ACI-based iterations are performed.

5. Implementation and Evaluation

To evaluate performance on disaggregated FPGAs, we employ two ZCU102 boards, each containing a Zynq UltraScale+ MPSoC, interconnected over a Small Form-factor Pluggable (SFP) 10-Gbps link. Each MPSoC hosts an ARM Cortex-A53 64-bit quad-core processor running at 1.2 GHz. One board serves as the host processor, whereas the second is the accelerator platform. The host board runs Ubuntu 16.04 on its Application Processing Unit (APU), while all communication is established through the SFP link. All

three accelerator cores deployed in ReFiRe are retrieved from *https://github.com/cornell-zhang/bnn-fpga*. Table 1 provides resource utilization per accelerator on the ZCU102 evaluation platform, hosting a Zynq Ultrascale+ MPSoC. We evaluate two alternative execution scenarios, the *Static_Architecture* and *PR_Architecture* (illustrated in Fig. 4) using the CIFAR-10 dataset.

Table 1. Resource utilization for the three BNN accelerator cores on the Zynq Ultrascale+ MPSoC

ACCEL.	LUTs	FFs	BRAMs	DSPs	Power (W)
FP_CONV	11609	13802	16	0	0.112
BIN_CONV	13208	5849	86	2	0.050
BIN_FC	4432	6148	20	2	0.086

5.1. Static_Architecture

We initially reproduce, using ReFiRe, the same static execution scenario that was evaluated by Zhao et al. [14] using SDSoC. Thus, we first deploy the accelerator cores *FP_CONV*, *BIN_CONV*, and *BIN_FC* through an initial configuration WINDOW. In this scenario, all 10,000 images we used for evaluation are processed sequentially, directing the layers output to the next, as dictated by the BNN architecture. This approach required 128 sec to complete. As a reference, we note that the SDSoC-based approach [14] using the exact same accelerators and number of images required 103.1 sec. The observed delay is due to data exchanges between remote FPGAs for ACI transfers and synchronization.

5.2. PR_Architecture

Next, we evaluate the DPR-based execution scenario by populating all AS with the same accelerator core and rely on the PARSEC and THREAD classes to invoke them in parallel per layer. The DPR overhead per AS (using ICAP [26]) is 7 ms (2.5 MB bitstream sizes). Note that, Zhao et al. [14] report 5.7 ms per image without using DPR. Thus, to yield a beneficial computation-to-PR ratio to exploit DPR using ReFiRe, we organize processing in batches. Figure 5 illustrates how performance improves with the batch size. As can be observed in the figure, DPR allows to outperform the fully static architecture when the batch size exceeds 25 images/batch. Evidently, processing a single image in-between DPR events yields the worst-case performance, requiring 917 seconds in total for the 10,000 images, when the static design with 1 instance per accelerator finishes in 128 seconds. When the batch size exceeds 300 images, DPR allows up to 3.1x faster execution, due to the four accelerator instances per layer. Note that, aggregate system performance increases almost linearly with the number of disaggregated FPGAs used, due to the beneficial computation-to-synchronization ratio that the ACI offers. The overhead to create and transfer an ACI to a remote FPGA is as low as 1.33 sec.

5.3. Comparison with other works

A comparison with previous FPGA accelerator designs for CNN and BNN models is provided in Table 2. Suda et al.[24] and Qui et al.[7] reported 117 GOPS/s and 136 GOPS/s, respectively, significantly lower than the performance attained through ReFiRe. Li et al.[25] achieved 594 GOPS/s, with 22.5 GOP/s/W efficiency, due to the increased power consumption of the design. Our work outperforms the reference approach pro-

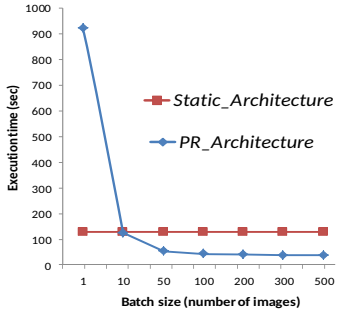


Figure 5. Execution time to process 10,000 images using the *Static_Architecture* and the *PR_Archtitecture* when the batch size (number of images in-between PR events) grows up to 500.

posed by Zhao et al. [14], achieving about 3.1 times higher performance and efficiency for the exact same set of accelerators. Umuroglou et al. [20] and Liang et al. [21] report considerably higher performance than all other approaches. Therefore, we intend to employ ReFiRe to further improve the performance of these accelerators by transparently introducing DPR and deploying disaggregated FPGAs.

Table 2. Performance comparison with other FPGA-based CNN/BNN accelerators. The presented accelerator system employs the same set of accelerator cores as Zhao et al. [14].

	Zhao et al.[14]	**This work**	Suda et al.[24]	Qiu et al.[7]	Li et al.[25]	Umuroglu et al.[20]	Liang et al.[21]
Platform	Zynq	**ZynqMP**	Stratix-V	Zynq	Virtex-7	Zynq	Stratix-V
	XC7Z020	**XCZU9EG**	5SGSD8	XC7Z045	VX690T	XC7Z045	5SGSD8
Clock(MHz)	143	**150**	120	150	156	200	150
Precision(bit)	Input: 8	**Input: 8**	8-16	16	16	Input: 8	Input: 8
	Weight: 1	**Weight: 1**				Weight: 1	Weight: 1
Model size (OPs)	1.24 G	**1.24 G**	30.9 G	30.76 G	1.45 G	112.5 M	1.23 G
Performance (GOP/s)	207.8	**667**	117	136	565.94	2465.5	9396.41
Power(W)	4.7	**5.97**	25.8	9.63	30.2	11.7	26.2
Efficiency (GOP/s/W)	44.2	**111.73**	4.57	14.22	22.15	210.72	358.64

6. Conclusions

Binarized Convolutional Neural Networks (BNNs) offer significant accuracy, performance and model compression over standard full-precision Convolutional Neural Networks (CNNs). This paper proposed a hardware accelerator architecture for BNNs on modern FPGA-based MPSoC devices deployed within a disaggregated-computing environment. We explored the trade-offs in exploiting Dynamic Partial Reconfiguration (DPR) to meet the performance, communication, and latency requirements. We find that throughput performance improves linearly with the number of accelerator devices, without requiring additional effort for communication or synchronization, neither on the host nor on the accelerator sides. We explored, a generic acceleration framework that improves performance of remote, fine-grained accelerators by encapsulating complex sequences of operations in arbitrarily long instructions called ACIs. We compared these accelerator instances against other FPGA-based BNN implementations in the literature. Our evaluation results show that disaggregation offers an attractive solution, which allows to expose near-peak accelerator performance at the application level, despite performing computations on remote nodes. Future work will focus on architectural improvements, exploring a low-precision network for a much larger and more complicated dataset like ImageNet and AlexNet.

References

[1] E. Nurvitadhi, D. Sheffield, , A. Mishra, G. Venkatesh, and D. Marr, "Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC," in *ICFPT*, 2016, pp. 77–84.

[2] C. Szegedy *et al.*, "Going Deeper with Convolutions," *CoRR*, vol. abs/1409.4842, 2014.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, 2016, pp. 770–778.

[4] A. Krizhevsky *et al.*, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems (NIPS)*, 2012, pp. 1097–1105.

[5] K. Simonyan *et al.*, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *CoRR*, vol. abs/1409.1556, 2014.

[6] X. Xie *et al.*, "Exploiting Sparsity to Accelerate Fully Connected Layers of CNN-Based Applications on Mobile SoCs," *ACM Trans. Embedded Comput. Syst.*, vol. 17, no. 2, pp. 37:1–37:25, 2018.

[7] J. Qiu *et al.*, "Going Deeper with Embedded FPGA Platform for Convolutional Neural Network," *FPGA*, 2016.

[8] F. N. Iandola *et al.*, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size," *CoRR*, vol. abs/1602.07360, 2016.

[9] M. Courbariaux, Y. Bengio, and J. David, "BinaryConnect: Training Deep Neural Networks with binary weights during propagations," *CoRR*, vol. abs/1511.00363, 2015.

[10] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading Digits in Natural Images with Unsupervised Feature Learning," in *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

[11] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks," *CoRR*, vol. abs/1603.05279, 2016.

[12] K. Katrinis *et al.*, "Rack-scale disaggregated cloud data centers: The dReDBox project vision," in *DATE 2016*. IEEE, 2016, pp. 690–695.

[13] A. M. Caulfield *et al.*, "A cloud-scale acceleration architecture," in *MICRO 2016*. IEEE, 2016, pp. 1–13.

[14] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang, "Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs," *FPGA*, 2017.

[15] V. Kathail *et al.*, "SDSoC: A Higher-level Programming Environment for Zynq SoC and Ultrascale+ MPSoC," *FPGA*, 2016.

[16] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *CoRR*, vol. abs/1502.03167, 2015.

[17] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," 2009.

[18] C. Zhang *et al.*, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," *FPGA*, 2015.

[19] S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[20] Y. Umuroglu *et al.*, "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference," *FPGA*, 2017.

[21] S. Liang *et al.*, "FP-BNN: Binarized neural network on FPGA," *Neurocomputing*, 2017.

[22] E. Pissadakis, N. Alachiotis, P. Skrimponis, D. Theodoropoulos, T. Korakis, and D. Pnevmatikatos, "ReFiRe: efficient deployment of Remote Fine-grained Reconfigurable accelerators," *ICFPT*, 2018.

[23] F. Kstner *et al.*, "Hardware/Software Codesign for Convolutional Neural Networks exploiting Dynamic Partial Reconfiguration on PYNQ," *IEEE International Parallel and Distributed Processing Symposium Workshops*, 2018.

[24] N. Suda *et al.*, "Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks," *FPGA*, pp. 16–25, 2016.

[25] H. Li *et al.*, "A High Performance FPGA-based Accelerator for Large-Scale Convolutional Neural Networks," *International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–9, 2016.

[26] Xilinx, "UltraScale Architecture Configuration," https://www.xilinx.com/support/documentation/user_guides/ug570-ultrascale-configuration.pdf, [Online; accessed 05-Jul-2018].

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200100

Porting a Lattice Boltzmann Simulation to FPGAs Using OmpSs

Enrico CALORE [a,1], Sebastiano Fabio SCHIFANO [a,b]
[a] *INFN Ferrara, Italy*
[b] *University of Ferrara, Italy*

Abstract. Reconfigurable computing, exploiting *Field Programmable Gate Arrays* (FPGA), has become of great interest for both academia and industry research thanks to the possibility to greatly accelerate a variety of applications. The interest has been further boosted by recent developments of FPGA programming frameworks which allows to design applications at a higher-level of abstraction, for example using directive based approaches.

In this work we describe our first experiences in porting to FPGAs an HPC application, used to simulate Rayleigh-Taylor instability of fluids with different density and temperature using Lattice Boltzmann Methods. This activity is done in the context of the FET HPC H2020 EuroEXA project which is developing an energy-efficient HPC system, at exa-scale level, based on Arm processors and FPGAs. In this work we use the *OmpSs* directive based programming model, one of the models available within the EuroEXA project. *OmpSs* is developed by the Barcelona Supercomputing Center (BSC) and allows to target FPGA devices as accelerators, but also commodity CPUs and GPUs, enabling code portability across different architectures. In particular, we describe the initial porting of this application, evaluating the programming efforts required, and assessing the preliminary performances on a Trenz development board hosting a Xilinx Zynq UltraScale+ MPSoC embedding a 16nm FinFET+ programmable logic and a multi-core Arm CPU.

Keywords. FPGA, OmpSs, EuroEXA, HPC, Lattice Boltzmann

1. Introduction

Reconfigurable computing using *Field Programmable Gate Arrays* (FPGA) is attracting lot of attention from the scientific community for its potential to accelerate a large variety of applications with interesting performance-energy ratios. However, the complexity of programming such devices has been one of the major issues preventing FPGAs to become widely adopted in scientific software communities. In fact, FPGAs have been commonly programmed using *Hardware Description Language* (HDL) such as VHDL and Verilog, which allow to describe arbitrary circuitry at *Register Transfer Level* (RTL). This approach is too low level for many application programmers, and has restricted the use of FPGA mainly to electronic engineering experts.

[1]Corresponding Author: Enrico Calore, INFN Ferrara, Via Saragat 1, 44121 Ferrara, Italy; E-mail: enrico.calore@fe.infn.it.

Despite of this, FPGAs have been successfully used in the past to boost the computing performance of several scientific applications. As an example: COPACOBANA [1] for code breaking; RIVYERA [2] for bioinformatics applications; EXTOLL [3] for communications and Janus [4,5] for spin-glasses simulations. These are all relevant projects using FPGAs as processing elements for HPC and scientific applications in general [6]. However, they required strong customization of applications, using data structures and implementations, very far from that used on commodity CPUs.

Recently, the resources available on FPGA chips have increased a lot, integrating high-end interfaces (e.g., PCIe, DDR3, GbitE, etc...) large static memories, large number of DSPs and also CPUs cores. The last feature in particular has raised the interest of many researchers for enabling FPGA-accelerated computing. A CPU is integrated on the device together with the programmable logic, and is able to run a full operating system, commonly based on GNU/Linux, allowing to start applications on the CPU to later offload specific functions on the FPGA.

In the last years, also the programming environments have been extensively developed. Languages such as OpenCL and High-Level Synthesis (HLS) frameworks based on *pragmas* directives are now available for different FPGAs, allowing to describe applications at algorithmic level [7]. This can be then interpreted and transcompiled by automatic software tools into a Register-Transfer Level (RTL) and finally synthesized to the gate level. In particular, for applications already developed for ordinary CPUs and accelerators, directive approaches allow to just annotate legacy C and Fortran codes with *pragmas* that guide the compilers in the synthesis process. Clearly, this approach is more abstract compared to a low level manual programming of the HDL code, and can results in less optimized designs in terms of timing and FPGA resources usage. Despite of this, the reduced programming effort required, combined with a faster design space exploration and a much higher software portability, make this approach very attractive and usable by larger application developers communities.

All of these factors contributed to the possibility of using FPGAs as computing accelerators, and many projects are now following this path. One of this is the EuroEXA project [2], a H2020 project funded by the EU, that following an hardware/software co-design approach, aims to port a rich mix of applications to its architecture. One of these applications consist in the simulation of fluids using Lattice Boltzmann Methods (LBM). The increasing popularity of LBM comes from its flexibility, allowing to study complex geometries and different types of boundary conditions, and from being particularly suitable for highly scalable implementations on massively parallel architectures [8].

In the EuroEXA project Xilinx FPGAs are adopted. Xilinx provides the VivadoHLS Design Suite to annotate C codes with proprietary HLS directives, allowing to offload a specific function to an FPGAs, automatically managing the host code compilation and the synthesis of the function to be offloaded. Anyhow, in this work we include a further level of programming abstraction over VivadoHLS and in particular we use the *OmpSs* directive based programming model in conjunction with its *OmpSs@FPGA* extension [9]. *OmpSs* allows to annotate the application code with directives to compile and offload a kernel on FPGAs, enabling accelerated computing, but given the same source code, it is also able to target other devices, such as GPUs or multi-core CPUs, easily enabling code portability [10].

[2] https://euroexa.eu/

The remainder of this contribution is organized as follows: in the next Section we introduce the EuroEXA project, in Sec. 3 we describe our LBM application, and in Sec. 4 we briefly overview *OmpSs@FPGA*. Sec. 5 describes our code porting to FPGA, Sec. 6 reports our results, and finally Sec. 7 highlights some concluding remarks.

2. The EuroEXA Project

The *Co-designed innovation and system for resilient exascale computing in Europe: from application to silicon* (EuroEXA) is a H2020 FET HPC project funded by the EU commission with a budget of ≈ 20M€. The aim of the project is to develop a prototype of an *exascale* level computing architecture suitable for both compute- and data-intensive applications, delivering world-leading energy-efficiency. To reach this goal this project proposes to adopt a cost-efficient, modular integration approach enabled by: novel iner-die links; FPGAs to leverage data-flow acceleration for compute, networking and storage; an intelligent memory compression technology; a unique geographically-addressed switching interconnect and novel Arm based compute units. As main computing elements are going to be adopted multi-core Arm processors combined with Xilinx UltraScale+ FPGAs, to be used both as compute accelerators and to implement an high bandwidth and low-latency interconnect between computing elements.

Form the software platform point of view, EuroEXA provides five high-level programming frameworks that enable FPGA-accelerated computing: Maxeler MaxCompilerMPT [3], OmpSs@FPGA [9], OpenStream [11], SDSoC or SDAccel [4] with OpenCL, and Vivado High Level Syntesis [5]. These frameworks are used to implement several key HPC applications across climate/weather, physics/energy and life-science/bioinformatics scientific domains. More details about the EuroEXA project can be obtained from its website: https://euroexa.eu.

In this work we describe our early steps towards the porting of our application within the EuroEXA Project using the OmpSs programming model. In preparation for the EuroEXA prototype, we are working on a Trenz TE8080 development board where we have developed our early implementations and performed preliminary performance measurements.

3. The Lattice Boltzmann Application

In this contribution we address CFD simulation applications based on the Lattice Boltzmann Method (LBM), a class of CFD solvers able to describe efficiently the physics of complex fluid flows, through a mesoscopic approach. LBM are stencil-based algorithms, discrete in space, time and momenta, operating on regular lattice grid. A set of synthetic pseudo-particles called *populations* are sitting at the edges of the lattice, and evolved for several time steps. At each time step, populations *propagate* from lattice-site to lattice-site, and then *collide* among each other updating their physical parameters. These two steps are the most compute intensive parts of actual LBM codes. In both rou-

[3] https://www.maxeler.com/solutions/low-latency/maxcompilermpt/
[4] https://www.xilinx.com/products/design-tools/all-programmable-abstractions.html
[5] https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

tines, there are no data dependencies between different lattice points, so they can execute in parallel on as many lattice sites as possible and according to the most convenient schedule. Boundary conditions, specific to each particular problem could be applied but have a truly minor computational impact on simulation time. A model labeled as *DxQy* describes a fluid in x dimensions using y populations per lattice point.

The specific model used in this work, the D2Q37, has been extensively used for convective turbulence studies [12], but has been also deeply optimized and used as a benchmarking application for several programming models and HPC hardware architectures. In the D2Q37 model the *propagate* function gather at each lattice site populations values from neighbors at distance up to 3 in the grid, generating a large number of sparse memory accesses and resulting to be strongly memory-intensive. The *collide* kernel performs ≈ 6600 double precision floating-point operations on data local to each lattice site, and is strongly compute-intensive with an arithmetic intensity greater than 10 [13]. Different implementations of this code have been designed and implemented, adopting several directive based languages to address both multi-core CPUs [14] and accelerators [8], such as GPUs [15] and also many-core devices [16].

4. The OmpSs Programming Model

In this work we have implemented our application using the *OmpSs* directives based programming model, developed at the BSC, and its *OmpSs@FPGA* extension [9].

OmpSs is very similar to the widely known OpenMP, and in fact it is a forerunner of OpenMP, where new features are introduced and developed before possibly getting pushed in the OpenMP standard. *OmpSs* is one of the tools selected to be used in the framework of the EuroEXA project to exploit FPGAs as accelerators, and allows to define task functions to be offloaded to such devices. It provides an automatic generation of a wrapper code handling data copies to and from the FPGA device, and manages flow dependencies and synchronizations. These data dependencies can be specified by the programmer using directives, as shown in Listing 1, where an example function is decorated with *pragmas* in order to be offloaded to an FPGA. Different buffers are set as input and outputs, giving also their sizes, in order to be copied in and out as needed. The function body to be actually offloaded onto FPGA, once processed by the *OmpSs* source to source toolchain, gets transformed into a bitstream by the VivadoHLS synthesis tool, allowing the programmer to add also proprietary HLS directives in the source code.

Thanks to the *OmpSs* directives, simply changing the offload target (which directly affect the final compiler to be used), the same source code can be compiled for several architectures, possibly targeting different accelerators. Interestingly enough, the use of *OmpSs* allows us also to exploit a wider set of tools developed at BSC, meant for performance analysis. In particular, we used Extrae [17], a tracing tool allowing to collect information during the execution of an application, such as: hardware counters; calls to MPI, OpenMP and *OmpSs* libraries; etc. To later utilize the acquired traces, we used also Paraver [18], a performance analysis tool which loading traces generated by Extrae provides a visual interface to analyze them. The traces can be displayed as timelines of the execution, as shown later, but can also be used to perform much more complex statistical analyses [19,20].

Listing 1: Example of a function performing the dot product operation, decorated with *OmpSs* directives, in order to be offloaded on a FPGA accelerator.

```
#pragma omp target device(fpga)
#pragma omp task in([BSIZE]v1, [BSIZE]v2) inout([1]result)
void dotProduct(float *v1, float *v2, float *result) {
  int resultLocal = result[0];
  for (size_t i = 0; i < BSIZE; ++i) {
    resultLocal += v1[i]*v2[i];
  }
  result[0] = resultLocal;
}
```

5. Implementation

Our application, when exploiting accelerators, is commonly implemented allocating the whole data domain into the device memory once for all, at the beginning of the simulation, and then performing several iterations of the algorithm following a double buffering approach [8]. Despite of this, when using FPGAs, the on-board memory is quite limited (although faster) with respect to other accelerators, thus we developed a blocking implementation, allowing to move slices of the whole lattice, to the FPGA device, in order to compute one slice at a time. To slice the lattice, gather and scatter operations are required in order to move just contiguous memory buffers, in and out from the FPGA BRAMs. The host part of this implementation is shown in Listing 2. Here we can see an outer loop over the iterations and an inner loop over different blocks of the lattice. Once the gather operation is completed on the host side, the *lbmBlocking* task function is called, which automatically handles the copy in and out of buffer arguments thanks to the OmpSs directives shown in Listing 3.

As described in Sec. 3, for each iteration, two different functions are commonly implemented: *propagate* and *collide*. In a first ported version, we directly implemented these two as inline functions, calling one after the other inside the *lbmBlocking* one. This requires a temporary intermediate buffer allocated in the FPGA's BRAMs which could be easily removed by merging the two function in a single one, saving about one third of the BRAM required.

Using proprietary VivadoHLS directives, as the ones shown in Listing 3, we have also optimized the placement of arrays in the BRAMs (using *HLS array_partition* directive), allowing for the concurrent access of multiple data items. Using *HLS pipeline* and *HLS unroll* directives we have been able also to achieve pipelining or unrolling of the loops performing the *collide* operation, increasing the performance as reported in Sec. 6.

On other parallel accelerators, such as GPUs, this application is commonly parallelized computing several lattice sites at the same time, exploiting the independence between the loops over the lattice sites [8]. On an FPGA this would translate to the unrolling and replication of the whole function body, which is not possible with the available resources on our development board. On the other side, to achieve a certain level of resources reuse, one may pipeline the execution over the lattice sites, allowing to start the computation of a new lattice site every few clock cycles. Unfortunately, at this stage, the resources of our target FPGA should be enough to pipeline the execution over the

Listing 2: Core of the LBM application showing the call to the function to be offloaded on the FPGA and computing one iteration on one block of the lattice.

```
for (i=0; i<NITER; i++) {
  for ( ix = HX; ix < HX+LX ; ix+=BCOL ) {

    // Gathering
    Bprv[...] = f1_soa[...];

    lbmBlocking( Bnxt, Bprv, param );
    #pragma omp taskwait

    // Scattering
    f2_soa[...] = Bnxt[...];

  } // Block loop
} // Iter loop
```

lattice sites, but the high routing congestion, due to the *collide* operation complexity, did not allowed us to produce a working bitstream.

From the portability point of view, interestingly enough, when compiling the application for architectures not using VivadoHLS, these directives are just ignored. In particular we compiled exactly the same code to run on the Arm cores of the Cortex A53 processor available in the same Trenz development board. Other directives could be added in the future, exploiting other directive based languages, to target other architectures.

6. Results

From the portability point of view, a first result is that we now have a single implementation able to be compiled for a multi-core CPU, just selecting *smp* as device target, or to offload the most time consuming part of our LBM application to FPGAs, selecting *fpga* as target device.

In this work, to test the application exploiting the FPGA offload on an actual hardware device, we have used a Trenz TE8080 development board [6], which hosts a Xilinx UltraScale+ ZU9 MPSoC. The FPGA in our Trenz board is much smaller, both in terms of resources and capabilities, wrt the one that will be used in the EuroEXA project, nevertheless, it is supported by *OmpSs* and it allows us to test and run our preliminary ported code and measure initial results. From the performance point of view, in fact, results are still very preliminary and even on this testbed a wide range of further optimization could be still explored. We report in Tab. 1 results measured with different implementations of our code showing the percentage of the ZU9 FPGA resources utilized, and the corresponding overall execution time divided by the lattice size (set to 256×256), giving the execution time required for each lattice site. We underline this is one of the handiness of using high level synthesis tools, which allow to easily test different implementations, possibly changing just *pragma* directives.

[6] https://shop.trenz-electronic.de/en/TE0808-04-9BE21-AS-TE0808-04-9BE21-AS-Starter-Kit

Listing 3: Sketch of the kernel function to be offloaded onto the FPGA, with *OmpSs* and HLS directives, corresponding to the last column implementation in Tab. 1.

```
#pragma omp target num_instances(1) device(fpga)
#pragma omp task out([BS]Bnxt) in([BH]Bprv, [PS]param)
void lbmBlocking(data_t Bnxt[BS],data_t Bprv[BH],data_t param[PS]) {

  #pragma HLS array_partition variable=param block factor=37

  for ( ix = 0; ix < BCOL; ix++) {
    for ( iy = HY; iy < (HY+LY); iy++) {

      #pragma HLS pipeline II=32
      #pragma HLS loop_flatten
      #pragma HLS expression_balance
      #pragma HLS dependence variable=Bnxt inter false

      data_t localPop[NPOP];

      #pragma HLS array_partition variable=localPop complete

      // PROPAGATE
      localPop[0] =  Bprv[              idxh - 3*NY + 1];
      localPop[1] =  Bprv[  1*popoffh + idxh - 3*NY    ];
      ...
      localPop[36] = Bprv[ 36*popoffh + idxh + 3*NY - 1];

      // COLLIDE
      for (p = 0; p < NPOP; p++) { Ops on localPop[] and param[] };
      ...
      for (p = 0; p < NPOP; p++) { Ops on localPop[] and param[] };

      Bnxt[] = ...;
      }
  }
```

The first version, on the leftmost column, refers to the original code annotated just with *OmpSs* directives, giving an execution time per site of $\approx 61\mu$sec. In the second version, we have added also HLS directives, pipelining or unrolling the inner loops involved in the *collide* operator, increasing the performance by a factor $\approx 5\times$, and reducing the time per lattice-site to $\approx 12.6\mu$sec In a third version we attempted to optimize for resources, merging the *propagate* and *collide* functions, and applying just the pipelining of the loops in the *collide* region. Moreover we also partitioned an array of constant parameters, splitting its content across different BRAMs, to allow to access different data items during the same clock cycle. As shown in Tab. 1, this results in a reduction of resources usage, especially for DSPs and BRAMs, without a negative impact on the time per site, which is even slightly better. In the last version we attempted to pipeline over the lattice sites (i.e., the outer loops). In this case, as reported by Vivado, the computing resources of the ZU9 FPGA should be enough to fit the design, if the *Initiation Interval* (II) – corresponding to the number of clock cycles to wait before filling in the pipeline a new lattice site – is kept greater or equal to 32, as shown in Listing 3. Anyhow, unfortunately,

Table 1. FPGA resources utilization and achieved overall execution time per lattice site at 200MHz, for different implementations of the offloaded function.

	First version	**Pipeline/Unroll Collide loops**	**Merged Funcs and Partition**	**Pipeline over sites**
DSP48E	2.8%	16.9%	4.8%	20.83%
BRAM_18K	29.2%	35.7%	11.4%	31.74%
LUT	9.1%	34.7%	38.28%	68.17%
FF	5.5%	15.1%	12.44%	58.66%
Exec. Time per lattice site	$60.62\mu s$	$12.65\mu s$	$12.4\mu s$	$\approx 0.16\mu s$ (Estimated)

the high amount of computing resources involved and the code complexity, result in high routing congestion levels and consequently in a failure of the final bitstream synthesis. Further increasing the `II` value allows to reduce the computing resources usage, but the routing congestion still impairs the synthesis, unless using `II` values in the same order of the pipeline depth. Neglecting routing issues, taking into account the FPGA clock frequency – set to 200MHz – and the minimum `II` value that allows to fit FPGA computing resources, we can estimate an execution time of $\approx 0.16\mu$sec per lattice site. This corresponds to a performance speed-up of about one order of magnitude. This is relevant in prospective, giving us an estimation of what we could expect to obtain e.g. on the larger Xilinx VU9 UltraScale+ FPGA, which could be adopted by the EuroEXA project.

To make a point of reference, we can compare the results achieved with the ones measured on a processor with a similar power envelop. In particular, running the same code on the Arm Cortex A53 embedded in the same MPSoC, we measure a value of $9.26\mu s$ per lattice site. This result is very close to the one measured on the ZU9 FPGA, but as already highlighted, using the larger FPGA available in the EuroEXA computing nodes, we expect to significantly increase this performance value.

Another interesting result is that *OmpSs* can be combined with a set of performance profiling tools, such as Extrae and Paraver. Extrae allows to collect execution traces that can be then visualize using Paraver. In Fig. 1 we show the execution traces of several launches of our code, corresponding to the different slices (or blocks) of the lattice. running on the Trenz board. We clearly see 5 different timelines, one for each core of the Arm CPU and one for the FPGA. The threads (in green) spawns the tasks (in red) which offload the *lbmBlocking* kernel that is executed on FPGA (in blue).

7. Conclusion and future works

Using *OmpSs* programming model and *OmpSs@FPGA* extension, after the initial setup of a working tool-chain involving OmpSs, Xilinx VivadoHLS SDK and Xilinx Petalinux (to generate bootable images for the Trenz board), we have been able, with minimal code modifications, to allow an actual HPC Lattice Boltzmann application to exploit a Xilinx FPGA as an accelerator. Interestingly, the same code can be compiled targeting different architectures, such as x86 and Arm multi-core CPUs.

Adding proprietary HLS directives, we have been able to increase the performance by $5\times$ wrt the initial version, without introducing major changes to the actual C code,

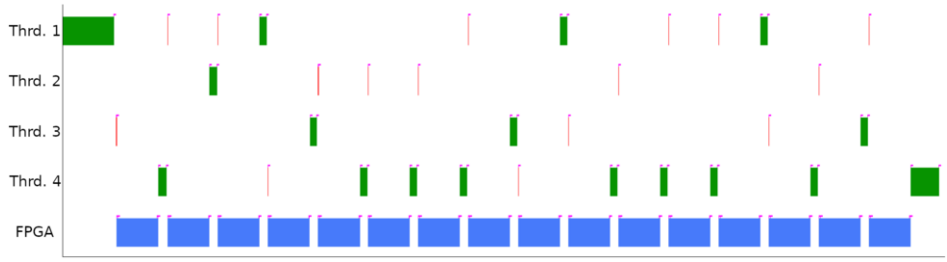


Figure 1. Paraver view of the execution timeline. The 4 top rows represent the 4 Arm cores performing the lattice initialization and the gather/scatter operations (Green) and managing the function offload to the FPGA (Red). In the bottom row is represented the execution timeline in the FPGA, each Blue box represent the fused Propagate-Collide computation on one lattice block.

starting from the original implementation of the two main functions of the LBM algorithm. With some modifications (i.e., merging *propagate* and *collide*) we were able to save some BRAM memory, allowing to increase the lattice slice we could process. On the Xilinx ZU9 FPGA we have used in this work the result achieved is similar to that measured running the code on the Arm cores of the same MPSoC; however we have estimated that using a larger FPGA, e.g the Xilinx VU9, we can speed-up the execution time by at least one order of magnitude. In particular, on the VU9 we expect to be able to pipeline over the lattice sites, thanks to the increased routing resources, and to reduce the minimum `II`, thanks to the higher amount of computing resources.

As future works, we plan to re-organize the loops involved in the *collide* kernel, to help to parallelize reduction operations involved in the inner loops. On the host side we aim to avoid gathering and scattering operations and succeed to overlap data transfers with computations. In particular, we aim to develop a multi-FPGA implementation, keeping one slice of lattice stored on each FPGA (e.g., in the VU9 on-board UltraRAM will be available), and moving back and forth from the host-DRAM only the lattice-slice halos for communications with neighbours. We also expect to work soon on a Xilinx VU9 in order to verify our estimations.

Acknowledgments: E.C. has been supported by the European Union's H2020 research and innovation programme under EuroEXA grant agreement No. 754337. This work has been done in the framework of the EuroEXA EU project and of the COKA, and COSA, INFN projects.

References

[1] Güneysu, T., Kasper, T., Novotný, M., Paar, C., Rupp, A.: Cryptanalysis with copacobana. IEEE Transactions on Computers 57(11), 1498–1513 (Nov 2008), doi:10.1109/TC.2008.80

[2] Wienbrandt, L.: Bioinformatics Applications on the FPGA-Based High-Performance Computer RIVYERA, pp. 81–103. Springer New York, New York, NY (2013), doi:10.1007/978-1-4614-1791-0_3

[3] Fröning, H.: Extoll and data movements in heterogeneous computing environments. In: Resch, M.M., Bez, W., Focht, E., Kobayashi, H., Patel, N. (eds.) Sustained Simulation Performance 2014. pp. 127–139. Springer International Publishing, Cham (2015), doi:10.1007/978-3-319-10626-7_11

[4] Belletti, F., Guidetti, M., Maiorano, A., Mantovani, F., Schifano, S., Tripiccione, R., Cotallo, M., Perez-Gaviro, S., Sciretti, D., Velasco, J., Cruz, A., Navarro, D., Tarancon, A., Fernandez, L., Martin-Mayor, V., Munoz-Sudupe, A., Yllanes, D., Gordillo-Guerrero, A., Ruiz-Lorenzo, J., Marinari, E., Parisi, G.,

Rossi, M., Zanier, G.: Janus: An FPGA-based system for high-performance scientific computing. Computing in Science and Engineering 11(1), 48–58 (2009), doi:10.1109/MCSE.2009.11

[5] Baity-Jesi, M., Baños, R., Cruz, A., Fernandez, L., Gil-Narvion, J., Gordillo-Guerrero, A., Iñiguez, D., Maiorano, A., Mantovani, F., Marinari, E., Martin-Mayor, V., Monforte-Garcia, J., Sudupe, A.M., Navarro, D., Parisi, G., Perez-Gaviro, S., Pivanti, M., Ricci-Tersenghi, F., Ruiz-Lorenzo, J., Schifano, S.F., Seoane, B., Tarancon, A., Tripiccione, R., Yllanes, D.: Janus II: A new generation application-driven computer for spin-system simulations. Computer Physics Communications 185(2), 550 – 559 (2014), doi:10.1016/j.cpc.2013.10.019

[6] Vanderbauwhede, W., Benkrid, K.: High-performance computing using FPGAs, vol. 3. Springer (2013), doi:10.1007/978-1-4614-1791-0

[7] Nane, R., Sima, V., Pilato, C., Choi, J., Fort, B., Canis, A., Chen, Y.T., Hsiao, H., Brown, S., Ferrandi, F., Anderson, J., Bertels, K.: A Survey and Evaluation of FPGA High-Level Synthesis Tools. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 35(10), 1591–1604 (Oct 2016), doi:10.1109/TCAD.2015.2513673

[8] Calore, E., Gabbana, A., Kraus, J., Pellegrini, E., Schifano, S.F., Tripiccione, R.: Massively parallel lattice-Boltzmann codes on large GPU clusters. Parallel Computing 58, 1 – 24 (2016), doi:10.1016/j.parco.2016.08.005

[9] Filgueras, A., Gil, E., Alvarez, C., Jimenez, D., Martorell, X., Langer, J., Noguera, J.: Heterogeneous tasking on SMP/FPGA SoCs: The case of OmpSs and the Zynq. In: 2013 IFIP/IEEE 21st International Conference on Very Large Scale Integration (VLSI-SoC). pp. 290–291 (Oct 2013), doi:10.1109/VLSI-SoC.2013.6673293

[10] Bosch, J., Filgueras, A., Vidal, M., Jimenez-Gonzalez, D., Alvarez, C., Martorell, X.: Exploiting parallelism on GPUs and FPGAs with OmpSs. In: Proceedings of the 1st Workshop on AutotuniNg and aDaptivity AppRoaches for Energy efficient HPC Systems. p. 4. ACM (2017), doi:10.1145/3152821.3152880

[11] Pop, A., Cohen, A.: Openstream: Expressiveness and data-flow compilation of openmp streaming programs. ACM Transactions on Architecture and Code Optimization (TACO) 9(4), 53 (2013), doi:10.1145/2400682.2400712

[12] Biferale, L., Mantovani, F., Sbragaglia, M., Scagliarini, A., Toschi, F., Tripiccione, R.: Second-order closure in stratified turbulence: Simulations and modeling of bulk and entrainment regions. Physical Review E 84(1), 016305 (2011), doi:10.1103/PhysRevE.84.016305

[13] Calore, E., Gabbana, A., Schifano, S.F., Tripiccione, R.: Evaluation of DVFS techniques on modern HPC processors and accelerators for energy-aware applications. Concurrency and Computation: Practice and Experience 29(12), 1–19 (2017), doi:10.1002/cpe.4143

[14] Mantovani, F., Pivanti, M., Schifano, S.F., Tripiccione, R.: Performance issues on many-core processors: A D2Q37 Lattice Boltzmann scheme as a test-case. Computers & Fluids 88, 743 – 752 (2013), doi:10.1016/j.compfluid.2013.05.014

[15] Calore, E., Gabbana, A., Kraus, J., Schifano, S.F., Tripiccione, R.: Performance and portability of accelerated lattice Boltzmann applications with OpenACC. Concurrency and Computation: Practice and Experience 28(12), 3485–3502 (2016), doi:10.1002/cpe.3862

[16] Calore, E., Gabbana, A., Schifano, S.F., Tripiccione, R.: Early experience on using knights landing processors for lattice boltzmann applications. In: Parallel Processing and Applied Mathematics: 12th International Conference, PPAM 2017, Lublin, Poland, September 10-13, 2017. Lecture Notes in Computer Science, vol. 1077, pp. 1–12 (2018), doi:10.1007/978-3-319-78024-5_45

[17] Servat, H., Llort, G., Huck, K., Giménez, J., Labarta, J.: Framework for a productive performance optimization. Parallel Computing 39(8), 336–353 (2013), doi:10.1016/j.parco.2013.05.004

[18] Pillet, V., Labarta, J., Cortes, T., Girona, S.: Paraver: A tool to visualize and analyze parallel code. In: Proceedings of WoTUG-18: transputer and occam developments. vol. 44, pp. 17–31 (1995)

[19] Mantovani, F., Calore, E.: Multi-node advanced performance and power analysis with paraver. In: Parallel Computing is Everywhere. Advances in Parallel Computing, vol. 32, pp. 723–732 (2018), doi:10.3233/978-1-61499-843-3-723

[20] Calore, E., Mantovani, F., Ruiz, D.: Advanced Performance Analysis of HPC Workloads on Cavium ThunderX. In: 2018 International Conference on High Performance Computing Simulation (HPCS). pp. 375–382 (July 2018), doi:10.1109/HPCS.2018.00068

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200101

A Processor Architecture for Executing Global Cellular Automata as Software

Christian RISTIG[a,1] and Christian SIEMERS[a]
[a] *Department of Informatics, Clausthal University of Technology, Germany*

Abstract. Cellular automata are a massively parallel programming model that are capable to solve many algorithmic problems efficiently. The complexity of defining a suitable cell rule for a concrete problem can be overcome by the use of the extended model of global cellular automata in conjunction with specialized compilers, to translate a high-level imperative programming language to cellular automata. Obviously, the execution on universal multicore processors does not make use of the full parallel potential of cellular automata and the workflow for direct hardware implementations is slow and hard to debug. In this paper, we propose a novel processor architecture that can execute a global cellular automaton as software and can still compete with other software or hardware implementations.

Keywords. FPGA, Cellular Automata, Processor Architecture, Parallel Processing, Dataflow

1. Introduction

Cellular automata (CA) are a massively parallel model that can easily be implemented in hardware [1]. There exist several application fields for cellular automata, e.g. image processing, machine learning, fluid dynamic or traffic simulation [2]. On the other side, writing a program for a cellular automaton that can solve a given problem is extremely challenging, as a cell has a strict and homogenous neighborhood and there is only one rule for all cells. Mortensen [3] presented a method to compile a high-level imperative programming language to the cellular automata model, so developers can write their algorithms in the usual way they do, and although benefit from the parallel execution. Unfortunately, the resulting automaton is somehow inefficient as information has to be passed from one cell to another over long distances, which is done by a message passing protocol.

To overcome the restrictions of a cellular automaton, the so-called global cellular automata (GCA) have been introduced [4] [5].

Definition. For a given natural number k, a global cellular automaton (GCA) is a 4-tuple (Z, Q, ν, δ) consisting of

- the set of cells Z,
- the neighborhood function $\nu: Z \times Q \to Z^k$,
- the set of cell states Q,
- the state transition function $\delta: Z \times Q \times Q^k \to Q$.

[1] Corresponding Author: Christian Ristig, Department of Informatics, Clausthal University of Technology, Julius-Albert-Str. 4, 38678 Clausthal-Zellerfeld, Germany. E-mail:christian.ristig@tu-clausthal.de

In contrast to a cellular automaton, the neighborhood of a cell does not only depend on the cell itself, but also of its current state. Thus, a cell might have a totally different neighborhood from one generation to another. The number of neighbors is denoted with k and might be constant or variable as well. Furthermore, the definition of the state transition function δ allows the use of individual rules for each cell. While it is easier to implement algorithms in software, an implementation in hardware is more complex because of the varying neighborhood. Several approaches have been presented in [6] [7] [8]. Common to all, there is always a compromise between generality and runtime. Specialized approaches are fast, but running a different algorithm is a complex process. In contrast, more general approaches have a great overhead in space and time.

Drieseberg et al. [9] use global cellular automata and combine them with the approach of Mortensen. They presented a compiler for a subset of the C programming language. It generates a GCA that can be run on a multicore CPU, a GPU or an FPGA. Experimental results show, that a speedup could be achieved compared to a program that was compiled with a standard C compiler and executed on an ordinary CPU. This was especially true for the implementation on an FPGA, which could achieve the highest speedup factor compared to the runtime of the same algorithm on a CPU. However, the development cycle includes the synthesis and place & route process of the hardware description and is very time-consuming. Additionally, the presented compiler creates a huge number of cells, already for small algorithms with a few lines of code. This results in a very slow or even impossible routing [10].

In this paper, we present a hardware architecture that executes global cellular automata written as software. The architecture is generalized and can execute a high number of cells. Nevertheless, it aims to reach high speedup factors compared to the execution of cellular automata on universal hardware. We also propose a mapping process of standard syntax elements of imperative programming languages (such as operators, if and while) to our cellular processor architecture, which results in much fewer cells than the approach of Drieseberg.

2. Hardware Architecture for Execution of Global Cellular Automata

In this section, we present a hardware architecture for a processor that is able to calculate a global cellular automaton $GCA = (Z, Q, v, \delta)$ where

- $Z \subset \mathbb{N}$ is a finite subset of the natural numbers (called the *cell IDs*),
- v is the neighborhood function with $k = 0$ or $k = 1$ depending on the local cell state,
- $Q = \{(pc, a, v, s) | pc \in \mathbb{N}; a, v \in \mathbb{Z}; s \in \mathcal{S}\}$ is the set of all possible cell states,
- δ is the state transition function (called the *rules*).

An element of Q is a 4-tuple where pc is the program counter, a is the accumulator, v the cell value and s the local cell state. The set $\mathcal{S} \coloneqq \mathcal{I} \cup \mathcal{V}$ is composed of the subset of invalid states $\mathcal{I}$, containing the states *Busy*, *LikelyTrue*, *LikelyFalse* and *Reset*, and of the subset of valid states $\mathcal{V}$, containing the states *Ready*, *True*, *False* and *LoopReset*.

In a global cellular automaton, the neighborhood of a cell depends not only of the cell ID, but also of its state. More precisely, it depends on the value of the program counter, which might change in every new generation. The definition of the GCA above

implies that a cell has either one neighbor cell at a given time or no neighbor at all. We agree that if a cell has a neighboring cell, it might use its cell value v or its local cell state s, but not its program counter and accumulator to calculate the cell's new cell state in Q.

The number of cells in Z and the state transition function δ highly depends on the function the cellular automaton computes. For a hardware architecture, we have to limit the number of cells and to define a set of general rules the designer of the function can choose from.

2.1. Ruleset

The ruleset can be divided into four categories: initialization, arithmetic/logic, comparison and control. Every rule may be annotated with a valid flag that has two effects when the rule actually changes the cell's program counter. First, the new value of the accumulator is copied to the cell's value v. Second, if the cells local state s is in $\mathcal{I}$, it is changed to a corresponding state in $\mathcal{V}$ according to Table 1. The local states *Reset* and *LoopReset* have a special meaning and are not affected by the valid flag.

Table 1. Local cell state changes when valid flag is set in a rule

Old State	New State
Busy, Ready	Ready
LikelyTrue, True	True
LikelyFalse, False	False

Initialization rules are used to set a definite value in the cell's accumulator. This value might be a constant (*set* rule) or the cell value $\bar{v}$ of a neighboring cell if its local state $\bar{s}$ is valid (*read* rule). There are also variants of the set and read rules which only set the accumulator when the condition $s \notin \mathcal{V}$ is met (*init* rule), or increase the cells program counter by two instead of one (*skip* rule). A complete list of the initialization rules and their impacts is shown in Table 2. They do not change the cell value or the local cell state if the rule is not annotated with a valid flag.

Table 2. List of initialization rules

Rule	Program Counter	Accumulator
Set	$pc = pc + 1$	$a = c$
Set and Skip	$pc = pc + 2$	$a = c$
Set Init	$pc = pc + 1$	$a = \begin{cases} c, & \mathrm{s} \notin \mathcal{V} \\ a, & \text{else} \end{cases}$
Read	$pc = \begin{cases} pc+1, & \bar{s} \in \mathcal{V} \\ pc, & else \end{cases}$	$a = \begin{cases} \bar{v}, & \bar{s} \in \mathcal{V} \\ a, & else \end{cases}$
Read and Skip	$pc = \begin{cases} pc+2, & \bar{s} \in \mathcal{V} \\ pc, & \text{else} \end{cases}$	$a = \begin{cases} \bar{v}, & \bar{s} \in \mathcal{V} \\ a, & \text{else} \end{cases}$
Read Init	$pc = \begin{cases} pc+1, & \mathrm{s} \in \mathcal{V} \vee \bar{s} \in \mathcal{V} \\ pc, & \text{else} \end{cases}$	$a = \begin{cases} \bar{v}, & \mathrm{s} \notin \mathcal{V} \\ a, & \text{else} \end{cases}$

Arithmetic/logic rules implement unary operators (denoted with $\ominus$) such as negation or binary operators (denoted with $\oplus$) such as addition or subtraction. The first operand is always the accumulator of the cell, the second is either a constant or the value of a neighboring cell. Like initialization rules, they have no impact on the cell value or the local cell state if a valid flag is not present, but all the other effects are presented in Table 3.

Table 3. List of arithmetic/logic rules

Rule	Program Counter	Accumulator
Unary operator	$pc = pc + 1$	$a = \ominus a$
Binary operator (constant)	$pc = pc + 1$	$a = a \oplus c$
Binary operator (neighbor cell)	$pc = \begin{cases} pc+1, & \bar{s} \in \mathcal{V} \\ pc, & else \end{cases}$	$a = \begin{cases} a \oplus \bar{v}, & \bar{s} \in \mathcal{V} \\ a, & else \end{cases}$

Comparison rules compare the accumulator with a constant or the value of another cell, and write the result into the accumulator as stated in Table 4. Afterwards, they also change the local cell state as defined in Eq. 1.

$$s = \begin{cases} LikelyFalse, & a = 0 \wedge s \in \mathcal{I} \\ False, & a = 0 \wedge s \in \mathcal{V} \\ LikelyTrue, & a \neq 0 \wedge s \in \mathcal{I} \\ True, & a \neq 0 \wedge s \in \mathcal{V} \end{cases} \tag{1}$$

Table 4. List of Comparison rules

Rule	Program Counter	Accumulator
Comparison with constant	$pc = pc + 1$	$\mathrm{a} = \begin{cases} 0, & a \preccurlyeq c \\ 1, & a \preccurlyeq c \end{cases}$
Comparison with neighbor	$pc = \begin{cases} pc+1, & \bar{s} \in \mathcal{V} \\ pc, & else \end{cases}$	$\mathrm{a} = \begin{cases} 0, & a \preccurlyeq \bar{v} \wedge \bar{s} \in \mathcal{V} \\ 1, & a \preccurlyeq \bar{v} \wedge \bar{s} \in \mathcal{V}\} \\ a, & \text{else} \end{cases}$

Control rules do not change the cells accumulator, but the program counter as well as the local cell state. The available rules are listed in Table 5 and can be divided into three groups:

- Rules that wait for a neighboring cell to take over a specific local state $\hat{s} \in \mathcal{V}$ (wait rule)
- Rules that increase the program counter by two instead of one when the own local state or the local cell state of a neighbor is *True* (skip rule)
- Rules for halting or resetting a cell

Table 5. List of control rules

Rule	Program Counter	Local Cell State
Wait rule	$pc = \begin{cases} pc+1, & \bar{s} = \hat{s}, \hat{s} \in \mathcal{V} \\ pc, & else \end{cases}$	No change
Skip rule (local)	$pc = \begin{cases} pc+2, & s \in \{LikelyTrue, True\} \\ pc+1, & sonst \end{cases}$	No change
Skip rule (neighbor)	$pc = \begin{cases} pc+2, & \bar{s} = True \\ pc+1, & \bar{s} \in \mathcal{V} \setminus \{True\} \\ pc, & else \end{cases}$	No change
Reset rule (local)	$pc = \begin{cases} 0, & s = Reset \\ pc, & else \end{cases}$	$s = \begin{cases} Busy, & s = Reset \\ Reset, & else \end{cases}$
Reset rule (neighbor)	$pc = \begin{cases} 0, & s = Reset \\ pc, & else \end{cases}$	$s = \begin{cases} Reset, & \bar{s} = Reset \\ Busy, & s = Reset \\ s, & else \end{cases}$
Loop (reset) rule	$pc = \begin{cases} 0, & \bar{s} = Reset \\ pc, & else \end{cases}$	$s = \begin{cases} Ready, & \bar{s} = Reset \\ LoopReset, & else \end{cases}$
Halt rule	No change	No change

2.2. Processor Architecture

The processor is built of several so-called *cell compute units* (CCUs) and an on-chip-network for interconnection, as presented in Figure 1. There is also a *control unit* (CU) connected to the network that has access to an external main memory. The CU is responsible for loading the individual cell rules from main memory into the CCUs before the automaton can be run. It also monitors the state of the automaton and signals the completion of work when all cells become inactive, thus there are no more state changes from one generation to another.

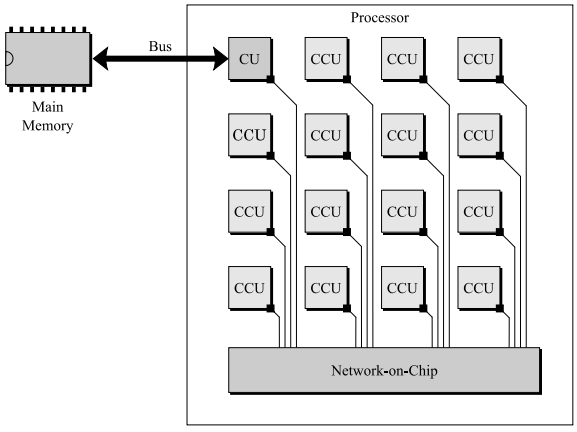


Figure 1. Overall processor architecture

A cell compute unit is designed for minimal hardware consumption. It uses a 1-operand-machine architecture, as presented in Figure 2, which is sufficient to execute a cell rule, because a rule has at most one operand that might be a constant (also: *immediate*) value or the value of a neighboring cell. Each CCU has four registers to store

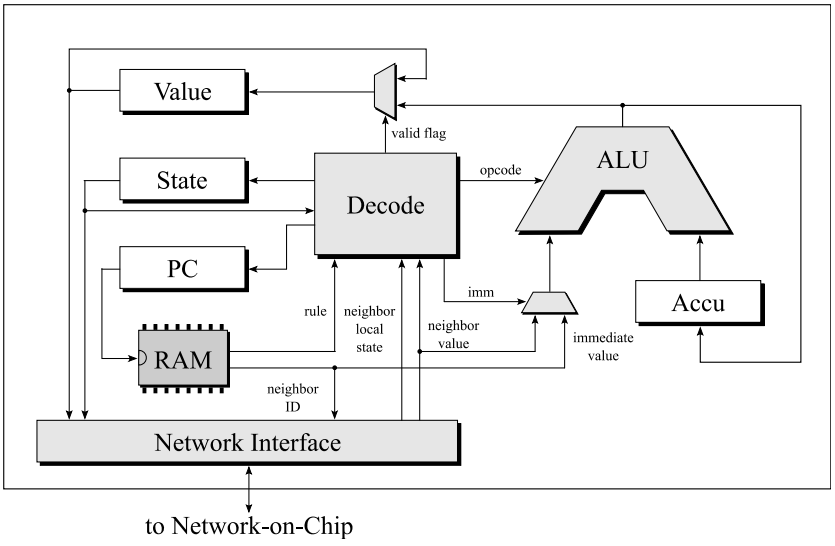


Figure 2. CCU architecture in details

the value of the cell's state tuple (pc, a, v, s). The program counter addresses a small memory belonging to the cell which is capable to hold a couple of rules and their operands. If a cell has a neighbor in the current generation, the ID of the neighbor is routed to the network-on-chip to request its local state and value. The cell's own local state and value are always provided to the network as well. A decoder unit decodes the rule and generates following control signals:

- an opcode for an arithmetic/logical unit (ALU)
- a signal to select the correct operand (imm)
- a signal to take over the ALU result as new cell value (valid flag)
- the next program counter
- the new local cell state

2.3. Mapping imperative programming languages to GCA ruleset

As already mentioned, writing a program for a cellular automaton is different than writing it in a programming language like C. Therefore, it is desirable to have a mapping algorithm from an imperative programming language to the ruleset of the presented hardware architecture. Based on the ideas of Drieseberg and Mortensen we propose the following procedure:

1. Transform the imperative program into static single assignment (SSA) form (as described in [11])
2. Build a dataflow graph of the transformed program (see example in Figure 3)
3. Replace each node in the dataflow graph with a cell and create the rules for each created cell
4. Optimize the resulting *cell graph* to reduce the number of cells

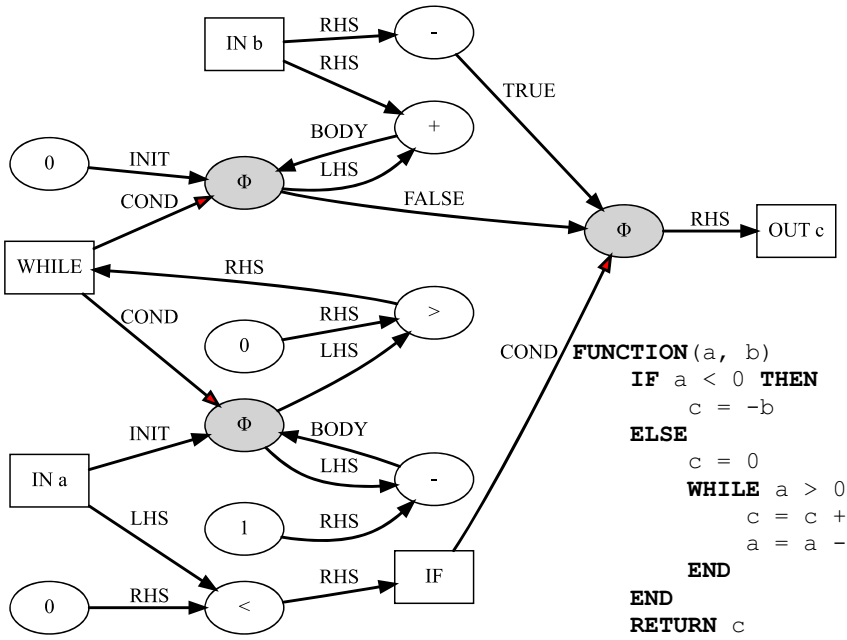


Figure 3. Example of a dataflow graph and the corresponding program in pseudocode

Step 1 includes several sub-procedures, like parsing the source code or building the control flow graph and (abstract) syntax trees, before the SSA form can be generated. Step 2 is very simple once the program is in SSA form. Additional optimizations (for instance removing unused variables, balancing expressions, etc.) could take place before. An example dataflow graph is presented in Figure 3. It consists of nine types of nodes that are generated from the keywords and literals in the original program code. The edges of the graph show the actual flow of data, beginning from the input nodes (IN) to the output nodes (OUT). They are annotated with additional information, so that the target node can handle the incoming data correctly. For example, a binary operator node needs to know the order of its operand, whereat LHS (left-hand side) denotes the left operand and RHS (right-hand side) the right one. Yet another example is the phi-nodes that belong to the while-node: they require the loop condition (COND) to decide if they have to choose the initial value, that is valid *before* the loop body is executed (INIT), or the value computed *in* the loop body (BODY). In step 3, the nodes of the dataflow graph are replaced with cells according to Figure 4. Every generated cell is assigned an ID that is used by other cells in their cell rules to access the cell's state. The cell rules are abbreviated with a self-explanatory mnemonic. Mnemonics written in bold font have the

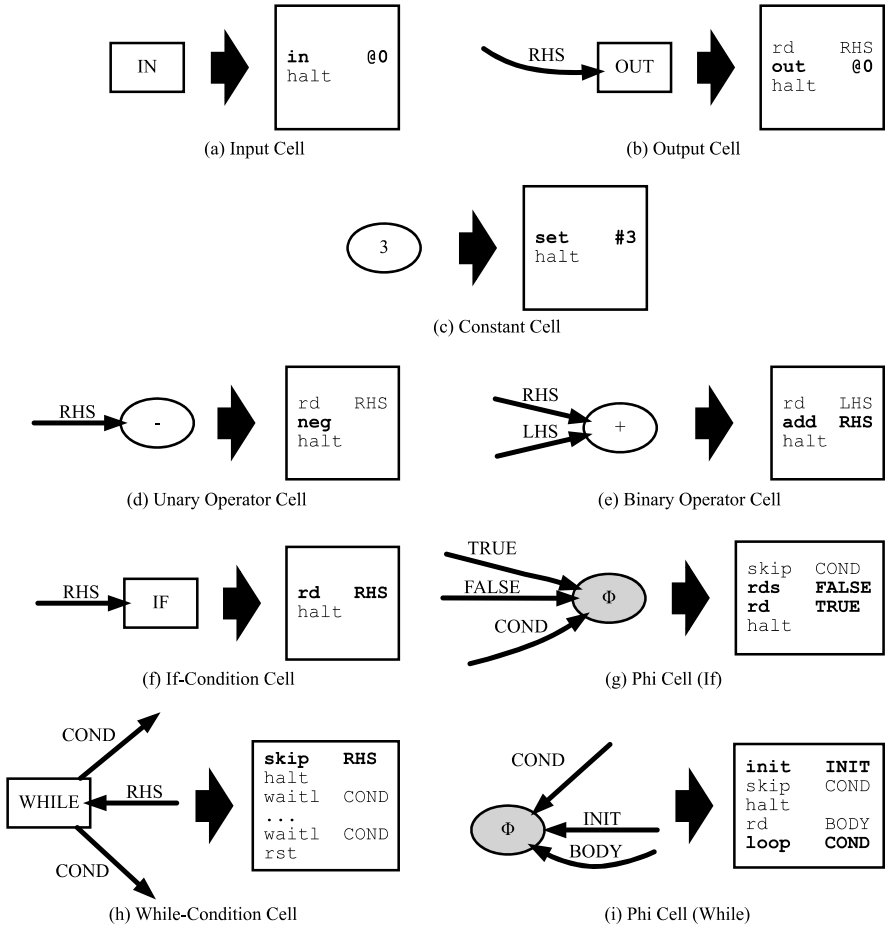


Figure 4. Translation of dataflow graph nodes to cells

valid flag set. Input and output cells have a special @i-operand that denotes to the i-th argument or return value, respectively. Numbers with a leading # symbol denote a literal value; without it they represent the ID of another cell. In some situations, an additional wait-rule must be prepended to a cell. This is when it wants to read from a phi-cell of a while loop outside of the loop body. Because the value of such a phi-cell is always valid during the execution of the loop, the cell outside must wait for the loop condition to be false (see Figure 5). Step 4 tries to eliminate cells by inserting their rules into another one. For example, literal nodes are easy to integrate into another cell, just by replacing a read access with a literal value. In Figure 5, the optimization process has been executed for the illustrated algorithm.

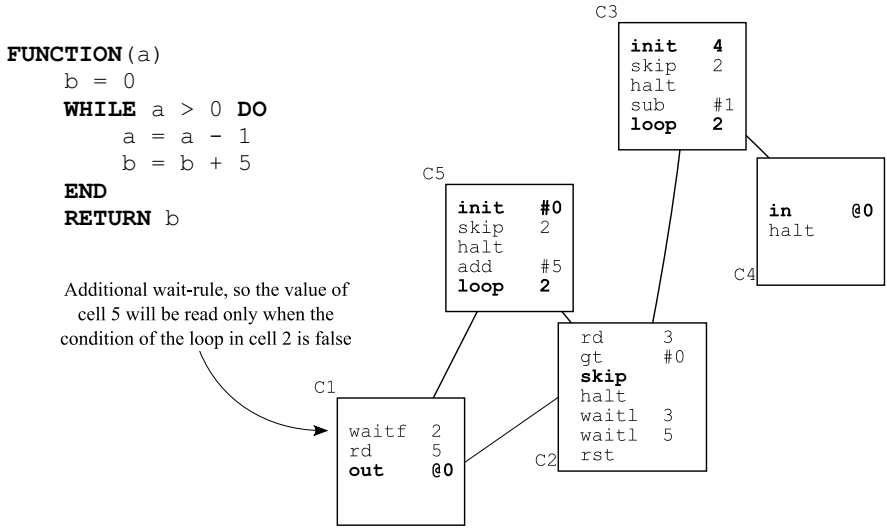


Figure 5. Optimized cell graph for an algorithm

3. Results

The presented processor architecture has been implemented in VHDL and synthesized with the Xilinx Vivado toolchain. A single cell compute unit with a 32-bit word size, including a multiplier unit using DSP slices, occupies around 480 lookup tables and 100 flip-flops. In a high-end 7-series Virtex FPGA with 2 million logic cells (type XC7V2000T) this corresponds to only 0.04% of logic consumed, so hundreds of CCUs can be integrated into a single FPGA. The local cell memory is implemented as LUT-RAM and uses 40 of the total 480 lookup tables (32 for the operand and 8 for the encoded rule). Depending on the FPGA technology, the possible number of rules per CCU varies. For a 6-input LUT this means 64 rules per cell.

For our experiments, we used a crossbar as network-on-chip. Although this network architecture uses a lot of FPGA logic (large multiplexer structures are needed for each single data bit), it has a low latency and is easy to implement. There exist lots more network architectures that might be suitable for the processor architecture (a good survey can be found in [12]), especially the so-called Benes-network is an appropriate candidate [13], as it is collision-free and resource-efficient. The complex routing algorithm can also be implemented in hardware [14].

The first algorithm we analyzed consist of a matrix multiplication of two 3x3 matrices. This operation has a high degree of parallelism as all 9 elements of the resulting matrix can be computed concurrently. Even the three multiplications needed to compute an element can run in parallel. The corresponding global cellular automaton has a total of 38 cells and was executed on a system with a maximum of 63 cells. Simulation shows that it took only five generations to compute the result, as illustrated in Figure 6. Cells highlighted with a red border did change their state in the current generation, and yellow cells are in a valid local state.

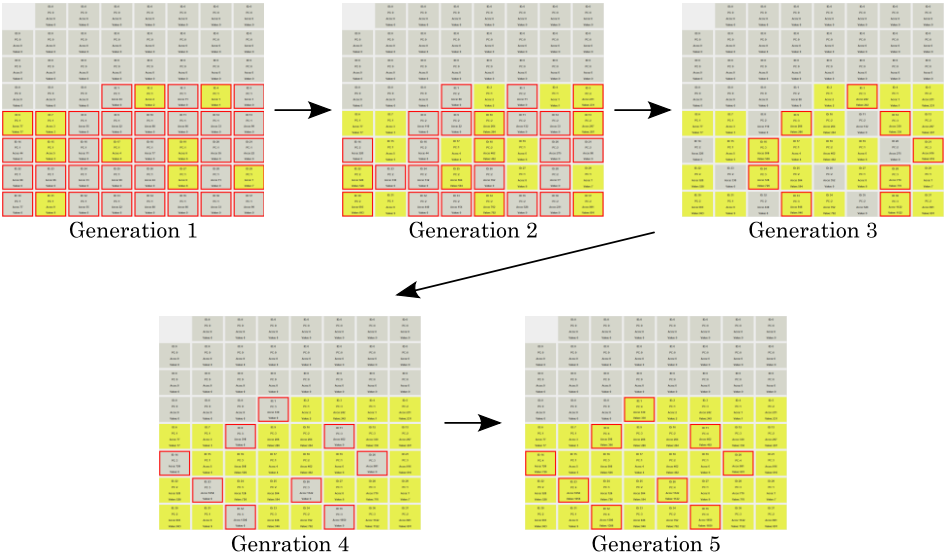


Figure 6. Execution of a matrix multiplication

Another algorithm we implemented was Stein's algorithm [15], the binary version of the *greatest common divisor (GCD)*, and compared the results with the FPGA implementation of Drieseberg et al. Their GCD global cellular automaton consists of 258 cells, whereas our optimized automaton consists of only 24 cells, which is less than 10%. We simulated the execution of the algorithm with the two input values 3528 and 3780. It took 217 generations to calculate the result of 252. This is also less, and only a third, of the number of generations compared to Drieseberg et al. It must be pointed out, that the number of generations is only an indicator to the actual execution time, as our simulation does not consider any hardware properties. Thus, the achievable clock frequency of the proposed processor architecture might be much slower due to propagation delays. Further research has to be done here. If a generation needs one clock cycle to compute and we assume a clock frequency of only 50 MHz, our automaton would need around 4 ms to calculate the GCD. Indeed, this is much slower than the FPGA implementation of Drieseberg et al. but still faster than the CPU and GPU variants.

4. Conclusion

In this paper, we presented a new processor architecture that is able to execute global cellular automata with a specialized ruleset as a software program. The ruleset is

designed in such a way that algorithms written in an imperative programming language can easily be mapped to a global cellular automaton with a low number of cells needed. The automation of this process is one of our next steps. In first experiments, we confirmed that the presented architecture can achieve faster execution times than a software implementation on a universal processor. This advantage outweighs even more, if the executed algorithm has a high degree of parallelism. Nevertheless, further research has to be done with real-world applications to fully prove the advantages of the architecture. Furthermore, the processor has to be implemented and evaluated on an FPGA to analyze the scalability of the architecture and to obtain actual execution times.

References

[1] M. Dascalu, "Cellular Automata Hardware Implementations - an Overview," *Romanian Journal of Information, Science and Technology,* 2016.

[2] A. C. Lima and J. C. Ferreira, "Automatic Generation of Cellular Automata on FPGA," *IX Jornadas sobre Sistemas Reconfiguráveis,* Februar 2013.

[3] M. Mortensen, High Level Parallel Programming Language Compiling to a Cellular Automata Processing Model, Aarhus: Master's thesis, Aarhus Universitet, 2007.

[4] R. Hoffmann, K. P. Völkmann and S. Waldschmidt, "Global Cellular Automata GCA: An Universal Extension of the CA Model," *ACRI 2000 "work in progress" session, Karlsruhe, Germany, Oct. 4th - 6th,* 2000.

[5] R. Hoffmann, K. P. Völkmann, S. Waldschmidt and W. Heenes, "GCA: Global Cellular Automata. A Flexible Parallel Model," *Parallel Computing Technologies, 6th International Conference, PaCT 2001, Novosibirsk, Russia, September 3-7, 2001, Proceedings,* 2001.

[6] R. Hoffmann, W. Heenes and M. Halbach, "Implementation of the Massively Parallel Model GCA," *Parallel Computing in Electrical Engineering,* 2004.

[7] W. Heenes, R. Hoffmann and J. Jendrsczok, "A Multiprocessor Architecture for the Massively Parallel Model GCA," *Parallel and Distributed Processing Symposium,* 2006.

[8] C. Schäck, W. Heenes and R. Hoffmann, "A Multiprocessor Architecture with an Omega Network for the Massively Parallel Model GCA," in *Lecture Notes in Computer Science (LNCS, volume 5657)*, Berlin, Heidelberg, Springer, 2009.

[9] J. Drieseberg and C. Siemers, "C to Cellular Automata and execution on CPU, GPU and FPGA," *International Conference on High Performance Computing & Simulation (HPCS),* pp. 216-222, 2012.

[10] J. Drieseberg, Abbildung sequentieller C-Programme auf parallel arbeitende Zellularautomaten, Clausthal-Zellerfeld: Dissertation, Technische Universität Clausthal, 2016.

[11] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," in *ACM Transactions on Programming Languages and Systems, Vol 13, No 4*, New York, 1991.

[12] T. Schwederski and M. Jurczyk, Verbindungsnetze: Strukturen und Eigenschaften, Stuttgart: Teubner, 1996.

[13] S. C. Stilkerich, C. Siemers and C. Ristig, "Appropriate Multi-core Architecture for Safety-critical Aerospace Applications - Certifiable Real-time Switching Network," *Proceedings of the 4th International Conference on Pervasive and Embedded Computing and Communication Systems - Volume 1: PECCS,* pp. 180-185, 2014.

[14] S. Aust, ConPar - Ein Echtzeitparallelrechner zur Rezentralisierung von Steuergeräten im Automobil, Clausthal-Zellefeld: Institut für Informatik TU Clausthal, 2013.

[15] J. Stein, "Computational problems associated with Racah algebra," *Journal of Computational Physics, Volume 1, Issue 3,* pp. 397-405, 1967.

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200102

Crossbar Implementation with Partial Reconfiguration for Stream Switching Applications on an FPGA

Yuichi KAWAMATA [a], Tomohiro KIDA [a], Yuichiro SHIBATA [a] and Kentaro SANO [b]
[a] *Graduate School of Engineering, Nagasaki University, Japan*
[b] *Riken Center for Computational Science, Japan*

Abstract. In this paper, we propose a network crossbar implementation using partial reconfiguration of an FPGA in a multi-FPGA cluster computing system. With a proposed framework, inter-FPGA network routing can be changed by reconfiguring the crossbar module by a partial reconfiguration mechanism. The purpose of this paper is to compare ordinary crossbar circuits and partial reconfiguration crossbar circuits, in terms of resource usage and the maximum operating frequency. As a result, by using partial reconfiguration, the maximum operating frequency is improved by 1.6 times while reducing required ALM resources by 13%, a proper bus sizes for a crossbar are selected.

Keywords. FPGA, Partial Reconfiguration, crossbar, Arria10

1. Introduction

Field Programmable Gate Arrays (FPGAs) have been attracting attention as a platform of a power-efficient custom computing because FPGAs can construct optimal data paths according to each application. Especially in recent years, with the improvement of the integration degree, FPGAs have been equipped with floating-point arithmetic cores and have devised wiring architecture to improve the operating frequency. Expectations are rising for applications in the high performance computing field [1][2].

Stencil calculation, which repeatedly applies arithmetic processing with data references of the same shape to data arranged in a grid, is a common design pattern used in various scientific calculations, and it is known that an FPGA-based system able to work efficiently in a streamwise framework [3][4][5]. Also, by connecting the operation pipelines in series and increasing the number of operations per memory access, it is able to improve the operation performance without increasing memory bandwidth required for DRAM[6]. Therefore, even in constructing a parallel system in which multiple FPGAs are interconnected, it is promising because the performance can be scaled without being restricted by the connection bandwidth between the FPGAs [7].

In order to build a multi-FPGA computing system, a communication mechanism is needed to exchange data between FPGAs. Many research projects have been carried out to extend the switch mechanism for network on chip (NoC) used for inter-module communication inside the chip, and to connect FPGAs [8] [9] [10] [11] [12]. These NoC-

derived switch mechanisms are mainly based on packet-based routing and have excellent flexibility. On the other hand, in multi-FPGA cluster computing systems that perform stream-based processing, depending on the application, it is not always necessarily required to route input packets to different destinations in a short time, as shown in the left figure of Figure 1. As shown in the right figure, it just needs to support routing as a stream for some amount of data.

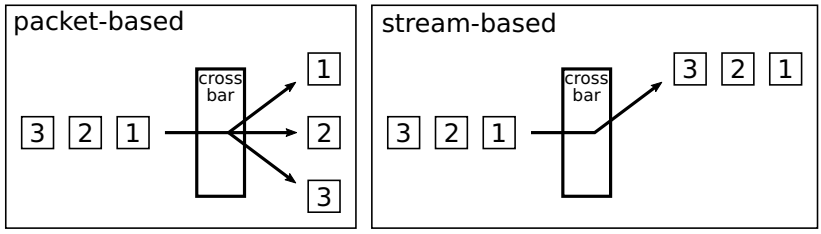


Figure 1. Comparison of routing

Since a general-purpose crossbar that can route any packets to any destinations is implemented in FPGAs as a set of multiplexers, it is thought that implementation efficiency decreases in terms of resource usage and frequency as the number of input / output bits increases. However, it is inflexible that only fixed routing can be performed for each application. That is, there is a trade-off between flexibility and performance / efficiency in crossbars for multi-FPGA systems, and there can be various design options.

In this paper, we propose a stream-based network crossbar that uses partial reconfiguration technology of FPGAs for path switching. Partial reconfiguration is a technology to change a specific circuit of FPGAs while other circuits in operation. Crossbars that use partial reconfiguration are expected to reduce resource usage and improve the maximum operating frequency, although their dynamic flexibility is limited compared to conventional general-purpose crossbars. In order to clarify these trade-off relationships and to evaluate the effect of partial reconfiguration for the crossbar, the conventional crossbar circuit is compared with the partial-reconfiguration-based crossbar in terms of resource usage and the maximum operating frequency, and the reconfiguration time required for partial reconfiguration is also evaluated. In this paper, assuming a multi-FPGA system with a two-dimensional torus as shown in Figure 2, we evaluate and verify a crossbar with a total of 5 inputs including 4 external inputs, and 1 internal input and 5 outputs as shown in Figure 3.

2. Partial Reconfiguration under Intel Environment

In this work, Partial Reconfiguration (PR) is performed with Intel FPGA Arria 10. In the experiment, JTAG is used to transfer configuration data and partial reconfiguration data. We use Intel's Quartus Prime Version 18.1 Pro Edition as a development environment.

2.1. Design procedure for partial reconfiguration design

The design procedure of the partial reconfiguration circuit in the Intel FPGA is as follows [13].

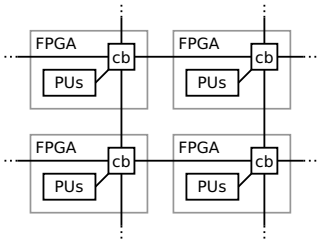


Figure 2. 2D torus consisting of FPGA

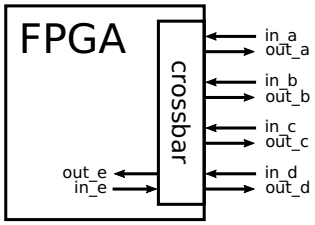


Figure 3. Example of stream connection

1. Circuit design and description
2. Creating Design Partition and LogicLock region
3. Allocating Placement region and Routing region
4. Adding the Partial Reconfiguration Controller IP Core
5. Creating Revisions
6. Compiling the Base Revision
7. Generating a qdb file
8. Compiling each persona

Details of the above procedures are described in the following.

2.1.1. Creating Design Partition and LogicLock region

Design Partition is created from partially reconfigured modules. From the created Design Partition, Quartus's LogicLock function [14] is used to fix the placement of the partially reconfigured modules. This fixed area is called LogicLock region. Since the designated modules are placed and routed only at the designated locations, the degree of freedom of placement and routing is reduced in the partial reconfiguration module, so the maximum operating frequency may be reduced compared to the case where the placement is not fixed. In this paper, we create LogicLock region and fix the crossbar module. And this design is compared with the ordinarily designed circuit and the partial reconfiguration circuit.

2.1.2. Allocating Placement region and Routing region

The LogicLock region has Placement region and Routing region. So we fix the location and size of the Placement region and the Routing region. The Placement region is a region for modules to be partially reconfigured, and the Routing region is a region for arranging paths connecting to the Placement region.

2.1.3. Adding the Partial Reconfiguration Controller IP Core

PR control mechanism creates and uses IP Core in Quartus. When performing partial reconfiguration, only one PR Controller IP is required on the FPGA [15]. The interface of PR Controller IP is shown in Figure 4.

In Figure 4 *nreset* is the asynchronous reset signal input for the PR Controller IP Core. *clk* is a clock for PR Controller IP Core, supporting up to 100 MHz. It begins a partial reconfiguration event when the *pr_start* signal changes from 0 to 1. It receives the next *pr_start* signal only when the *freeze* signal is low (0). It inputs configuration data

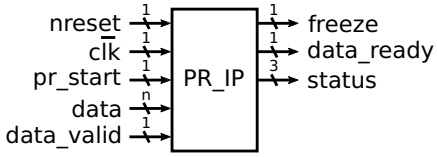


Figure 4. PR Controller IP

for partial reconfiguration to the *data* port. Data width can be selected from 1, 8, 16, and 32 bits. *data_valid* indicates that valid data has been input to the *data* port. *freeze* outputs a high (1) signal during partial reconfiguration. *data_ready* indicates that the *data* port is ready to receive data. *status* is a 3-bit error output indicating the status of partial reconfiguration event. When performs partial reconfiguration via the JTAG interface, the PR Controller IP exchanges signals with the JTAG interface, so insertion of *clk*, *pr_start*, *data*, *data_valid*, and *data_ready* values into these is ignored.

2.1.4. Compiling Revisions and Personas

In the partial reconfiguration design flow, Quartus uses project revision format. There are two versions, Base and Persona Implementation. The Base revision is designed for the entire circuit, and the Persona Implementation revision is designed for the partially reconfigured module. Usually there are only one Base revision and multiple Persona Implementation revisions. The Base revision is compiled first. This compilation operation includes logic synthesis, placement and routing, timing analysis, configuration data generation, etc. The compiled base revision is written out into a qdb database file. The functional module to be partially reconfigured is called persona. Partial reconfiguration data using the qdb file created in the previous step is generated.

3. Evaluated Implementation

In this paper, in addition to the usual design, the LogicLock (hereinafter LL) circuit with fixed area for crossbar module and Partial Reconfiguration (hereinafter PR) circuit which partially reconfigures the crossbar module were created, as shown in Figure 5.

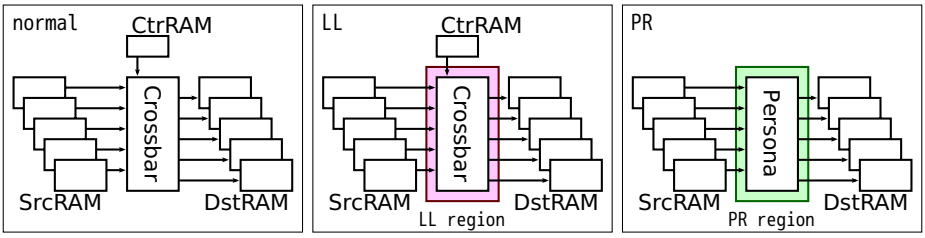


Figure 5. Outline of each circuit

In the FPGA design created this time, stream data is generated from SrcRAM, which is M20K Embedded Memory, and stream data is stored in DstRAM through the crossbar. Also, the crossbar is controlled from CtrRAM.

All crossbar modules other than PR circuit mount full crossbars. The bit widths for stream data of 8 bits, 32 bits, 64 bits, 256 bits, 1024 bits and 4096 bits are evaluated.

3.1. Implementation of full crossbar

The implementation of the full crossbar is shown in Figure 6. It has 5 inputs and 5

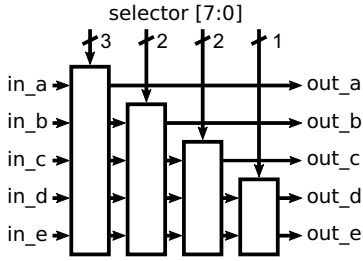


Figure 6. Implementation of full crossbar

outputs, and connection of the inputs and outputs is changed by the 8 bit selection line. The upper 3 bits select 1 output from the leftmost crossbar, and the other outputs are connected to the crossbar on the right without changing the order. In this way, 8-bit connection lines are divided into 3 bits, 2 bits, 2 bits and 1 bit, used as selection lines for each crossbar. In this structure, multiple inputs cannot be connected to a same output.

3.2. persona

Three personas (ST, RT, and X) shown in Figure 7 are evaluated. Since the PR crossbar

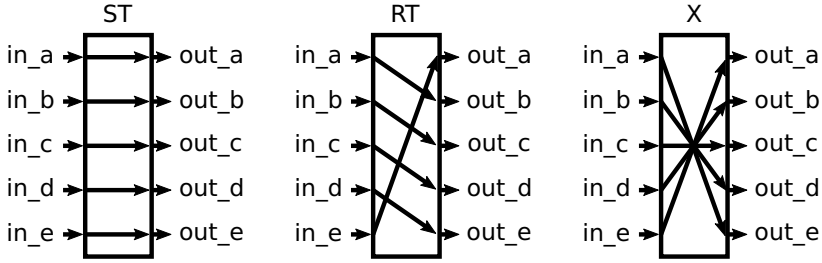


Figure 7. Implemented personas

circuit changes the routing by exchanging the persona, the selection line for crossbar control is not necessary. Therefore, the selection line is not implemented in the PR circuit.

In persona ST, inputs and outputs are connected in the same order. Applying this persona in all FPGAs will make each FPGA independent. In persona RT, the inputs are connected to the output next to that in persona ST. The bottom input is connected to the top output. Considering Figure 2, a system as shown in Figure 8 can be configured. In persona X, the inputs and outputs are connected in reverse order.

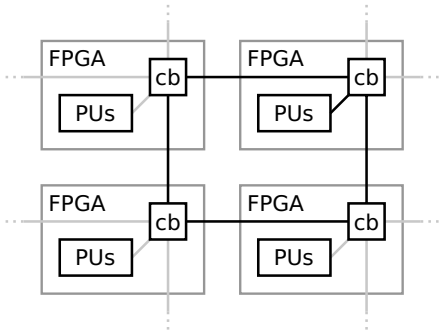


Figure 8. Example of Persona RT routing

3.3. Embedded Memory

We used M20K to store bit stream data. By using M20K as single port RAM, In-System Memory Content Editor can be used to read from / write to the RAM. In the evaluation experiments, the circuit operation was verified by writing data to SrcRAM and reading data from DstRAM via In-System Memory Content Editor. The number of words is set to 8 for all the crossbar designs.

3.4. Reconfiguration area

In the evaluation, the LL circuit and PR circuit with the same bus bits were implemented in the same area position, with setting the same area size. The LogicLock region and the partial reconfiguration area were set as shown in Table 1. The area size was set to

Table 1. Comparison of each area

	Width	Height	Origin
8bit LL circuit	10	10	X88_Y8
8bit PR circuit	10	10	X88_Y8
32bit LL circuit	10	10	X88_Y8
32bit PR circuit	10	10	X88_Y8
64bit LL circuit	10	10	X88_Y8
64bit PR circuit	10	10	X88_Y8
256bit LL circuit	10	30	X88_Y8
256bit PR circuit	10	30	X88_Y8
1024bit LL circuit	10	110	X88_Y8
1024bit PR circuit	10	110	X88_Y8
4096bit LL circuit	28	210	X35_Y11
4096bit PR circuit	28	210	X35_Y11

10×10 for circuits up to 64 bits. For 256 bits and 1024 bits, since it was not possible to implement in 10×10 area, the area was expanded in the Y direction. For 4096 bits, the area was expanded in both X and Y directions, and the position of the reference point (Origin) was also changed so that the area could be set.

4. Evaluation and Consideration

We evaluate and discuss the above mentioned circuits. The 1024 bit RT, 4096 bit RT and 4096 bit X can not be implemented because they can not be placed and routed. The evaluation environment is shown below.

- FPGA : Intel Arria10 10AX115N2F45E1SG FPGA
- CPU : Intel Core(TM) i7-8700K
- MEM : DDR4 16GB
- OS : CentOS 7.5.

4.1. Maximum operating frequency

Figure 9 and Table 2 show the maximum operating frequency of the circuit evaluated this time.

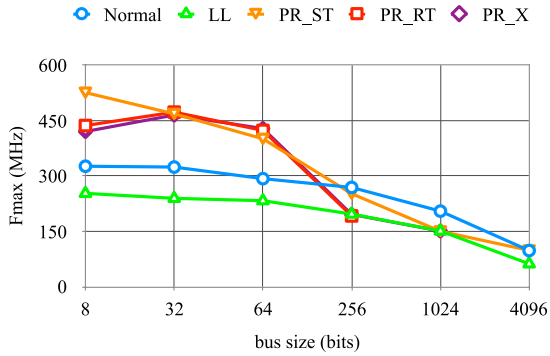


Figure 9. Maximum operating frequency comparison

Table 2. Maximum operating frequency (MHz)

bus size (bits)	8	32	64	256	1024	4096
normal circuit	326.26	324.04	292.65	268.89	204.79	98.12
LL circuit	252.91	239.41	233.05	196.89	151.01	62.25
PR circuit ST	525.49	467.63	400.48	251.95	150.26	98.73
PR circuit RT	436.30	472.59	423.19	191.86	-	-
PR circuit X	420.17	464.68	428.27	196.00	150.99	-

For 64 bits or less, the crossbar implemented by PR results in a higher maximum operating frequency than the normal circuit. On the other hand, for 256 bits or more, the frequency is slower than the normal circuit. The LL circuits are slower than the normal circuit for all the bits. A different PR circuit persona achieves a different maximum operating frequency, the order of the achieved frequency is also different depending on bit numbers. This is because a sufficient area can be allocated to the PR region for the designs up to 64 bits PR, and a high degree of freedom for routing is kept. For larger size designs, the freedom for routing is limited and thus the maximum frequency is degraded. Moreover, since the place where the data streams can access to the RAM, is fixed, the design without RAM can result in higher maximum operating frequency.

4.2. Amount of resource used

A comparison of the ALM usage of the crossbar modules evaluated this time is shown in Figure 10 and Table 3. Figure 10 plots the relative usage which is normalized to the

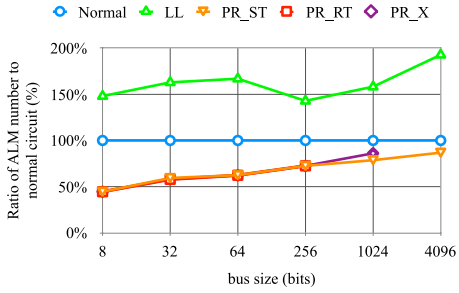


Figure 10. Comparison of ALM useage

Table 3. Resource usage of crossbar module (ALMs)

bus size(bits)	8	32	64	256	1024	4096
normal circuit	101	276	514	1911	6511	26686
LL circuit	150	450	858	2729	10300	51421
PR circuit ST	45	164	321	1386	5122	23168
PR circuit RT	45	160	321	1386	-	-
PR circuit X	45	160	320	1386	5605	-

amount of ALMs used in the normal circuit. The PR circuit was smaller than the normal circuit for all bit numbers, and the LL circuit was larger than the normal circuit. In the ALM usage for the PR crossbar module is about 45% of that for the normal circuit when the bus size is 8 bits. This ratio becomes higher as the bus size increases, and reaches approximately 87% at 4096 bits. On the other hand, differences in ALM usages due to change of personas for PR circuits are relative low, since there is only a difference in connection between inputs and outputs in the PR circuit.

Table 4 shows the amount of resources used for the entire FPGA design for evaluated designs. Even though the PR circuit includes a PR controller IP circuit in addition to

Table 4. Amount to Total resource use

	ALM	Register	RAM
32bit normal circuit	1138	1138	21
32bit LL circuit	1307	1149	21
32bit PR circuit ST	988	1109	20
4096bit normal circuit	47950	42961	2051
4096bit LL circuit	73038	42361	2051
4096bit PR circuit ST	41706	42912	2050

the normal circuit, the PR circuit is smaller than the normal circuit in terms of every resource. The usage of ALM is about 87% of the normal circuit when the bus size is 32 bits and 4096 bits.

4.3. Generated file size

Table 5 compares bit stream file sizes fort the evaluated designs. The sof (sram object

Table 5. File size (MB) and reconfiguration time (sec)

	sof	rbf : ST	rbf : RT	rbf : X	Configuration	PR :ST	PR : RT	PR : X
8bit normal circuit	36	-	-	-	21.24	-	-	-
8bit LL circuit	36	-	-	-	21.02	-	-	-
8bit PR circuit	36	5.9	5.9	5.9	21.12	7.68	7.69	7.62
32bit normal circuit	36	-	-	-	20.99	-	-	-
32bit LL circuit	36	-	-	-	21.08	-	-	-
32bit PR circuit	36	6.1	6.1	6.1	21.12	7.93	8.23	7.70
64bit normal circuit	36	-	-	-	21.05	-	-	-
64bit LL circuit	36	-	-	-	21.21	-	-	-
64bit PR circuit	36	6.1	6.1	6.1	21.30	7.95	7.86	7.84
256bit normal circuit	36	-	-	-	21.18	-	-	-
256bit LL circuit	36	-	-	-	21.28	-	-	-
256bit PR circuit	36	17	16	17	21.26	18.20	19.00	19.06
1024bit normal circuit	36	-	-	-	21.25	-	-	-
1024bit LL circuit	36	-	-	-	21.28	-	-	-
1024bit PR circuit	36	54	-	54	21.37	56.65	-	56.46
4096bit normal circuit	36	-	-	-	21.36	-	-	-
4096bit LL circuit	36	-	-	-	21.36	-	-	-
4096bit PR circuit	36	114	-	-	21.36	163.34	-	-

file) is used for entire configuration, and the rbf (raw binary file) is used for partial reconfiguration. Therefore, rbf is not generated for normal circuits and LL circuits. The rbf file sizes for designs of 256 bits or more are larger than those for the designs up to 64 bits, probably due to a larger PR region. Even with the same area, the file size is larger for 32 bits and 64 bits compared to the design with 8 bits. Moreover, difference in rbf file size was shown due to change of personas even for the same bus size. On the other hand, all designs have the same size of sof. This evaluation results mean that crossbars with a wide bus size require a large area and large on-chip memory capacity to store the bit stream data.

4.4. Reconfiguration time

A comparison of the reconfiguration time via JTAG interface is shown in Table 5. The data in the table were obtained as an average of 5 measurement results. The measured values include not only circuit reconfiguration time but also a startup overhead of the Quartust tool. There is no significant difference in the configuration time as well as sof size. On the other hand, in partial reconfiguration, the increase in the reconfiguration time was observed after 256 bits, where the rbf size is large. When the bus size is 64 bits, the partial reconfiguration time is about 37% of the normal configuration time, and it increases to 86%, 265%, and 765% when it is 256 bits, 1024 bits, and 4096 bits, respectively.

Faster partial reconfiguration is possible by using internal memory. If partial reconfiguration is performed at 3.2 Gbps, which is the theoretical maximum performance of PR controller IP, it can be estimated that the time required to change a 5.9 MB of an 8-bit crossbar module persona is about 15 milliseconds. In addition, the 4096-bit crossbar persona, which requires the largest rbf size of 115 MB among the evaluated designs in this time, can be theoretically reconfigured in about 285 milliseconds.

5. Conclusion

In this paper, we described a crossbar for stream data using partial reconfiguration. The number of required ALMs can be reduced by 13% when the bus size is 32 bits and 4096 bits, compared to a usual full crossbar. This means that partial reconfiguration is effective for reducing the resources used in crossbar design. So it can be said that it is effective when you want to reduce the resources used in the design. The partial reconfiguration also improves the maximum operating frequency when a sufficient partial reconfiguration area is allocated for routing. For example, the maximum operating frequency is improved by 1.6 times for an 8-bit crossbar design. However, for wider bus size such as 256 bits, the maximum operating frequency degraded compared to the normal circuit. Partial reconfiguration with JTAG interface took several seconds in this experiment. Therefore, this approach is not suitable for applications that frequently change the routing. One of our important future work is to verify high-speed partial reconfiguration from internal memory.

References

[1] Kentaro Sano and Satoru Yamamoto. FPGA-Based Scalable and Power-Efficient Fluid Simulation using Floating-Point DSP Blocks. *IEEE Transactions on Parallel and Distributed Systems*, 28:2823–2837, 2017.

[2] Czajkowski, Tomasz and Aydonat, Utku and Denisenko, Dmitry and Freeman, John and Kinsner, Michael and Neto, David and Wong, Jason and Yiannacouras, Peter and P. Singh, Deshanand. From OpenCL to high-performance hardware on FPGAs. *Proceedings - 22nd International Conference on Field Programmable Logic and Applications, FPL 2012*, pages 531–534, 08 2012.

[3] Kentaro Sano. FPGA-Based Systolic Computational-Memory Array for Scalable Stencil Computations. In *High-Performance Computing Using FPGAs*, pages 279–303, 2013.

[4] Yukinori Sato, Yasushi Inoguchi, Wayne Luk, and Tadao Nakamura. Evaluating reconfigurable dataflow computing using the Himeno benchmark. In *Proc. ReConFig*, pages 1–7, 2012.

[5] Heiner Giefers, Christian Plessl, and Jens Förstner. Accelerating Finite Difference Time Domain Simulations with Reconfigurable Dataflow Computers. In *Proc. HEART*, pages 33–38, 2013.

[6] Keisuke Dohi, Koji Okina, Rie Soejima, Yuichiro Shibata, and Kiyoshi Oguri. Performance modeling of stencil computing on a stream-based FPGA accelerator for efficient d esign space exploration. *IEICE Transactions on Information and Systems*, 98–D(2):298–308, 2015.

[7] Kentaro Sano, Yoshiaki Hatsuda, and Satoru Yamamoto. Multi-fpga accelerator for scalable stencil computation with constant memory bandwidth. *Parallel and Distributed Systems, IEEE Transactions on*, 25:695–705, 03 2014.

[8] Sagar Latti. FPGA Implementation of Four Port Router for Network on Chip. *International Research Journal of Engineering and Technology (IRJET)*, 03:887–880, 2016.

[9] Andreas Ehliar, Dake Liu. *A Network on Chip based gigabit Ethernet router implemented on an FPGA*, volume 03. SSoCC, 2006.

[10] Andreas Ehliar, Dake Liu. An FPGA based open source Network-on-Chip architecture. *FPL*, 03:800–803, 2007.

[11] Roman Gindin, Israel Cidon, Idit Keidar. NoC-based FPGA: Architecture and routing. *NOCS*, pages 253–264, 2007.

[12] David Bafumba-Lokilo, Yvon Savaria, Jean-Pierre David. Generic Crossbar Network on Chip for FPGA MPSoCs. *IEEE*, pages 269–272, 2008.

[13] Intel. *AN 797: Partially Reconfiguring a Design on Intel Arria 10 GX FPGA Development Board*. Intel, 2018.

[14] Altera. *15. Analyzing and Optimizing the Design Floorplan with the Chip Planner*. Altera, 2013.

[15] Intel. *Intel Quartus Prime Pro Edition User Guide Partial Reconfiguration*. Intel, 2018.

Tools and Infrastructure for Reproducibility in Data-Intensive Applications

This page intentionally left blank

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200104

Cryptographic Methods with a *Pli Cacheté*

Towards the Computational Assurance of Integrity

Thatcher L COLLINS [a,1]
[a] *Department of Applied Mathematics, University of Washington, Seattle, USA*

Abstract. Unreproducibility stemming from a loss of data integrity can be prevented with hash functions, secure sketches, and Benford's Law when combined with the historical practice of a *Pli Cacheté* where scientific discoveries were archived with a 3rd party to later prove the date of discovery. Including the distinct systems of preregistation and data provenance tracking becomes the starting point for the creation of a complete ontology of scientific documentation. The ultimate goals in such a system–ideally mandated–would rule out several forms of dishonesty, catch computational and database errors, catch honest mistakes, and allow for automated data audits of large collaborative open science projects.

Keywords. reproducibility, hash function, secure sketch, fuzzy extractor, data fraud, library science, open science, applied ontology, publication bias, plagiarism, data provenance, preregistration, Benford's Law

1. Introduction

When integrity breaks down in a scientific setting, the mess can involve legal action, investigations, accusations, and negative media coverage. To prevent that, a systematic and unbiased way to prevent fraud or inadvertent corruption of the data or the results is proposed. The *European Code of Conduct for Research Integrity* defines integrity as "**Reliability** in ensuring the quality of research, reflected in the design, the methodology, the analysis and the use of resources. **Honesty** in developing, undertaking, reviewing, reporting and communicating research in a transparent, fair, full and unbiased way. **Respect** for colleagues, research participants, society, ecosystems, cultural heritage and the environment. **Accountability** for the research from idea to publication, for its management and organisation, for training, supervision and mentoring, and for its wider impacts"[1]. Except for respect, training and mentoring, a loss of integrity can lead to unreproducible science. Within that subset of integrity issues, the hardest part is finding the resultant hidden changes. Thus the practical manifestation of integrity in data-driven science is: **no unintentional changes, no secret changes**. That is, every significant change in content is intentional and tracked over time and space; data provenance expands to include provenance for the application of the scientific method.

[1]E-mail: *thtchr@uw.edu.*; https://orcid.org/0000-0003-0591-0823; Project Page: https://osf.io/asbtg/

Fraud includes any human misconduct that changes the science (most often for data but not necessarily), including plagiarism, manipulation, or fabrication. For this paper, **corruption** always refers to the sort with data, not political corruption which is simply a motivation or qualitative description of the machinations of an act of fraud. In a broad-form definition for what most people call plagiarism, The Council of Science Editors (CSE) defines **piracy** as "the unauthorized reproduction or use of ideas, data, or methods from others without adequate permission or acknowledgment," even by secretly repurposing one's own work. A similarly broad definition was described in the 13th century by Persian Poet Shams-e-Qays with a useful categorization of **plagiarism**: 1) Verbatim (*Intihal*): exact copy; 2) "Flayed" (*Salkh*): changing the order, re-arranging; 3) Conceptual (*Elmam*): "approaching" the exact copy in concept; and 4) Domain Transfer (*Naql*): unattributed reuse in a new domain [2]. The CSE's technical definition of plagiarism only encompasses verbatim and flayed plagiarism, the others fall under piracy [3]. Is there sufficient evidence of fraud to justify all of this work? In *Chemistry of Materials*, a 2019 study of chemistry articles published from 2017-2018 found that 42% of 331 retracted chemistry papers were retracted because of plagiarism and another 16% were retracted due to falsified data. Only 16% were due to honest errors [4] [5].

2. Hashing from the Start

A hash function maps a digital object to a finite uniform string called a **checksum** or a **hash** with no discernible relationship to the original object, acting as a secret identification code. Two objects that are exactly the same will always produce the same hash. Two identical hashes are likely to come from the same object, but not necessarily (because an infinite list of potential objects mapped to a finite set of strings will sometimes produce a **collision**). One change to an object will produce a completely different hash. Hashes allow for rapid checks of changes in content[6]. Hashing prevents Shams-e-Qays's verbatim-type plagiarism.

For version control such as Git and data management, hashing is already essential [7]. In a recent overview of reproducibility systems, The Whole Tale Project [8] mentioned "an optional checksum" of data. In this model, integrity checks **within** a system (like Git) are standard, but integrity checks **between** different research components are optional. If a mandatory automatic checksum is standard elsewhere, why should it be optional for science? Moreover these are **internal** integrity checks, but scientific integrity would be better ensured by including **external** 3rd party checks. After publication or at the completion of the scientific process, data and other artifacts might go to a repository, many of which create a checksum at this point to ensure the integrity of the data as it is stored long-term or moved around for other uses. CoreTrustSeal requires checksums in their certification of scientific repositories [9]. Dryad, Zenodo, 3TU.Datacentrum (4TU), and OSF all use checksums. Dryad is notable for including an audit before final acceptance into the database [11]. Libraries often include checksums in their Data Management Policies (DMPs) [11]. The Open Science Chain is designed to handle provenance and integrity after publication where data reuse is very complicated [12]. The advantages of using a simple hash algorithm (or collection of simple ones) is that they are: portable, found on every modern operating system, unlikely to become obsolete, and fast.

3. *Pli Cacheté*: Caching all the Hashing

A checksum is only a snapshot certification of content, not timing. Unimpeachable certification of existence in time requires physical possession by a 3rd party. Thomas Erren makes a compelling case to create a modern version of a 3rd party system called a *Pli Cacheté* (French for a sealed envelope) first brought to prominence by the French *Academie des Sciences* in the 1700s which accepted draft deposits of scientific work. Erren suggests reviving the *Pli Cacheté* so that researchers have "the opportunity to claim priority of sealed scientific rationale and data which may not be substantiated enough and might mislead when published too early or even erroneously" [13]. Notice that it helps the scientist by preventing a politically charged debate over the primacy of scientific discovery (accurate recognition of which is a type of integrity), but also, it encourages systematic, careful research; "reliable" in the sense of the *European Code of Conduct*.

Expand this concept to each stage of the scientific process including each day of data collection, where a simple checksum from the hash function is given to a library (rather than an academic journal as suggested by Erren) or other 3rd party such as an open repository. Instead of simply collecting the various components needed to reproduce a scientific result, an academic publisher (or peer reviewer) can hash the received contents and audit their checksum against the 3rd party checksums. Hashing a step and sending it to a library takes less time than reading this article; easy to do, but easy to forget (this is the great implementation challenge).

The full power of a cryptographic hash comes when used as a mandatory audit at every stage of the scientific stack:

1. Hashing the **hypothesis** prevents *post-hoc* storytelling or changing the hypothesis to fit the data;
2. Hashing the text of the domain-specific **data collection methods** (e.g. lab techniques) certifies that methods to be reproduced do not suffer from differences in memory; or have not changed significantly during the study;
3. Hashing the **data** as soon as it is recorded prevents manipulating the data or changing the outlier policy;
4. Hashing **additional stages** (such as noise reduction) further adds to the difficulty of fabrication and falsification;
5. Hashing the **conclusions and results** preserves the patent rights and scientific credit; and prevents errors of publishing too soon or without sufficient evidence.

The North American Scientific Integrity Consortium highlights a primary problem: "there are impediments and disadvantages of open science that must be acknowledged, including concerns with intellectual property, matters of national security, and the potential loss of confidentiality of research participants in human clinical trials" [14]. The *Pli Cacheté* system maintains privacy even as hashes of the data can ensure integrity of private or secret data. In support of this approach, an editorial in the *Journal Nature* includes "better record-keeping" in their proposed solutions [15]

4. Similar Systems Found in Law

The Paris Convention for the Protection of Industrial Property (1883) includes a provision–still in force–that an inventor has 6-12 months to file, if desired, in the other

"Contracting States," retaining the original filing date as the date of discovery. But because the clock on the monopoly period in the US does not start until the US patent application is filed, inventors can file in Europe first, then just under 12 months later in the US, granting the inventor an extra year of patent protection [17]. In 1995, US law changed to "give U.S. applicants parity with foreign applicants under the GATT Uruguay Round Agreements." Since then, US applicants can file a **provisional patent** in a sealed envelope for up to a year, providing the same year-long grace period to filers of US patents [16]. The provisional application is not opened till the full application is filed, just like the traditional *Pli Cacheté*.

Second, common law and many other legal systems have rules for the immediate acceptance of documentary evidence (**self-authenticating** (USA), self-proving (Scotland), public instruments (British Common Law), and authentic instruments (EU)) [18]. Third, carbon copies create exact copies at the time of creation for the rapid authentication of business documents. A *Pli Cacheté*, broadened to be like self-authenticating documents can help avoid or ease the resolution of litigation, bureaucratic adjudication, and messy public battles. These similar systems are summarized in Table 1.

Table 1: Summary of Similar Systems

	Pli Cacheté	Patents	Carbon Copies	Self-Authenticated
Domain	science	technology	business	law
Purpose	integrity	monopoly rights	disputes	litigation
3rd Party	library	patent office	(varies)	government

5. In the Classroom

A 1994 study found that "Eighty-nine percent of students surveyed admitted they had cheated" [19]. Without evidence, Canada's York University has a Teaching Policy which states "Academic dishonesty is a serious problem in undergraduate labs. This is partly because the culture of lab courses sometimes fosters plagiarism." Their solution is similar to the Pli Cacheté but uses Teaching Assistants as a 3rd party: "students obtain the TAs signature on all pages of their original lab notes and data, and submit those notes with their lab report" or else a "carbon copy may then be ripped out and handed to the TA before the student leaves the class" [20]. Thus, academia is already using self-authentication to solve integrity problems. Lab classes might also require submission of a pre-lab beforehand, where a pre-lab mirrors preregistration.

6. Comparison to Preregistration and Data Provenance

Preregistration (also called registered reports) is a reaction to the nonpublication of negative science, also called **publication bias** [21] or the **file drawer effect**. The Open Science Framework has guidelines for open science that state "preregistration of studies is a means of making research more discoverable even if it does not get published. Preregistration of Analysis Plans certifies the distinction between confirmatory and exploratory research" [22]. This preregistration of analysis overlaps with the caching of a study plan, but the holder of the deposit is a journal (2nd party relationship. Preregistration, at a minimum, acknowledges the existence of negative results. At its best, it comes

with a commitment to publish regardless of the result. A call-to-action from the 2019 Hong Kong Conference for Research Integrity states "Value the reporting of all research, regardless of the results" [23]. In the event of a conflict of interest with the journal, a 3rd party *Pli Cacheté* preserves integrity.

One subtle integrity problem, suggested by Klein et al, is that "while embargoes on preregistrations can mitigate the fear of being scooped, flexibility in the release of pre-registered documents limits transparency. For example, researchers may strategically release only those documents that fit the narrative they wish to convey once the results are in. It is therefore preferable to encourage transparency from the outset" [21]. But a hash of the preregistered documents would still allow reviewers and the public to verify the completeness of an embargoed preregistration.

Data provenance is "the derivation history of a data product starting from its original sources" as well as the information needed to process, identify, and distinguish a dataset [24]. Both provenance and *Pli Cacheté* indicate the time and place of the origin of a data product; *Pli Cacheté* authenticates the time, while provenance describes the details of the journey (helpful to an auditor for investigating fraud). *Pli Cacheté* hashes the contents while provenance describes the contents. Data provenance within a project might be handled by Git; data provenance after a project might be handled by a separate metadata file or a scientific data blockchain such as Open Science Chain (OSC)[12]. Currently, Git is for tracking the creation of something while OSC is for tracking reuse and modification **after** creation. But a *Pli Cacheté* would use universally-available hash already designed for rapid match checks, allowing an integrity check in any other system.

Are all three systems necessary? Consider the standard questions in Table 2 used by journalists to ensure an accurate report: who, what, where, why, and how. Outside of that standard list is the question of corroboration: does an independent source, a second reporter, or some other previously unknown document corroborate the answers to the standard list? Compare the coverage of these questions by the three systems:

Table 2: The Three *P*s of Scientific Documentation

Question	Pli Cachete	Preregistration	Provenance
who	x	x	x
what	x	x	x
where			x
when	x		x
why		x	
how		x	
corroboration	x		
mid-stream	x		x
ownership	3rd party	2nd party	1st party

Each system's information is held by a different party **during** the scientific process, with every question answered somewhere. The goal is to use this completeness to solve reproducibility problems and aid integrity investigations.

7. Tackling the File Drawer Effect

Scargle defines publication bias and the file drawer effect to be when the probability of being published "depends on the statistical significance of its results" [25]. In effect,

amalgamated research (including metaresearch and multi-grid systems) become biased by an unrepresentative sample. Clinical studies often require preregistration by law (with no guarantee of publication). In *Selective Publication of Antidepressant Trials and its Influence on Apparent Efficacy* [26], not only is the existence and statistical effect of publication bias demonstrated, but also acknowledged is the switching of domains after a **secondary effect** shows positive results. Whether or not an amalgamated study using secondary results is biased is up to others to evaluate, but data provenance and preregistration give researchers the investigative information they need. If by fraud, a study is changed (or fabricated) to be positive, this moves a study from negative to positive. In Table 3, the three integrity systems illuminate how to prevent the three file drawer effects.

Table 3: The Filedrawer Effect

	published	not published	domain change	fraud
relationship to claim	positive	negative	positive	positive
solution	–	preregistrtion	provenance	pli cachete

8. Conflict Resolution Amongst Scientists

A dispute can occur within a scientific group during the scientific process or even much later, including during a retraction process when editors decide which scientists to blame. The *Bullied Into Bad By Science* Campaign was started in 2017 by "postdocs and a reader in the humanities and sciences at the University of Cambridge" who were **concerned about the desperate need for publishing reform** to increase transparency, reproducibility, timeliness, and academic rigour of the production and dissemination of scholarly outputs [31] Because, they say (and cite evidence) that early career researchers (ECRs) "**are often pressured into publishing against their ethics** through threats that we would not get a job/grant unless we publish in particular journals". Their petition has a amendment added by Anne Schell (not necessarily endorsed by the initial signers) that specifically addresses integrity: "Stop pressuring ECRs into conducting/writing up underpowered, non-preregistered, p-hacked, HARKed studies. In other words: stop teaching/advising/pressuring people to mutilate data into a 'publishable' form when that distances it from actual science." Preregistration can indeed prevent a lot of this pressure. Pressure can come **after** preregistration, where data could manipulated to conform with the preregistered hypothesis. This highlights the need to hash data **as it is created**. Researchers can unilaterally post (OSF, github, or a library) hashes of their data without jeopardizing the privacy of the scientific group. Later, if asked to manipulate data, this scientist could point to the 3rd party hashes and explain how easily they would be caught: legal and cryptographic checkmate.

In the summer of 2019 at the University of Florida, Computer Architecture PhD candidate Huixiang Chen committed suicide–according to the suicide note posted by his friends and academic colleagues–because he saw no way out of a dispute over the integrity of data [28]. Despite the extra attention this action has elicited, this is not a good way to resolve an academic dispute; this is also precisely the type of situation an integrity system should try to prevent. In no way does the mention of this tragedy imply the guilt or innocence of the people involved. Chen claimed that a joint paper that had been accepted included experiments that had never been conducted, so he was tasked "to make up for the missing experiments" [28]. If journals, funders, departments,

or universities required that experiments be hashed and cashed with a 3rd party when they are created; and the peer reviewers checked the *Pli Cacheté* before acceptance for publication; then disputes of this nature would be resolved easily or never happen at all. As the International Journal of Medical Journal Editors states, "Perceptions of conflict of interest are as important as actual conflicts of interest," [29] similarly, perceptions of misconduct are almost as important as actual misconduct. Openness and authenticated documentation protects everyone from accusations and perceptions of misconduct.

In North America, the Scientific Integrity Consortium aims to form a comprehensive approach to integrity in science, focusing on the best practices for the bureaucracies that manage scientists. A *Pli Cacheté* can complement their work because they identify an "urgent need to refocus the scientific communitys efforts on policing itself" [14]. Anyone can choose to use a unilateral *Pli Cacheté*, a form of policing oneself that prevents investigations altogether. While requiring it would be a bureaucratic solution, it minimizes the need for bureaucratic monitoring and control.

9. Cryptographic Techniques

Consider the backwards problem to evaluate integrity solely on the basis of the data and hypotheses revealed just when an article is published. Potential integrity problems include: 1) computational bugs or errors introduced during data storage, processing or analysis; 2) intentional changing of the data points to affect the final result; and 3) data plagiarism (possibly with data owned by the scientist but dishonestly reused).

9.1. Secure Sketches and FIBE

To easily defeat a hash check for plagiarism, change any one thing, then the checksums will be different. Shams-e-Qays might have called this type of plagiarism "flayed" data. One solution is to borrow a method from biometric passwords called Fuzzy Identity Based Encryption (FIBE). Typical passwords and hashes must be exact to gain access. But a fingerprint scan, for example, is a huge data file with an error deviating from the "true" fingerprint[30]. Instead, one FIBE method creates a **secure sketch** for the older data set with a chosen tolerance for deviation. If both both data sets can unlock the secure sketch, then they are indeed very close and candidates for investigation[30].

These techniques "apply not just to biometric information, but to any keying material that, unlike traditional cryptographic keys, is (1) not reproducible precisely and (2) not distributed uniformly." Thus FIBE can also be used to honestly evaluate data that contain slight variations when reproduced. The **secure sketch** is a ciphertext which "produces public information about its input w that does not reveal w, and yet allows exact recovery of w given another value that is close to w." A **fuzzy extractor** is a similar (near) uniform length ciphertext. Fuzzy cryptographic methods incorporate metric spaces and distance functions where the distance between w and w' is a parameter chosen at encryption time[30]. An enticing idea is a fuzzy data auditing system (prototyped in Li et al [31]) for use in a large multi-grid system [32] but with tolerance for error because "apparently routine data manipulation workflows become rife with mundane complexities as researchers struggle to assemble large, complex datasets." [33].

9.2. Benford's Law

Benford's Law is an empirical observation (mathematical proofs of which are hotly debated) where 1 is the most common first digit, 2 is the 2nd most common, descending monotonically to 9. An unadulterated data set will often demonstrate an exponentially descending Benford Curve. Manual adjustments to data can substantially change the distribution of numeral frequency, serving as an indicator for potential fraud. Two data sets which have exactly the same deviation from the Benford Curve would be another indicator. Benford's Law is particularly useful for detection fraud in data sets for regression[34]. Storing copies of Benford Curves in a *Pli Cacheté* would serve as an additional integrity check, one that–in the long term–would not depend on having a reproducible hash function.

In the detection of plagiarism, Benford's Law and secure sketches have a powerful synergy. While merely one change defeats a plagiarism audit with a checksum, a secure sketch would require many more changes. But with more manufactured changes, the Benford Curve would likely be more erratic–and therefore possibly fraudulent. Defeat a Secure Sketch, get caught by a Benford Curve.

9.3. Resampling Detection

A method to detect stretching and rescaling is the Expectation-Maximization algorithm which "is applied to estimate the interpolation kernel parameters, and a probability map (called *p*-map) that is achieved for each pixel provides its probability to be correlated to neighbouring pixels. The presence of interpolated pixels results in the periodicity of the map, clearly visible in the frequency domain" [35]. This is an argument for the caching of raw data. Bowman and Keene advocate for the preservation of raw data in general because it "will allow the researchers to view the entire spectrum of what was done rather than simply what was reported" [36].

9.4. Auditing Amalgamated Research

A four step plagiarism audit process emerges: 1) Does the checksum match another dataset?; 2)Does the secure sketch match another dataset?; 3) Is the Benford curve erratic?; 4) Is there evidence of resampling if the data is raw? This is a check of a proposed dataset against previously adopted datasets. Passing this, a new hash of the proposed dataset should be checked against the original hash stored with in a *Pli Cacheté* (hopefully from the date of its creation). Failing this check leads to an investigation: 1) If the secure sketch still matches, then the changes are minor; and 2) Are the Benford Curves consistent with a random change, erratic change, or remain a consistent Benford Curve?

10. Conclusions

While a *Pli Cacheté* can be immediately and unilaterally, some promising directions for future research include:

1. Establish an effective and universal secure sketch or fuzzy extractor (like SHA);
2. Find or create a resampling test suitable for scientific data audits;

3. Create an open repository of fraudulent and corrupted data for testing;
4. Audit an already constructed amalgamated research database; and
5. Create a standardized ontology of reproducible science.

The biggest barrier to implementation is the social challenge of convincing a scientific community to adopt any mandatory methods. Herein lies the importance of the backwards problem: as investigations reveal fraud or unintentional errors, then the case for the forward problem of preventing the loss of integrity becomes more compelling.

References

[1] The European Code of Conduct for Research Integrity. 2017. http://ec.europa.eu/research/participants/data/ref/h2020/other/hi/h2020-ethics_code-of-conduct_en.pdf

[2] Sadeghi, R. 2019. The attitude of scholars has not changed towards plagiarism since the medieval period: Definition of plagiarism according to Shams-e-Qays, thirteenth-century Persian literary scientist. Research Ethics. Vol. 15(2) 13. https://doi.org/10.1177/1747016116654065

[3] Laine, C and The Council of Science Editors. 2012. Description of Research Misconduct. Approved March 30th, 2012. https://www.councilscienceeditors.org/resource-library/editorial-policies/white-paper-on-publication-ethics/3-1-description-of-research-misconduct/#312

[4] Coudert, F. 2019. Correcting the Scientific Record: Retraction Practices in Chemistry and Materials Science. Chem. Mater.201931103593-3598. Publication Date:May 28, 2019. https://doi.org/10.1021/acs.chemmater.9b00897

[5] Chawla, D. S. 2019. Retracted chemistry studies most often plagued with plagiarism. Chemical Engineering News. Published: May 31, 2019. https://cen.acs.org/research-integrity/misconduct/Retracted-chemistry-studies-often-plagued/97/web/2019/05

[6] Sean Thorpe, Indrajit Ray, Tyrone Grandison, Abbie Barbir. Formal Hash Compression Provenance Techniques For The Preservation Of The Virtual Machine Log Auditor Environment. The International Journal of Information Science and Computer Application (IJISCA), Vol 1, pp 1-10. https://www.tyronegrandison.org/uploads/1/8/8/1/18817082/newfinalijisca_papercameraready2012.pdf

[7] Blazic, AJ. Trusted Archive Authority–Long Term Trusted Archive Service. EurOpen Conference Processings, Czech Open Systems Users' Group, pp.107-119 May, 2001. https://www.europen.cz/Anot/30/hlavni.pdf#page=107

[8] A. Brinckman, K. Chard, N. Gaffney, M. Hategan, M.B. Jones, K. Kowalik, et al. 2019. Computing environments for reproducibility: capturing the "Whole Tale" Future Gener. Comput. Syst., 94, pp. 854-867 https://doi.org/10.1016/J.FUTURE.2017.12.029

[9] The Swiss Centre of Expertise in the Social Sciences (FORS). Implementation of the CoreTrustSeal for the Repository DARIS. Mar. 20, 2018. https://www.coretrustseal.org/wp-content/uploads/2018/03/DARIS.pdf

[10] Assante, M., Candela, L., Castelli, D. and Tani, A. 2016. Are Scientific Data Repositories Coping with Research Data Publishing? Data Science Journal, 15, p.6. http://doi.org/10.5334/dsj-2016-006

[11] University of Minnesota Libraries. Data Management Plans (DMPs). Retrieved September 8th, 2019. https://www.lib.umn.edu/datamanagement/DMP

[12] Sivagnanam, S., Nandigam, V. and Lin, K. 2019. Introducing the Open Science Chain: Protecting Integrity and Provenance of Research Data. In Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning) (p. 18). ACM. https://dl.acm.org/citation.cfm?id=3332203 and https://www.opensciencechain.org

[13] Erren TC. 2008. On establishing priority of ideas: revisiting the pli cacheté (deposition of a sealed envelope). Medical Hypotheses 2008;71(1):8e10 https://doi.org/10.1016/j.mehy.2008.08.013

[14] Kretser, A., Murphy, D., Bertuzzi, S. et al. 2019. Sci Eng Ethics. 25: 327. https://doi.org/10.1007/s11948-019-00094-3

[15] Editorial Board. Research integrity is much more than misconduct. 2019. Nature 570, 5. https://www.doi.org/10.1038/d41586-019-01727-0

[16] United States Patent and Trade Office (USPTO). Provisional Application for Patent. Retrieved September 8th, 2019. https://www.uspto.gov/patents-getting-started/patent-basics/types-patent-applications/provisional-application-patent

[17] World Intellectual Property Organization (WIPO). Retrieved September 8th 2019. Summary of the Paris Convention for the Protection of Industrial Property (1883). https://www.wipo.int/treaties/en/ip/paris/summary_paris.html

[18] Tracy, J. E. 1939. The introduction of documentary evidence. Iowa L. Rev., 24(3), 436-463. https://heinonline.org/HOL/Page?collection=journals&handle=hein.journals/ilr24&id=458&men_tab=srchresults

[19] Graham, M.A. 1994. Cheating at Small Colleges. Journal of College Student Development, 35(4), pp.25560. https://eric.ed.gov/?id=EJ489082

[20] York University, Canada. Retreived September 8th, 2019. Academic Dishonesty in Laboratory Environments. https://teachingcommons.yorku.ca/resources/teaching-strategies/academic-integrity/academic-dishonesty-in-laboratory-environments/

[21] Klein, O., Hardwicke, T. E., Aust, F., Breuer, J., Danielsson, H., Hofelich Mohr, A., Frank, M. C. 2018. A Practical Guide for Transparency in Psychological Science. Collabra: Psychology, 4(1), 20. http://doi.org/10.1525/collabra.158

[22] Center for Open Science. 2018. Guidelines for Transparency and Openness Promotion in Journal Policies and Practices "The TOP Guidelines." Version 1.0.1 https://osf.io/9f6gx/

[23] Mohar, David and Glasziou, Paul. 2019. How can we improve organizational assessment of researchers? World Conference on Research Integrity, Hong Kong, China. http://www.wcri2019.org/uploads/files/archive_plenary/day_4_june_5/concluding_plenary_david_moher_and_paul_glasziou_hong_kong_principles_final_plenary_session.pdf

[24] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. 2005. A survey of data provenance in e-science. SIGMOD Rec. 34, 3 (September 2005), 31-36. https://doi.org/10.1145/1084805.1084812

[25] Scargle, J.D., 2000. Publication Bias: The File-Drawer Problem in Scientific Inference. Journal of scientific exploration, 14(1), p.91. https://arxiv.org/abs/physics/9909033

[26] Turner, E. H., Matthews, A. M., Linardatos, E., Tell, R. A., and Rosenthal, R. 2008. Selective Publication of Antidepressant Trials and its Influence on Apparent Efficacy. N Engl J Med; 358:252-260. https://doi.org/10.1056/NEJMsa065779

[27] Logan, C., et al. 2017. Retrieved September 8th, 2019. http://bulliedintobadscience.org

[28] Anonymous. 2019. The Hidden Story Behind the Suicide PhD Candidate Huixiang Chen. Medium, June 29th, 2019. https://medium.com/@huixiangvoice/the-hidden-story-behind-the-suicide-phd-candidate-huixiang-chen-236cd39f79d3

[29] International Committee of Medical Journal Editors. Website Viewed September 8th, 2019. Conflicts of Interest. http://www.icmje.org/recommendations/browse/roles-and-responsibilities/author-responsibilities--conflicts-of-interest.html

[30] Y Dodis, R Ostrovsky, L. Reyzin, A Smith. 2008. Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data. A preliminary version appeared in Eurocrypt 2004 [DRS04]. SIAM Journal on Computing, 38(1):97139 http://web.cs.ucla.edu/~rafail/PUBLIC/89.pdf

[31] Y. Li, Y. Yu, G. Min, W. Susilo, J. Ni and K. R. Choo, 2019. Fuzzy Identity-Based Data Integrity Auditing for Reliable Cloud Storage Systems. IEEE Transactions on Dependable and Secure Computing, vol. 16, no. 1, pp. 72-83, 1 Jan.-Feb. https://doi.org/10.1109/TDSC.2017.2662216

[32] G. Tylissanakis and Y. Cotronis, "Data Provenance and Reproducibility in Grid Based Scientific Workflows," 2009 Workshops at the Grid and Pervasive Computing Conference, Geneva, 2009, pp. 42-49. https://doi.org/10.1109/GPC.2009.16

[33] K. Chard et al. 2016. I'll take that to go: Big data bags and minimal identifiers for exchange of large, complex datasets. 2016 IEEE International Conference on Big Data, Washington, DC, 2016, pp. 319-328. https://doi.org/10.1109/BigData.2016.7840618

[34] Jamain, A. Benford's Law [thesis]. 2001. Imperial College of London and Ecole Nationale Superieure. https://wwwf.imperial.ac.uk/~nadams/classificationgroup/Benfords-Law.pdf

[35] Piva, A. An Overview on Image Forensics. ISRN Signal Processing, vol. 2013, Article ID 496701, 22 pages, 2013. https://doi.org/10.1155/2013/496701

[36] Bowman, N. D. and Keene, J. R. 2018. A Layered Framework for Considering Open Science Practices. Communication Research Reports, 35:4, 363-372. https://doi.org/10.1080/08824096.2018.1513273

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200105

Replicating Machine Learning Experiments in Materials Science

Line POUCHARD [a,1], Yuewei LIN [a] Hubertus VAN DAM [a]
[a] *Brookhaven National Laboratory, Upton, NY 11973-5000*

Abstract. Transparency and reproducibility are important aspects of validation for Machine Learning (ML) models that are not fully understood and applies independently of the application domain. We offer a case study of reproducibility that highlights the challenges encountered when attempting to reproduce analyzes obtained with Machine Learning methods in materials informatics. Our study explores prediction results obtained with ML models and issues in training data serving as input. We discuss challenges related to theory-driven and numerical errors in training data, lack of reproducibility across platforms and versions, and effects of randomness when varying hyperparameters. In addition to model accuracy, a main metric of interest in the ML community, our results show that model sensitivity may be equally important for applying ML in domain applications such a materials science.

Keywords. reproducibility, machine learning, materials science, materials informatics

1. Introduction

The design of new materials depends upon the ability to combine precursor materials to make samples and test them using expensive characterization techniques to reconstruct molecular and atomic structures and deduce interesting properties. Discovery through empirical process is slow and largely depends for its success on the intuition of individual scientists. Materials informatics, seeking to elucidate structure-property relationships using complex, multi-scale information in a physically meaningful, statistically robust manner, is becoming more data-intensive due to the advent of high throughput detectors and more complex models [1], [2]. In this context, accelerating discovery requires reducing the combinatorial explosion of the number of potential candidates in the search space for making samples of interest. Machine Learning (ML) methods are increasingly used to predict the relationship between structures and properties and provide guidance to experimentalists for suggesting potentially useful combinations [3]–[5]. Neural Networks have witnessed a revival in the ML community thanks to new methods preventing overfitting, new training methods and the use of computer hardware with GPUs, so that their predictive power has superseded that of other methods, for instance in drug discovery [6], [7]. In order to harness the benefits of new generation ML algorithms in materials informatics and broaden the path of accelerated discovery it is necessary to understand their limits and establish transparency in how results are obtained. Better ways

[1]Corresponding Author: Line Pouchard; E-mail:pouchard@bnl.gov

of explaining results obtained with ML methods will lead towards better reproducibility of results in materials science, and thus better confidence in the ability of ML methods to predict new candidates for experimentation.

Transparency and reproducibility are important aspects of validation for Machine Learning models that are not fully understood and applies across the board independently of the application domain. The National Academy of Science defines reproducibility as obtaining consistent computational results using the same input data, computational steps, methods, code and conditions of analysis [8]. Replicability implies consistent results across studies aimed at answering the same scientific questions. Reproducibility and replicability of scientific results improve when researchers provide access to their codes, methods, data, and execution environments. A survey points to the fact that, of 400 Artificial Intelligence papers recently presented at major AI conferences, only 6% share their algorithms codes, and a third share their data [9].

Understanding the limits of computational reproducibility when dealing with complex mathematical models such as ML models goes beyond making accessible scientific code, training data, and hyperparameters. ML methods present specific challenges in reproducibility related to the building of models, the effects of random seeds, and the choice of platforms and execution environments [10]. ML models that are inherently non-deterministic also pose a problem for the validation of results. If the training sets are divided for cross-fold validation and testing, the partitioning of the sets will affect reproducibility. In materials science, the lack of open access and the heterogeneity of experimental data that describe only few aspects of a material have led to the use of computational structures calculated from physics first principles (ab initio) for the training of models. Experimental data can provide the ground truth to evaluate the accuracy of algorithms but these data are typically small, hard to come by, and not necessarily representative of the problem at hand.

In addition, there is another factor limiting the reproducibility of ML models in materials informatics. Data used as input, especially when they come from atomic and molecular structures computed from theory, can introduce biases due to theoretical approximations. These biases may or may not influence the outcomes for the ML models depending on where the sensitivity of these models lies. In diverse teams, the scientists who build the models are not the ones developing the scientific simulation codes that produce input data. They may not be aware of the presence of these biases and their potential for skewing models, as this requires in-depth knowledge of the parameters, theoretical methods and implementations used for calculating the input data. On the one hand the theoretical approximations made when calculating computational structures are known to those producing these structures, but not available to others. On the other, scientists who build ML models are presumably aware of the sensitivity of their models but not of the theoretical approximations in their input data. The disconnect introduced at the interface of these two groups of actors may result in poor performance prediction that remains unexplained by scrutinizing the ML models alone.

In this paper, we offer a case study of reproducibility that highlights the challenges encountered when attempting to reproduce analyzes and results obtained with Machine Learning methods in materials informatics. Section 2 presents the rationale for using these methods in materials discovery and highlights issues of reproducibility in the training data used by ML models. Section 3 presents several experiments reproducing results. The first experiment uses QM9, a publicly available dataset of computed small organic

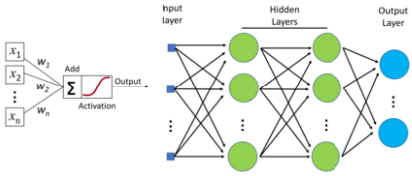


Figure 1. An illustration of a MLP structure.

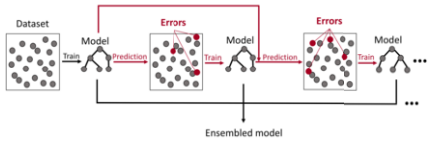


Figure 2. An illustration of a GBT structure.

molecules. It illustrates the fact that, even when data and methods are known, reproducibility is still a challenge, due to hidden theoretical assumptions and lack of transparency. The second set of experiments uses regression-based analysis, including a neural network (MLP) and a Gradient Boasted Tree (GBT), where computational structures are used for training models, and experimental structures for validation (Figures 1 and 2). Section 4 discusses the results and Section 5 concludes with some pointers to future work.

In this paper we make the following contributions:

- an inventory of issues related to reproducility challenges in the use of ML in materials informatics
- an exploration of theoretical assumptions and uncertainties linked to training data
- several experiments reproducing results obtained with ML methods and computational data in materials informatics
- a comparison of results obtained with several common ML platforms (Tensorflow, PyTorch, lightGBM)
- an exploration of the information required for understanding discrepencies in results
- a discussion of reproducibility challenges when using ML in materials discovery

2. The use of ML in materials science

2.1. Motivation

Many technological advances depend on materials with properties of particular interest. Materials science is a field of research that takes the properties of interest and looks for or designs new materials and characterizes them in the pursuit of those with the desired properties. From a theoretical perspective it is understood that the structure of a material determines its properties. The structure of a material is given by the chemical composition and the positions of atoms as well as the length scales of material features (such as grain sizes, fiber thickness, surface roughness, etc.). At present, there are roughly two approaches to studying materials, an experimental approach, and a computational one. Use of ML algorithms represents a new third approach (Figure 3).

In experimental materials science the general workflow consists of making a sample of a candidate material and experimentally assess its properties (Figure 3a). There are a broad range of properties that might be of interest and an equally broad range of experiments to assess them. Examples are catalytic activity which may be measured in a flow cell, charging characteristics that may be measured in a battery cell, the color characteristics of LEDs, the thermopower of thermoelectric materials, critical temperature and

critical current in superconductors, etc. The measured properties have to be understood in terms of the structure of the sample to be able to propose other candidate materials that may better match the set of desired properties. An additional set of experiments particularly targets the structure elucidation but in some cases, e.g. X-ray spectroscopy, experiments alone may not be sufficient to determine the structure.

In computational materials science theoretical models provide a way to compute properties of interest from a given material structure (Figure 3b). In particular ab-initio models based on Schrödinger's equation provide a path to calculating a broad range of properties. Resulting structures are validated against experimental results. A caveat is that these models are typically computationally very intensive and non-trivial approximations are needed to compute results with reasonable compute resources. Nevertheless, these computational models can be used to calculate materials properties. Comparison of the computationally obtained results with the measured properties can help determine the structure of the material sample from which the measurements were obtained.

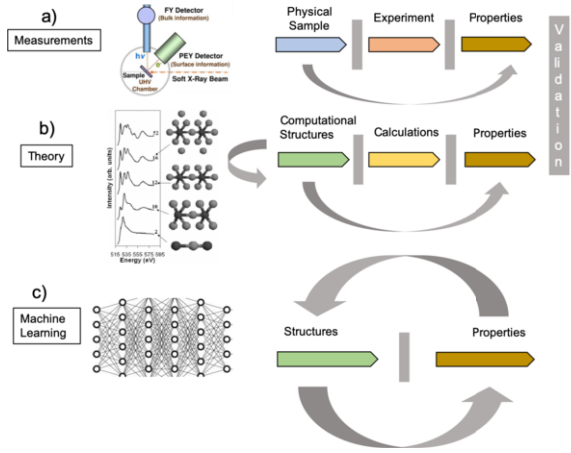


Figure 3.Discovering the structure-property relationship in materials informatics.

However, the problem that remains is that Schrödinger's equations only provides a path to compute the properties from structures. In practical materials science problems the measured properties are given and the material structure needs to be solved. Schrödinger's equation does not provide a convenient formalism to solve such inverse problems. By contrast, machine learning can be used to train a model that correlates input data to output data (Figure 3c). In principle, machine learning does not care about the direction of the relationship. It can be used with material structure as inputs and properties as outputs, but it can also learn the inverse model with the properties as inputs and the structure as outputs. Based on this realization machine learning can be used in essentially two modes. First, it can used as an alternative to ab-initio calculations to predict materials properties from structures, but at a much lower computational cost. Second, it can be used to build models for inverse relationships for which there are few, if any, alternative models available. Both kinds of applications could prove very valuable to moving materials science forward.

2.2. Theory reproducibility and artifacts in training data

In the previous section the importance of machine learning to help solve materials science problems was explained. For machine learning to be successful a key ingredient is

the availability of diverse sets of accurate training data. The accuracy of the training data is particularly important as ML models typically have no way of knowing the underlying physics they aim to "learn". Instead, the training data is supposed to be a representation of that physics in the form of a large number of individual examples. Systematic errors in the training data will lead to these errors becoming ingrained in the ML model. The availability of mature simulation codes, significant compute resources as well as software for automatically running and analyzing simulations make it possible to generate such data sets automatically. This is the approach taken in a few research projects already [11]–[14]. The outstanding problem then is to ensure that the data obtained in this way is sufficiently accurate. Solving Schrödinger's equation is only practical after making some approximations. These approximations lead to artifacts in the computed results and these artifacts have non-trivial relationships to the underlying approximations. For example, Density-Functional Theory (DFT) has been claimed to be in principle exact. But when using the currently available density functionals, the models of electron structure suffer from unphysical self-interaction errors and strong correlation effects that are poorly described. In another example, configuration interaction (CI) methods with fixed excitation levels, such as singles-doubles (SDCI), produce electron correlation energies that scale incorrectly with the number of electrons.

In fact one may state that the important expertise of practitioners in the field is related to these artifacts and therefore to knowing in which cases and how the results are affected by them. In cases where preferred methods leave doubt about the results it is common to try more advanced methods to reduce uncertainty. This expertise is only available to the ML experts who design models in well-functioning inter-disciplinary teams that share expertise and knowledge. When data obtained by such computations is made publicly available for re-use, the lack of transparency may lead to inaccurate predictions in ML results.

3. Study: Re-running ML models

We designed several experiments in reproducibility to illustrate the issues encountered when using Machine Learning. The first experiment illustrates the lack of transparency when calculating the computational structures and properties of small organic molecules that can be used as training data (section 3.1). The second experiment tests common platforms for training models under various versions of each platform (section 3.2). The third experiment tests the models themselves under various conditions (section 3.3).

3.1. QM9 experiment on the accuracy of training data

Small organic molecules are used in *de novo* drug design and a number of studies with large sets of computational molecules have been published [15] that can be used for training data or benchmarking existing codes. QM9, one of these data sets, was built using DFT methods that are supported by a wide range of quantum chemistry codes. The QM9 data set contains the subset of $C_7H_{10}O_2$ isomers consisting of 6,095 molecules. This subset of molecules is small enough in numbers and the molecules are small enough in size that calculations on this set can be repeated with relatively modest resources. In their paper [16], [17] the authors document two important issues that may affect the

results of simulations performed to obtain computational structures for training data. Here these issues are accepted and the work focuses on reproducing the calculations within the reported limits:

- that the reliability and accuracy of DFT depends on the chemical composition and atomistic configurations in molecules and materials,
- and that most reported calculations are done on small molecules implying the existence of a selection bias.

The structures published in QM9 were obtained by translating SMILES strings into structures, and performing subsequent geometry optimization. The final part of the geometry optimization was done with DFT using the Gaussian 09 code [18] but similar capabilities are available in most Gaussian basis set quantum chemistry codes. We used NWChem, an open source package that implements this capability [19]. As these codes have been stable for some time, and the final structures are minimal energy structures it should be straightforward to test the following hypothesis: a geometry optimization started at the published structure should converge at the first point with the same total energy if the same energy expression and basis set (input data) are used.

The basis sets are specifically formatted for the code itself and some codes make historically developed basis sets available with the codes. This is the case for Gaussian09, where the 6-31G(2df,p) basis set used in this experiment was historically developed by Pople and his collaborators on the Gaussian project [20]–[23]. The basis set made available in NWChem comes from the Basis Set Exchange [24]–[26], a community project collecting basis sets and making them publicly available in formats allowable for their respective codes. This project relies on publicly accessible data for the specification of the basis set. Some basis sets have been revisited and refined over time and so discrepancies between a built-in version of a basis set in one code versus that of another code are possible.

Inspite of well documented sources of uncertainty in DFT calculations, for atoms that are spherically symmetric, such as the ones we tested here, highly accurate results are generally achievable. However, our results were significantly different from the results published by the authors of the QM9 data set.

The difference between the two sets of results can largely be attributed to a technical detail related to the way the handling of the angular momentum of the basis functions may be chosen in different codes. Most codes allow one to choose between either Cartesian or spherical harmonic basis functions. Gaussian09 is different in that it allows one to choose between Cartesian and spherical harmonic basis functions independently for d-functions and other angular momentum functions. For example, in Gaussian one can choose to use Cartesian d-functions together with spherical harmonic functions for all higher angular momentum functions. The authors know of no other code that allows this kind of flexibility.

The basis set chosen for calculating the QM9 data set is one that exploits this particular Gaussian feature. It uses Cartesian d-functions and spherical harmonic f-functions. This means that the calculations reported for the QM9 data set can only be reproduced with the Gaussian code and no other code. As Gaussian09 is proprietary code, the authors of the QM9 paper are not at liberty to publish their implementation nor the input data made available with the code. This lack of transparency can affect the re-use of their data sets as training data for ML that may amplify the uncertainties.

3.2. Training models

In a previous experiment by co-author Lin, ML models were used to predict Coordination Numbers (CN) known to characterize size and 3D shape of nanoparticles [27], [28] (Figure 4). Training sets are built using computational data produced from ab initio methods. The model can then be used on experimental spectra to help determine the properties of experimental particles. In the experimental process, XANES spectra, a type of properties, are measured (4a). Coordination Numbers (CN) are calculated (not shown). The computational approach calculates XANES spectra and Coordination Numbers from computational structures (4b). After validation, computational XANES spectra and CNs are used to train the model. Predicted CN (box on the right) are compared to Calculated CN to validate the model. In the ML approach, the trained model can be used with large amounts of experimental spectra pouring out of high throughput detectors to predict expected CN (4c).

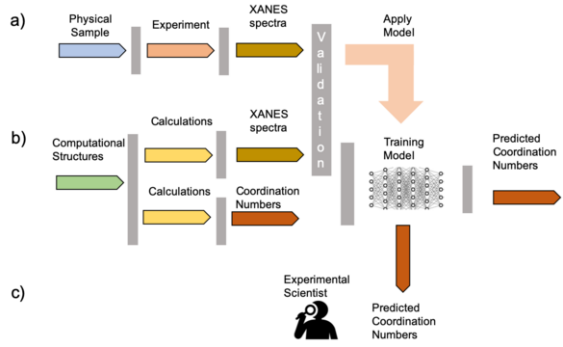


Figure 4.Application of ML for guiding high throughput experiments

Specifically, the machine learning models are defined as a nonlinear function which maps the spectra vector to the coordination number vector. It is a typical regression task. In this work, we evaluated three major powerful and widely used regression models, (1) Gradient Boosted Trees (GBT), an efficient machine learning model that ensembles a set of decision trees; (2) multilayer perceptron (MLP), a classic fully connected neural network; and (3) the most popular type of neural network, (one-dimensional) convolutional neural networks (1D-CNN). For the reproducibility experiment, we focus on GBT and MLP with a relatively deep structure of 5 layers with 400, 400, 200, 200, and 100 nodes respectively ([28].

3.3. Reproducibility across several different platforms

We first evaluate the reproducibility of different machine learning platforms and their versions, and the results are shown in Table 1. Here we test platforms used for the MLP (with Tensorflow and PyTorch) and GBT (with lightGBM) models. Specifically we keep the same random seed for each model to avoid the randomness in model training, and only use different platforms (Tensorflow, PyTorch and lightGBM) with their different versions. We train models on the same training dataset. In Table 1 we define models as reproducible if predictions on the same testing dataset are exactly the same. Our experiment shows that different platforms cannot reproduce exactly the same model, while different versions of the same platform show good reproducibility for TensorFlow and PyTorch.

		Tensorflow		PyTorch		lightGBM		
		1.9.0	1.14.0	1.2.0	1.13.0	2.2.0	2.2.1	2.3.0
Tensorflow	1.9.0	Y	Y	N	N	-	-	-
	1.14.0	Y	Y	N	N	-	-	-
PyTorch	1.2.0	N	N	Y	Y	-	-	-
	1.3.0	N	N	Y	Y	-	-	-
lightGBM	2.2.0	-	-	-	-	Y	Y	N
	2.2.1	-	-	-	-	Y	Y	N
	2.3.0	-	-	-	-	N	N	Y

Table 1. The reproducibility across different deep learning platforms and versions.

3.4. Influence of some random factors in machine learning training

In this section, we investigate the influence of two random factors in two machine learning models that we applied to our Coordination Number prediction task, the multilayer perceptron (MLP) and gradient boosted trees (GBT). We measure the influence of one random factor at a time by fixing all the hyperparameters and other random factors (with the appropriate random seeds), and let free the factor under consideration. We train the model N ($i \in \{1,...N\}$) times, and with each trained model, we obtain an accuracy x_i on the test data. If accuracy numbers x_1^N are close to each other, we say the random factor has a small influence, in other words, the model is robust in terms of the random factor. Specifically, the metrics we used to measure the dispersion of accuracy numbers are: 1) the Coefficient of Variation (CV), also known as Relative Standard Deviation (RSD). CV is defined as standard deviation divided by mean; and 2) the Mean Absolute Difference (MAD), defined as $\frac{1}{N(N-1)}\Sigma_{i=1}^{N}\Sigma_{j=1}^{N}|x_i - x_j|$. In this work, we set $N = 5$.

3.4.1. Influence of random factors in MLP

In this section, we investigate the influence of two random factors in MLP, i.e., data order and weight initialization.

Influence of different data orders Most modern machine learning models use stochastic (or mini-batch) gradient descent (SGD) to iteratively optimize the loss function. As a result, data is fed into the model in a random order for training, and this data order adds randomness to the models. Theoretically, in gradients of different orders of (mini-batch) samples, the optimizer uses different paths to get local minimums, and those are usually different. To see how the orders affect the accuracy of the trained model, we keep all the other factors fixed and only leave the freedom for the data order. The results are: the CV of five models is 0.0933, and the MAD is 0.0464.

Influence of different weight initializations The optimizer of the modern machine learning models usually uses a random start point (weights initialization) to start the optimization process which also results in different local minimums. To see how the weight initialization affects the accuracy of the trained model, we keep all the other factors and only leave the freedom for weight initialization. In this task, as we did in last section, we also trained the model five times, and apply the mean of absolute z scores to measure the deviation of all five trained models. The results are: the CV of five models is 0.0349, and the MAD is 0.0171.

Model	MLP		GBT	
Random factor	Data order	Weight init.	Feature selection	Data selection
CV	0.0933	0.0349	0.0035	0.0063
MAD	0.0464	0.0171	0.0023	0.0043

Table 2. The influence of the random factors in model training.

3.4.2. Influence of random factors in GBT

In this section, we investigate the influence of two random factors in GBT, i.e., the feature random selections and data random selections.

Influence of random feature selections In each iteration (tree) of the training, GBT may only use a fraction of the randomly selected features to speed up the learning process while reducing overfitting. We set the features fraction at 0.5, i.e., the half of the features are randomly selected to trained in each iteration. The results are: the CV of five models is 0.0035, and the MAD is 0.0023.

Influence of random data selections In each iteration (tree) of the training, GBT may only also use a fraction of the randomly selected data to speed up the learning process. We set the data fraction is 0.5, i.e., half of the data are randomly selected to trained in each iteration. The results are: the CV of five models is 0.0063, and the MAD is 0.0043.

4. Discussion of challenges

There are many reproducibility issues in materials informatics workflows that can affect the accuracy of results when using ML algorithms. We discuss them below in sequential order:

- Theory-driven errors in the calculation of computational structures
- Numerical errors due to scalability
- Lack of reproducibility across platforms and versions
- Effects of randomness related to training the model itself
- Effects of randomness caused by domain shift when experimental results are used in the test set of a model trained on computational structures.

In cases where the fundamental approximations are not expected to be problematic it is critical to realize that important conclusions are drawn from comparisons of different results. If these results are obtained from calculations that are based on the same approximations then the comparison is likely to benefit from partial cancellations of errors. Therefore it is important to understand what errors are made in a given method and design the calculations to carefully control these errors. This in turn requires consistent choices of input data such as basis sets, energy expressions, convergence criteria, cutoffs, etc. In addition it may require comparisons of results between different codes, as well as validation of the results against more advanced methods.

In computational modeling progress in validating methods has been made through the design and development of a number of test sets. In practice the test sets are typically small (on the order of a hundred atomistic systems) but they usually include typical as well as known difficult systems for a particular property. Based on this diversity it is often

assumed that methods that do well on these sets will work well in general. In practice this assumption has to be somewhat qualified. In order for these tests to be readily usable the test cases have to be reasonably small, so that the tests can be run quickly with many different methods, codes and parameter choices. However, with increasingly large atomistic systems can come increasingly severe numerical issues. Hence the performance of a method on a small system may not be indicative for the performance on a large system. These numerical problems are mainly discovered during applications that reach beyond the scale of prior applications, simply for the reason that it is too expensive to systematically test methods on large problems.

Training models is much less computationally expensive than calculating theoretical properties, thus making it an attractive solution. Once built, models can be used to parse "on-the-fly" large amounts of experimental spectra. This is advantageous during the course of an experiment at the beamline, where scientists typically only spend a few four-hour sessions. Models can enable them to guide the course of their experiments and correct it if needed by comparing their new data out of the detector with the predicted results. When this guidance becomes automated, and computationally driven methods steer the course of experiments, the experiment itself is on the way to become autonomous, a long-term goal of the materials science community. However, this will only become possible when criteria for defining and evaluating reproducibility in ML results are well established.

It is common knowledge that training a machine learning model multiple times, even with the same data set, does not usually produce the same model, as different training and testing errors are produced with each run. Several factors influence the randomness introduced at training as seen in the previous section. Model hyperparameters, such as network structure, layer number, neuron number, optimizer, learning rate, batch size, epoch number, activation function, can be treated as a configuration file and easily written into the trained model or an external file. Some platforms can now store hyperparameters and models [29] but a common data model enabling reproducibility does not exist in materials informatics.

In Table 1, we found that TensorFlow and PyTorch cannot reproduce exactly the same model, while different versions of the same platform showed good reproducibility. For lightGBM platform, different versions may or may not reproduce exactly the same model. We would like to emphasize that our evaluations are only based on the specific platforms and versions we test in this paper. Reproducibility across different platform versions really depends on implementation updates of related functions for specific versions, such as optimizers and I/O. In other words, the fact that specific version pairs can or cannot produce the same results may just be due to our accidental choice of version pairs using the same or different implementations of related functions.

We also evaluated the reproducibility of the MLP and GBT based on some random factors. Table 2 showed that GBT has better robustness than MLP, i.e., with certain randomness, the accuracy of GBT is more consistent than MLP. It suggests that the performance of GBT may be more stable than MLP in practice. Although this is generally agreed upon in the ML community, no theoretical proof for it exists. In addition, the best results in terms of accuracy are usually the ones reported, regardless of their robustness to randomness. When ML results are re-used for additional conclusions or guiding experiments, decisions based on both robustness and accuracy will be needed.

With ML models, there are two general classes of errors. The first one that we investigated here concerns how the models perform when trained on the same training dataset with the same hyper parameters. The second one (domain shift) refers to transfer learning or domain adaption and typically draws more attention from ML researchers [30]. Researchers study how models use a training dataset not necessarily representative of the custom data to which models are applied. In our case, models trained on computational structures are used to make predictions about experimental data. Our experiment shows that the first class of errors should not be ignored by practitioners interested in applying these models in their domain science.

5. Future work

In addition to evaluate the influence of the random factors of models in high level, it is also interesting to explore the randomness in low level implementations, such cuDNN deterministic factor. In addition, and independently of the model itself, system level configurations, such as operation system, GPU drivers/library version, eg. CUDA, cuDNN versions also influence the reproducibility of results. The provenance of model execution needs to be extracted and made available.

6. Acknowledgements

This work was performed using resources of Brookhaven National Laboratory operated for the U.S. Department of Energy by Brookhaven Science Associates under contract number DE-SC0012704. The authors gratefully acknowledge some funding support from the Brookhaven National Laboratory under the Laboratory Directed Research and Development 18-05 FY 18-20.

References

[1] K. Rajan, "Materials informatics," *Materials Today*, vol. 8, no. 10, pp. 38–45, 2005. DOI: 10.1016/S1369-7021(05)71123-8.

[2] J. Hill, G. Mulholland, K. Persson, R. Seshadri, C. Wolverton, and B. Meredig, "Materials science with large-scale data and informatics: Unlocking new opportunities," *Mrs Bulletin*, vol. 41, no. 5, pp. 399–409, 2016. DOI: 10.1557/mrs.2016.93.

[3] E. O. Pyzer-Knapp, K. Li, and A. Aspuru-Guzik, "Learning from the harvard clean energy project: The use of neural networks to accelerate materials discovery," *Advanced Functional Materials*, vol. 25, no. 41, pp. 6495–6502, 2015. DOI: 10.1002/adfm.201501919.

[4] N. Wagner and J. M. Rondinelli, "Theory-guided machine learning in materials science," *Frontiers in Materials*, vol. 3, p. 28, 2016. DOI: 10.3389/fmats.2016.00028.

[5] P. Raccuglia, K. C. Elbert, P. D. Adler, C. Falk, M. B. Wenny, A. Mollo, M. Zeller, S. A. Friedler, J. Schrier, and A. J. Norquist, "Machine-learning-assisted materials discovery using failed experiments," *Nature*, vol. 533, no. 7601, p. 73, 2016. DOI: 10.1038/nature17439.

[6] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015. DOI: 10.1038/nature14539.

[7] J. Ma, R. P. Sheridan, A. Liaw, G. E. Dahl, and V. Svetnik, "Deep neural nets as a method for quantitative structure–activity relationships," *Journal of chemical information and modeling*, vol. 55, no. 2, pp. 263–274, 2015. DOI: 10.1021/ci500747n.

[8] E. National Academies of Sciences and Medicine, *Reproducibility and Replicability in Science*. May 2019. DOI: 10.17226/25303.

[9] O. E. Gundersen and S. Kjensmo, "State of the art: Reproducibility in artificial intelligence," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[10] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, "Deep reinforcement learning that matters," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[11] A. Jain, S. P. Ong, G. Hautier, W. Chen, W. D. Richards, S. Dacek, S. Cholia, D. Gunter, D. Skinner, G. Ceder, and K. A. Persson, "The Materials Project: A materials genome approach to accelerating materials innovation," *APL Materials*, vol. 1, no. 1, p. 011 002, 2013, ISSN: 2166532X. DOI: 10.1063/1.4812323. [Online]. Available: http://doi.org/10.1063/1.4812323.

[12] S. Kirklin, J. E. Saal, B. Meredig, A. Thompson, J. W. Doak, M. Aykol, and C. Wolverton, "The open quantum materials database (oqmd): Assessing the accuracy of dft formation energies," *npj Computational Materials*, vol. 1, p. 15 010, 2015. DOI: 10.1038/npjcompumats.2015.10. [Online]. Available: https://doi.org/10.1038/npjcompumats.2015.10.

[13] S. Curtarolo, W. Setyawan, S. Wang, J. Xue, K. Yang, R. H. Taylor, L. J. Nelson, G. L. Hart, S. Sanvito, M. Buongiorno-Nardelli, N. Mingo, and O. Levy, "Aflowlib.org: A distributed materials properties repository from high-throughput ab initio calculations," *Computational Materials Science*, vol. 58, pp. 227–235, 2012, ISSN: 0927-0256. DOI: 10.1016/j.commatsci.2012.02.002. [Online]. Available: https://doi.org/10.1016/j.commatsci.2012.02.002.

[14] Computational Atomic-scale Materials Design. (2014). Computational materials repository, [Online]. Available: https://cmr.fysik.dtu.dk/ (visited on).

[15] (2013). Quantum-machine.org, [Online]. Available: https://http://quantum-machine.org/ (visited on).

[16] L. Ruddigkeit, R. van Deursen, L. C. Blum, and J.-L. Reymond, "Enumeration of 166 billion organic small molecules in the chemical universe database gdb-17," *Journal of Chemical Information and Modeling*, vol. 52, no. 11, pp. 2864–2875, 2012. DOI: 10.1021/ci300415d. [Online]. Available: https://doi.org/10.1021/ci300415d.

[17] R. Ramakrishnan, P. O. Dral, M. Rupp, and O. A. von Lilienfeld, "Quantum chemistry structures and properties of 134 kilo molecules," *Scientific Data*, vol. 1, 2014. DOI: 10.1038/sdata.2014.22. [Online]. Available: https://doi.org/10.1038/sdata.2014.22.

[18] M. J. Frisch, G. W. Trucks, H. B. Schlegel, G. E. Scuseria, M. A. Robb, J. R. Cheeseman, G. Scalmani, V. Barone, G. A. Petersson, H. Nakatsuji, X. Li, M. Caricato, A. Marenich, J. Bloino, B. G. Janesko, R. Gomperts, B. Mennucci, H. P. Hratchian, J. V. Ortiz, A. F. Izmaylov, J. L. Sonnenberg, D. Williams-Young, F. Ding, F. Lipparini, F. Egidi, J. Goings, B. Peng, A. Petrone, T. Henderson, D. Ranasinghe, V. G. Zakrzewski, J. Gao, N. Rega, G. Zheng, W. Liang, M. Hada, M. Ehara, K. Toyota, R. Fukuda, J. Hasegawa, M. Ishida, T. Nakajima, Y. Honda, O. Kitao, H. Nakai, T. Vreven, K. Throssell, J. A. Montgomery, Jr., J. E. Peralta, F. Ogliaro, M. Bearpark, J. J. Heyd, E. Brothers, K. N. Kudin, V. N. Staroverov, T. Keith, R. Kobayashi, J. Normand, K. Raghavachari, A. Rendell, J. C. Burant, S. S. Iyengar, J. Tomasi, M. Cossi, J. M. Millam, M. Klene, C. Adamo, R. Cammi, J. W. Ochterski, R. L. Martin, K. Morokuma, O. Farkas, J. B. Foresman, and D. J. Fox, *Gaussian 09*, 2016.

[19] M. Valiev, E. Bylaska, N. Govind, K. Kowalski, T. Straatsma, H. V. Dam, D. Wang, J. Nieplocha, E. Apra, T. Windus, and W. de Jong, "Nwchem: A comprehensive and scalable

open-source solution for large scale molecular simulations," *Computer Physics Communications*, vol. 181, no. 9, pp. 1477–1489, 2010, ISSN: 0010-4655. DOI: 10.1016/j.cpc.2010.04.018. [Online]. Available: https://doi.org/10.1016/j.cpc.2010.04.018.

[20] R. Ditchfield, W. J. Hehre, and J. A. Pople, "Self-consistent molecular-orbital methods. ix. an extended gaussian-type basis for molecular-orbital studies of organic molecules," *J. Chem. Phys.*, vol. 54, 1971. DOI: 10.1063/1.1674902.

[21] M. J. Frisch, J. A. Pople, and J. S. Binkley, "Self-consistent molecular orbital methods 25. supplementary functions for gaussian basis sets," *J. Chem. Phys.*, vol. 80, 1984. DOI: 10.1063/1.447079.

[22] W. J. Hehre, R. Ditchfield, and J. A. Pople, "Self-consistent molecular orbital methods. xii. further extensions of gaussian-type basis sets for use in molecular orbital studies of organic molecules," *J. Chem. Phys.*, vol. 56, 1972. DOI: 10.1063/1.1677527.

[23] R. Krishnan, J. S. Binkley, R. Seeger, and J. A. Pople, "Self-consistent molecular orbital methods. xx. a basis set for correlated wave functions," *J. Chem. Phys.*, vol. 72, 1980. DOI: 10.1063/1.438955.

[24] B. P. Pritchard, D. Altarawy, B. Didier, T. D. Gibson, and T. L. Windus, "New basis set exchange: An open, up-to-date resource for the molecular sciences community," *Journal of Chemical Information and Modeling*, vol. 0, no. 0, null, 2019. DOI: 10.1021/acs.jcim.9b00725. [Online]. Available: https://doi.org/10.1021/acs.jcim.9b00725.

[25] D. Feller, "The role of databases in support of computational chemistry calculations," *J. Comput. Chem.*, vol. 17, 1996. DOI: 10.1002/(SICI)1096-987X(199610)17:13<1571::AID-JCC9>3.0.CO;2-P.

[26] K. L. Schuchardt, B. T. Didier, T. Elsethagen, L. Sun, V. Gurumoorthi, J. Chase, J. Li, and T. L. Windus, "Basis set exchange: A community database for computational sciences," *J. Chem. Inf. Model.*, vol. 47, 2007. DOI: 10.1021/ci600510j.

[27] J. Timoshenko, D. Lu, Y. Lin, and A. I. Frenkel, "Supervised machine-learning-based determination of three-dimensional structure of metallic nanoparticles," *The Journal of Physical Chemistry Letters*, vol. 8, no. 20, pp. 5091–5098, 2017. DOI: 10.1021/acs.jpclett.7b02364.

[28] Y. Lin, M. Topsakal, J. Timoshenko, L. Deyu, S. Yoo, and A. I. Frenkel, "Machine Learning Assisted Structure Determination of Metallic Nanoparticles," in *Handbook on Big Data and Machine Learning in the Physical Sciences. Vol 2: Advanced Analysis Solutions for Leading Experimental Techniques*, K. Kleese Van Dam, S. Campbell, K. Yager, and R. Farnsworth, Eds., World Scientific, Nov. 2019.

[29] R. Chard, Z. Li, K. Chard, L. Ward, Y. Babuji, A. Woodard, S. Tuecke, B. Blaiszik, M. Franklin, and I. Foster, "Dlhub: Model and data serving for science," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 283–292. DOI: 10.1109/IPDPS.2019.00038.

[30] Y. Lin, J. Chen, Y. Cao, Y. Zhou, L. Zhang, Y. Y. Tang, and S. Wang, "Cross-domain recognition by identifying joint subspaces of source domain and target domain," *IEEE Transactions on Cybernetics*, vol. 47, no. 4, pp. 1090–1101, 2016.

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200106

Documenting Computing Environments for Reproducible Experiments

Jason CHUAH, Madeline DEEDS, Tanu MALIK [a,1],
Youngdon CHOI, Jonathan L. GOODALL [b]
[a] *School of Computing, DePaul University, Chicago, IL 60637*
[b] *Dept. of Engineering Systems and Environment, Univ. of Virginia, Charlottesville, VA 22904*

Abstract. Establishing the reproducibility of an experiment often requires repeating the experiment in its native computing environment. Containerization tools provide declarative interfaces for documenting native computing environments. Declarative documentation, however, may not precisely recreate the native computing environment because of human errors or dependency conflicts. An alternative is to trace the native computing environment during application execution. Tracing, however, does not generate declarative documentation.

In this paper, we preserve the native computing environment via tracing and and automatically generate declarative documentation using trace logs. Our method distinguishes between inputs, outputs, user and system dependencies for a variety of programming languages. It then maps traced dependencies to standard package names and their versions via querying of standard package repositories. We use standard package names to generate comprehensive declarative documentation of the container. We verify the efficacy of this approach by preserving the native computing environments of several scientific projects submitted on Zenodo and GitHub, and generating their declarative documentation. We measure precision and recall by comparing with author-provided documentation. Our approach highlights over- and under-documentation in scientific experiments.

1. Introduction

Experiments in computational research are vital for establishing and validating an idea. A computational experiment typically has three components: goals, means, and claims [1]. While goals and claims are typically text-based, the means of an experiment, including experimental environment, procedures, and execution, are primarily computational. Sharing the means of an experiment is increasingly being recognized as critical toward establishing the reproducibility of results [2]. Previously sharing the means of an experiment implied sharing code and data relating to an experiment. There is increasing consensus that to establish reproducibility of results, authors must also share a description of computing environments, such as the documentation of used system libraries, configuration files, and parameters. Other users can use documentation about computing environments to build and extend environments.

[1]Corresponding Author: School of Computing, DePaul University, Chicago, IL 60637; E-mail: tanu@cdm.depaul.edu

Documenting computing environments, however, can be challenging. Typically, a user determines primary applications within an experiment's scope. The applications often depend upon complex software packages, which internally depend upon other packages. Documenting a computing environment often requires a user to know all packages and their dependencies including specific release versions. This can be too onerous for users who have not installed or built the application in different computing environments.

Recently, two prominent methods have emerged that document computing environments:

- The *container* method is a declarative method to describe application dependencies using a set of known packages. The known packages are determined either from documentation or from having a general familiarity of the application. If part of an application does not belong to any known package, the user documents this part manually.
- In the *tracing* method, a system observes the execution of an application and tracks direct or indirect references to binaries, input data files, and dependencies. The automatically tracked files comprises of all accessed package dependencies in a computing environment.

The container method is coarse-grained and is useful for documenting computing environments of standard applications, such as database servers, web servers, compilers, where an application builds from well-known packages. Systems such as Docker [3] and Singularity [4] adopt this approach. The tracing method is more fine-grained and is more useful for ad hoc, user-compiled applications where the user has either never built the application from source or does not recollect the complete dependency toolchain. Systems such as Sciunit [5,6], ReproZip [7], and CARE [8] adopt the latter approach for documenting dependencies.

While both methods document computing environments, using the documentation for establishing reproducibility of experiments is a challenge. Container methods rarely specify version numbers of binaries or packages; neither do the container engines (such as Docker) verify if the built container environment is the same as the experiment's native environment. Tracing methods are too fine-grained—at the granularity of each file in the package—and thus lose package-level semantics. Thus, while tracing methods guarantee exact native computing environments, without an accompanying documentation it is difficult to change or extend such environments.

In this paper, we define a reproducible experiment as a shared experiment that is repeated for verification and modified for establishing reproducibility. To repeat an experiment, we preserve its native computing environment via tracing. We document this environment in terms of declarative container-specific instructions. To generate such instructions, we first distinguish between inputs, outputs, user and system dependencies in a trace log for a variety of programming languages. We then map traced dependencies to standard package names and their versions via querying of standard package repositories to generate container-compatible documentation.

The advantage of this approach is that it repeats computational experiments exactly; The container contains only the necessary traced files, and its contents are declaratively documented for extensions and reproducibility of the experiment. We verify the efficacy of this approach by tracing the computing environments of several scientific projects submitted on Zenodo and GitHub. We compare their generated documentation with author-provided documentation and also if the container execution provides similar output. Results show instances of over- and under-documentation in scientific experiments.

We organize the rest of this paper as follows. We describe container and tracing methods to document computing environments in Section 2. We describe our process for generating declarative documentation in Section 3. We present our experiments in Section 4, and conclude in Section 5.

2. Documenting Computing Environments

In this section we describe the container and tracing methods for documenting computing environments. We use Docker [3] and Sciunit [5,6] as representative methods to describe the documentation.

2.1. Containers and Docker

Containers are an OS-level virtualization technique in which the virtual environment shares the OS kernel of the host environment. A container virtual environment isolates processes and files using namespaces, chroot, and cgroups. Docker is a container engine that allows users to create and maintain containers.

A *dockerfile* is a text-based file with declarative instructions defining the contents of a Docker container. The sequential instructions specify the order of execution for creating a desired image. Users typically build an image using a Dockerfile as an argument to the Docker build command. We summarize available instructions that help to document the native computing environment of an application. A Docker container can inherit infrastructure definitions from another container (`FROM` instruction). This can either be an operating system container, such as Ubuntu, but also any other existing container (e.g., with a pre-installed JDK installation). For maintenance, a dockerfile should provide the name and email of an active maintainer (`MAINTAINER` instruction). The `ENV` instruction sets environment variables. `ADD` and `COPY` instruction allow to place files into the container. A user may document a file as a URL, relative to the current path or as a zip file, which unpacks the archive within a container. `RUN` allows to execute any shell command within the container, and is often used to retrieve dependencies, and install and compile software. A container's main running process is the `ENTRYPOINT` and/or `CMD` at the end of the Dockerfile, which may subsequently fork other processes. Each instruction results in a layer in the Docker image. Thus a well documented dockerfile is a programmatic specification of the dependencies of an application.

2.2. Tracing and Sciunit

Application virtualization creates a sandbox in which it copies all files and environment variables referenced by an application. Similar to a container, a sandbox shares the host's kernel but unlike a container processes are not isolated—only files are. The sandbox engine monitors the running application process using *strace* and then copies each referenced file within a sandboxed directory. Strace internally attaches itself to the main application process using the *ptrace* system call, which monitors all the system calls of the running process [5]. Ptrace intercepts each system call to determine the running state of the process. The sandbox engine uses the arguments to file-system specific system calls to copy accessed files. Sciunit is an engine for creating and maintaining sandboxed applications.

The sandboxed creates a log of the traced system calls, which is a file-level documentation of the native computing environment. The log begins with a special "root path" which is where the application directory resides in the host system. The log contains all the dependencies identified during the reference execution audit. The sandbox engine locates the dependencies at the same path within the special root path as it identifies them in the original system. The trace log also contains interactions between processes if they *fork* or *exec* other processes and between processes and files when files are read and written. The log also stores the logical range of times that processes interacted with other processes or with files. Figure 1 shows a sample trace log of an high-energy physics application available on Zenodo [9]. Semantically the log comprises binaries, system libraries, configuration files, input data files, or temporary cache files, which must be distinguished into packages, sub-packages, and inputs to generate declarative documentation.

```
# @agent: maddie
# @machine: Linux cdm 4.15.0-66-generic #75-Ubuntu SMP x86_64 x86_64 x86_64 GNU/Linux
# @namespace: cde-root
# @subns: 1
# @fullns: cde-root.1
# @parentns: (null)
1572557755 14098 EXECVE 14104 /davix-tester /davix/build/test/functional ["./davix-tester", "--help"]
1572557755 14104 EXECVE2 -1
1572557755 14104 MEM 434176
1572557755 14104 READ /etc/ld.so.cache
1572557755 14104 CLOSE /etc/ld.so.cache
1572557755 14104 READ /usr/lib/x86_64-linux-gnu/libgtk3-nocsd.so.0
1572557755 14104 CLOSE /usr/lib/x86_64-linux-gnu/libgtk3-nocsd.so.0
1572557755 14104 READ /davix/build/src/libdavix.so.0
1572557755 14104 CLOSE /davix/build/src/libdavix.so.0
1572557755 14104 READ /usr/lib/x86_64-linux-gnu/libstdc++.so.6
1572557755 14104 CLOSE /usr/lib/x86_64-linux-gnu/libstdc++.so.6
1572557755 14104 READ /lib/x86_64-linux-gnu/libgcc_s.so.1
1572557755 14104 CLOSE /lib/x86_64-linux-gnu/libgcc_s.so.1
1572557755 14104 READ /lib/x86_64-linux-gnu/libc.so.6
1572557755 14104 CLOSE /lib/x86_64-linux-gnu/libc.so.6
1572557755 14104 READ /lib/x86_64-linux-gnu/libdl.so.2
1572557755 14104 CLOSE /lib/x86_64-linux-gnu/libdl.so.2
1572557755 14104 READ /lib/x86_64-linux-gnu/libpthread.so.0
1572557755 14104 CLOSE /lib/x86_64-linux-gnu/libpthread.so.0
1572557755 14104 READ /usr/lib/x86_64-linux-gnu/libssl.so.1.1
1572557755 14104 CLOSE /usr/lib/x86_64-linux-gnu/libssl.so.1.1
1572557755 14104 READ /usr/lib/x86_64-linux-gnu/libcrypto.so.1.1
1572557755 14104 CLOSE /usr/lib/x86_64-linux-gnu/libcrypto.so.1.1
1572557755 14104 READ /usr/lib/x86_64-linux-gnu/libxml2.so.2
1572557755 14104 CLOSE /usr/lib/x86_64-linux-gnu/libxml2.so.2
```

Figure 1. Description of trace logs in column order: (i) timestamp, (ii) process identifier, (iii) type of system call traced, and (iv) accessed file path name

3. Using Trace Logs to Generate Documentation

We describe how to automatically generate a Dockerfile from a trace log.

3.1. Generating Dockerfile Instructions

The workflow to generate a Dockerfile from trace logs is first encapsulating a computational artifact into a sciunit, and then using the trace log to distinguish between various entities namely inputs, outputs, processes, and system/user dependencies. Our method uses a representation of the trace log in terms of lineage graph to determine these entities and, once distinguished, maps them to declarative commands in the Dockerfile. We

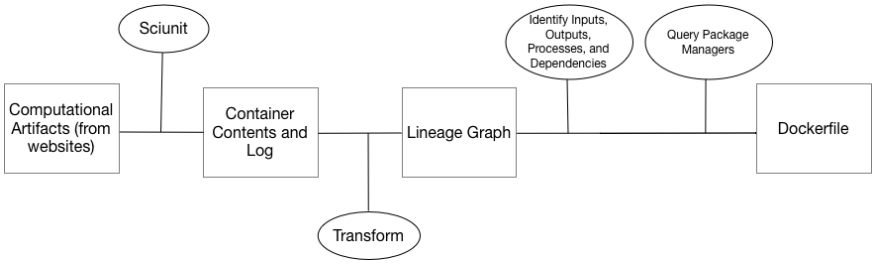


Figure 2. Workflow to generate Dockerfile declarative instructions from trace logs

manually compare generated documentation with author-supplied documentation and by building and executing the container. Figure 2 shows the workflow.

A trace logs maps to lineage graph via logged `READ/WRITE/EXECVE` calls. In particular, files represent entities and processes represent activities. Each `READ/WRITE/EXECVE` represent a data dependency. Given a lineage graph it can be used simply to determine inputs as nodes with no in-going edges, outputs as nodes with no out-going edges, and processes as nodes with process-to-process edges. The log is noisy in that it also contains information about temporary files, outputs, and process memory execution. We filter such files as this is execution-specific information and not relevant for documentation.

Distinguished files are converted to declarative instructions. Since each Dockerfile must begin with a `FROM` command, and the log provides OS information in its header as part of `@machine` command, our method instantiates an @machine specific base image. Identified input datasets and binaries are documented as `ADD` statements; Environment specific information is documented using the `ENV` command.

A bulk of the trace log consists of references to dependencies, which are either part of standard system packages or are user-defined. Distinguishing between system and user dependencies is crucial as the system documents them using different declarations. In particular, user dependencies are copied via the `COPY` command, and system dependencies are documented using the `RUN` command. One may perhaps copy system dependencies too using the `COPY` command. However, if all system dependencies are documented similar to user dependencies it leads to only one layer of the docker image. A one-command Dockerfile provides poor indication about the complexity of the software or the quality of the Dockerfile [10]. Knowing the documentation precisely is helpful if the user wishes to extend the experiment. In such cases a more verbose documentation can indicate if extended dependencies will conflict with the container contents.

Unlike a user dependency which is mentioned directly as part of a `COPY` command, a system dependency cannot be directly mapped to a `RUN` command. We must identify the corresponding package that maps to this dependency. Packages contain more than one dependency and for all identified dependencies only the corresponding package need to be stated in the `RUN` command. For instance, if the trace log specifies a path to *libcrypto.so.1.1*, then the corresponding `RUN` command is to invoke the package manager to install *libssl* as in `RUN apt-get install libssl1.1`. *libssl* also maps to *libssl.so.1.1* and *libpthread.so.0*.

To generate the above `RUN` instruction, we need to map between a dependency and its package name. For this we use programming language-specific package managers. Our method currently works for C/C++, Python and R languages. For instance C/C++

libraries are searched using *apt-get* and *yum* package managers, and Python libraries are searched through *pip*. Package managers provide a search interface to determine a package name from a dependency file[2]. Sometimes the search returns multiple packages, and our current policy is to document the first package obtained as part of the search result. Querying language-specific repository for each package is costly (˜1 min for each package query). For this we create a local database to curate all queries packages and known sub-packages to avoid repeated querying.

The process of mapping between a dependency and its package depends on how the language maintains the packages in the file system. While the most common paths where libraries are installed are /usr/lib and /lib, depending upon the language packages may be found under sub-directories site-packages, dist-packages or site-library (Python and R) or under architecture specific directories (C/C++). Typically a package name follows these paths along with the version information.

```
# information received from:
# https://zenodo.org/record/3379611
# MAINTAINER georgios.bitzes@cern.ch

# this is assumed for now.
FROM ubuntu

# build-essential: contains gcc/g++ compilers
#
# dependendencies identified from:
# libssl1.1 ------> libssl.so.1.1
# libxml2 --------> libxml.so.2
#
RUN apt-get update \
        && apt-get install -y build-essential \
        && apt-get install -y libssl1.1 \
        && apt-get install -y libxml2

# project name is automatically determined from the parent directory
WORKDIR /davix

# places the executed binary and relative dependency in the working directory '/davix'
ADD davix-tester libdavix.so.0 ./

# executes the original program with the same arguments
CMD ["./davix-tester", "--help"]
```

Figure 3. A automaticaly generated Dockerfile for Davix

Detecting packages using path information may result in false positives. In particular, Python and R interpreters check for the existence of several packages during loading. The container log is not able to distinguish between paths in which a package is checked for its existence and a path in which the the content of the package is used (e.g. when libraries are indeed imported into source code). This distinction is possible by tracing the logs at a finer granularity and determining if content was indeed read, but that increases the overhead of tracing. Since the paths of checked packaged is same as the path of a used package, we distinguish them by identifying patterns of usage. For instance, if a package is simply checked then there is an entry in specific paths such as *dist-info*, *egg-info* for Python and R, but no sub-package path or file path within a package is present. Such false positives do not arise with respect to C/C++.

[2]For R, the CRAN repository only provides a GUI-based search interface which required web-scraping to build a local database.

Finally, the sciunit contents are copied into a Docker container but the instructions for creating path directories are not documented as it does not provide any information about the native computing environment. Instead we provide as comments the location of directories which are of relevance to the user. We also use comments to document versions of packages when package managers do not install specific versions of packages. Figure 3 shows an example of a generated Dockerfile from the trace log shown in Figure 1. We would like to highlight that in case of Python applications it is sufficient to use trace logs to generate a setup file instead of a Dockerfile, and the Dockerfile can simply reference to executing the setup file. Such optimizations are outside the current scope.

4. Experiments

We collected scientific computational artifacts from GitHub and Zenodo. GitHub and Zenodo [9] artifacts are not shared using any virtualization. Our experimental setup consists of the following steps: (i) download and manually install a project; (ii) determine if they execute successfully in a new environment, possibly generating an output; (iii) execute the application to containerize under Sciunit; (iv) use the Sciunit log file to generate a Dockerfile that builds a Docker container consisting of necessary dependencies, data, and source code; and finally (v) execute the Docker and Sciunit containers to determine is same output is produced. Since our setup depended on generating a valid result, we downloaded, in total, about 100 repositories from GitHub and Zenodo. However, we could build and successfully execute only 29 repositories. Out of these, 19 are Zenodo repositories and 10 are GitHub repositories. The other repositories reported an error which we did not try to fix. The successful repositories consisted of 19 Python applications, 10 C/C++ applications, and 1 R application. The first three columns of the Table 1 shows the information. We would like to re-emphasize that since we are not original authors of these applications, we assumed the container result as the correct one if it matched with the execution.

Language	Source	# of repositories	# of Dockerfile built	# of Sciunit executed	# of Docker execution
C/C++	Zenodo	10	10	10	7
Python	Zenodo & GitHub	18	18	18	14
R	GitHub	1	1	1	1

Table 1. Repository Description

We measure two kinds of experiments: (i) if the generated Dockerfile generates the same output, and (ii) if the author provided documentation corresponds to the Dockerfile documentation. The former is determined by first building the Dockerfile and then by manually comparing the contents of the output in the container with the native execution. Table 1 shows that we are able to build Dockerfiles for all projects. However some of the C/C++ and R projects did not produce the same output. In our dataset, most C/C++ projects that did not generate a correct output had a networking component which we believe was not sufficiently documented. Currently the trace log does not audit network

events. The single R project that did not run was due to poor mapping of the dependency to a package name. This owes to the poor search interface of the R package manager. Python projects were the most stable in terms of result comparison.

To measure if the author provided documentation corresponds to the Dockerfile documentation. we measure the quality of documentation in terms of precision and recall. Precision is defined as the ratio of the number of user-listed packages identified in the trace log to all that our method can potentially identify, and recall is defined as the ratio of the number of user-listed packages to all those that our method identified. In other words, precision computes the number of application-specific packages identified amongst all identified packages, and recall computes the number of packages identified by our system which are also listed by the author in the documentation. Equations (1) and (2) state them formally.

$$Precision = \frac{Identified\ Packages}{Total\ Identified\ Packages} \tag{1}$$

$$Recall = \frac{Identified\ Packages}{User\ Listed\ Packages} \tag{2}$$

We show how these measure compare with listed dependencies in two scientific projects in Python and R, respectively. pySUMMA is a wrapper for the SUMMA [11,12] computational hydrology model in which authors list seven packages and their versions as were found compatible in their environment. Figure 4 shows the listed dependencies. As available on the GitHub repository [13], pySUMMA is not encapsulated in a container. We downloaded pySUMMA in a virtual environment and installed it with all its dependencies and their versions as specified. Figure 4 shows the five packages and their corresponding versions that were identified as application dependencies. In particular, the model does not identify *seaborn* and *jupyterthemes*: the file paths from the log show that *seaborn* is a sub-package of *matplotlib*, and so it is not a package that the author must explicitly install. *jupyterthemes* on the other hand appeared as a *dist-info* path. So even though *jupyterthemes* was listed it was not actually used. Several other packages are listed in the log, such as NetCDF41.4.2 and geopandas0.4.0/ These packages were identified within the logs and subsequently added by the authors in the setup.py file. Several other packages internal to the Python interpreter are listed as 'Python Built-in Packages'. These identified packages come bundled with standard Python environment. In Figure 4, precision is 0.083 as out of 60 packages identified, the author only listed 7. Recall is 0.714 as out of 7 that the author listed on the provided README file, only 5 were identified.

Our approach works similarly for documenting an R application. Food Inspection Evaluation (FIE) is an R application in Table 3 and includes user-defined, standard and commercial packages. However, in this case, the GitHub repository (not actively maintained) [14] does not list any R packages. It lists only a few C libraries. By downloading and creating a sciunit, we identified all R user-defined and standard packages and C dependencies. For lack of space, we omit the result, and direct the user to our Github repository of analyzed packages.

Table 2 shows the precision and recall for C/C++ and Python Zenodo applications respectively. Tables 3 shows the precision and recall results for some of the Python and R GitHub applications. Most of GitHub applications are poorly documented. For these

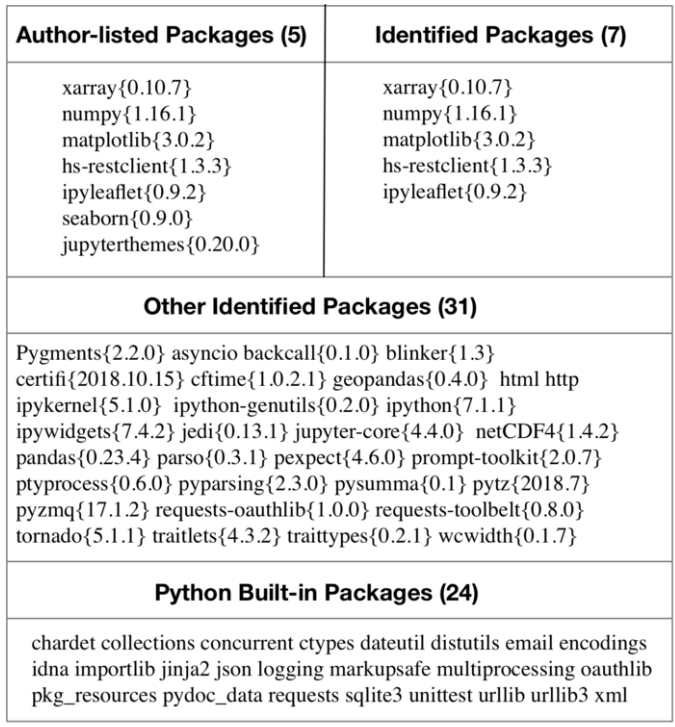


Figure 4. Listed Vs Identified Packages in pySUMMA. 2 of the Author-listed packages were not identified as they are not used by the program. We also identified 31 other packages that the author does not list.

GiHub/Zenodo Object Name	Precision	Recall
cpp-atlas	0.222	0.66
hdt-CPP	0.4	0.4
simple-web-server	0.33	0.5
research-ocr	0.024	1
c-blockchain	1	1
scram	0.25	0.66
activia	1	1
Dgtal	0	0
Davix	0.28	1
causaltrail	0	0

Zenodo Object Name	Precision	Recall
clam	0.12	0.5
informers	0.208	0.714
pydov: v0.3.0	0.12	0.75
lmfit-py 0.9.14	0.167	0.8
jungleweather	0.02	1.0
pianoplayer	0.069	1
GraSPy 0.1	0.12	1
fbpic	0.156	1
pyBathySfM v4.0	0.208	1

Table 2. Precision/Recall of C/C++ Zenodo applications (left) Precision/Recall of Python Zenodo applications (right)

applications, recall is assumed to be 1 as we consider our manual virtual machine installation as the total number of author-listed packages. All Zenodo applications were documented in that authors do list install requirements. For applications on both repositories, precision value is low since we constantly report many sub-packages that are not reported by the author. However, our recall value is on the high end as we identify most of the author-listed or required packages for the application to run.

GitHub Object Name	Precision	Recall
zagats	0.12	0.5
snake	0.208	0.714
gooselife	0.12	0.75
pySUMMA	0.167	0.8
newmeric	0.02	1.0
asplos	0.069	1
craps	0.12	1
imdb-deeplearn	0.156	1
FIE	0.208	1
Image morphing	0.208	1

Table 3. Precision/Recall of Python & R GitHub applications

5. Conclusions

In this paper we developed a model to interpret container logs and document dependencies of applications. Although precision is low, high recall shows that necessary dependencies captured during tracing can be used to build a comprehensive and verbose dockerfile. We believe this mapping from a trace provides low-overhead for users to create and maintain containers, which is increasingly important for conducting reproducible research. For full documentation of our experiments and artifacts please visit: https://tanum@bitbucket.org/geotrust/trace-descriptions.git

Acknowledgements

This work is supported in part by the Better Scientific Software Fellowship to Malik and by NSF grants ICER-1639759 and CNS-1846418.

References

[1] V. Stodden and *et. al.*, *Implementing reproducible research*. Cambridge University Press, 2013.
[2] National Academies of Sciences, Engineering, and Medicine, *Reproducibility and Replicability in Science.* Washington, DC: The National Academies Press, 2019. https://doi.org/10.17226/25303.
[3] *Docker* https://www.docker.com/, **Online; accessed 8-Jan-2019**, 2019.
[4] G. Kurtzer and *et. al.*, *Singularity: Scientific containers for mobility of compute*, PLoS One,12(5), 2017.
[5] D. H. Ton That and *et. al.*, *Sciunits: Reusable Research Objects*, IEEE eScience, 2017.
[6] Q. Pham, T. Malik, and I. Foster, *Using Provenance for Repeatability*, In USENIX Theory and Practice of Provenance (TaPP), 2013.
[7] F. Chirigati, D. Shasha, and J. Freire, *ReproZip: Using Provenance to Support Computational Reproducibility*, In USENIX Theory and Practice of Provenance (TaPP), 2013.
[8] Y. Janin and *et. al.*, *CARE, the Comprehensive Archiver for Reproducible Execution*, In the Workshop on Reproducible Research Methodologies and New Publication Models in Computer Engineering, 2014.
[9] *Zenodo* https://zenodo.org, **Online; accessed 8-Jan-2019**, 2019.
[10] J. Cito, G. Schermann, and *et. al.*, *An Empirical Analysis of the Docker Container Ecosystem on GitHub*, In International Conference on Mining Software Repositories (MSR), 2017.
[11] M. P. Clark, B. Nijssen, and et. al., A unified approach to process-based hydrologic modeling, Part 1: Modeling concept. Water Resources Research, 51.
[12] M. P. Clark, B. Nijssen, and et. al., A unified approach for process-based hydrologic modeling, Part 2: Model implementation and example applications. Water Resources Research, 51.
[13] pySUMMA. https://github.com/uva-hydroinformatics/pysumma
[14] Food Inspection Evaluation. https://github.com/Chicago/food-inspections-evaluation

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200107

Toward Enabling Reproducibility for Data-Intensive Research Using the Whole Tale Platform

Kyle CHARD [a], Niall GAFFNEY [b], Mihael HATEGAN [a], Kacper KOWALIK [c], Bertram LUDÄSCHER [d], Timothy MCPHILLIPS [d], Jarek NABRZYSKI [e], Victoria STODDEN [d,1], Ian TAYLOR [e], Thomas THELEN [f], Matthew J. TURK [d] and Craig WILLIS [c]

[a] *University of Chicago*
[b] *Texas Advanced Computing Center*
[c] *National Center for Supercomputing Applications*
[d] *University of Illinois at Urbana-Champaign*
[e] *University of Notre Dame*
[f] *University of California at Santa Barbara*

Abstract.
Whole Tale `http://wholetale.org` is a web-based, open-source platform for reproducible research supporting the creation, sharing, execution, and verification of "Tales" for the scientific research community. Tales are executable research objects that capture the code, data, and environment along with narrative and workflow information needed to re-create computational results from scientific studies. Creating reproducible research objects that enable reproducibility, transparency, and re-execution for computational experiments requiring significant compute resources or utilizing massive data is an especially challenging open problem. We describe opportunities, challenges, and solutions to facilitating reproducibility for data-and compute-intensive research, that we call "Tales at Scale," using the Whole Tale computing platform. We highlight challenges and solutions in frontend responsiveness needs, gaps in current middleware design and implementation, network restrictions, containerization, and data access. Finally, we discuss challenges in packaging computational experiment implementations for portable data-intensive Tales and outline future work.

Keywords. reproducible research, scalability, science as a service, platform as a service, scientific computing, computational science, scientific workflows, replicability, reproducibility, big data, data provenance, cyberinfrastructure, transparency

1. Introduction

In this work we explore barriers and opportunities for extending the Whole Tale infrastructure [1] to facilitate reproducible data-intensive research at scale. Creating re-

[1]Corresponding Author. E-mail: vcs@stodden.net. This material is based upon work supported by the National Science Foundation under Grant No. 1541450.

producible research objects that capture artifacts such as data, software, and sufficient details from a computational experiment to enable reproducibility, transparency, and re-execution is a challenge the research community is addressing in a variety of ways [2,3,4,5,6,7,8,9,10]. However reproducing results that require significant compute resources, rely on specialized hardware, or utilize massive data, is an especially challenging open problem. We start by describing the Renaissance Simulations [11] to provide a concrete motivating scenario for this work. We then present the current implementation of the Whole Tale reproducible research platform, and define the notion of a "Tale" as a reproducible research object published by the Whole Tale system [12,13]. We enumerate possible execution models for data-intensive computational experiments that rely on significant compute resources, specialized hardware, and/or massive data. We discuss challenges and solutions inherent in extending the Whole Tale platform this way, including frontend responsiveness needs, gaps in middleware design and implementation, network restrictions, containerization, and data access. We close with a discussion of the role of computational reproducibility in the search for scientific correctness.

We acknowledge the following contributions to this work, following the CASRAI CRediT (Contributor Roles Taxonomy) convention (see `https://casrai.org/credit/`). Conceptualization: Chard, Gaffney, Hategan, Kowalik, Willis; Funding acquisition: Ludäscher, Stodden, Turk, Chard, Nabrzyski, Gaffney; Project management: Kowalik, Willis; Software: Hategan, McPhillips, Kowalik, Taylor, Thelen, Willis; Supervision: Chard, Gaffney, Kowalik, Ludäscher, Nabrzyski, Stodden, Turk, Willis; Visualization: Hategan; Writing (original draft): Hategan, Stodden; Writing (review & editing): Chard, Ludäscher, Hategan, Stodden, Kowalik, Willis.

2. Motivating Scenario: Renaissance Simulations Laboratory

To motivate the discussion, we describe a real-world scenario based on the Renaissance Simulations. The Renaissance Simulations (RS) are some of the largest and most detailed simulations of the formation and evolution of the first galaxies. Created after three years using more than 100 million core hours, the resulting simulations enable the exploration of a variety of research questions concerning structure formation and chemical evolution in the early universe. However, the complexity, depth, and size of these simulations require researchers to have access to specialized resources for analysis. The Renaissance Simulations Laboratory (RSL) is a virtual laboratory devoted to providing access to over 70 TB of raw data and derived data products including halo catalogs, merger trees, and mock observations. RSL exposes data available on systems at the San Diego Supercomputing Center (SDSC) via Jupyter web-based interactive environments and, in a later phase of the project, enable launching jobs on SDSC Comet and other XSEDE resources [15]. Analyses conducted via RSL are intended to be shared and published to foster a collaborative research community in keeping with the long history of openness and transparency of computational research artifacts in computational astronomy.

This scenario illustrates three key challenges related to our vision of "Tales at Scale," which we define as executable research objects that capture the code, data, and environment along with narrative and workflow information needed to re-create computational results from data- and compute-intensive scientific studies. Creating such reproducible research objects to enable reproducibility, transparency, and re-execution for computa-

tional experiments requiring significant compute resources or utilizing massive data is an especially challenging open problem. We describe opportunities, challenges, and solutions to facilitating reproducibility for data-and compute-intensive research using the Whole Tale computing platform. First, the RS data are very large, impractical to transfer, and requires large-scale resources to analyze. Second, the research community leverages interactive Jupyter environments for both exploratory and primary analytical work with some analysis requiring batch compute resources. Third, the community is interested in sharing resulting research artifacts (e.g. code, derived data) for both re-execution and re-use. There are many research communities that would benefit from general-purpose infrastructure, like RSL, that enables researchers to 1) perform exploratory work via large-scale interactive environments, 2) publish reproducible research artifacts based on experiments that require HPC workloads, and 3) do both in an environment that does not require transferring data to new systems. RSL and the Whole Tale project share common platform components and Whole Tale is using the RSL as a driver for the design and implementation of solutions to common problems of computational reproducibility at scale.

3. The Whole Tale Project: Goals, Infrastructure, and Tales

In related work we have presented the Whole Tale project: a web-based and open source cyberinfrastructure platform that enables the generation and publishing of reproducible research artifacts, which we call "Tales," objects encapsulating code, data, and computational environment information [12,13]. The goal of the Whole Tale project is to enable reproducibility for computation- and data-enabled research. The approach is to transform the discovery process by uniting data products, computational pipelines, and research articles into an integrated whole. Whole Tale supports research experiments in situ, through popular analysis environments such as Jupyter and RStudio interfaces, and captures information including the code, data, provenance, and execution environment used to produce research findings. This information is then packaged by Whole Tale into an archival format called a Tale which defines a standard for executable and reproducible research objects [14]. The Tale stores explicit references to data and code used in computational experiments, both for reproducibility purposes and to permit the citation of the specific versions used in any subsequent research. A Tale can be submitted or published to an external research repository and assigned a persistent identifier by the repository. The Whole Tale platform allows users to interactively create and edit Tales and to re-run a Tale to reproduce and verify results as obtained by the original Tale creator.

The Whole Tale platform itself consists of a set of services, collectively termed the Whole Tale Services, which serve the main web interface and implement the backend services needed to support the functionality of the web interface. Whole Tale is developed as an open source project and can be deployed by third parties. The Whole Tale Services at `https:\\wholetale.org`are currently deployed on persistent resources that take the form of a number of virtual machines supporting the underlying databases, service containers, and other components. Currently, research in the Whole Tale environment is supported by a Docker swarm cluster on Jetstream virtual machines [16,17], referred to at the "Whole Tale cluster." The Whole Tale Services are responsible for launching and managing Tales. Tale "Frontends," such as a Jupyter or RStudio notebook, run

on Tale Frontend Resources, which may be co-located with Whole Tale resources. Tales that require heavy computational or data resources, the focus of this article, are considered to contain HPC Workloads, which typically consists of CPU-intensive serial or parallel (e.g., MPI) code, or code meant to be run on GPU hardware. The lifecycle management of Tale frontends and eventually High Performance Computing (HPC) workloads is handled by specialized middleware, discussed from the perspective of "Tales at Scale" will be presented in Section 5.2. The Whole Tale platform today focuses on interactive environments as Tale Frontends for both exploratory work and re-execution of published artifacts. Through this, system users are able to work in environments that they are comfortable with while gaining the benefits of the Whole Tale platform to package the results of their work. These interactive environments also provide the opportunity to instrument parts of the research process, such as software dependency identification and computational provenance. As discussed below, the focus on interactivity can be at odds with support for HPC resources where the primary interactive environment is typically a secure shell such as SSH.

4. Possible Execution Models Allowing Access to External Resources

A visual representation of the conceptual architecture described in the previous section is shown in Figure 1. In the current Whole Tale implementation, the three types of resources shown in Figure 1 are only logically distinct and currently consist of Docker containers. The middleware connecting the core services with the Tale frontends is a wrapper around Docker API calls. Tales, in turn, can launch applications that are pre-built into the Tale images using standard system calls, thereby running them inside the same container as the Tale. However, CPU-intensive applications are, in principle, still limited to being run inside the Tale containers. This limitation can be worked around by connecting to external resources and launch custom HPC workloads. Such a solution, however, hinders the ability of a Tale to capture the entirety of the environment in which it was created, negatively impacting reproducibility. This issue will be discussed in more detail in Section 5.

The architecture presented in the previous section is general in that it does not specify the precise meaning of the resources involved. The Whole Tale platform only dictates that the core services be deployed on some persistent resources. We can distinguish a number of models that refine this general architecture by assigning concrete resources to the logical ones, enumerated below. We briefly list some of their benefits and downsides, with a more detailed analysis following in Section 5. We discuss six possible approaches in turn.

The first model deploys the Tale frontend and HPC workloads on the Whole Tale cluster (Figure 2). The frontend runs on resources that are part of the Whole Tale deployment and users can launch local HPC jobs using standard system calls, jobs which would run inside the same container as the Tale. The resources available for the HPC jobs are limited to what is provided by default through the Whole Tale deployment cluster.

The second possible model deploys the Tale frontend on an HPC compute node (Figure 3). This option involves running the Tale frontend e.g. a Jupyter notebook or R-studio IDE, on compute nodes in an HPC cluster. The notebooks would then be able to launch independent HPC jobs using standard system calls. As in the previous model,

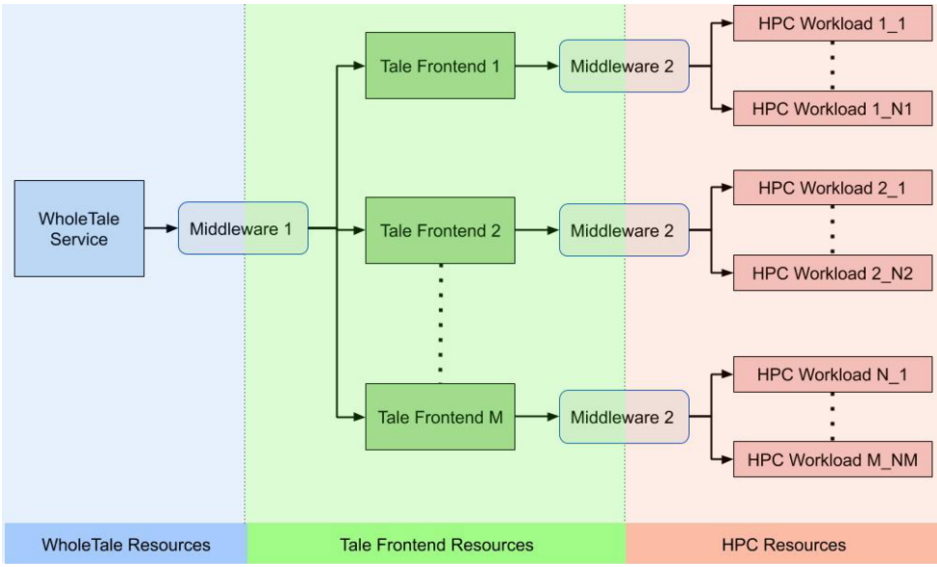


Figure 1. Conceptual high level Whole Tale architecture for executing Tales on HPC resources.

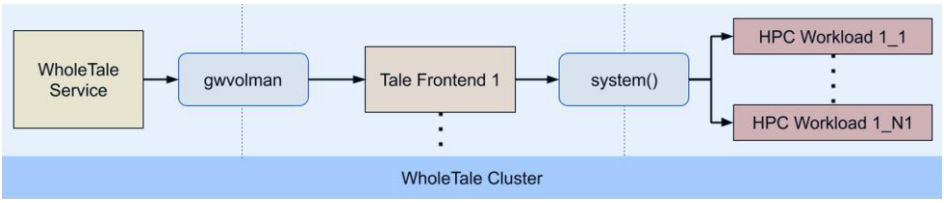


Figure 2. Tale Frontend on Whole Tale Deployment Cluster. The middleware used by the core Whole Tale services to manage Tales is named "gwvolman."

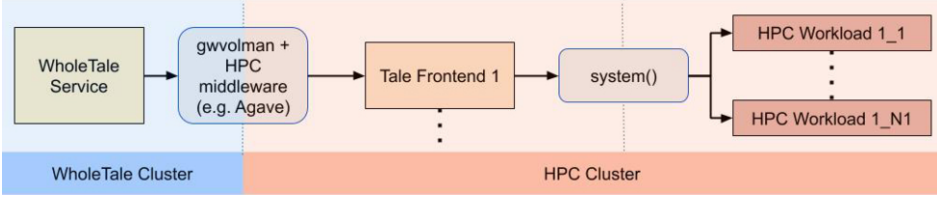


Figure 3. Tale Frontend on single HPC Compute Node.

HPC jobs would be local to Tale frontends, but can now benefit from the HPC hardware on which the frontends run.

Our third model deploys the Tale frontend on an HPC compute node with local LRM (cluster queuing system) access (Figure 4). This is a similar scenario to that shown in Figure 3, but would allow submission of HPC jobs to the queuing system of the cluster. This would enable scaling of the HPC jobs beyond what is provided by the compute resources available to the Tale frontend.

We can extend the third model to deploy the Tale frontend on HPC compute nodes

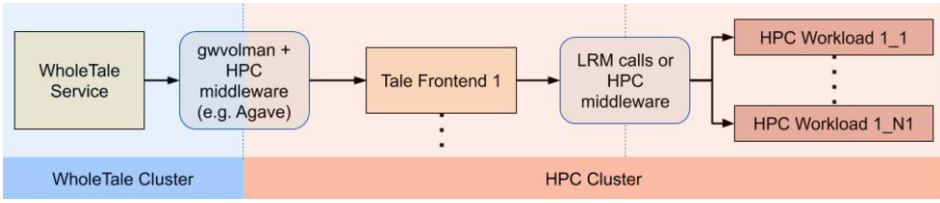


Figure 4. Tale Frontend on HPC Compute Node with Local LRM (cluster queuing system) Access.

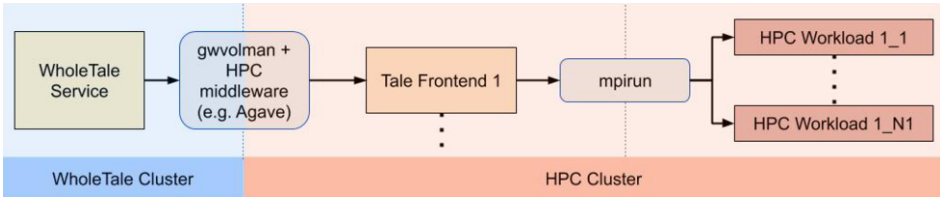


Figure 5. Tale Frontend on HPC Compute Nodes with MPI.

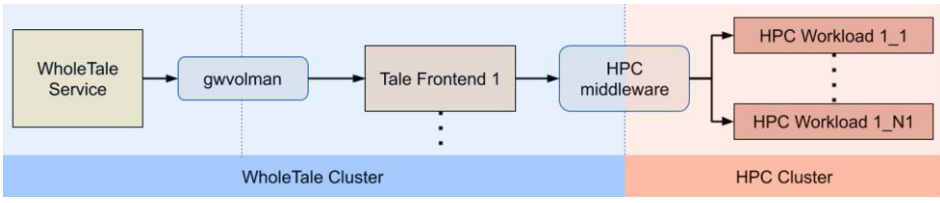


Figure 6. Tale Frontend on Whole Tale Cluster with Remote LRM Access.

with MPI (Figure 5). This involves launching the Tale frontend as an MPI job. The cluster LRM (queuing system) would allocate the number of nodes requested at the submission of the Tale frontend job and set the appropriate MPI environment. The Tale frontend would run on the lead node allocated to the MPI job by the LRM and would be able to, using "mpirun" or other cluster specific tools, launch MPI subjobs on the nodes allocated to the MPI job.

Our penutimate model deploys the Tale frontend on the Whole Tale cluster with remote LRM access (Figure 6). In this scenario, Tale frontends continue to run alongside Whole Tale core services, but HPC jobs can be submitted to remote clusters via the middleware.

The final model is a decoupled Tale frontend with LRM Remote Access. In this model, Tale frontends are allowed to run on various resources, including the Whole Tale cluster, a cloud provider, or an HPC cluster. HPC jobs, in turn, can run on any resources supported by the middleware. This model would be useful in allowing users to bypass the limitations present in the default resources provided by the Whole Tale infrastructure. A user with cloud access could request that a Tale be run on cloud resources under the user's account. Furthermore, users with access to HPC clusters could schedule HPC jobs on those resources.

5. Infrastructure Challenges for Implementing "Tales at Scale" on Whole Tale

In this section we consider challenges to consider when implementing the "Tales at Scale," model including frontend responsiveness, HPC network restrictions, and containerization.

5.1. Maintaining Responsiveness of Tale Frontends

From a usability perspective, Tale frontends need to be launched relatively quickly and predictably. When running Tale frontends on the Whole Tale cluster, Tale frontends become available in a matter of seconds. The process of building the Tale image and launching the container is not instant, but roughly equal to the amount of time it takes docker to load the Tale image from a local registry. However, if the Tale frontend is launched on a typical HPC resource, the process involves the additional step of submitting a job to the LRM and waiting for the LRM to schedule the job on compute nodes. The nature of HPC workloads (often long-running processes) means that LRMs are rarely optimized for quickly scheduling jobs. In addition, the scheduling time is also affected by the number and size of jobs belonging to other users that are already queued by the LRMs. HPC resources may also cycle through scheduled maintenance making them intermittently unavailable. These factors can make the Tale launching time on HPC resources unpredictable, and measured very rarely in seconds.

Strategies to mitigate Frontend launch responsiveness include advanced reservations and pilot jobs. Advanced reservations must be negotiated with HPC resource owners ahead of time. While it may be possible to negotiate long term advanced reservations, this cannot usually be done in an automatic fashion and is repeated whenever a new HPC resource is used. By contrast, an approach based on pilot jobs would involve maintaining a dynamic pool of placeholder HPC workloads that are already scheduled on HPC resources and can pick up actual HPC workloads instantly. From the responsiveness standpoint, solutions which place Tale frontends on the Whole Tale cluster are, therefore, preferred.

5.2. Scalability and Longevity of Middleware

Middleware that supports the "Tales at Scale" idea described in this article will allow the Whole Tale core services to launch Tales, and users to launch HPC jobs from Tales. Some examples of middleware include the Globus Toolkit [18], "Simple API for Grid Applications" (SAGA) [19], DRMAA (see Global Grid Forum `http://www.gridforum.org/`), and Agave (see The Agave Platform `http://developer.agaveapi.co/`). Most current middleware provides some level of abstraction over individual HPC resources. That is, to the programmer using the middleware, different resources are exposed through a unified API which abstracts away details such as the type of LRM that a specific HPC resource employs. The nature of the application using the middleware as well as the HPC resources will impose certain constraints on middleware implementation. There are two important issues to consider in our Tales at Scale case: middleware scalability and longevity.

A problem arises from the fact that scalability may not have been explicitly considered as an integral part of remote HPC submission library design. Such libraries may be

designed and tested with the idea of a user submitting and monitoring a relatively low number of jobs at a time. When used by a high throughput system however, such as a portal or a workflow system, problems can arise. For example middleware that wraps LRM command line tools to monitor jobs, such as "qstat," may only periodically do so, even when an invocation occurs for every job handled by the middleware, if the number of jobs is small. When the number of jobs is large however, the repeated invocation of "qstat" can overwhelm the LRM.

An additional issue preventing middleware from scaling is the management of secure connections. Establishing secure network connections requires a number of cryptographic steps which tend to be CPU-bound. If each job operation (submission, status query) requires the establishment of a secure connection and many such operators are performed on a single CPU, such as a single middleware client used by a portal or a service deployed on an HPC resource, the rate at which the operations can be performed can be limited. Addressing this issue requires aggregating operations under connections that are kept alive for some time. This is only useful however if the same middleware client/service instance are used. Associating a different client with each Tale instance can potentially reduce the benefits that would come from the sharing of secure connections, but may be necessary for different authentication credentials.

Finally, the lifetime of a remote job as seen from the client side is fundamentally asynchronous. The process involves doing some work until the job information is transmitted to the remote side, followed by waiting until the job completes or fails. It is convenient in many cases to treat such remote jobs as a synchronous process since it leads to simpler code. However, this approach can be wasteful since it involves allocating a thread for each job, a thread which spends most of its lifetime doing nothing while tying up resources. It may be important to note that both the middleware API and the implementation must be asynchronous in order to derive a scalability benefit. For example, the Python implementation of the SAGA provides an asynchronous API layer implemented as a wrapper around a thread based, synchronous core, which is unsatisfactory.

A second important feature of middleware is longevity. By longevity we mean the ability of middleware to continue to perform its intended purpose. From a reproducibility perspective, we must be particularly sensitive to this. Middleware typically used to access HPC resources can lack long term support and funding. Additionally, the incentives required to support stable software produced in a research environment do not always align with the needs of long term executability and reproducibility. A corollary is that "*-as-a-service" solutions tend to be insufficient since not only does the client side, including for example Whole Tale, lack long term support, but so does the service side. Even assuming open source and that the possibility of maintaining the client side component of the middleware exists, the service side may still pose challenges, in particular when deployment is on resources that are not under the control of either Whole Tale or users. For example, TeraGrid, the precursor to XSEDE, allowed HPC job submission to resources using specialized services which are not used by XSEDE. As a consequence, software written to work on TeraGrid using the corresponding client libraries would be unable to function today on XSEDE due to a lack of corresponding services. However, preservation of Tales can enable transparency for HPC workflows, even when re-executability is not possible.

Longevity is a difficult problem but solutions are possible. The Whole Tale project could develop a "middleware insulation service" which would allow Tales to program against an API that is fixed in time. The Whole Tale team would then maintain working

bindings from the insulation service to current HPC middleware. This requires that the Whole Tale project itself is sufficiently supported in the future, which is not a certainty. Alternatively, HPC middleware could be based on protocols that are well known and likely to endure the test of time. Specifically, SSH is nearly universally available on HPC clusters and if the list of LRM implementations is relatively small with stable interfaces, middleware that uses SSH to connect to HPC resources and invokes the relevant LRM commands would be usable even if the core Whole Tale services were unavailable. With the ability to save Tales in a format that allows them to be downloaded from repositories independently of the existence of a working Whole Tale deployment, a feature which we presently support, there is a potential path toward the long term utility of "Tales at Scale." To increase the potential long term utility of "Tales at Scale" HPC runs could be made increasingly transparent through provenance capture via the Whole Tale platform, even when re-executability may not be possible.

5.3. Managing HPC Network Restrictions

Many HPC clusters restrict incoming network connections to compute nodes from outside the cluster. Tale frontends however require incoming network connections in order to expose their user interface. Consequently, a general solution involving Tale frontends on compute nodes requires some form of proxying of connections from the Whole Tale cluster to HPC cluster compute nodes. Restrictions on incoming network connections may likely be a result of local security policies and therefore proxying, even if authenticated, may be seen as an unwelcome circumvention of such policies potentially requiring engagement with HPC resource managers. Many computing centers are now deploying Jupyter environments which opens the possibility of leveraging these resources directly.

5.4. Containerization and HPC Workloads

For the purpose of containerization, we can divide HPC workloads into unoptimized applications, optimized applications, and mixed applications. Unoptimized applications refer to various tools that use general CPU instructions and can be compiled (or are interpreted) and run on most types of hardware. Optimized applications are designed to benefit from specialized CPU instructions, such as Streaming SIMD Extensions (SSE) instructions or GPU hardware. Mixed applications consist of unoptimized driving code with optimized cores provided by specialized libraries.

From the standpoint of computational reproducibility, optimization poses a fundamental problem, since it benefits from specialization for particular hardware, whereas the ability to re-run code at different points in time and on different hardware requires abstraction. For mixed and unoptimized applications, containerization can capture the environment and dependencies of the unoptimized parts. On the other hand, when dealing with optimized applications or libraries, recording such code in optimized form leads to a dependence on specific hardware which can affect the ability for the code to be re-run if the specific hardware becomes unavailable. Even without that issue, typical optimized HPC applications are compiled for specific hardware and statically linked against libraries specific to that hardware to improve performance, rendering containers unnecessary. Unfortunately providing only precompiled executables in a Tale introduces opacity in the computation that underlies the research, implying that the relevant source code

should always be included in a Tale to enables transparency as well as reproducibility, regardless of whether otherwise reproducible binary code is included or not. In addition to including the source code in a published Tale, several possible choices regarding the inclusion of optimized binary code exist:

1. Package generic, pre-compiled, statically linked executables with Tales. These Tales will therefore be tied to a specific architecture (e.g. AMD64) and may perform suboptimally on specific HPC resources that share the architecture but not necessarily the fine details of custom instruction sets.
2. Package multiple versions of pre-compiled, statically linked executables with Tales, one for each target HPC resource. This provides performance optimality, but is tied to a specific set of resources.
3. Package source code and generic libraries required to compile the source code, which may perform suboptimally on specific HPC resources, but is not necessarily tied to a specific architecture.
4. Provide as much ancillary infrastructure to allow on-demand compilation against custom HPC libraries when a Tale is run. This may include compilation on target HPC resources or cross-compilation including possibly copies of libraries/compilers optimized for target HPC resources.

Each of these choices involves tradeoffs between reproducibility/portability and Tale size and complexity. For example, GCC on BlueGene/Q machines produces unoptimized code that, three years ago, was significantly slower than code produced by the IBM XL compiler which used specialized PPC vector instructions. However, the XL compiler is proprietary and, even if available for free, the license of proprietary tools may or may not permit redistribution in a form suitable for direct inclusion in Tales.

In addition, an instance of an experiment must decide on a concrete set of dependencies. This can lead to an overspecification, for example including irrelevant constraints on the dependencies, such as specific versions of libraries. This can be problematic if repeatability is affected when such constraints are relaxed. To give a fabricated by not atypical example, imagine an experiment uses a linear algebra library to multiply matrices and specifies that version 1.0.2 of the library should be used. When the experiment is re-run with version 1.0.2 of the library, the same results are obtained. However, when version 1.0.3 is used, the results differ. This example shows how altering dependencies can change scientific results obtained at the application level.

5.5. Data access and Data Quasi-locality

The current implementation of Whole Tale uses a cache (internally called the Data Management System, or DMS) which brings data from external collections to Whole Tale resources. It does the relevant transfer when users attempt to open the corresponding files from inside a Tale. However, different strategies are available and initiating transfers when a Tale is created is possible. This would result in a quasi locality of data since, after the initial transfer, the data is local. There are limitations to this type of solution. One is that data repositories may contain significantly more data than would be feasible to transfer and store on Whole Tale resources. On the other hand, for many applications only a small subset of the data in external repositories would be actively used at any given time. Data access and usage patterns remain an open question which could inform the viability of particular solutions.

In the event that Tale frontends and/or HPC workloads run on HPC resources on which copies of data are already available, the current Whole Tale implementation would be inefficient, since each file would be transferred once to Whole Tale resources and once for each Tale frontend instance that accesses the file. Bypassing this mechanism requires that we develop an alternative solution to the current DMS. The precise details depend on how such data are exposed on HPC resources. There are two main options. The first occurs when data are accessible on HPC resources from a POSIX file system. In this case, the solution consists simply of instructing the container system, if one is used, to mount the relevant files/directories inside the Tale container. The second option is that data are available on HPC resources through a non-POSIX interface. In this scenario, Tale initialization (or HPC task initialization) must invoke the relevant tools to transfer the files to a locally accessible location on the HPC resource.

6. Conclusions and Future Work

We have attempted to describe the challenges inherent in extending the Whole Tale platform to enable reproducible data-intensive research at scale including frontend responsiveness needs, gaps in current middleware design and implementation, network restrictions, containerization, and data access. We have provided several examples outlining possible steps forward for the Whole Tale system in advancing reproducibility for data- and compute-intensive scientific applications, related to work on the adoption of common workflow systems and the capture of detailed experiment provenance information for reproducibility [20,21,22,23,24,25].

The "Tale" specification [14], mentioned at the outset, is designed to capture and communicate information needed to reproduce the scientific results it contains (at least contemporaneously), and includes all code, data, libraries, and environment information (collectively, dependencies) necessary for an experiment to be re-run. However, regenerating identical computational results when an experiment is repeated does not necessarily indicate scientific correctness. Rerunning faulty experiments can produce the same faulty results. The task of relaxing dependencies to a necessary minimum is complicated by the fact that non-trivial software may require overspecified dependencies due to reasons that are not relevant to the experiment. For example, libraries may themselves have more or less strict constraints on their dependencies, which are due to code paths that are not exercised by a particular experiment. It is perhaps important to recognize that the role of projects like Whole Tale is primarily one of accurate recorders that remove the guesswork from the task of understanding the context in which the original experimenters obtained their results. By contrast, the task of ensuring scientific correctness is mainly left to the scientific community at large. This suggests that, faced with many technical challenges in supporting the re-executability of complex HPC workloads, a more practical short term goal might be to focus on enabling the systematic capture of provenance information, recording the conditions under which experiments were performed. This instrumentation process would still benefit from access to the interactive environment in which the experiment is run.

Through its creation of "Tales," the Whole Tale platform embraces reproducibility as a form of packaging for transparency. Today, the Whole Tale system can be used to package data- and compute-intensive research artifacts and provide an interactive en-

vironment for exploration. The Whole Tale platform could also be extended to enable researchers to launch interactive environments during the exploratory phase of their research as well which may require providing access to specialized compute resources or data. Whole Tale can achieve this goal today in the context of running on a single node. We have also described the more challenging case of capturing information regarding code, data, and the computational environment via the interactive environment provided by Whole Tale, including provenance information [25], with the goal of enabling re-execution for data-intensive workloads at scale.

References

[1] Brinckman A, Chard K, Gaffney N, Hategan M, Jones MB, Kowalik K, et al. Computing environments for reproducibility: Capturing the "Whole Tale". Future Generation Comp Syst. 2019;94:854–867. doi:10.1016/j.future.2017.12.029.

[2] Donoho DL, Maleki A, Rahman IU, Shahram M, Stodden V. Reproducible Research in Computational Harmonic Analysis. Computing in Science Engineering. 2009;11(1):8–18. doi:10.1109/MCSE.2009.15.

[3] Stodden V, Borwein JM, Bailey DH. 'Setting the Default to Reproducible' in Computational Science Research; 2013. Available from: https://sinews.siam.org/Details-Page/setting-the-default-to-reproducible-in-computational-science-research.

[4] Collberg C, Proebsting TA. Repeatability in Computer Systems Research. Commun ACM. 2016;59(3):62–69. doi:10.1145/2812803.

[5] Jimenez I, Sevilla M, Watkins N, Maltzahn C, Lofstead J, Mohror K, et al. Standing on the Shoulders of Giants by Managing Scientific Experiments Like Software. ;login: The USENIX Magazine. 2016;41(4).

[6] Yuan Z, Ton That DH, Kothari S, Fils G, Malik T. Utilizing Provenance in Reusable Research Objects. Informatics. 2018;5(1). doi:10.3390/informatics5010014.

[7] Chirigati F, Rampin R, Shasha D, Freire J. ReproZip: Computational Reproducibility With Ease. In: Proceedings of the 2016 International Conference on Management of Data. SIGMOD '16. New York, NY, USA: ACM; 2016. p. 2085–2088. Available from: http://doi.acm.org/10.1145/2882903.2899401.

[8] Monajemi H, Donoho DL, Stodden V. Making massive computational experiments painless. In: 2016 IEEE International Conference on Big Data (Big Data); 2016. p. 2368–2373.

[9] Cranmer K, Yavin I. RECAST — extending the impact of existing analyses. Journal of High Energy Physics. 2011;4:38. doi:10.1007/JHEP04(2011)038.

[10] Sandve GK, Nekrutenko A, Taylor J, Hovig E. Ten Simple Rules for Reproducible Computational Research. PLOS Computational Biology. 2013;9(10):1–4. doi:10.1371/journal.pcbi.1003285.

[11] O'Shea BW, Wise JH, Xu H, Norman ML. Probing the Ultraviolet Luminosity Function of the Earliest Galaxies With the Renaissance Simulations. The Astrophysical Journal. 2015;807(1):L12. doi:10.1088/2041-8205/807/1/l12.

[12] Chard K, Gaffney N, Jones MB, Kowalik K, Ludäscher B, Nabrzyski J, et al. Implementing Computational Reproducibility in the Whole Tale Environment. In: Proceedings of the 2nd International Workshop on Practical Reproducible Evaluation of Computer Systems, P-RECS@HPDC 2019, Phoenix, AZ, USA, June 24, 2019.; 2019. p. 17–22. Available from: https://doi.org/10.1145/3322790.3330594.

[13] Chard K, Gaffney N, Jones MB, Kowalik K, Ludäscher B, Nabrzyski J, et al. Implementing Computational Reproducibility in the Whole Tale Environment. In: Proceedings of the 2Nd International Workshop on Practical Reproducible Evaluation of Computer Systems. P-RECS '19. New York, NY, USA: ACM; 2019. p. 17–22. Available from: http://doi.acm.org/10.1145/3322790.3330594.

[14] Chard K, Gaffney N, Jones MB, Kowalik K, Ludäscher B, Nabrzyski J, et al. Implementing Computational Reproducibility in the Whole Tale Environment. In: Proceedings of the 2Nd International Workshop on Practical Reproducible Evaluation of Computer Systems. P-RECS '19. New York, NY, USA: ACM; 2019. p. 17–22. Available from: http://doi.acm.org/10.1145/3322790.3330594.

[15] Kluyver T, Ragan-Kelley B, Pacrez F, Granger B, Bussonnier M, Frederic J, et al. In: Jupyter Notebooks – a publishing format for reproducible computational workflows; 2016. p. 87–90.

[16] Stewart, C.A., Cockerill, T.M., Foster, I., Hancock, D., Merchant, N., Skidmore, E., Stanzione, D., Taylor, J., Tuecke, S., Turner, G., Vaughn, M., and Gaffney, N.I. Jetstream: a self-provisioned, scalable science and engineering cloud environment. 2015, In Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure. St. Louis, Missouri. ACM: 2792774. p. 1-8. doi:10.1145/2792745.2792774.

[17] John Towns, Timothy Cockerill, Maytal Dahan, Ian Foster, Kelly Gaither, Andrew Grimshaw, Victor Hazlewood, Scott Lathrop, Dave Lifka, Gregory D. Peterson, Ralph Roskies, J. Ray Scott, Nancy Wilkins-Diehr XSEDE: Accelerating Scientific Discovery. Computing in Science & Engineering, vol.16, no. 5, pp. 62-74, Sept.-Oct. 2014. doi:10.1109/MCSE.2014.80

[18] Foster IT. Globus Toolkit Version 4: Software for Service-Oriented Systems. J Comput Sci Technol. 2006;21(4):513–520. doi:10.1007/s11390-006-0513-y.

[19] Garreau M, Faurot W. Redux in Action. 1st ed. Greenwich, CT, USA: Manning Publications Co.; 2018.

[20] Afgan E, Baker D, van den Beek M, Blankenberg D, Bouvier D, Čech M, et al. The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2016 update. Nucleic Acids Research. 2016;44(W1):W3–W10. doi:10.1093/nar/gkw343.

[21] Ludäscher B, Altintas I, Berkley C, Higgins D, Jaeger E, Jones M, et al. Scientific Workflow Management and the Kepler System. Concurr Comput : Pract Exper. 2006;18(10):1039–1065. doi:10.1002/cpe.v18:10.

[22] Deelman E, Vahi K, Juve G, Rynge M, Callaghan S, Maechling PJ, et al. Pegasus, a workflow management system for science automation. Future Generation Computer Systems. 2015;46:17 – 35. doi:https://doi.org/10.1016/j.future.2014.10.008.

[23] Hunold S, Träff JL. On the State and Importance of Reproducible Experimental Research in Parallel Computing. CoRR. 2013;abs/1308.3648.

[24] Fursin G, Lokhmotov A, Savenko D, Upton E. A Collective Knowledge workflow for collaborative research into multi-objective autotuning and machine learning techniques. CoRR. 2018;abs/1801.08024.

[25] Pouchard L, Baldwin S, Elsethagen T, Jha S, Raju B, Stephan E, et al. Computational reproducibility of scientific workflows at extreme scales. The International Journal of High Performance Computing Applications. 2019;33(5):763–776. doi:10.1177/1094342019839124.

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

Subject Index

Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

Author Index

This page intentionally left blank