Julian Martin Kunkel
Thomas Ludwig
Hans Werner Meuer (Eds.)

LNCS 7905

Supercomputing

28th International Supercomputing Conference, ISC 2013
Leipzig, Germany, June 2013
Proceedings

Lecture Notes in Computer Science 7905

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Julian Martin Kunkel Thomas Ludwig
Hans Werner Meuer (Eds.)

Supercomputing

28th International Supercomputing Conference, ISC 2013
Leipzig, Germany, June 16-20, 2013
Proceedings

Springer

Volume Editors

Julian Martin Kunkel
University of Hamburg
Department of Informatics
Bundestraße 45a
20146 Hamburg, Germany
E-mail: julian.martin.kunkel@informatik.uni-hamburg.de

Thomas Ludwig
Deutsches Klimarechenzentrum
Bundestraße 45a
20146 Hamburg, Germany
E-mail: ludwig@dkrz.de

Hans Werner Meuer
University of Mannheim, Germany
and
Prometeus GmbH
Fliederstraße 2
74915 Waibstadt, Germany
E-mail: hans.meuer@isc-events.com

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-38749-4 e-ISBN 978-3-642-38750-0
DOI 10.1007/978-3-642-38750-0
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2013939202

CR Subject Classification (1998): C.1, B.8, C.4, C.3, I.3.1, C.5, F.2, D.3.4, D.4, C.2.4

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

The International Supercomputing Conference, founded in 1986 as the Supercomputer Seminar by Professor Hans Meuer, has been held annually for the last 27 years. Originally, the seminar brought together a group of 81 scientists and industrial partners who all shared an interest in high-performance computing. Since then, the annual conference has become a major international event within the HPC community. Accompanying its growth in size over the years, the conference has moved from Mannheim via Heidelberg, Dresden, and Hamburg to Leipzig. With 2,400 attendees and 175 exhibitors from over 50 countries in 2012, we are optimistic that this steady growth of interest will also turn ISC'13 into a powerful and memorable event.

In 2007 we decided to strengthen the scientific part of the conference by presenting selected talks on relevant research results within the HPC field. These research paper sessions began as a separate day preceding the conference, where slides and accompanying papers were made available via the conference website. The research paper sessions have since evolved into an integral part of the conference, and this year the scientific presentations are scheduled over three days. The call for participation was issued in winter 2012, inviting researchers and developers to submit the latest results of their work as full research papers to the sessions Program Committee.

A total of 89 paper abstracts were submitted from authors all over the world. In a peer-review process an international committee selected the best 35 papers for publication and for presentation in the research paper sessions. This year's review process was modified to improve the overall quality of the reviews by including an early abstract submission and author rebuttal phase.

We are pleased to announce that many fascinating topics in HPC were presented this year. The papers address the following issues regarding the development of an environment for exascale supercomputers:

- Scalable applications with 50K+ cores
- Performance improvements in algorithms
- Accelerators
- Performance analysis and optimization
- Library development
- Administration and management of supercomputers
- Energy efficiency
- Parallel I/O
- Grid and cloud

We believe that this selection is highly appealing and that the presentations will foster inspiring discussions with the audience.

As in the previous years, two independent award committees selected two papers considered to be of exceptional quality and worthy of special recognition.

- The Gauss Centre for Supercomputing sponsors the Gauss Award. This award is assigned to the most outstanding paper in the field of scalable supercomputing.
- PRACE, the Partnership for Advanced Computing in Europe, awards a prize to the best scientific paper by a European student or scientist.

The award winners are announced on the website of ISC'13.

We would like to express our gratitude to all our colleagues for submitting papers to the ISC scientific sessions, as well as to the members of the Program Committee for organizing this year's attractive program.

June 2013

Julian M. Kunkel
Thomas Ludwig
Hans Meuer

Organization

Program Committee

Pavan Balaji	Argonne National Laboratory, USA
Venkatramani Balaji	Princeton University, USA
Mahdi Bohlouli	University of Siegen, Germany
Xing Cai	Simula Research Laboratory, Norway
Anne Elster	Norwegian University of Science and Technology, Norway
Michael Gerndt	Technische Universität München, Germany
Lutz Gross	University of Queensland, Australia
David Ham	Imperial College London, UK
Frank Hannig	Friedrich Alexander University Erlangen-Nürnberg, Germany
Magne Haveraaen	University of Bergen, Norway
Huynh Phung Huynh	A*STAR, IHPC, Singapore
Kenichi Itakura	JAMSTEC, Japan
Oleksiy Koshulko	Glushkov Institute of Cybernetics NAS, Ukraine
Julian M. Kunkel	Deutsches Klimarechenzentrum, Germany
Dong Li	Oak Ridge National Lab, USA
Fang-Pang Lin	National Center for High-Performance Computing, Taiwan
Thomas Ludwig	Deutsches Klimarechenzentrum, Germany
Muniyappa Manjunathaiah	University of Reading, UK
Simon McIntosh-Smith	University of Bristol, UK
Bernd Mohr	Jülich Supercomputing Center, Germany
Pierre Moinier	BAE Systems, UK
Alexander Moskovsky	RSC SKIF, Russia
Matthias Müller	RWTH Aachen, Germany
Kengo Nakajima	University of Tokyo, Japan
Alexander Nemukhin	M.V. Lomonosov Moscow State University, Russia
Ramon Nou	Barcelona Supercomputing Center, Spain
Julio Ortega	University of Granada, Spain
Ying Qian	King Abdullah University of Science and Technology, Saudi Arabia
Michael M. Resch	HLRS Stuttgart, Germany
Henk Sips	Delft University of Technology, The Netherlands
Alexandros Stamatakis	HITS, Germany

Guangming Tan	Chinese Academy of Sciences, China
Osamu Tatebe	University of Tsukuba, Japan
Jeyan Thiyagalingam	Oxford University, UK
Matthew E. Tolentino	Intel, USA
Yuichi Tsujita	Kinki University, Japan
Zhonglei Wang	Intel Mobile Communications, Germany
Thomas Wild	Technische Universität München, Germany
Ying Zhao	Tsinghua University, China
Mikhail Zhizhin	CIRES/NOAA Boulder and Space Research Institute, Russia

External Reviewers

Ian Bland	Huynh-Bach Khoa	Christian Schmitt
Konstantinos Chasapis	Mian Lu	Konstantin Solnushkin
Mihai-C. Duta	Richard Membarth	Marc Wiedemann
Joel Fenwick	Xiaoxuan Meng	Michaela Zimmer
Andriy Golovinsky	Gihan Mudalige	
Jeff Hammond	Antonio Pena	
Qi Hu	Istvan Reguly	

Table of Contents

591 TFLOPS Multi-trillion Particles Simulation on SuperMUC

Wolfgang Eckhardt[1], Alexander Heinecke[1],
Reinhold Bader[2], Matthias Brehm[2], Nicolay Hammer[2], Herbert Huber[2],
Hans-Georg Kleinhenz[2], Jadran Vrabec[3], Hans Hasse[4], Martin Horsch[4],
Martin Bernreuther[5], Colin W. Glass[5], Christoph Niethammer[5],
Arndt Bode[1,2], and Hans-Joachim Bungartz[1,2]

[1] Technische Universität München, Boltzmannstr. 3, D-85748 Garching, Germany
[2] Leibniz-Rechenzentrum der Bayerischen Akademie der Wissenschaften, Boltzmannstr. 1, D-85748 Garching, Germany
[3] University of Paderborn, Warburger Str. 100, D-33098 Paderborn, Germany
[4] Laboratory of Engineering Thermodynamics (LTD), TU Kaiserslautern, Erwin-Schrödinger-Str. 44, D-67663 Kaiserslautern, Germany
[5] High Performance Computing Centre Stuttgart (HLRS), Nobelstr. 19, D-70569 Stuttgart, Germany

Abstract. Anticipating large-scale molecular dynamics simulations (MD) in nano-fluidics, we conduct performance and scalability studies of an optimized version of the code `ls1 mardyn`. We present our implementation requiring only 32 Bytes per molecule, which allows us to run the, to our knowledge, largest MD simulation to date. Our optimizations tailored to the Intel Sandy Bridge processor are explained, including vectorization as well as shared-memory parallelization to make use of Hyperthreading. Finally we present results for weak and strong scaling experiments on up to 146016 Cores of SuperMUC at the Leibniz Supercomputing Centre, achieving a speed-up of 133k times which corresponds to an absolute performance of 591.2 TFLOPS.

Keywords: molecular dynamics simulations, highly scalable simulation, vectorization, Intel AVX, SuperMUC.

1 Introduction and Related Work

MD simulation has become a recognized tool in engineering and natural sciences, complementing theory and experiment. Despite its development for over half a century, scientists still quest for ever larger and longer simulation runs to cover processes on greater length and time scales. Due to the massive parallelism MD typically exhibits, it is a preeminent task for high-performance computing.

An application requiring large-scale simulations is the investigation of nucleation processes, where the spontaneous emergence of a new phase is studied [8]. To enable such simulations, we optimized our program derived from the code `ls1 mardyn`. A description of `ls1 mardyn` focusing on use cases, software structure and load balancing considerations can be found in [1]. Based on the further development of the memory optimization described in [3], an extremely low memory

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 1–12, 2013.

requirement of only 32 Bytes per molecule has been achieved, which allows us to carry out the to our knowledge largest MD simulation to date on SuperMUC at Leibniz Supercomputing Centre. In order to run these large-scale simulations at satisfactory performance, we tuned the implementation of the molecular interactions outlined in [2] to the Intel Sandy Bridge processor and added a newly developed shared-memory parallelization to make use of Intel Hyperthreading. Thereby, this contribution continues a series of publications on extreme-scale MD. In 2000, Roth [18] performed a simulation of $5 \cdot 10^9$ molecules, the largest simulation ever at that time. Kadau and Germann [4, 10] followed up, holding the current world record with 10^{12} particles. These simulations demonstrated the state of the art on the one hand, and showed the scalability and performance of the respective codes. More recent examples include the simulation of blood flow [15] as well as the force calculation of $3 \cdot 10^{12}$ particles by Kabadshow in 2011 [9], however without calculating particle trajectories.

The remainder of the paper is organized as follows: this Section describes the computational model of our simulation code. Section 2 describes the architecture of SuperMUC, Section 3 details the implementation with respect to vectorization and memory-efficiency, and Section 4 presents the results.

The fluid under consideration is modeled as a system of N discrete particles. Only particles i and j separated by a distance r_{ij} that is smaller than a cut-off radius r_c interact pairwise through the truncated and shifted Lennard-Jones potential [19], which is determined by the usual Lennard-Jones-12-6 potential (LJ-12-6) $U_{LJ}\left(r_{ij}\right)$ with the potential parameters ϵ and σ:

$$U_{LJ}\left(r_{ij}\right) = 4\epsilon \cdot \left(\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right).$$

The interactions with all neighbors results in a force $F_i = \sum_{j \in particles} F_{ij}(r_{ij})$ on each of the particles, which is evaluated only once per particle pair, due to Newton's law $F_{ij} = -F_{ji}$.

In MD, the most time-consuming step is the force calculation. To efficiently search for neighboring particles, the linked-cell algorithm is employed in a similar way as in [10]. The computational domain is subdivided into cubic cells with an edge length r_c. Consequently, for a given particle, the distances to all other particles contained in the same cell as well as in the (in 3D) 26 adjacent cells have to be computed. This results in a linear complexity of the force calculation. The particles' data are stored in dynamic arrays, i.e. contiguous memory blocks, per cell, to avoid additional memory for pointers. Thus, the organization of the linked-cells data structure causes only small overhead.

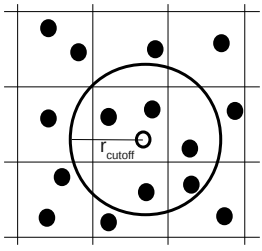


Fig. 1. Schematic of the linked-cell algorithm (2D)

In accordance with preceding large-scale simulations [4], single-precision variables are used for the calculation. For a particle we store only its position ($3 \cdot 4$ Bytes), velocity ($3 \cdot 4$ Bytes) and an identifier (8 Bytes), i.e. 32 Bytes in total.

The force vector does not need to be stored permanently, because the time integration of the equations of motion is carried out on the fly, as detailed in Section 3.

To evaluate our implementation, single-center Lennard-Jones particles were distributed on a regular grid according to a body-centered cubic lattice, with a liquid-like number density of $\rho\sigma^3 = 0.78$, and the cut-off radius was specified to be $r_c = 3.5\sigma$. The time step length was set to 1 fs.

For the MPI parallelization, we employ a spatial domain decomposition scheme. For n processes, the domain is divided in n equally-sized sub-domains, which are assigned to one process each. Each sub-domain is surrounded by a layer of ghost cells residing on neighboring processes, so the particles at the process boundaries have to be exchanged at the beginning of each time step.

2 SuperMUC – The World's Largest x86 Machine

2.1 System Topology

We optimized our MD code on the micro-architecture level for a specific processor: the Intel Sandy Bridge EP driving SuperMUC operated at the Leibniz Supercomputing Centre in Munich. This system features 147456 cores and is at present the biggest x86 system worldwide with a theoretical double precision peak performance of more than 3 PFLOPS, placed #6 on the current Top500 list. The system was assembled by IBM and features a highly efficient hot-water cooling solution. In contrast to supercomputers offered by Cray, SGI or even IBM's own BlueGene, the machine is based on a high-performance commodity network: a FDR-10 infiniband pruned tree topology by Mellanox. Each of the 18 leafs, or islands, consists of 512 nodes with 16 cores at 2.7 GHz clock speed (turbo mode is disabled) sharing 32GB of main memory. Within one island, all nodes can communicate at full FDR-10 data-rate. In case of inter-island communication, four nodes share one uplink to the spine switch. Since the machine is operated diskless, a significant fraction of the nodes' memory has to be reserved for the operation environment.

2.2 Intel Sandy Bridge Architecture

After a bird's eye view on the entire system, we now focus on its heart, the Intel Sandy Bridge EP processor that was introduced in January 2012, featuring a new vector instruction set called AVX. In order to execute code with high performance and to increase the core's instructions per clock, major changes to the previous core micro-architecture code-named Nehalem have been applied. These changes are highlighted by italic characters in Fig. 2. Since the vector-instruction width has been doubled with AVX (AVX is available with two vector widths: AVX128 and AVX256), also the load port's (port 2) width needs to be doubled. However, doubling a load port's width would impose tremendous changes to the entire chip architecture. In order to avoid this, Intel changed two ports by additionally implementing in each port the other port's functionality as shown for

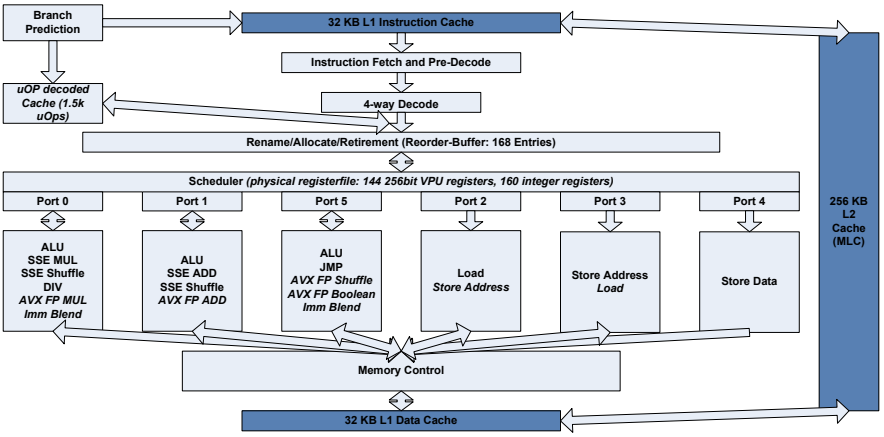


Fig. 2. Intel Sandy Bridge, changes w.r.t. Intel Nehalem are highlighted with italic characters: trace-cache for decoded instructions, AVX support and physical register file

ports 2 and 3. Through this trick, the load bandwidth has been doubled from 16 Bytes to 32 Bytes per cycle at the price of reduced instruction level parallelism. Changes to the ALUs are straightforward: ports 0, 1 and 5 were simply doubled, and they provide classic SSE functionality for AVX instructions and extensions for blend and mask operations. However, this bandwidth improvement still does not allow for an efficient exploitation of AVX256 instructions as this would require a 64 Bytes per cycle load and 32 Bytes per cycle store bandwidth. This increase will be implemented with the up-coming Haswell micro-architecture [14]. Due to 32 Bytes load bandwidth and the non-destructive AVX128 instruction set, AVX128 codes can often yield the same performance as AVX256 on Sandy Bridge but much better than SSE4.2 on an equally clocked Nehalem chip. This can also be attributed to the fact that 16 Bytes load instructions have a three times higher throughput (0.33 cycles) than 32 Bytes load instructions (here ports 2 and 3 have to be paired and cannot be used independently). According to experiments we did with different applications and kernels using AVX256 on Sandy Bridge, the full performance enhancement of 2× speed-up can be just exploited for kernels which can be perfectly register-blocked, e.g. DGEMM [7]. If in contrast only a standard 1D-blocking is possible, roughly a 1.5-1.6 × speed-up can be achieved in comparison to AVX128 [6].

Up to Nehalem, each unit had dedicated memory for storing register contents for executing operations on them. A so-called out-of-order unit took care of the correctness of the execution pipeline. With AVX, a register allocation in each compute unit of the core would be too expensive in terms of transistors required, therefore a so-called *register file* was implemented: Register contents are stored in a central directory. Shadow registers and pointers allow for an efficient out-of-order execution. Furthermore, a general performance enhancement was added

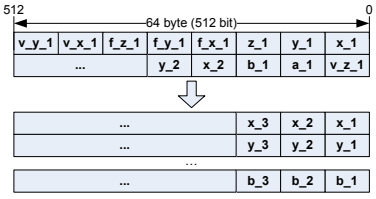
to the Sandy Bridge architecture: a cache for decoded instructions. This trace-cache like cache boosts the performance of kernels with small loop bodies, such as the force calculation in MD. Furthermore, the Sandy Bridge EP cores feature Intel's SMT implementation called *Hyperthreading Technology* which helps to increase the core's utilization in workload scenarios where the instruction mix is not optimal or the application is suffering from high memory latencies.

3 Implementation

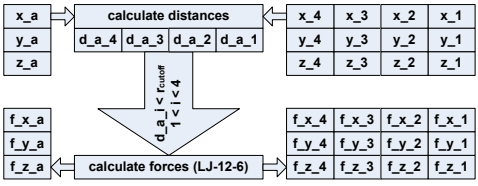
3.1 Vectorization of the Compute Kernel

Since our simulation code is written in C++ and therefore applies standard object-oriented design principles with cells and particles being single entities, we follow an approach of memory organization and vectorization, first sketched in [2]. That work describes, by using a simple proxy application and not the entire `ls1 mardyn` code base, how the LJ-12-6 force calculation inside a linked cell algorithm can be vectorized on x86 processors. That prototype implementation does not feature important statistical measurements such as virial pressure and potential energy which we added in this work.

The object-oriented memory layout is cache-efficient by design because particles belonging to a cell are stored closely together. However, implementing particles in a cell as a so-called *array of structures (AoS)* forbids easy vectorization, at least without gather and scatter operations (see [5]) which, unfortunately, are not available on Intel Sandy Bridge. Only in simple cases (e.g., updates of one member, etc.) this drawback does not matter, because prefetch logic inside the hardware loads only cache-lines containing data which have to be modified.



(a) *AoS to SoA conversion*: In order to allow for efficient vectorization, corresponding elements have to be stored for data streaming access.



(b) *Kernel vectorization*: The vectorization of the LJ-12-6 force calucation is optimized by duplicating one particle and streaming four other particles.

Fig. 3. Optimizing LJ-12-6 force calculation by SoA storage scheme and vectorization

Implementing the LJ-12-6 force calculation on AoS-structures poses major challenges: The upper part of Fig. 3a shows elements scattered across several cache-lines. Taking into account that only a small portion of the members is needed for the force calculation, a temporary *structure of arrays (SoA)* can be

constructed in order to address cache-line pollution and vectorization opportunities, illustrated in the lower part of Fig. 3a. Figure 3b sketches the applied vectorization of the LJ-12-6 calculations. In contrast to other methods which vectorize across the spatial coordinates [11–13], the present approach can exploit vector-units of arbitrary length.

In this work, single-precision AVX128 instructions were employed. The calculation is performed on particle pairs, therefore we broadcast-load the required data of one particle in the first register (a), the second register is filled by data from four other particles (1, 2, 3 and 4). Dealing with four particle pairs at once, we can theoretically reduce the number of operations by a factor of four. Since the force calculation may be required for all, some or none of the pairs in the vector register, we need to apply some pre- and post-processing performed by regular logical operations: It has to be determined, if for any particle pair the distance is smaller than r_c (pre-processing), because only then the force calculation has to be executed. If the force calculation has been executed, the calculated results need to be zeroed by a mask for all particle pairs whose distance is larger than r_c (post-processing). In order to ensure vectorization of the kernel we employed intrinsics. Due to the cut-off radius `if`-condition inside the inner-most loop, current compilers (gcc and icc) deny to vectorize the loop structure iterating over particles in cell-pairs. For the chosen simulation scenario (cut-off radius $r_c = 3.5\sigma$) a speed-up of 3 $\times$ is possible on a single core by using the proposed SoA-structure and vectorization.

With increasing vector length, this masking technique becomes the major bottleneck. Here, it can easily happen that more elements are being masked than elements which have to be computed. Therefore, moving to a wider vector-instruction set may result in more instructions being executed. However, if the vector-instruction set features *gather* and *scatter* instructions, this issue can be overcome because only the particle pairs taking part in the interaction are processed, which has been successfully demonstrated by Rapaport with the layered-linked-cell algorithm [16, 17]. The first x86 processor which offers full gather/scatter support is the so-called Xeon Phi coprocessor. Enabling `ls1 mardyn` for Xeon Phi is ongoing research.

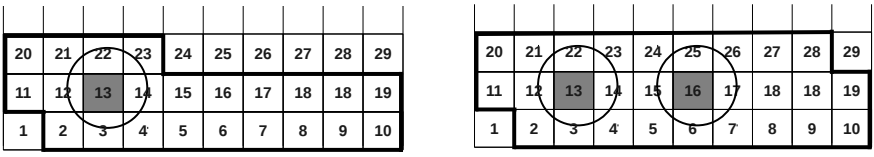
A different issue inhibiting the most efficient usage of the Sandy Bridge core is the lack of instruction level parallelism in the compute kernel. The evaluation of distance, potential energy and force on the particles requires significantly more multiplications than additions, thus the ADD unit cannot be fully utilized. Even worse, the calculation of the power-12-term of the LJ-12-6 requires a sequence of dependent multiplications. Therefore, the superscalarity of a Sandy Bridge core can not be exploited optimally, a fact we address by using Hypterthreading Technology as described below.

We restricted ourselves to AVX128 instructions for several reasons. In Section 2.2 we described that Intel Sandy Bridge is not able to handle AVX256 instructions at full speed. This fact would also forbid to use Hyperthreading efficiently as currently ports 2 and 3 inside the core can be used by different threads. Switching to AVX256, these ports are operated in paired mode, available to just

one of both threads. Furthermore, we showed in the outlook of [2] that AVX256 instruction are only beneficial when increasing the cut-off radius. Last but not least we want to ensure that `ls1 mardyn` runs best on various x86 platforms. Besides Intel Sandy Bridge, AMD Interlagos plays an important role since this chip is used as processor in most of Cray's supercomputers. AMD Interlagos features two 128bit FPUs shared between two integer units. Therefore an AVX128 code is essential for best performance on Interlagos. With the current code base we only expect slight changes when moving to an Interlagos based machine.

3.2 Memory and Utilization Optimizations

In order to achieve the low memory requirement of only 32 Byte per molecule, we refined the linked-cells algorithm with the sliding window that was introduced in [3]. It is based on the observation that the access pattern of the cells can be described by a sliding window, which moves through the domain. After a cell has been searched for interacting particles for the first time in a time step, its data will be required for several successive force calculations with particles in neighboring cells. If the force calculation proceeds according to the cells' index as depicted in Fig. 4a, these data accesses happen within a short time period, until the interactions with all neighbors have been computed. While the cells in the window are accessed several times, they naturally move in and out of the window in FIFO order.



(a) Sliding window (cells in bold black frame) in 2D. Particles in cells in the window will be accessed several times, cells 2 through 23 are covered by the window in FIFO order. For the force calculation for the molecules in cell 13, cell 23 is searched for interacting particles for the first time in this iteration. The particles in cell 2 are checked for the last time for interactions.

(b) Extension of the sliding window for multi-threading. By increasing the window by 5 cells, two threads can independently work on three cells each: thread 1 works on cells 13, 14, 15; thread 2 works on cells 16, 17, 18. To avoid that threads work on same cells (e. g., thread 1 on the cell pair 15–25, thread 2 on 16–25), a barrier is required after each thread finished its first cell.

Fig. 4. Basic idea of the sliding window algorithm and extension for multi-threading

Particle data outside the the sliding window are stored in form of C++ objects in AoS-manner, only with position, velocity and an identifier. Per cell, particle objects are stored in dynamic arrays. When the sliding window is shifted further and covers a new cell, the positions and velocities of the particles in that cell are

converted to SoA-representation. Additionally, arrays for the forces have to be allocated. The force calculation is now performed on the particles as described above. When a cell has been considered for the last time during an iteration, its particles are converted back to AoS-layout. Therefore, the calculation of forces, potential energy and virial pressure can be performed memory- and runtime-efficiently on the SoA, while the remaining parts of the simulation code can be kept unchanged according to their object-oriented layout. To avoid the overhead of repeated memory (de-)allocations when particle data in a cell are converted, we initially allocate dynamic arrays fitted to the maximum number of particles per cell for each cell in the window, and reuse that memory. Since the sliding window covers three layers of cells, these buffers consume a comparably small amount of memory, while the vast majority of the particles is stored memory-efficiently. At this point, it becomes apparent that the traversal order imposed by the sliding window also supports cache reusage: when particle data are converted to SoA-representation, that data are placed in the cache and will be reused several times soon after.

In order to reduce the memory requirement to 32 Byte per particle and to further improve the hardware utilization, this algorithm needs two further revisions: the time integration has to be performed on the fly, and opportunity for multi-threading needs to be created. Since the forces are not stored with the molecule objects, the time integration has to be performed during that conversion, i. e., the particles' new positions and velocities have to be calculated at that moment. Nevertheless, the correct traversal of the particles is ensured, because cells that have been converted are not required for the force calculation during this time step any more and the update of the linked-cells data structure, i.e. the assignment of particles to cells, takes place only between two time steps.

As stated above, the LJ-12-6 kernel is not well instruction-balanced, impeding the use of the superscalarity of a Sandy Bridge core. In order to make use of Hyperthreading Technology, we implemented a lightweight shared-memory parallelization. By extending the size of the sliding window as shown in Fig. 4b, two threads can perform calculations concurrently on three independent cells. Exploiting Newtons third law $F_{ij} = -F_{ji}$ for the force calculation and considering cell pairs only once, it must be avoided that threads work on directly neighboring cells simultaneously. Therefore, a barrier, causing comparably little overhead on a Hyperthreading core, is required after each thread has processed the first of its three cells. This allows the execution of one MPI rank per core with two (OpenMP-)threads to create sufficient instruction level parallelism, leading to a 12% performance improvement.

4 Strong and Weak Scaling on SuperMUC

In order to evaluate the performance of the MD simulation code `ls1 mardyn`, we executed different tests on SuperMUC. With respect to strong scaling behavior, we ran a scenario with $N = 4.8 \cdot 10^9$ particles, which perfectly fits onto 8 nodes; 18 GB per node are needed for particle data. Fig. 5 shows that a very good

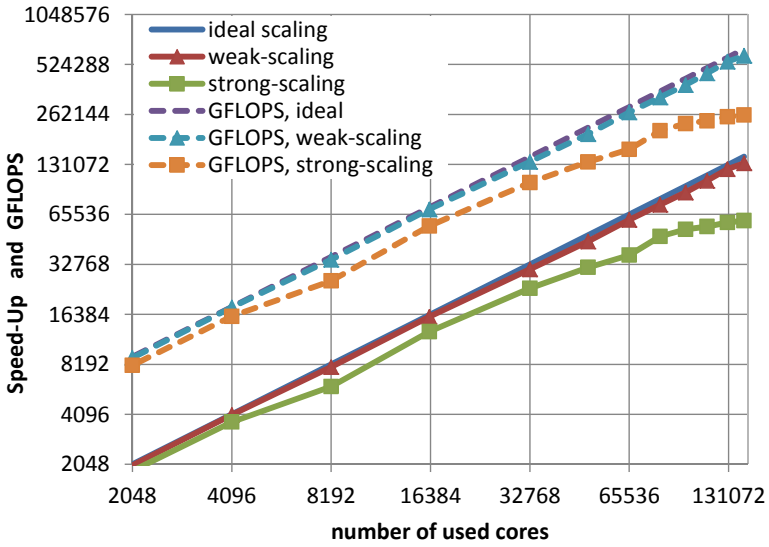


Fig. 5. Weak and strong scaling for 2048 to 146016 cores with respect to speed-up and GFLOPS on SuperMUC. Ideal scaling was achieved in case of weak scaling whereas as a parallel efficiency of 42% was obtained in the strong scaling tests. We cut off the plot at 2048 cores, here we obtained a parallel efficiency of 91.1% in case of strong scaling (compared to 128 cores) and 98.6% in case of weak scaling (compared to one core).

scaling was achieved for up to 146016 cores using 292032 threads at a parallel efficiency of 42 % comparing 128 to 146016 cores.

In this case, less than 20 MB ($5.2 \cdot 10^5$ particles) of main memory per node, which fits basically into the processors' caches, are used. This excellent scaling behavior can be explained by analyzing Fig. 6. Here we measured achievable GFLOPS depending on the number of particles simulated on 8 nodes. Already for $N = 3 \cdot 10^8$ particles (approx. 8% of the available memory) we are able to hit the performance of roughly 550 GFLOPS which we also obtained for $N = 4.8 \cdot 10^9$.

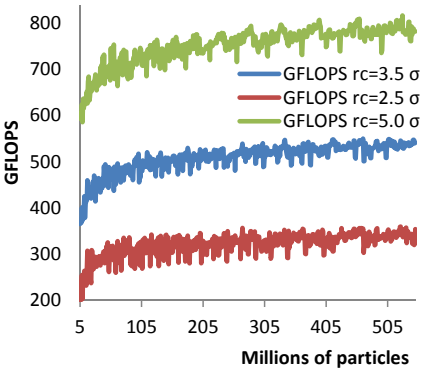


Fig. 6. GFLOPS dependeding on particle count and cut-off on 128 cores

It should be pointed out that the performance only decreases slightly for systems containing fewer particles (reducing the particle system size by a factor of 100): for $N = 10^7$ (which corresponds to the strong scaling setting in case of 146016 cores w.r.t. particles per node) we see a drop by 27% which only increases to the mentioned 58% when moving from 128 to 146016 cores. We have

to note that the overall simulation time in this case was 1.5 s for 10 time steps, thereof 0.43 s were communication time. Since 0.43 s are roughly 29% of 1.5 s, it becomes clear that the biggest fractions of the 58% decrease are stemming from low particle counts per process and relatively high communication costs.

Moreover, we performed a weak scaling analysis which is, to our knowledge, the largest MD simulation to date. Due to MPI buffers on all nodes, we were not able to keep the high number of particles per node ($6.0 \cdot 10^8$) and were forced to reduce it to $4.52 \cdot 10^8$. Particularly, buffers for eager communication turned out to be the most limiting factor. Although we reduced them to a bare minimum (64 MB buffer space for each process), roughly 1 GB per node had to be reserved as we use one MPI rank per core. By keeping Fig. 6 in mind, we know that this slight reduction has no negative impact on the overall performance of our simulation. In case of 146016 cores we were able to run a simulation of $4.125 \cdot 10^{12}$ particles with one time step taking roughly 40 s. For this scenario, a speed-up of 133183 × (compared to a single core) with an absolute performance of 591.2 TFLOPS was achieved, which corresponds to 9.4% peak performance efficiency.

These performance numbers can be easily improved by increasing simulation parameters like the cut-off radius r_c which results in a higher vector-register utilization. However, preceding publications [18, 10, 4] used cut-off radii within the interval $2.5\sigma < r_c < 5.0\sigma$. Therefore we restricted ourselves to $r_c = 3.5\sigma$ in order to ensure fairness, please consult Fig. 6 for a performance comparison of `ls1 mardyn` for different cutoff radii in this interval.

5 Conclusions

In this paper we showed that MD simulations can be scaled up to more than 140000 cores and a multi-trillion ($4.125 \cdot 10^{12}$) number of particles on modern supercomputers. Due to the sliding window technique, only 32 Bytes are required per particle, and with the help of a shared memory parallelization and a carefully optimized force calculation kernel we achieved 591.2 TFLOPS, which is 9.4% of the system's theoretical peak performance.

We achieved not only perfect weak scaling, but also excellent strong scaling results together with a good performance of the kernel also for comparably small particle numbers per core. These properties are essential for the investigation of large inhomogeneous molecular systems. Such scenarios are characterized by highly heterogeneous particle distributions, which requires a powerful load balancing method implementation. Therefore, we are working on the incorporation of the load balancing from the original `ls1 mardyn` code.

As indicated during the force kernel's discussion, the current kernel implementation suffers from not fully exploited vector-registers. Increasing the net-usage of vector-registers is subject of ongoing research. The most promising instruction set is currently provided by the Intel Xeon Phi coprocessor which features a full blown gather/scatter implementation.

Beside tuning `ls1 mardyn` for better performance on emerging architectures, energy efficiency with focus on the energy to solution ratio is an additional

research direction, especially when targeting MD scenarios with millions of time steps. Since SuperMUC is capable of dynamic frequency scaling, it provides an optimal testbed for such activities.

References

1. Buchholz, M., Bungartz, H.-J., Vrabec, J.: Software design for a highly parallel molecular dynamics simulation framework in chemical engineering. Journal of Computational Science 2(2), 124–129 (2011)
2. Eckhardt, W., Heinecke, A.: An efficient vectorization of linked-cell particle simulations. In: ACM International Conference on Computing Frontiers, Cagliari, pp. 241–243 (May 2012)
3. Eckhardt, W., Neckel, T.: Memory-efficient implementation of a rigid-body molecular dynamics simulation. In: Proceedings of the 11th International Symposium on Parallel and Distributed Computing - ISPDC 2012, Munich, pp. 103–110. IEEE (2012)
4. Germann, T.C., Kadau, K.: Trillion-atom molecular dynamics becomes a reality. International Journal of Modern Physics C 19(09), 1315–1319 (2008)
5. Gou, C., Kuzmanov, G., Gaydadjiev, G.N.: SAMS multi-layout memory: providing multiple views of data to boost SIMD performance. In: Proceedings of the 24th ACM International Conference on Supercomputing, ICS 2010, pp. 179–188. ACM, New York (2010)
6. Heinecke, A., Pflüger, D.: Emerging architectures enable to boost massively parallel data mining using adaptive sparse grids. International Journal of Parallel Programming 41(3), 357–399 (2013)
7. Heinecke, A., Trinitis, C.: Cache-oblivious matrix algorithms in the age of multi- and many-cores. Concurrency and Computation: Practice and Experience (2013); accepted for publication
8. Horsch, M., Vrabec, J., Bernreuther, M., Grottel, S., Reina, G., Wix, A., Schaber, K., Hasse, H.: Homogeneous nucleation in supersaturated vapors of methane, ethane, and carbon dioxide predicted by brute force molecular dynamics. The Journal of Chemical Physics 128(16), 164510 (2008)
9. Kabadshow, I., Dachsel, H., Hammond, J.: Poster: Passing the three trillion particle limit with an error-controlled fast multipole method. In: Proceedings of the 2011 Companion on High Performance Computing Networking, Storage and Analysis Companion, SC 2011 Companion, pp. 73–74. ACM, New York (2011)
10. Kadau, K., Germann, T.C., Lomdahl, P.S.: Molecular dynamics comes of age: 320 billion atom simulation on bluegene/l. International Journal of Modern Physics C 17(12), 1755–1761 (2006)
11. Lindahl, E., Hess, B., van der Spoel, D.: Gromacs 3.0: a package for molecular simulation and trajectory analysis. Journal of Molecular Modeling 7, 306–317 (2001)
12. Olivier, S., Prins, J., Derby, J., Vu, K.: Porting the gromacs molecular dynamics code to the cell processor. In: IEEE International Parallel and Distributed Processing Symposium, IPDPS 2007, pp. 1–8 (March 2007)
13. Peng, L., Kunaseth, M., Dursun, H., Nomura, K.-i., Wang, W., Kalia, R., Nakano, A., Vashishta, P.: Exploiting hierarchical parallelisms for molecular dynamics simulation on multicore clusters. The Journal of Supercomputing 57, 20–33 (2011)
14. Piazza, T., Jiang, H., Hammarlund, P., Singhal, R.: Technology Insight: Intel(R) Next Generation Microarchitecture Code Name Haswell (September 2012)

15. Rahimian, A., Lashuk, I., Veerapaneni, S., Chandramowlishwaran, A., Malhotra, D., Moon, L., Sampath, R., Shringarpure, A., Vetter, J., Vuduc, R., Zorin, D., Biros, G.: Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2010, pp. 1–11. IEEE Computer Society, Washington, DC (2010)
16. Rapaport, D.C.: The Art of Molecular Dynamics Simulation. Cambridge University Press (2004)
17. Rapaport, D.C.: Multibillion-atom molecular dynamics simulation: Design considerations for vector-parallel processing. Computer Physics Communications 174(7), 521–529 (2006)
18. Roth, J., Gähler, F., Trebin, H.-R.: A molecular dynamics run with 5 180 116 000 particles. International Journal of Modern Physics C 11(02), 317–322 (2000)
19. Vrabec, J., Kedia, G.K., Fuchs, G., Hasse, H.: Comprehensive study of the vapour-liquid coexistence of the truncated and shifted lennard-jones fluid including planar and spherical interface properties. Molecular Physics 104(9), 1509–1527 (2006)

Up to 700k GPU Cores, Kepler, and the Exascale Future for Simulations of Star Clusters Around Black Holes

Peter Berczik[1,2,3], Rainer Spurzem[1,3,4], Shiyan Zhong[1], Long Wang[4,1], Keigo Nitadori[5], Tsuyoshi Hamada[6], and Alexander Veles[2]

[1] National Astronomical Observatories of China, CAS, Beijing, P.R. China
berczik@nao.cas.cn
[2] Main Astronomical Observatory, NASU, Kyiv, Ukraine
[3] Astronomisches Rechen-Institut, ZAH, Univ. of Heidelberg, Germany
[4] Kavli Institute for Astronomy and Astrophysics, Peking University, Beijing, P.R. China
[5] RIKEN Institute, Tokyo, Japan
[6] Nagasaki Advanced Computing Center, Nagasaki, Japan

Abstract. We present benchmarks on high precision direct astrophysical N-body simulations using up to several 100k GPU cores; their soft and strong scaling behaves very well at that scale and allows further increase of the core number in the future path to Exascale computing. Our simulations use large GPU clusters both in China (Chinese Academy of Sciences) as well as in Germany (Judge/Milkyway cluster at FZ Jülich). Also we present first results on the performance gain by the new Kepler K20 GPU technology, which we have tested in two small experimental systems, and which also runs in the titan supercomputer in the United States, currently the fastest computer in the world. Our high resolution astrophysical N-body simulations are used for simulations of star clusters and galactic nuclei with central black holes. Some key issues in theoretical physics and astrophysics are addressed with them, such as galaxy formation and evolution, massive black hole formation, gravitational wave emission. The models have to cover thousands or more orbital time scales for the order of several million bodies. The total numerical effort is comparable if not higher than for the more widely known cosmological N-body simulations. Due to a complex structure in time (hierarchical blocked time steps) our codes are not considered "brute force".

1 Introduction

Theoretical numerical modeling has become a third pillar of sciences in addition to theory and experiment (in case of astrophysics the experiment is mostly substituted by observations). Numerical modeling allows one to compare theory with experimental or observational data in unprecedented detail, and it also provides theoretical insight into physical processes at work in complex systems. Similarly, data processing of e.g. astrophysical observations comprises the use of complex

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 13–25, 2013.

software pipelines to bring raw data into a form digestible for observational astronomers and ready for exchange and publication. Required algorithms are, for example, mathematical transformations like Fourier analyses of time series or spatial structures, complex template analyses or huge matrix-vector operations. Here fast access to and transmission of data, too, require supercomputing capacities. However, sufficient resolution of multi-scale physical processes still poses a formidable challenge, such as in the examples of few-body correlations in large astrophysical many-body systems, or in the case of turbulence in physical and astrophysical flows.

We are undergoing a new revolution on parallel processor technologies, and a change in parallel programming paradigms, which may help to advance current software towards the Exaflop/s scale and help better resolving and understanding typical multi-scale problems. The current revolution in parallel programming has been mostly catalyzed by the use of graphical processing units (GPU) for general purpose programming, but it is not clear whether this will remain the case in the future. GPU's have become widely used nowadays to accelerate a broad range of applications, including computational physics and astrophysics, image/video processing, engineering simulations, quantum chemistry, just to name a few [10,25,24,5,15]. GPU's are rapidly emerging as a powerful and cost-effective platform for high performance parallel computing. The GPU Technology Conferences held annually in San Jose (and offsprings in other parts of the world)[1,2] regularly provides a snapshot of the breadth and depth of present day GPU (super)computing applications. Recent GPU's, such as the NVIDIA Kepler K20 Computing Processor, offer 2496 CUDA processor cores and extremely fast on-chip-memory chip, as compared to only 4-8 cores on a standard Intel or AMD CPU. Groups of cores have access to very fast shared memory pieces. A single Kepler Tesla K20 device supports double precision operations fully with a peak speed of about 1 Tflop/s (double precision) and a little less than 4 Tflop/s (single precision). In this paper we use a code which still uses the single precision operations, which was originally developed for previous GPU architectures, which had no or very inefficient support for double precision. We circumvented this by emulation of a few critical parts of the code with the double precision operations "emulation" using a combination of few single precision operations (see: [22]). More details can be found in the Ph.D. thesis of one of us (Keigo Nitadori), "New approaches to high-performance N-body simulations with high-order integrator, new parallel algorithm, and efficient use of SIMD hardware", Univ. of Tokyo, 2009.

Dynamical modeling of dense star clusters with and without massive black holes poses extraordinary physical and numerical challenges. One of them is that gravity cannot be shielded such as electromagnetic forces in plasmas, therefore long-range interactions go across the entire system and couple non-linearly with small scales. High-order integration schemes and direct force computations for large numbers of particles have to be used to properly resolve all physical processes in the system. On small scales inevitably correlations form already early

[1] `http://www.gputechconf.com`
[2] `http://www.nvidia.com/gtc`

during the process of star formation in a molecular cloud. Such systems are dynamically extremely rich, they exhibit a strong sensitivity to initial conditions and regions of phase space with deterministic chaos.

Direct N-Body Codes in astrophysical applications for galactic nuclei, galactic dynamics and star cluster dynamics usually have a kernel in which direct particle-particle forces are evaluated. Astrophysical structures can develop also the high density contrasts. High-density regions created by gravitational collapse co-exist with low-density fields, as is known from structure formation in the universe or the turbulent structure of the interstellar medium. A high-order time integrator in connection with individual, hierarchically blocked time steps for particles in a direct N-body simulation provides the best compromise between accuracy, efficiency and scalability [21,1,2,23,14]. With GPU hardware up to a few million bodies could be reached for our models [6,7,12]. Note that while [11] already mention that their algorithm can be used to compute gravitational forces between particles to high accuracy, [21] find that the self-adaptive hierarchical time-step structure inherited from Aarseth's codes improves the performance for spatially structured systems by $\mathcal{O}(\mathcal{N})$ - it means that at least for astrophysical applications with high density contrast FMM is not a priori more efficient than direct N-body (which sometimes is called "brute force", but that should only be used if a shared time step is used, which is not the case in our codes). One could explain this result by comparing the efficient spatial decomposition of forces (in FMM, using a simple shared time step) with the equally efficient temporal decomposition (in direct N-body, using a simple spatial force calculation). Some exemplary research papers on dynamics of black holes in dense stellar systems, which illustrate the application domain of our code, can be found in [16,17].

2 Hardware

We present here results obtained from our GPU clusters in China and Germany: the `Laohu` at NAOC/CAS in Beijing with 85 Dual Intel Xeon nodes and 170 NVIDIA Tesla C1060 GPU's), the `Mole-8.5` at IPE/CAS in Beijing with 362 Dual Intel Xeon nodes integrating totally 2088 GPU's (NVIDIA Tesla C2050), and the Milkyway Judge cluster at Jülich supercomputing center (JSC) in Germany, with 200 Dual Intel Xeon nodes and 400 NVIDIA Tesla M2070 GPU's. In addition to that a first benchmark on a recently installed cluster with GPU's based on the Kepler architecture is added.

3 Software

The test code which we use for benchmarking on our clusters is a direct N-body simulation code for astrophysics, using a high order Hermite integration scheme and individual block time steps (the code supports time integration of particle orbits with 4^{th}, 6^{th} and 8^{th} order schemes). The code is called φGPU, it has been developed from our earlier published version φGRAPE [14] (which originally use the `GRAPE6a` cards as a hardware accelerator for the calculus of the particles mutual gravitational interaction).

The code is fully parallelized using the MPI library, and on each node using many cores of the special hardware. The code was mainly developed and tested by two of us (Keigo Nitadori and Peter Berczik, see also [13]) and is based on an earlier `C` code version[3] for `GRAPE6a` clusters [14]. The new code is written from scratch in `C++` and based on [22] earlier CPU serial N-body code (yebisu). More details and also the φGPU code public version will be published in an upcoming publication [8].

The MPI parallelization was done in the same "j" particle parallelization mode as in the earlier φGRAPE code [14]. All the particles are divided equally between the working nodes (using the `MPI_Bcast()` commands) and in each node we calculate only the fractional forces for the, so call, "active" – "i" particles at the current time step. Due to the hierarchical block time step scheme the average number $\langle N_{\rm act} \rangle$ of active particles (due for a new force computation at a given time level) is usually small compared to the total particle number N, but its actual value can vary from $1 \dots N$. The full forces from all the particles acting on the active particles we get after using the global `MPI_Allreduce()` communication routines.

The present version of φGPU code we used and tested only with the recent GNU compilers (ver. 4.x). We use native GPU support and direct code access to the GPU using only the NVIDIA native CUDA library[4]. We use CUDA 3.2 and 4.0, the code is rather robust against the different CUDA versions. One first test was done with a single Kepler K20 and CUDA 5.0 (see Fig. 4). Multi GPU support is achieved through MPI parallelization. Each MPI process uses only a single GPU, but we can start two MPI processes per node (to use effectively for example the dual quad core CPU's and the multi GPU's in the NAOC, CAS and IPE, CAS GPU cluster). In this case each MPI process uses its own GPU inside the node. Communication always (even for the processes inside one node) works via MPI. We do not use any of the possible OMP (multi-thread) features of recent gcc 4.x compilers inside one node.

The φGPU code uses a blocked hierarchical individual time step scheme (HITS) and a Hermite high order time integration scheme of at least $4^{\rm th}$ order for integration of the equation of motions for all particles [20]. At every time we integrate the motion only for $\langle N_{\rm act} \rangle$ particles, a number which is usually much less compared to the total number of particles N. The average value $\langle N_{\rm act} \rangle$ depends on the details of the algorithm and on the particle configuration integrated. According to a simple theoretical estimate it is $\langle N_{\rm act} \rangle \propto N^{2/3}$ [18], but the real value of the exponent deviates from 2/3, depending on the initial model and details of the time step choice [21] (see our actual measurements given by equation (8).

For the time step itself first the so-called individual "Aarseth" time step is used [1]:

[3] `ftp://ftp.ari.uni-heidelberg.de/staff/berczik/phi-GRAPE/`
[4] `http://www.nvidia.com/object/cuda_home_new.html`

$$\Delta t = \sqrt{\eta \, \frac{|\boldsymbol{a}||\boldsymbol{a}^{(2)}| + |\boldsymbol{a}^{(1)}|^2}{|\boldsymbol{a}^{(1)}||\boldsymbol{a}^{(3)}| + |\boldsymbol{a}^{(2)}|^2}}, \tag{1}$$

where $\boldsymbol{a}^{(k)}$ is the $k^{\rm th}$ derivative of acceleration and η is a parameter which controls the accuracy. Usually, a value around 0.02 is used for η. This time step choice is known to work well with fourth-order schemes, but it is less efficient for higher-order schemes [19]. In our φGPU code we use the generalized "Aarseth" type criterion already proposed in the paper [22]:

$$\Delta t = \eta_p \, \left(\frac{A^{(1)}}{A^{(p-2)}} \right)^{1/(p-3)} \tag{2}$$

where

$$A^{(k)} = \sqrt{|\boldsymbol{a}^{(k-1)}||\boldsymbol{a}^{(k+1)}| + |\boldsymbol{a}^{(k)}|^2}. \tag{3}$$

Here, p is the order of the integrator. We moved the accuracy parameter η_p out of the fractional power, so that the time step is directly proportional to η_p. The enumerator is the same as that for the Aarseth criterion for the fourth-order scheme, and for the denominators we used the terms of highest orders available. The fractional power is chosen to give the correct dimension of time.

For efficient parallelization and vectorization we introduce the block time step scheme [20]. Particles are grouped together in time by replacing their original individual time steps Δt_i with a common block time step $\Delta t_{i,\rm b} = (1/2)^n$, where n is chosen according to

$$\left(\frac{1}{2}\right)^n \leq \Delta t_i < \left(\frac{1}{2}\right)^{n-1}. \tag{4}$$

The commensurability is enforced by requiring that $t_i/\Delta t_i$ be an integer.

For the performance analysis we run the code for a fixed physical time span, one time unit (TU) in our dimensionless N-body units, which is approximately one orbital time around the half-mass radius of the system. We count time steps and floating point operations and also analyse the cumulative number $\sum N_{\rm act}$ of active particles, as well as the average number of active particles during at intermediate time points $\langle N_{\rm act} \rangle$. $n_{\rm ts}$ is the number of advancements of time in the simulation:

$$\langle N_{\rm act} \rangle \equiv \frac{\sum N_{\rm act}}{n_{\rm ts}} \tag{5}$$

Empirically, we get the following best fits for the number of steps:

$$\begin{aligned} n_{\rm ts}(4^{\rm th}) &\propto N^{0.575}, \\ n_{\rm ts}(6^{\rm th}) &\propto N^{0.585}, \\ n_{\rm ts}(8^{\rm th}) &\propto N^{0.581}. \end{aligned} \tag{6}$$

$$\begin{array}{l} \sum N_{\rm act}(4^{\rm th}) \propto N^{1.185}, \\ \sum N_{\rm act}(6^{\rm th}) \propto N^{1.259}, \\ \sum N_{\rm act}(8^{\rm th}) \propto N^{1.273}. \end{array} \tag{7}$$

Hence:

$$\begin{array}{l} \langle N_{\rm act} \rangle (4^{\rm th}) \propto N^{0.613}, \\ \langle N_{\rm act} \rangle (6^{\rm th}) \propto N^{0.681}, \\ \langle N_{\rm act} \rangle (8^{\rm th}) \propto N^{0.687}. \end{array} \tag{8}$$

These numbers depend on several parameters, such as the chosen time step parameter, and the density and velocity distribution of the particles (here we use a rather simple example of a Plummer model, which has a homogeneous core and a halo with decreasing density outside). For the more detail study of this dependences we refer the reader to the paper [21] and also to our more detailed analyis in preparation[8].

In a simple theoretical model our code should asymptotically scale with N^2, so we would expect $N \cdot \langle N_{\rm act} \rangle \cdot n_{\rm ts} \propto N^2$. However, the measurements (see equation (7)) deliver a slightly smaller number $\langle N_{\rm act} \rangle \cdot n_{\rm ts} \propto N^{1+x}$, with $x_{4^{\rm th}} = 0.18$, $x_{6^{\rm th}} = 0.26$ and $x_{8^{\rm th}} = 0.27$, which is in good agreement with the earlier results of [21].

4 Results of Benchmarks

In this section we describe some results of our extensive performance testing of the 6^{th} order HITS scheme on several different GPU clusters. The wall clock time $T_{\rm TOT}$ needed for our particle based algorithm to advance the simulation by the fixed time integration interval (1 TU, see definition above) is decomposed into several components:

$$T_{\rm TOT} = T_{\rm host} + T_{\rm GPU} + T_{\rm comm} + T_{\rm MPI} \tag{9}$$

We have from left to right: the computing time spent on the host – $T_{\rm host}$, on the GPU – $T_{\rm GPU}$, the communication time to send data between host and GPU – $T_{\rm comm}$, and the communication time for MPI data exchange between the nodes – $T_{\rm MPI}$. In our present implementation all components are blocking, so there is no hiding of communication. This could be improved in further code versions, but for now it eases profiling.

We use a detailed timing model for the determination of the wall clock time needed for different components of our code on CPU and GPU, which is then fitted to the measured timing data. Its full definition is given in Table. 1.

In practice we see that only two terms play the dominant role (∼90% of the time) in the understanding of the strong and weak scaling behavior of our code. These are the:

– Force computation time (on GPU) – $T_{\rm GPU}$:
$n_{\rm ts} \cdot \mathcal{O}(N \cdot \langle N_{\rm act} \rangle / N_{\rm GPU})$.

Table 1. Breaking down the computational tasks in a parallel direct N-body code with individual hierarchical block time steps. At every block time step level we denote $\langle N_{\rm act}\rangle \leq N$ particles, which should be advanced by the high order corrector as active or "i" particles, while the field particles, which exert forces on the "i" particles to be computed are denoted as "j" particles. Note that the number of "j" particles in our present code is always $N/N_{\rm GPU}$ (in each node, where $Ngpu$ is the number of GPU's used for the simulation). We also have timing components for low-order prediction of all "j" particles and distinguish communication of data from host to GPU and return, and through the MPI message passing network.

Components of our timing model for φGPU code. One time step integration		
task	expected scaling	timing variable
active particle determination	$\mathcal{O}(N_{\rm act}\log(N_{\rm act}))$	$T_{\rm host}$
all particle prediction	$\mathcal{O}(N/N_{\rm GPU})$	$T_{\rm host}$
active ("i") particle prediction	$\mathcal{O}(N_{\rm act})$	$T_{\rm host}$
"j" part. send. to GPU	$\mathcal{O}(N/N_{\rm GPU})$	$T_{\rm comm}$
"i" part. send. to GPU	$\mathcal{O}(N_{\rm act})$	$T_{\rm comm}$
force computation on GPU	$\mathcal{O}(N\cdot N_{\rm act}/N_{\rm GPU})$	$T_{\rm GPU}$
receive the force from GPU	$\mathcal{O}(N_{\rm act})$	$T_{\rm comm}$
MPI global communication	$\mathcal{O}((\tau_{\rm lat}+N_{\rm act})\log(N_{\rm GPU}))$	$T_{\rm MPI}$
correction/advancing "i" particle	$\mathcal{O}(N_{\rm act})$	$T_{\rm host}$

– Message passing communication time – $T_{\rm MPI}$:
$n_{\rm ts}\cdot\mathcal{O}((\tau_{\rm lat}+\langle N_{\rm act}\rangle)\log(N_{\rm GPU}))$.

Within the $T_{\rm MPI}$ we can distinguish a bandwidth dependent part (scaling as $\langle N_{\rm act}\rangle\cdot\log(N_{\rm GPU})$) and a latency dependent part (scaling as $\tau_{\rm lat}\cdot\log(N_{\rm GPU})$).

Hence, for this short paper we only discuss the simplified form of the equation (9):

$$T_{\rm TOT}\approx T_{\rm GPU}+T_{\rm MPI} \tag{10}$$

or:

$$T_{\rm TOT}\approx\frac{\alpha N\sum N_{\rm act}}{N_{\rm GPU}}+\beta(n_{\rm ts}\tau_{\rm lat}+\sum N_{\rm act})\log(N_{\rm GPU}) \tag{11}$$

The latency is only relevant for a downturn of efficiency for strong scaling at relatively large numbers of $N_{\rm GPU}$. Starting in the strong scaling curves from the dominant term at small $N_{\rm GPU}$ there is a linearly rising part in Fig. 1., just the force computation on GPU, while the turnover to a flat curve is dominated by the time of MPI communication between the computing nodes – $T_{\rm MPI}$.

To find a model for our performance measurements we use the ansatz:

$$P=\frac{\text{(total flop operations)}}{T_{\rm TOT}}\approx\frac{\gamma\cdot N\cdot\sum N_{\rm act}}{T_{\rm TOT}} \tag{12}$$

where $T_{\rm TOT}$ is the computational wall clock time needed to integrate the system for 1 TU. Here γ defines how many floating point operations our particular Hermite scheme requires per particle per interaction per step. Based on the detail flop count (see Table. 1. in [22]) we have: $\gamma_{4^{\rm th}} = 60$, $\gamma_{6^{\rm th}} = 97$ and $\gamma_{8^{\rm th}} = 144$.

Replacing here the $T_{\rm TOT}$ with the approximation from equation (11) we have:

$$P \approx \frac{\gamma\ N\ \sum N_{\rm act}}{\frac{\alpha N \sum N_{\rm act}}{N_{\rm GPU}} + \beta(n_{\rm ts}\tau_{\rm lat} + \sum N_{\rm act})\log(N_{\rm GPU})} \tag{13}$$

The reader with interest in more detail how this formula can be theoretically derived for general purpose parallel computers is referred to [9]. The α, β and $\tau_{\rm lat}$ are hardware time constants for the floating point calculation on GPU, for the bandwidth of the interconnect hardware used for message passing and its latency, respectively.

Putting to the equation (13) our earlier power approximations for the $n_{\rm ts}$ and $\sum N_{\rm act}$ as a function of total particle numbers N (see equations (6) and (7)) we get P as a function only of N and $N_{\rm GPU}$:

$$P \approx \frac{\gamma\ N^{2+x}}{\frac{\alpha N^{2+x}}{N_{\rm GPU}} + \beta(N^{0.33+x}\tau_{\rm lat} + N^{1+x})\log(N_{\rm GPU})} \tag{14}$$

The parameter $x = 0.26$ is a particular result for our case of the $6^{\rm th}$ order HITS and the particular initial model used for the N-body system, virial Plummer's model as in [21].

Comparison of the functional form of Eq. 14 yields to a very good match for $x = 0.26$, as can be seen in Fig. 1. The dotted gray lines for 0.5M, 1M, 2M, 4M and 6M particles show our approximation formula results. The real maximum performance data which we get for the largest 1536 GPU simulations (using in total about 700k GPU cores) for these above particle numbers are (approximately): 29 Tflop/s, 58 Tflop/s, 147 Tflop/s, 239 Tflop/s and 315 Tflop/s.

The parameters α, β and $\tau_{\rm lat}$ can be determined for each particular hardware used. The timing formula (14) can then be used to approximate our φGPU 6^{th} order code calculation "speed" for any other number of particles, GPU's, or different hardware parameters (possible faster MPI LAN on the other GPU clusters).

For example, on the `Mole-8.5` system, we see, that for $N = 6$M particles if we are using $N_{\rm GPU} = 2000$ GPU cards we expect to get ≈330 Tflop/s (see: Fig. 1.). If we use our scaling formula for the much higher node-to-node bandwidth (4 times faster MPI LAN) of the `Tianhe-1A`[5] system at National Supercomputing Center, Tianjin, China (this is the number one supercomputer according to the Top500 list of November 2010, with ∼7000 NVIDIA Fermi Tesla C2050 GPU's and 160 Gbit/s node-to-node bandwidth) we can possibly reach sustained performance of ∼1.1 Petaflop/s (see the faster LAN approximation lines on: Fig. 1.). This is subject of our near future research application.

To our knowledge the direct N-body simulation with four million and six million bodies in the framework of a so-called Aarseth style high precision N-body

[5] `http://en.wikipedia.org/wiki/Tianhe-I#Tianhe-1A`

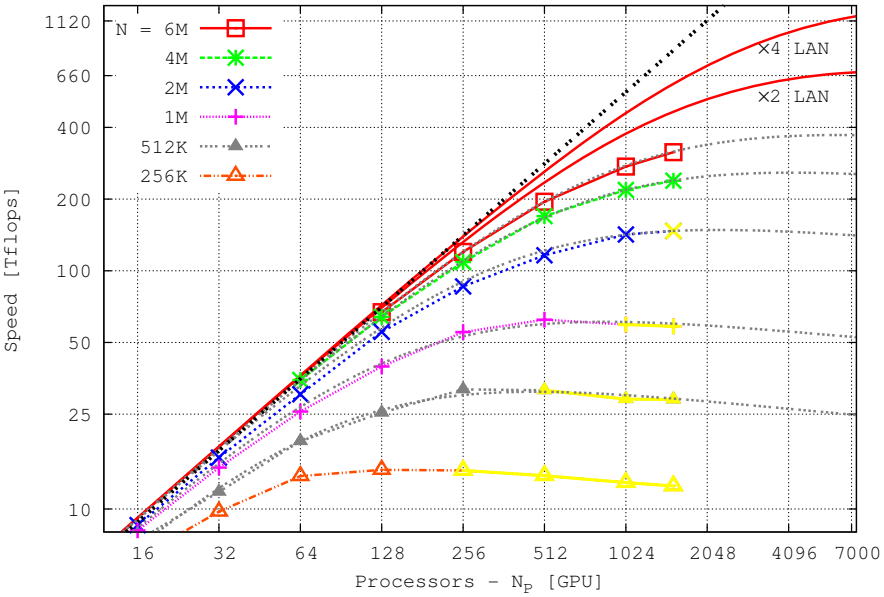


Fig. 1. Strong scaling for different problem sizes on `Mole-8.5` cluster, each line corresponds to a different problem size (particle number), which is given in the key. Here as in all the following plots the symbols (like quadrat, star, triangle, cross) represent our actual runs and measurements; the curves having the same colour represent the fit by our timing model; their grey extensions represent extrapolations beyond our actual tests. If one would use all ∼2000 GPU's on the system a sustained speed of ≈0.33 Petaflop/s is feasible (for 6M particles). We also show the approximation lines for the 6M particle runs on the cluster with the possible 2 times and 4 times faster MPI LAN (for example the `Tianhe-1A` GPU cluster with up to ∼7000 GPU's). On such a system we expect to get the maximum ∼1.1 Petaflop/s performance for 6M particles. Our largest run uses 1600 GPU's equivalent to about 700k GPU cores. The yellow color of the data points indicated the runs where we have the number of particles per GPU 1k or even less. This small number of particles clearly not give us a good usage of the current many cores Fermi GPU's, so we try to have in the real astrophysical simulations always this number larger as 1k.

code (high order Hermite scheme, hierarchical block time step, integrating an astrophysically relevant virial Plummer model with core-halo structure in density for a certain physical time) is the largest of such simulation which exists so far.

However, the presently used parallel MPI-CUDA GPU code φGPU is on the algorithmic level of NBODY1 [1] - though it is already strongly used in production, useful features such as regularization of few-body encounters and an Ahmad-Cohen neighbor scheme [4] are not yet implemented. Only with those the code would be equivalent to NBODY6, which is the most efficient code for single workstations [1,3], eventually with acceleration on a single node by one or two GPU's (work by Aarseth & Nitadori, see NBODY6[6]).

[6] `http://www.ast.cam.ac.uk/~sverre/web/pages/nbody.htm`

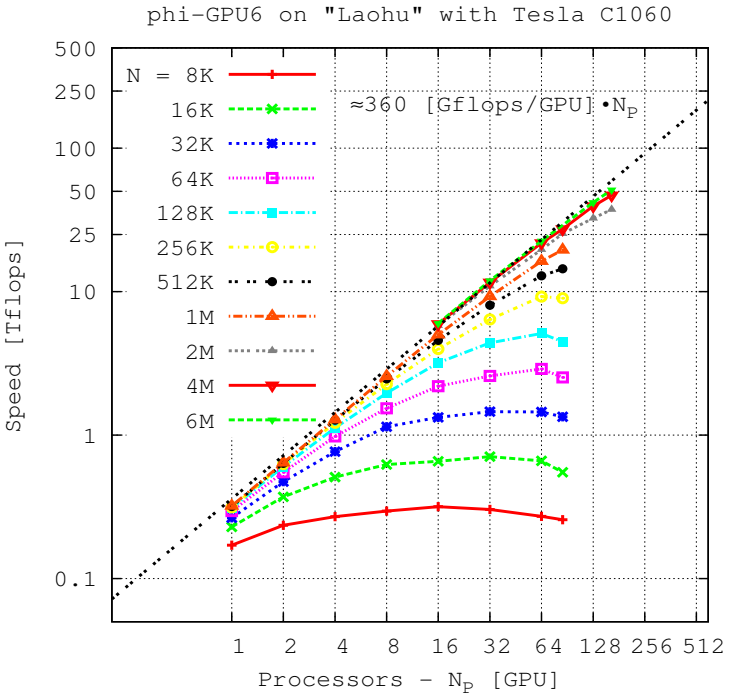


Fig. 2. Left: Same benchmark simulations as in Fig. 1, but for the laohu cluster in Beijing, using the older C1060 GPUs. We reach about 330 Gflop/s per GPU, a total of 51.2 Tflop/s on 164 GPUs.

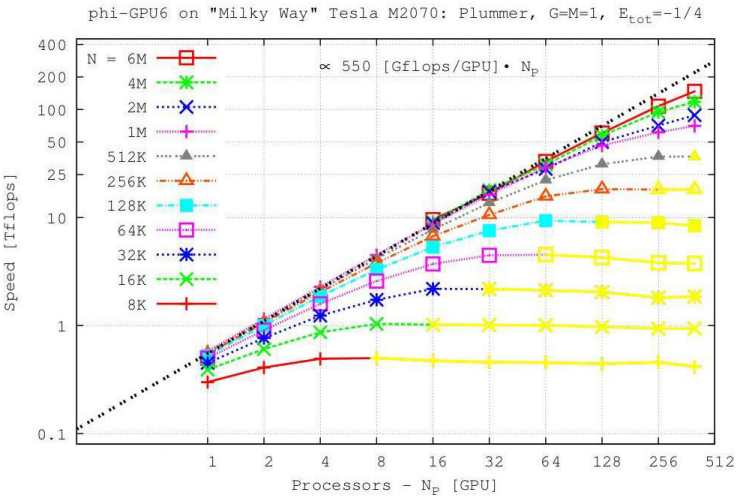


Fig. 3. Same as previous figure, but now using the most recent Milky Way GPU cluster at JSC; as in the case of Mole-8.5 we reach about 550 Gflop/s per GPU card, and a total sustained performance of 150 Tflop/s for six million bodies on 400 GPU's, which is equal to about 160k GPU cores.

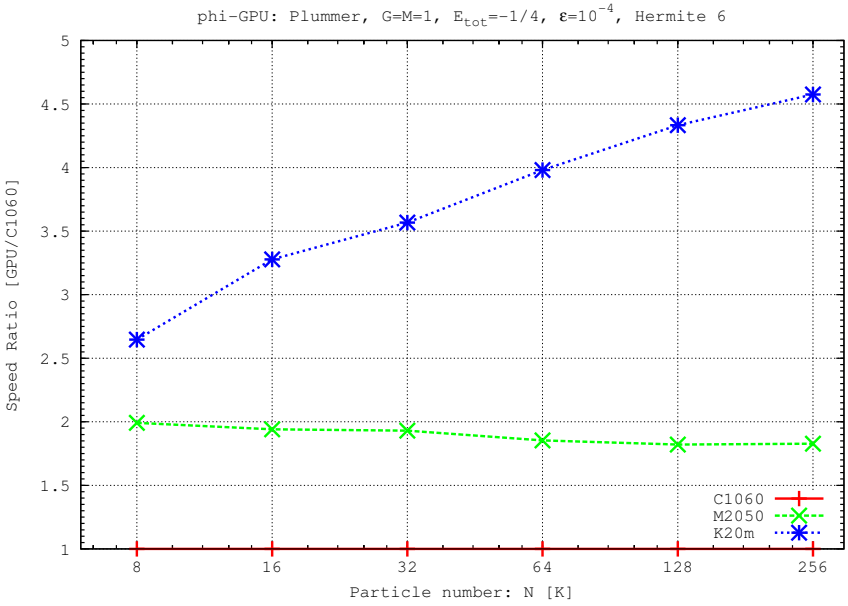


Fig. 4. Here we report a preliminary result from a benchmark test of our code on one Kepler K20 card; we compare with the performance on Fermi C2050 (used in the Mole-8.5 cluster), and the oldest Tesla C1060 GPU (used in the laohu cluster of 2009) - the latter is used as a normalization reference. We plot the speed ratio of our usual benchmarking simulation used in the previous figures, as a function of particle number. From this we see the sustained performance of a Kepler K20 would be about 1.4 - 1.5 Tflop/s.

We have shown that our GPU clusters for the very favorable direct N-body application reach about one third of the theoretical peak speed sustained for a real application code with individual block time steps.

5 Conclusions

We have presented exemplary implementation of parallel code using many graphical processing units as accelerators, so combining message passing parallelization with many-core parallelization and discussed their benchmarks using up to 1536 Fermi Tesla C2050 GPU's with more than 700k GPU thread cores in parallel. On this hardware we reach a sustained speed of 550 Gflop/s per GPU card; similar results are obtained from the Milkyway cluster at JSC, which uses 400 M2070 GPUs. At the University of Heidelberg we have just started first benchmarking using the new Kepler K20 GPU architecture; the code described in this paper reached a sustained performance of about 1500 Gflop/s, which is a speed up of 3 compared to Fermi architecture, and of 5 compared to the older C1060

GPUs. However, we only show here a first benchmark (Fig. 4), because a small cluster with 20 Kepler K20 GPUs in Heidelberg is not yet fully operational; in the future more promising benchmarks could be achieved on large GPU clusters with Kepler architecture, such as the Titan system at ORNL. From our figure one can see that the maximum sustained performance per Kepler GPU card for our code (which is reached when the communication balances further speed-up) has not yet been reached at 1.5 Tflop/s. Also the question how much of the Kepler performance stems from just scaling up the number of cores and how much originates from using CUDA 5.0 has to be checked in future work.

For direct high-accuracy gravitating N-body simulations we discussed how self-gravity, because it cannot be shielded, generates inevitably strong multi-scale structures in space and time, spanning many orders of magnitude. This requires special codes, which nevertheless scale with a high efficiency on GPU clusters. So our codes are examples that it is possible to reach the sub-Petaflop/s scale in sustained speed for realistic application software with large GPU clusters, and all results for soft and strong scaling show that more than a million GPU cores can be efficiently used. Whether our programming models can be scaled up for future hardware and the Exaflop/s scale, however, remains yet to be studied.

Acknowledgments. Chinese Academy of Sciences has supported this work by a Visiting Professorship for Senior International Scientists, Grant Number 2009S1-5 (RS), and National Astronomical Observatory of China (NAOC) of CAS by the Silk Road Project (P.B. and R.S.).

The special supercomputer `Laohu` at the High Performance Computing Center at National Astronomical Observatories of China, funded by Ministry of Finance under the grant ZDYZ2008-2, has been used. We thank the computer system support team at NAOC (Li Changhua, Cui Chenzhou) for their support to run the `Laohu` cluster.

Simulations were also performed on the GRACE supercomputer (grants I/80 041-043 and I/81 396 of the Volkswagen Foundation and 823.219-439/30 and /36 of the Ministry of Science, Research and the Arts of Baden-Württemberg).

P.B. and I.B. acknowledges financial support by the Deutsche Forschungsgemeinschaft (DFG) through SFB 881 "The Milky Way System" at the Ruprecht-Karls-Universitat Heidelberg. Significant co-funding and collaboration of the team of Jülich Supercomputing Center in acquiring and operating the Milkyway GPU cluster is gratefully acknowledged.

P.B. and A.V. acknowledges the special support by the NASU under the Main Astronomical Observatory GRID/GPU computing cluster project. P.B.'s studies are also partially supported by the program Cosmomicrophysics of NASU.

References

1. Aarseth, S.J.: From NBODY1 to NBODY6: The Growth of an Industry. Publications of the Astronomical Society of the Pacific 111, 1333–1346 (1999)
2. Aarseth, S.J.: Star Cluster Simulations: the State of the Art. Celestial Mechanics and Dynamical Astronomy 73, 127–137 (1999)

3. Aarseth, S.J.: Gravitational N-Body Simulations (November 2003)
4. Ahmad, A., Cohen, L.: A numerical integration scheme for the N-body gravitational problem. Journal of Computational Physics 12, 389–402 (1973)
5. Akeley, K., Nguyen, H.: GPU Gems 3 (2007)
6. Berczik, P., Merritt, D., Spurzem, R.: Long-Term Evolution of Massive Black Hole Binaries. II. Binary Evolution in Low-Density Galaxies. The Astrophysical Journal 633, 680–687 (2005)
7. Berczik, P., Merritt, D., Spurzem, R., Bischof, H.: Efficient Merger of Binary Supermassive Black Holes in Nonaxisymmetric Galaxies. The Astrophysical Journal Letters 642, L21–L24 (2006)
8. Berczik, P., Nitadori, K., Hamada, T., Spurzem, R.: The Parallel GPU N-Body Code φGPUİn: New Astronomy (2013) in preparation
9. Dorband, E.N., Hemsendorf, M., Merritt, D.: Systolic and hyper-systolic algorithms for the gravitational N-body problem, with an application to Brownian motion. Journal of Computational Physics 185, 484–511 (2003)
10. Egri, G., Fodor, Z., Hoelbling, C., Katz, S., Nogradi, D., Szabo, K.: Lattice QCD as a video game. Computer Physics Communications 177, 631–639 (2007)
11. Greengard, L., Rokhlin, V.: A fast algorithm for particle simulations. Journal of Computational Physics 73, 325–348 (1987)
12. Gualandris, A., Merritt, D.: Ejection of Supermassive Black Holes from Galaxy Cores. The Astrophysical Journal 678, 780–797 (2008)
13. Hamada, T., Iitaka, T.: The Chamomile Scheme: An Optimized Algorithm for N-body simulations on Programmable Graphics Processing Units. ArXiv Astrophysics e-prints (March 2007)
14. Harfst, S., Gualandris, A., Merritt, D., Spurzem, R., Portegies Zwart, S., Berczik, P.: Performance analysis of direct N-body algorithms on special-purpose supercomputers. New Astronomy 12, 357–377 (2007)
15. Hwu, W.-M.-W.: GPU Computing Gems (2011)
16. Khan, F.M., Preto, M., Berczik, P., Berentzen, I., Just, A., Spurzem, R.: Mergers of Unequal-mass Galaxies: Supermassive Black Hole Binary Evolution and Structure of Merger Remnants. The Astrophysical Journal 749, 147 (2012)
17. Li, S., Liu, F.K., Berczik, P., Chen, X., Spurzem, R.: Interaction of Recoiling Supermassive Black Holes with Stars in Galactic Nuclei. The Astrophysical Journal 748, 65 (2012)
18. Makino, J.: A Modified Aarseth Code for GRAPE and Vector Processors. Proceedings of Astronomical Society of Japan 43, 859–876 (1991)
19. Makino, J.: Optimal order and time-step criterion for Aarseth-type N-body integrators. The Astrophysical Journal 369, 200–212 (1991)
20. Makino, J., Aarseth, S.J.: On a Hermite integrator with Ahmad-Cohen scheme for gravitational many-body problems. Publications of the Astronomical Society of Japan 44, 141–151 (1992)
21. Makino, J., Hut, P.: Performance analysis of direct N-body calculations. The Astrophysical Journal Supplement Series 68, 833–856 (1988)
22. Nitadori, K., Makino, J.: Sixth- and eighth-order Hermite integrator for N-body simulations. New Astronomy 13, 498–507 (2008)
23. Spurzem, R.: Direct N-body Simulations. Journal of Computational and Applied Mathematics 109, 407–432 (1999)
24. Yang, J., Wang, Y., Chen, Y.: Journal of Computational Physics 221, 799 (2007)
25. Yasuda, K.: Journal of Computational Chemistry 29, 334 (2007)

Parallelizing a High-Order CFD Software for 3D, Multi-block, Structural Grids on the TianHe-1A Supercomputer

Chuanfu Xu[1], Xiaogang Deng[1], Lilun Zhang[1], Yi Jiang[2], Wei Cao[1], Jianbin Fang[3], Yonggang Che[1], Yongxian Wang[1], and Wei Liu[1]

[1] School of Computer, National University of Defense Technology, Changsha 410073, China
xuchuanfu@nudt.edu.cn
[2] State Key Laboratory of Aerodynamics, China Aerodynamics Research and Development Center, Mianyang 621000, China
[3] Parallel and Distributed Systems Group, Delft University of Technology, Delft 2628CD, The Netherlands

Abstract. In this paper, with MPI+CUDA, we present a dual-level parallelization of a high-order CFD software for 3D, multi-block structural girds on the TianHe-1A supercomputer. A self-developed compact high-order finite difference scheme HDCS is used in the CFD software. Our GPU parallelization can efficiently exploit both fine-grained data-level parallelism within a grid block and coarse-grained task-level parallelism among multiple grid blocks. Further, we perform multiple systematic optimizations for the high-order CFD scheme at the CUDA-device level and the cluster level. We present the performance results using up to 256 GPUs (with 114K+ processing cores) on TianHe-1A. We can achieve a speedup of over 10 when comparing our GPU code on a Tesla M2050 with the serial code on an Xeon X5670, and our implementation scales well on TianHe-1A. With our method, we successfully simulate a flow over a high-lift airfoil configuration using 400 GPUs. To the authors' best knowledge, our work involves the largest-scale simulation on GPU-accelerated systems that solves a realistic CFD problem with complex configurations and high-order schemes.

Keywords: GPU parallelization, high-order CFD, multi-block structural grid, heterogeneous system.

1 Introduction

Although low-order (e.g., second-order) schemes for computational fluid dynamics (CFD) have been widely used in engineering applications, they are insufficient in capturing small disturbances in an environment containing sharp gradients. To ensure the high-resolution and fidelity of a numerical simulation, it is imperative that CFD researchers develop and apply robust and high-order CFD

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 26–39, 2013.

methods that can deal with complex flows in complex domains. Among others, compact high-order finite difference schemes based on a compact stencil are very attractive for flows with multiscales (e.g., aeroacoustics and turbulence), due to their high formal order, good spectral resolution and flexibility[1].

Despite the rapid development of high-order algorithms in CFD, the applications of high-order finite difference schemes on complex configurations are still few. One main factor which hinders the widely application of these methods is the complexity of grids. Particularly, the Geometric Conservation Law (GCL)[2] and block-interface conditions, which can be neglected for low-order schemes must be treated carefully for high-order ones when the configurations are complex. In order to apply high-order finite difference schemes on complex multi-block grids, we have developed a Conservative Metric Method (CMM)[3] to calculate the grid derivatives, and employed a Characteristic-Based Interface Condition (CBIC)[4] to fulfill high-order multi-block computing. Based on CMM and CBIC, we have developed a Hybrid cell-edge and cell-node Dissipative Compact Scheme (HDCS)[5] recently and implemented it in our high-order CFD software HOSTA (High-Order SimulaTor for Aerodynamics) for multi-block structural grids (see Sect.2).

HOSTA has been successfully applied to a wide range of flow simulations so far, showing its flexibility and robustness[6]. However, to parallelize HOSTA on supercomputer like TianHe-1A[7] (a GPU-accelerated massive parallel processing system) is challenging. Developers often need to manage different levels of parallelisms using different parallel programming models (e.g., NVIDIA's Compute Unified Device Architecture (CUDA)[8] for GPUs and MPI or OpenMP for CPUs) for heterogeneous compute devices. Further, when performing complex high-order, multi-block grid CFD simulations, we need extensive implementation and optimization efforts to achieve high performance and efficiency.

The work that we present here demonstrates a comprehensive effort to efficiently accelerate **large-scale simulations** of **realistic CFD problems** using both **complex multi-block grids** and **high-order CFD schemes** on the **TianHe-1A supercomputer** (see Sect.3). In the past 8 months, we have successfully parallelized HOSTA (using the HDCS scheme) with MPI+CUDA. When parallelizing HOSTA on a single GPU, we exploit dual-level parallelisms: fine-grained parallelism by using a CUDA thread to compute a cell within a grid block, and coarse-grained parallelism by using multiple CUDA streams to compute multiple blocks. At the CUDA-device level, we also use a kernel-decomposition optimization to further enhance the performance. For efficient simulations on large-scale GPUs, we use non-blocking MPI, CUDA multi-stream and CUDA events to maximize the overlapping of kernel computation, intra-node data transfer and inter-node communication. The GPU-enabled HOSTA shows promising strong and weak scalability for large-scale parallel tests on TianHe-1A. Finally, we simulate a flow over a high-lift airfoil configuration to evaluate the high-order behavior of HDCS. To our best knowledge, this is the largest-scale simulation on GPU-accelerated systems that solves a realistic CFD problem with both complex configurations and high-order schemes so far.

The remainder of the paper is organized as follows. Section 2 briefly describes the numerical methods and HOSTA implementation. In Section 3, we detail our MPI-CUDA implementation and optimizations, and present the performance results and the validation results. In Section 4, we introduce some related work. Finally we conclude this paper in Section 5.

2 Numerical Methods and HOSTA Implementation

```
Do nstep=nstepst,nsteped     !time-marching loop
  Do iter=1,nsubmax          !sub-iteration for unsteady flows
    Call boundary_conditions()      !boundary conditions
    Call Exchange_BC(PV)            !exchange primitive variables(PV) for boundaries
    Call Exchange_Singular(PV)      !exchange PV for singularities
    Call calc_spectral_radius()     !calculate delta of spectral radius
    Call calc_time_step()           !calculate delta of time step
    !begin of calculation of RHS
    Call calc_gradient()            !calculate gradient of PV (DPV)
    Call Exchange_BC(DPV)           !exchange DPVs for boundaries
    Call Exchange_Singular(DPV)     !exchange DPVs for singularities
    Call calc_inviscid()            !calculate the inviscid fluxes
    Call calc_viscous()             !calculate the viscous fluxes
    Call calc_source()              !calculate the source flux
    Call Exchange_BC(RHS)           !exchange RHSs for boundaries
    Call Exchange_Singular(RHS)     !exchange RHSs for singularities
    ! end of calculation of RHS
    Call sol_jacobi()               !Jacobi solver for delta of conservative variables(DQ)
    Call Exchange_BC(DQ)            !exchange DQs for boundaries
    Call Exchange_Singular(DQ)      !exchange DQs for singularities
    Call Update()                   !update PV···
    Call Residual()                 !calculate residual
  End do                     !end of sub-iteration
End do                       !end of time-marching
```

Fig. 1. The main pseudocode for the time-marching loop of HOSTA

In curvilinear coordinates the governing equations (Euler or Navier-Stokes) in strong conservative form are:

$$\frac{\partial \tilde{Q}}{\partial \tau} + \frac{\partial \tilde{F}}{\partial \xi} + \frac{\partial \tilde{G}}{\partial \eta} + \frac{\partial \tilde{H}}{\partial \zeta} = 0 \tag{1}$$

where $\tilde{F}$, $\tilde{G}$ and $\tilde{H}$ are the fluxes along the ξ, η and ζ direction respectively; $\tilde{Q}$ is the conservative variable.

Let us first consider the discretization of the inviscid flux derivative along the ξ direction. The discretization for the other inviscid fluxes can be computed in a similar way. The seventh order HDCS scheme called HDCS-E8T7 can be expressed as follows:

$$\begin{aligned}\frac{\partial \tilde{F}_i}{\partial \xi} = {} & \frac{256}{175h}(\tilde{F}_{i+1/2} - \tilde{F}_{i-1/2}) - \frac{1}{4h}(\tilde{F}_{i+1} - \tilde{F}_{i-1}) \\ & + \frac{1}{100h}(\tilde{F}_{i+2} - \tilde{F}_{i-2}) - \frac{1}{2100h}(\tilde{F}_{i+3} - \tilde{F}_{i-3})\end{aligned} \tag{2}$$

where h is the grid size, $\tilde{F}_{i\pm1/2} = \tilde{F}(U_{i\pm1/2})$ are the cell-edge fluxes, and $\tilde{F}_{i+m} = \tilde{F}(U_{i+m})$ are the cell-node fluxes. The cell-edge variables ($U_{i\pm1/2}$) are interpolated, and the numerical flux $\tilde{F}_{i\pm1/2}$ can be evaluated by cell-edge variables, which is similar to that of the WCNS (Weighted Compact Nonlinear Schemes)[9]:

$$\tilde{F}_{i\pm1/2} = \tilde{F}(U^L_{i\pm1/2}, U^R_{i\pm1/2}) \tag{3}$$

where $U^L_{i\pm1/2}$ and $U^R_{i\pm1/2}$ are the left-hand and right-hand cell-edge variables. For HDCS-E8T7, the seventh-order dissipative compact interpolation needs to solve the following system of tri-diagonal equations:

$$\begin{aligned} \tfrac{5}{14}(1-\alpha)U^L_{i-1/2} + U^L_{i+1/2} + \tfrac{5}{14}(1+\alpha)U^L_{i+3/2} &= \tfrac{25}{32}(U_{i+1}+U_i) \\ +\tfrac{5}{64}(U_{i+2}+U_{i-1}) - \tfrac{1}{448}(U_{i+3}+U_{i-2}) + \alpha[\tfrac{25}{64}(U_{i+1}-U_i) & \\ +\tfrac{15}{128}(U_{i+2}-U_{i-1}) - \tfrac{5}{896}(U_{i+3}-U_{i-2})] & \end{aligned} \tag{4}$$

where α is the dissipative parameter to control numerical dissipation. For the viscous fluxes, we also use a sixth order central difference scheme (see [9] for more details).

HOSTA is a production-level in-house CFD software containing more than 25,000 lines of FORTRAN90 codes. We present the main pseudo-code for the timing-marching loop of HOSTA in Fig.1. The HDCS scheme is implemented when calculating the viscous (*calc_viscous*) fluxes and the inviscid fluxes (*calc_inviscid*). We have implemented explicit Runge-Kutta method and several implicit time marching methods in HOSTA, but in the following GPU parallelization, we focus on the Jacobi iterative method (*Sol_jacobi*). Note that in each time step HOSTA performs four exchanges of boundary and singularity data to ensure the robustness of high-order schemes (see Fig.1).

3 MPI-CUDA Implementation and Optimizations for HOSTA

When parallelizing HOSTA via MPI, we partition a complex grid with single or multiple computational domains into multiple grid blocks for better load balance and then distribute them to MPI processes (as shown in Fig.2(a)). For a MPI-CUDA implementation, a collection of grid blocks owned by a MPI process will be computed by a GPU. In this section, we begin by parallelizing and optimizing HOSTA on a single GPU, and then move forward to large-scale GPU-accelerated systems. Finally we evaluate and validate our solutions on TianHe-1A.

3.1 Parallelization and Optimizations on a Single GPU

On a single GPU, we present a dual-level parallelization: fine-grained data parallelism within a block and coarse-grained task parallelism among multiple blocks. For the fine-grained parallelism, we use two approaches (3D kernel configuration or 2D kernel configuration) according to data dependency among cells in a block.

If there is no data dependency between cells (e.g., the procedure *Sol_jacobi*), each GPU thread calculates a cell independently. A 3D grid block is mapped to a GPU grid in CUDA as illustrated in Fig.2(b): the processing of grid cells on I,J and K direction (coordinate) are mapped to GPU threads on X,Y and Z dimension respectively; a grid block of size (NI,NJ,NK) is logically decomposed to some *sub_blocks* of size (x_blk,y_blk,z_blk) , each *sub_block* is computed by a GPU thread block of the same size, and those thread blocks compose a GPU grid of size ($\lceil NI/x\_blk \rceil$,$\lceil NJ/y\_blk \rceil$, $\lceil NK/z\_blk \rceil$), where $\lceil x \rceil$ is the minimum integer that is larger than x. When implementing HOSTA, complex stencil computations for the viscous (*calc_viscous*) and inviscid (*calc_inviscid*) fluxes are decomposed along the I, J and K directions. Thus, data dependencies only exist in the corresponding direction. Since CUDA has no global synchronization, we choose to use a 2D kernel configuration or 2D decomposition for this case, i.e., we use one GPU thread to compute all the cells on a cell line. For example, in the I direction, the 2D thread block is configured as (1,y_blk,z_blk) and the size of GPU grid is (1,$\lceil NJ/y\_blk \rceil$, $\lceil NK/z\_blk \rceil$).

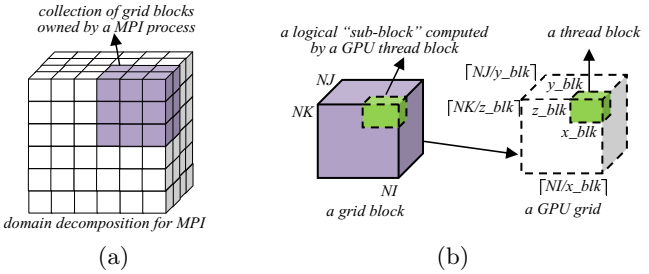


Fig. 2. Domain decomposition for MPI-CUDA parallelization

When there is a data dependency, we propose and use a kernel decomposition strategy to substitute a large 2D kernel with several small 3D kernels. A 3D configuration can ensure more GPU threads on the same dimension to access the global memory in a coalesced manner and thus improve performance. The kernel decomposition is mainly implemented in the computation of viscous and inviscid fluxes. For example, by carefully analyzing the data dependency, the initial 2D *I_viscous_kernel* to compute I direction's viscous fluxes is decomposed to three 3D kernels: *I_viscous_kernel_1* to compute the cell-edge metrics and primitives, and the cell-node viscous fluxes, *I_viscous_kernel_2* to compute the cell-edge viscous fluxes and *I_viscous_kernel_3* to compute the cell-node derivative of viscous fluxes. However, due to the interpolation in HDCS-E8T7, the initial 2D *I_inviscid_kernel* to compute I direction's inviscid fluxes is decomposed to four small kernels (three 3D kernels and one *semi*-3D kernel): *I_inviscid_kernel_1* to compute the cell-edge metrics, *I_inviscid_kernel_2* to reconstruct the left-hand and right-hand cell-edge primitives, *I_inviscid_kernel_3* to compute the cell-edge and cell-node inviscid fluxes and *I_inviscid_kernel_4* to compute the cell-node

derivative of inviscid fluxes. We implement the *semi*-3D *I_inviscid_kernel_2* by changing data structure and inter-exchanging loops. As Fig.3(a) shows, initially separated primitive variables $U_m(i,j,k)$ $(1 \leq m \leq 5)$ for the cell-node (i,j,k) form a system of tri-diagonal equations and there is a data dependency among neighboring statements. Fig.3(b) shows the new code snippet. A merged primitive variable $PV(m,i,j,k)$ replaces the five separate primitive variables to form another loop for m. Then we exchange loop m and loop i, and an independent "M" direction is available to be parallelized on GPU. Thus, a *semi*-3D kernel configuration of a 3D thread block (5,y_blk,z_blk) and a 2D GPU grid (1,$\lceil NJ/y\_blk \rceil$, $\lceil NK/z\_blk \rceil$) can be used for the decomposed kernel.

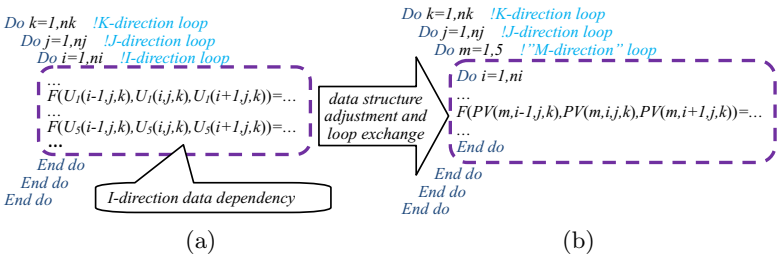


Fig. 3. Data structure adjustment and loop exchange for HDCS-E8T7

We implement the coarse-grained parallelism based on CUDA streams. Because there is no data dependency among grid blocks between exchanges of boundary/singularity data, we bind each block to a CUDA stream and issue all the streams simultaneously to the GPU. As Fig.4 shows, all the operations for block i such as computation, host to device (H2D) data copy and device to host (D2H) data copy are associated with stream i. The multi-stream implementation can fully exploit the potential power of modern GPU architecture. For example, when a stream is accessing global memory, the kernel engine can schedule and execute warps from other streams to hide memory access latency. More importantly, the kernel execution and PCI-E data transfer of different streams can be substantially overlapped, especially for GPUs with separate copy engines for H2D and D2H such as Tesla M2050 in TianHe-1A. Our multi-stream design is independent with the fine-grained GPU parallelization within a block. Furthermore, it can be used to overlap the GPU computation, data transfer and MPI communication as Sect.3.2 describes.

3.2 Moving Forward to Large-Scale GPU-Accelerated Systems

Since a compute node (of TianHe-1A) contains only one GPU, and thus, we choose to use one MPI process on each node for large-scale simulations on multiple GPUs. Considering a 3D block with six boundaries (or ghost zones), the data for a single boundary is continuously stored in the device memory, while the

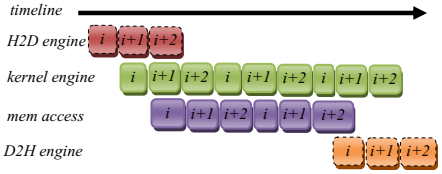


Fig. 4. Multi-stream execution for multiple grid blocks

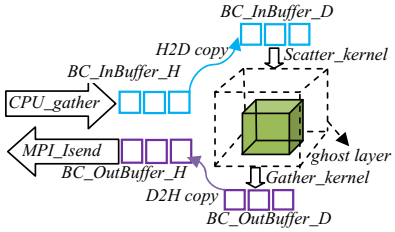


Fig. 5. Gather-scatter optimization for boundary data transfer of a 3D block

data for different boundaries is not. Further, CUDA API for data transfer can only copy continuous data elements for a time. Thus, we use a Scatter-Gather optimization to minimize the times of data transfer for boundary/singularity data of a 3D grid block via PCI-E, as Fig.5 illustrated. Before performing D2H copy, we use a gather kernel (i.e., *Gather_kernel*) to collect all non-continuous data to a continuous device buffer *BC_OutBuffer_D*, and then the entire buffer is copied to host buffer *BC_OutBuffer_H*. Correspondingly, before performing H2D copy, we use a gather procedure (i.e., *CPU_gather*) on the CPU to pack all the updated boundary data to a host buffer *BC_InBuffer_H*, and then the entire buffer is transferred to device buffer *BC_InBuffer_D*. Finally, we use a scatter kernel (i.e., *Scatter_kernel*) to distribute the data elements to each boundary.

Furthermore, we overlap the kernel execution, data transfer and MPI communication using CUDA multi-stream, non-blocking MPI and CUDA events. Fig.6 shows a schematic for the computation of the gradient of primitives, the inviscid and viscous fluxes. When the GPU finishes all the stream/block's operations for the gradient of primitives, a CUDA event with the same stream ID is recorded to represent data dependency between MPI communication and the boundary data computed by the stream. Before the host calls *MPI_Isend* to send the boundary data for a block, it must query the device to make sure that the event associated with the block/stream has been executed. As we can see from Fig.6, the data copy for block *i*, the non-blocking MPI send for block (*i-1*) and the computation of the gradient of primitives for block (*i+1*) are largely overlapped. When the GPU is executing the kernel or performing data copy, the CPU can also call *MPI_Irecv* to receive boundary data from blocks on other nodes (e.g., block (*i+1*), as illustrated by Fig.6). Note that *MPI_Waitall* must be called to ensure that *MPI_Irecv* has finished receiving data.

3.3 Performance Evaluation and Validation

We ported HOSTA to the GPU using a CUDA C and Fortran90 mixed implementation. Thus, our performance results on the CPU are all obtained from the Fortran90 implementation. We ported all the procedures (more than 16000 lines of CUDA C codes) in the time-marching loop except the procedures for boundary conditions and MPI communication (red codes in Fig.1). In our implementation,

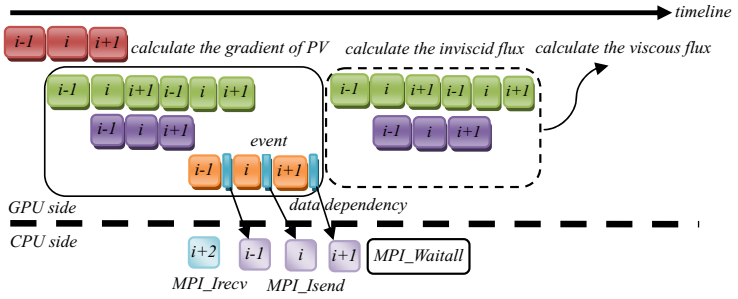


Fig. 6. The overlapping of kernel execution, data transfer and MPI communication when computing the gradient of primitives, the inviscid and viscous fluxes

we often maximize the kernel's performance when setting (x_blk, y_blk, z_blk) to (16,4,4) (i.e., the number of threads in a thread block is 256) and thus we choose it as the default setting in the following tests.

We use the TianHe-1A supercomputer as our test platform. Each of TianHe-1A compute nodes contains a Tesla M2050 and two six-core 2.93GHz Xeon X5670s. A customized high-speed interconnection network is used for the inter-node communication with a bi-directional bandwidth of 160Gbps and a latency of $1.57\mu s$. For more information about TianHe-1A, please refer to [7]. We use MPICH2-GLEX for MPI communication. The CUDA version is 4.2, and the compilers for FORTRAN and C are icc11.1 and ifort 11.1. All the code is compiled with -O3 optimization flag.

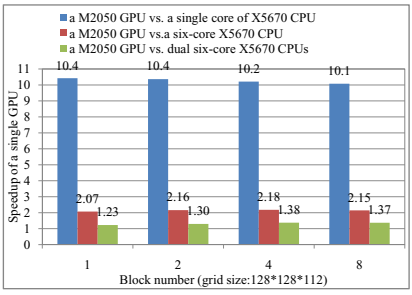


Fig. 7. Single GPU speedup for fixed grid size

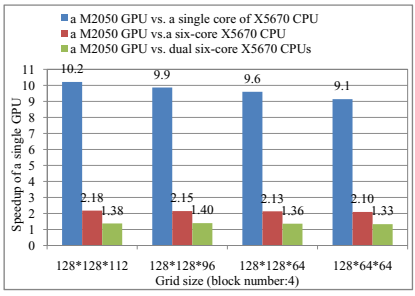


Fig. 8. Single GPU speedup for different grid sizes

In Fig.7, we present the speedup of a M2050 GPU over a single core of X5670 CPU, and the speedups of a M2050 GPU over a six-core X5670 CPU using 6 OpenMP threads and dual six-core X5670 CPUs using 12 OpenMP threads. The grid size is 128*128*112, and the block number is varying from 1 to 8. We can see that our GPU code can achieve a speedup of more than 10 over the serial

CPU code. Note that the price of a M2050 is similar to the price of an X5670, and a speedup of about 2.1 when comparing a M2050 to a six-core X5670 is also comparable to the results of paper [10] as far as similar-priced comparison of CPU and GPU is concerned. A speedup of about 1.3 when comparing a M2050 to dual X5670s further validates GPU's cost-effectiveness. We also observed slight performance degradation for both GPU and CPU when the block number is increased for a fixed grid size. This can be explained by the fact that multiple blocks will incur extra OpenMP and kernel overheads.

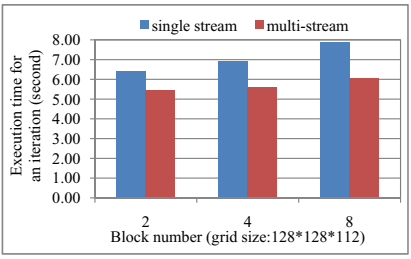


Fig. 9. Performance comparison for single stream and multi-stream implementation

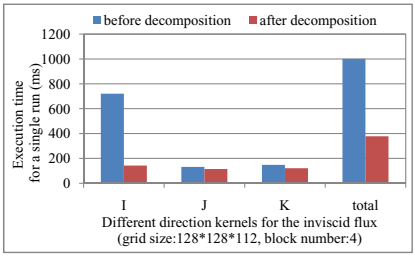


Fig. 10. Performance comparison for before and after kernel decomposition

In Fig.8, we present performance results for different grid sizes and the block number is fixed to 4. We get better performance for larger problem sizes. This is because higher workloads can better overlap the computation and global memory access for GPUs. Fig.9 shows the results of the multi-stream optimization. We see a 20% to 30% performance enhancement for a whole iteration. But our multi-stream implementation requires multiple blocks on a GPU to be associated with multiple streams and the extra GPU memory space to store the intermediate results for simultaneously executed blocks. Fig.10 shows the results when the kernel decomposition is adopted for the computation of the inviscid fluxes. We can reduce about 80% execution time for the I direction kernel, but for the J and K directions, the performance improvements are only about 15%. This is because the decomposition in the I direction can ensure more GPU threads to access the global memory in a coalesced manner.

In Fig.11, we shows the strong scaling results with and without overlapping. We obtain the results without the overlapping described in Sect.3.2 by directly copying the whole grid block between GPU and CPU. We use the performance achieved on 32 GPUs as a baseline and each GPU simulates a 128*128*112 grid with 4 blocks for the baseline test. Thus our total grid size for strong scalability test is fixed to 58720256 (namely 32*128*128*112). The grid is evenly partitioned to grid blocks and distributed to GPUs. The GPU number is scaled from 32 to 256 and the block number per GPU is fixed to 4. We see that the overlapping of computation, data transfer and MPI communication plays an important role for good scalability. We observe a speedup of about 5.5 when scaling from 32 GPUs

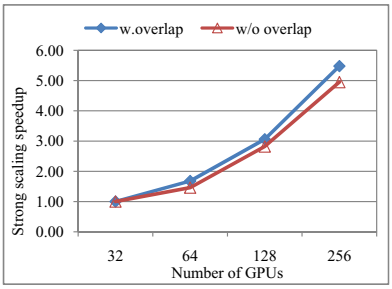


Fig. 11. Strong scaling speedup on TianHe-1A

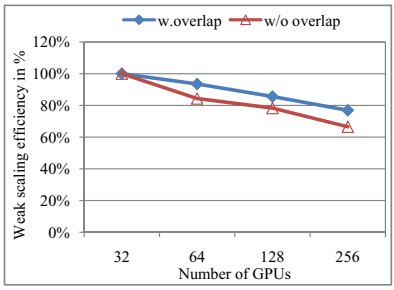


Fig. 12. Weak scaling efficiency on TianHe-1A

to 256 GPUs with overlapping, and a speedup of about 5.0 without overlapping, demonstrating a promising result for strong scalability test on large-scale GPU-accelerated systems. Fig.12 presents the weak scaling efficiency results with and without overlapping. The problem size for each GPU is fixed to 128*128*112 with 4 blocks. Again we use the performance achieved on 32 GPUs as a baseline and the GPU number is increased from 32 to 256. We lose about 23% (from the perfect weak scaling efficiency of 100%) using our overlapping strategy and 256 GPUs, while the efficiency loss is up to 33.5% for the non-overlapping one.

We use the same EET high-lift configuration and the same conditions as tested in the LTPT facility at NASA LaRC[11] to validate HDCS. Fig.13 shows the grid structure. The incoming Mach number M=0.17, and the chord length c=0.457m, the corresponding chord Reyonlds number is 1.71×10^6, the angle of attack is 4°. The effects of explicit LES models are imitated by the truncation error of HDCS to simulate the turbulent flow. The computational grid contains approximately 800, 000, 000 grid points. The dual time stepping scheme with 60 subiterations based on Jacobi iterative method is used for the time integration. The computation has been successfully performed on TianHe-1A using 400 GPUs and advanced with a time step of 2.5×10^{-6}s for total 30,000 time steps. Flow statistics are collected for 10, 000 time steps. Fig.14 demonstrates the iso-surfaces of the second invariant colored by density contours for the instantaneous flow. The second invariant Q is:

$$Q = (\Omega_{ij}\Omega_{ij} + S_{ij}S_{ij})/2 \tag{5}$$

where $\Omega_{ij} = (u_{i,j} - u_{j,i})/2$ and $S_{ij} = (u_{i,j} + u_{j,i})/2$. Due to the fine mesh, we can see the details about the vortex structure. The computational mean spanwise vorticity comparing with the experimental measurement from [11] is shown in Fig.15. We clearly see that the shear layer separating from the leading edge is demonstrated by the computation.

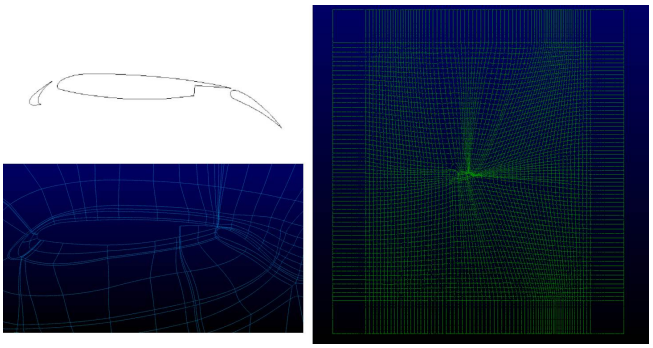
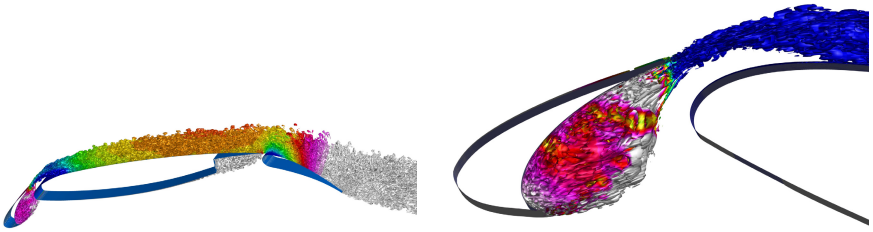
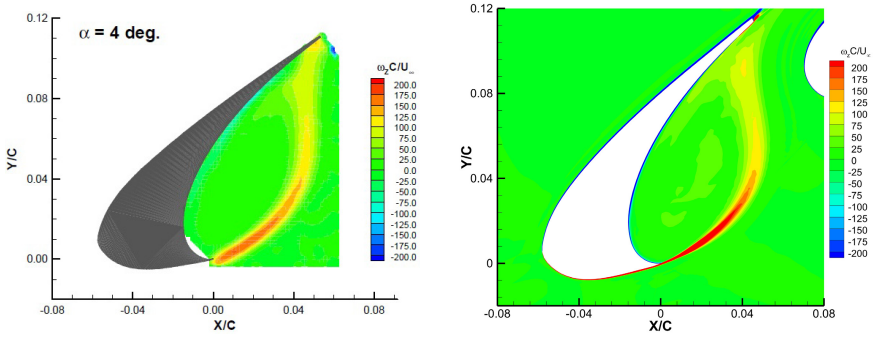


Fig. 13. Grid structure of the high-lift airfoil configuration



(a)Whole field (b)Near the slat

Fig. 14. Iso-surfaces of the second invariant of velocity gradient tensor for instantaneous vortex structure



(a)Experiment (b)Computation

Fig. 15. Time-averaged spanwise-vorticity distribution

4 Related Work

In the past twenty years, there have been many studies in developing and applying high-order compact finite difference schemes for CFD. Lele[1] has developed several central compact schemes with spectral-like resolution. Visbal and Gaitonde[12] use filters to prevent numerical oscillations of central compact schemes. Recently, Lele's central compact schemes have been successfully applied by Rizzetta et al.[13] for the simulation of low speed flows. In order to deal with shock wave problems, Adams et al.[14] have developed a compact-ENO (Essentially Non-Oscillatory) scheme. Pirozzoli[15] has developed a compact-WENO (Weighted Essentially Non-Oscillatory) scheme which was further improved by Ren et al.[16]. Deng et al. have developed the WCNS[9]. But as mentioned by Deng et al.[6], meeting GCL for complex configurations is difficult, which limits the applications of high-order finite difference schemes. For this, HDCS with inherent dissipation was derived and implemented in HOSTA for the high-resolution flow simulation on complex geometry. HDCS employs a new central compact scheme to fulfil the GCL and adds dissipation on the central scheme by high-order dissipative interpolation of cell-edge variables (see Sect.2).

Prior work has also shown the experiences of porting CFD codes to GPUs, with impressive speed-ups. Phillips et al.[17] have developed a 2D compressible Euler solver on a cluster of GPUs for rapid aerodynamic performance prediction. Appa et al.[18] implemented and optimized an unstructured 3D explicit finite volume CFD code on multi-core CPUs and GPUs for efficient aerodynamic design. Corrigan et al.[19] have ported an adaptive, edge-based finite element code FEFLO to GPU clusters in a semi-automated fashion: the existing Fortran-MPI code is preserved while the translator inserts data transfer calls as required. Brandvik and Pullan[22] implemented a multi-GPU enabled Navier-Stokes solver for flows in turbomachines. They reported almost linear weak scaling for typical turbomachinery cases using up to 16 GPUs. All the above studies were tested on small-scale GPU platforms using low-order CFD methods. For large-scale GPU clusters, Jacobsen et al.[20] parallelized a CFD solver for incompressible fluid flows. They used up to 128 GPUs for parallel scalability test. They further demonstrated the large eddy simulation of a turbulent channel flow on 256 GPUs[21], but they only simulated a lid-driven cavity problem using low-order schemes.

Due to the complexity, high-order CFD schemes generally require extra implementation and optimization efforts on GPUs. Tutkun et al.[23] have implemented a six-order compact finite difference scheme on a single GPU. Antoniou et al.[24] implemented a high-order solver that can run on multi-GPUs for the compressible turbulence using WENO. But the solver can only run on a single node platform containing 4 GPUs for very simple domains like a 2D or 3D box. Results show that their single-precision implementation can achieve a speedup of 53 when comparing 4 Tesla C1070 GPUs with a single core of an Xeon X5450 CPU. Castonguay et al.[25] published the parallelization of the first high-order, compressible viscous flow solver for mixed unstructured grids by using MPI and CUDA, where the Vincent-Castonguay-Jameson-Huynh method is used. A flow

over SD7003 airfoil and a flow over sphere is simulated using 32 GPUs. Appleyard et al.[26] reported a MPI-CUDA implementation to accelerate the solution of the level set equations for interface tracking using a HOUC (High-Order Upstream Central) scheme. But they only demonstrated performance results using 4 GPUs. Zaspel et al.[10] described the implementation of an incompressible double-precision two-phase solver on GPU clusters using a fifth-order WENO scheme. The test problem is a rising bubble of air inside a tank of water with surface tension effects and parallel performance results are reported using up to 48 GPUs. To summarize, we see that the above GPU-enabled CFD simulations using high-order schemes are still preliminary as far as the grid complexity, problem size and parallel scale are comprehensively concerned.

5 Conclusion and Future Work

The complexity of high-order CFD simulation for complex multi-block grids makes it very difficult to parallelize on large-scale heterogeneous HPC systems. In this work, we parallelize and optimize HOSTA, a high-order CFD software for multi-block structural grids. We successfully simulate a flow over a high-lift airfoil configuration on TianHe-1A using 400 GPUs. We conclude that TianHe-1A is a suitable platform to run the HOSTA-like CFD software in terms of performance and scalability, although programming it takes many efforts. Thus, we want to develop a heterogenous programming and auto-tuning framework for CFD-like scientific applications based on the current work. Furthermore, we plan to collaborate CPUs and GPUs for large-scale calculation on multi-block grids to further improve HOSTA's performance.

Acknowledgement. This paper was supported by the National Science Foundation of China under Grant No.11272352, the National Basic Research Program of China under Grant No. 2009CB723803 and the Open Research Program of China State Key Laboratory of Aerodynamics.

References

1. Lele, S.K.: Compact finite difference schemes with spectral-like resolution. J. Comput. Phys. 103, 16–42 (1992)
2. Trulio, J.G., Trigger, K.R.: Numerical solution of the one-dimensional hydrodynamic equations in an arbitrary time-dependent coordinate system. Technical Report UCLR-6522. University of California Lawrence Radiation laboratory (1961)
3. Deng, X.G., Mao, M.L., Tu, G.H., Liu, H.Y., Zhang, H.X.: Geometric conservation law and applications to high-order finite difference schemes with stationary grids. J. Comput. Phys. 230, 1100–1115 (2011)
4. Deng, X.G., Mao, M.L., Tu, G.H., et al.: Extending the fifth-order weighted compact nonlinear scheme to complex grids with characteristic-based interface conditions. AIAA Journal 48(12), 2840–2851 (2010)

5. Deng, X.G., Jiang, Y., Mao, M.L., Liu, H.Y., Tu, G.H.: Developing hybrid cell-edge and cell-node dissipative compact scheme for complex geometry flows. In: The Ninth Asian Computational Fluid Dynamics Conference (2012)
6. Deng, X.G., Mao, M.L., Tu, G.H., et al.: High-order and high accurate CFD methods and their applications for complex grid problems. Commun. Comput. Phys. 11, 1081–1102 (2012)
7. Yang, X.J., Liao, X.K., Lu, K., et al.: The TianHe-1A supercomputer: its hardware and software. Journal of Computer Science and Technology 26, 344–351 (2011)
8. NVIDIA CUDA: CUDA C programming guide v4.2 (2012)
9. Deng, X.G., Zhang, H.X.: Developing high-order weighted compact nonlinear schemes. J. Comput. Phys. 165, 22–44 (2000)
10. Zaspel, P., Griebel, M.: Solving incompressible two-phase flows on multi-GPU clusters. Comput. & Fluids (2012)
11. Jenkins, L.N., Khorrami, M.R., Choudhari, M.: Characterization of unsteady flow structures near leading-edge slat: part I. PIV measurements. AIAA paper 2004-2801 (2004)
12. Visbal, M.R., Gaitonde, D.V.: High-order accurate methods for complex unsteady subsonic Flows. AIAA Journal 37, 1231–1239 (1999)
13. Rizzetta, D., Visbal, M., Morgan, P.: A high-order compact finite difference scheme for large-eddy simulation of active flow control. Progress in Aerospace Sciences 44, 397–426 (2008)
14. Adams, N.A., Shariff, K.: A High-resolution hybrid compact-ENO scheme for shock-turbulence interaction problems. J. Comput. Phys. 127 (1996)
15. Pirozzoli, S.: Conservative hybrid compact-WENO schemes for shock-turbulence interaction. J. Comput. Phys. 179, 81–117 (2002)
16. Ren, Y., Liu, M., Zhang, H.: A characteristic-wise hybrid compact-WENO schemes for solving hyperbolic conservations. J. Comput. Phys. 192, 365–386 (2005)
17. Phillips, E.H., Zhang, Y., Davis, R.L., Owens, J.D.: Rapid aerodynamic performance prediction on a cluster of graphics processing units. AIAA paper 2009-565 (2009)
18. Appa, J., Sharpe, J., Moinier, P.: An unstructured 3D CFD code optimised for multicore and graphics processing units. In: MRSC 2010 (2010)
19. Corrigan, A., Lohner, R.: Porting of FEFLO to multi-GPU clusters. AIAA paper 2011-0948 (2011)
20. Jacobsen, D.A., Thibault, J.C., Senocak, I.: An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters. AIAA paper 2010-0522 (2010)
21. DeLeon, R., Jacobsen, D., Senocak, I.: Large-eddy simulations of turbulent incompressible flows on GPU clusters. Computing in Science & Engine 15, 26–33 (2013)
22. Brandvik, T., Pullan, G.: An accelerated 3D Navier-Stokes solver for flows in turbomachines. In: ASME Turbo Expo 2009: Power for Land, Sea and Air (2009)
23. Tutkun, B., Edis, F.O.: A GPU application for high-order compact finite difference scheme. Computers & Fluids 55, 29–35 (2012)
24. Antoniou, A.S., Karantasis, K.I., Polychronopoulos, E.D.: Acceleration of a finite-difference WENO scheme for large-scale simulations on many-core architectures. AIAA paper 2010-0525 (2010)
25. Castonguay, P., Williams, D.M., Vincent, P.E., Lopez, M., Jameson, A.: On the development of a high-order, multi-GPU enabled, compressible viscous flow solver for mixed unstructured grids. AIAA paper 2011-3229 (2011)
26. Appleyard, J., Drikakis, D.: Higher-order CFD and interface tracking methods on highly-parallel MPI and GPU systems. Computers & Fluids 46, 101–105 (2011)

Lattice QCD on Intel® Xeon Phi™ Coprocessors

Bálint Joó[1], Dhiraj D. Kalamkar[2], Karthikeyan Vaidyanathan[2], Mikhail Smelyanskiy[3], Kiran Pamnany[2], Victor W. Lee[3], Pradeep Dubey[3], and William Watson III[1]

[1] Thomas Jefferson National Accelerator Facility, Newport News, VA, U.S.A.
[2] Parallel Computing Lab., Intel Corporation, Bangalore, India
[3] Parallel Computing Lab., Intel Corporation, Santa Clara, CA, U.S.A.

Abstract. Lattice Quantum Chromodynamics (LQCD) is currently the only known model independent, non perturbative computational method for calculations in the theory of the strong interactions, and is of importance in studies of nuclear and high energy physics. LQCD codes use large fractions of supercomputing cycles worldwide and are often amongst the first to be ported to new high performance computing architectures. The recently released Intel Xeon Phi architecture from Intel Corporation features parallelism at the level of many x86-based cores, multiple threads per core, and vector processing units. In this contribution, we describe our experiences with optimizing a key LQCD kernel for the Xeon Phi architecture. On a single node, using single precision, our Dslash kernel sustains a performance of up to 320 GFLOPS, while our Conjugate Gradients solver sustains up to 237 GFLOPS. Furthermore we demonstrate a fully 'native' multi-node LQCD implementation running entirely on KNC nodes with minimum involvement of the host CPU. Our multi-node implementation of the solver has been strong scaled to 3.9 TFLOPS on 32 KNCs.

1 Introduction

A computationally challenging problem in Lattice Quantum Chromodynamics (LQCD) is the solution of the discretized Dirac equation in the presence of an SU(3) gauge field. Its key operation is the multiplication of a vector by a sparse matrix known as the Dslash operator. LQCD is an important calculational tool in nuclear and high energy physics.

The current trend in high performance computing is to couple commodity processors with various types of computational accelerators, which offers dramatic increases in both compute density, memory bandwidth and energy efficiency.

In this paper we describe the implementation and tuning of some key LQCD operations: Dslash and the solution of the Dirac Equation, for Intel's recently released Intel® Xeon Phi™ Coprocessor, codenamed Knights Corner (KNC). Our implementation exploits the salient architectural features of KNC, such as large caches, inter-core communication as well as hardware support for irregular memory accesses. By using a KNC-friendly lattice data layout together with a parallel cache blocked algorithm, our implementation of Dslash kernel on KNC sustains up to 320 GFLOPS on a single node, in single precision, which corresponds to nearly 87% of achievable performance. Furthermore, we demonstrate a fully 'native' multi-node LQCD implementation running entirely on KNC nodes, with minimum involvement of the host CPU. Our multi-node

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 40–54, 2013.

implementations of the Dslash operator and the Conjugate Gradients [1] solver have been strong scaled to 5 TFLOPS and 3.9 TFLOPS, respectively, on 32 KNCs.

2 Background

2.1 Lattice QCD

In this section we provide relevant details of LQCD for our work. For further details, many excellent references are available, e.g [2]. LQCD operates on an $N_d = 4$ dimensional space time lattice, with $V = L_x L_y L_z L_t$ sites, where L_x, L_y, L_z and L_t are the dimensions of the lattice in the X, Y, Z and T directions respectively. Quark fields are ascribed to the sites of the lattice while gluon fields are ascribed to the links between sites. The interaction of quarks and glouns is given by the Fermion matrix M. In the Wilson [3] formulation of quarks, M is given by:

$$M = (N_d+m) - \frac{1}{2}D, \text{with } D = \sum_{\mu=1}^{4} \left((1-\gamma_\mu)\otimes U_x^\mu \, \delta_{x+\hat{\mu},x'} + (1+\gamma_\mu)\otimes U_{x-\hat{\mu}}^{\mu\dagger} \, \delta_{x-\hat{\mu},x'}\right) \tag{1}$$

where D is the *Wilson-Dslash* operator (WD). In Eq. 1 the sum is over directions μ, m is a quark mass parameter, U_x^μ is the gauge link matrix connecting site x with its neighbor in the μ direction, and γ_μ are elements of a Dirac spin-algebra. The propagation of quarks in a gluon field is given by the Dirac equation: $M\psi = \chi$ where ψ and χ are spinors, which at each lattice site x are complex matrices carrying a spin index $\alpha \in [0, 1, 2, 3]$ and color index $a \in [0, 1, 2]$. Applying D to a spinor can be viewed as a nearest neighbour stencil operation. To facilitate the solution of M, an even-odd preconditioning is typically used, wherein a lattice is checkerboarded into even and odd sub-lattices and one solves the Schur complement system $\tilde{M}_{oo}\tilde{\psi}_o = \tilde{\chi}_o$ only on one checkeboard (in this case 'odd'). Here $\tilde{M}_{oo}$ is the Schur complement of M after checkerboarding given by:

$$\tilde{M}_{oo} = (N_d + m)\mathbb{I}_{oo} - \frac{1}{4(N_d + m)} D_{oe} D_{eo} \tag{2}$$

where subscripts oe, eo indicate that the operator maps odd sites to even, even to odd respectively. The system is large and sparse and is typically solved by an interative solver such as Conjguate Gradients (CG) [1] or BiCGStab [4].

The key operation is the sparse matrix vector multiplication $D\psi$. The naive arithmetic intensity of the Dslash operator is 1320 flops / 1440 bytes = 0.92 flops/byte in single precision, however, due to nearest neighbor nature of the Dslash operator, there is substantial reuse amongst spinors. Properly exploiting this reuse with caches reduces memory traffic, and can almost double the arithmetic intensity [5]. In addition, memory traffic can be reduced further by performing 2-row gauge field compression, by making use of the $SU(3)$ nature of the links and storing only two rows of the matrix and reconstucting the third row on the fly [6].

2.2 Intel® Xeon Phi™ Coprocessor Architecture

The recently released Intel® Xeon Phi™ Coprocessor architecture features many in-order cores on a single die. Each core has 4-way simultaneous multithreading (SMT or hyper-threading) support to help hide memory and multi-cycle instruction latency. In addition, each core has 32 vector registers, 512 bits wide, and its vector unit executes a 16-wide (8-wide) single (double) precision vector instruction in a clock cycle, which can be paired with scalar instructions and data prefetches. KNC has two levels of cache: a single-cycle access 32 KB first level data cache (L1) per core and a larger globally coherent second level cache (L2) that is partitioned among the cores, with a private 512 KB partition per core. KNC also has hardware support for irregular data accesses and several flavors of prefetch instructions for each level of the memory hierarchy.

KNC is physically mounted on a PCIe slot and has dedicated GDDR memory. Communication between the host CPU and KNC is therefore done explicitly through message passing. However, unlike many other coprocessors, it runs a complete Linux-based operating system, with full paging and virtual memory support, and features a shared memory model across all threads and hardware cache coherence. Thus, in addition to common programming models for coprocessors, KNC supports more traditional multi-processor programming models such as OpenMP [7].

3 QCD Implementation on KNC

Our library is written in C++ and threading is carried out using OpenMP threads. The library implements the Wilson Dslash, the even-odd preconditioned Wilson operator and a CG solver. We consider the code in two parts: a high level part which is concerned with parallelizing over threads, and performs the loop structure for our cache blocking strategy, and a 'back end' part which takes care of working on a vector of lattice sites.

To achieve high performance on KNC, one must take full advantage of its vector capabilities. A SIMD friendly, *partial structure of arrays* (SOA) layout such as described in [5] can run with high vector efficiency, however it requires that a *scanline* (line of sites in X) be a multiple of the hardware vector unit length (vec). This can restrict the application of the code to problems where $L_{xh} = L_x/2$, the X-width of a checkerboard, is a multiple of vec. The larger vec is, the more restrictive this becomes. Our first KNC implementation was written this way. However, we have developed a more general approach, described below, to address this limitation. Specifically, we allow the inner array length soa of the SOA (or SOA length) to be a factor of vec. This has the advantage of allowing more general problem sizes, but since it mixes X and Y dimensions it can complicate communications in those directions. Our code is templated on floating point type, vector length vec and SOA length soa. The back end codes are generated using a code generator which we will describe below, and are hooked into the main library using template specialization. We focused specifically on the cases of $vec = 16$, $soa = 4, 8, 16$ for Xeon Phi and $vec = 8$, $soa = 4, 8$ for SNB-EP (using AVX).

3.1 Data Structures and Indexing

Our primary data structures are $SU(3)$ gauge fields and 4-spinors as discussed earlier. Each of these fields is associated with a lattice site. In the case of the gauge fields we

```
typedef float SU3MatrixBlock[8][3][3][2][vec];
typedef float FourSpinorBlock[3][4][2][soa];
```

Fig. 1. The structures of the fields used for a single block of computation

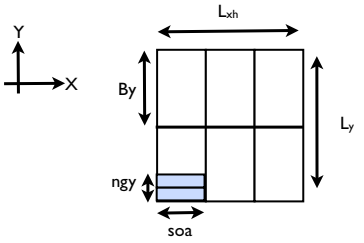


Fig. 2. Our vectorization scheme: An X-Y planar view shows blocks of length soa in X, and ngy lines in Y. In this instance $ngy = 2$ and $vec = 2soa$.

associate with a site the 8 links emenating from it (forward and backwards in each of 4 directions). We then split the lattice into blocks of soa sites. In the case of spinors we have $V_B = L_{xh}L_yL_zL_t/soa$ such spinor-blocks per checkerboard. We require that soa divide L_{xh} exactly. Our back end kernels process ngy blocks, each of length soa sites, where $ngy = vec/soa$. A single processing step can thus process a full vec sites worth of data, made up of spinor blocks of length soa sites from ngy different y coordinates. Hence we must also have that L_y be divisible by ngy. The situation is illustrated in Fig. 2. The gauge fields could be packed similarly, however since they are typically used in several dslash applications, it is worth repacking the ngy blocks of length soa into a single block of length vec up-front, allowing them to be read as a single vector. Hence the gauge field has $V_{BG} = L_{xh}L_yL_zL_t/vec$ blocks. We show the C++ definitions of the spinor and gauge block datatypes in figure 1.

To reduce the effects of associativity conflict misses, we pad our lattice arrays, by adding Pad_{xy} blocks onto the end of every XY plane and Pad_{xyz} blocks onto the end of every XYZ time-slice. Hence a spinor site with coordinates (x, y, z, t) is indexed as follows: We locate the XY plane for the z and t indices using the formula $xyBase = t\ Pxyz + z\ Pxy$ with $Pxy = (L_{xh}L_y/soa) + Pad_{xy}$, and $Pxyz = L_zPxy + Pad_{xyz}$. Within the XY plane, we first split the x coordinate into an x-block index; $xb = x/soa$ and an index within the block: $xi = x \mod soa$. The offset to the block is now $xb + nsoa \times y$ with $nsoa = L_{xh}/soa$. Given an array `spinor` of objects of type `FourSpinorBlock` as defined in Fig. 1, with V_B array elements, the site would be indexed as `spinor[ xb + nsoa*y + xyBase ][c][s][r][xi]` where `c`, `s` and `r` are indices for the color, spin and complex component respectively. In our code, we compute $xyBase$ and then separate offsets in terms of floating point numbers to be used by gathering loads. Indexing gauge fields is slightly different, since ngy lines of y are packed together. In this case the block offset would be computed as $xb + (nsoa * y + xyBase)/ngy$.

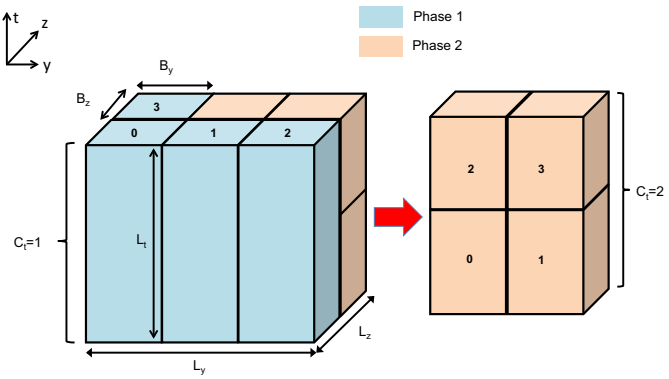


Fig. 3. Cache Blocking: The lattice is blocked in the Y and Z dimensions with block sizes B_y and B_z respetively. In this example there are 6 blocks to be processed by 4 cores. The number on each block is the index of the core which will process it. The first four blocks are mapped round-robin to the cores in phase 1 (blue). In phase 2 the remaining 2 blocks are each split in the temporal direction over 2 cores ($C_t = 2$) to use all 4 cores (orange).

3.2 Code Generator

We wrote a simple code generator to overcome two programming challenges: having a portable abstraction for producing intrinsics to generate code for both Xeon Phi, and AVX. The second challenge was to schedule software prefetches with the main computation. The generation of L1 prefetch instructions is part of the code generating routines. The L2 prefetches are generated in a separate pass and the two instruction streams are intermixed. The current version of the generator supports single precision vector instructions for Xeon Phi and AVX. It is straightforward to extend the generator to support double precision and other vector architectures. Gathering (scattering) spinors from (to) their $soa \times ngy$ XY blocks is supported both via the gather intrinsics on Xeon Phi, or via a sequence of masked load-unpack, or pack-store instructions. To take advantage of streaming stores we use in-register shuffles to pack full cache lines, which are then streamed to memory using nontemporal store instructions.

3.3 Cache Blocking and Block-to-Core Mapping

Our code implements a variant of 3.5D blocking [8]. To deal with the per-core private caches of Xeon Phi, we define cache blocking dimensions B_y and B_z such that at minimum 3 T-slices of $L_{xh}B_yB_z$ sites fit into the private L2 cache of each core. The second problem we addressed was to maximize the number of cores that can be used from the N_c cores available, while maintaining load-balance. Our scheme for assigning blocks to cores, which is pre-computed at initialization, is as follows:

Given B_y and B_z, we have in total $N_b = l_y l_z$ blocks, where $l_y = L_y/B_y$ and $l_z = L_z/B_z$. The blocks are assigned to cores in potentially several phases. Initially we set the number of remaining blocks N_r to N_b. Then, in each phase p, if $N_r \geq N_c$, we assign one block to each core, and allow the core to stream through all L_t T-slices.

We then decrement N_r to $N_r - N_c$ for the next phase and repeat this process until either all the blocks are assigned, or fewer blocks remain than available cores: $N_r < N_c$.

At this point, we will start using multiple cores per block by dividing each block into C_t^p parts along the T-direction and assigning a core to each part. If N_r divides N_c exactly, we choose $C_t^p = N_c/N_r$ and finish the assignments. Otherwise we first calculate candidate values $C_t^l = floor(N_c/N_r)$ and $C_t^u = ceil(N_c/N_r)$ respectively. Choosing C_t^l partitions, would allow us to finish all the N_r remaning blocks, but at the expense of under utilizing the N_c cores, since we would use only $N_r C_t^l$ cores, which is less than N_c. Instead, choosing C_t^u partitions, would allow processing only $floor(N_c/C_t^u)$ blocks in this phase (which are less than the remaning N_r blocks), but utilize most of the cores. We would then keep the remaining $N_r - floor(N_c/C_t^u)$ blocks for the next phase, however, in this situation we would have even fewer blocks to assign to the cores in the next phase, which would lead to a further reduction of work per core in that phase. We use a simple heuristic to pick between C_t^l and C_t^u: If C_t^u is greater than some threshold T_t we will assume that picking C_t^u would result in each core having too little work, so we pick C_t^l instead and terminate the process. Otherwise we pick C_t^u, decrement N_r to $N_r - floor(N_c/C_t^u)$ and carry on to the next phase. We found $T_t = 4$ was optimal on both Xeon and Xeon Phi. It is useful to keep track of the number of blocks to be processed by the phase defined as $C_{yz}^p = \min(N_r, N_c/C_t)$.

The scheme can be elaborated, to allow the user to specify a minimum value of C_t^p (denoted $minCt$) at the start of the mapping process to allow the handling of *non-uniform memory access* (NUMA) affinity issues. Since our data structures are divided in the T-dimension accross different NUMA domains, we set $minCt$ to the number of NUMA domains and we re-define N_c to be the number of cores per NUMA domain throughout our mapping process. Now, for cases where we would not have split the blocks previously, we will split them into $minCt$ parts with a core (from a different NUMA domain) being assigned to each part. For the phases where originally we would have split the blocks, we now compute C_t^p with the re-defined N_c and we multiply the result by $minCt$. It should be clear that the scheme we described originally is the case when $minCt = 1$ which is appropriate for uniform memory access architectures such as for Xeon Phi whereas for a dual socket Xeon we would have $minCt = 2$.

We illustrate the blocking scheme in Fig. 3, using a case of 6 blocks to be mapped to 4 cores as an example. The work splits into 2 phases: phase 1 processes the blocks colored in light blue. With $minCt = 1$, each of these blocks will be processed by a single core without splitting the time direction. The remaining 2 blocks will be processed by phase 2. In this phase, we have that N_r divides N_c exactly so we choose $C_t^2 = N_c/N_r = 2$ and finish the assignment process. Although the number of cores does not exactly divide the original number of blocks, all the cores are utilized in both phases in this example.

In each phase, C_t^p should be kept to the minimum, to minimize T-boundary exchange overhead between cores. In this scheme cores working on the same T-slice in the same phase, on adjacent $B_y \times B_z$ blocks will share boundary data via inter-core communication, without going back to memory. However, in order to ensure constructive sharing, some degree of synchronization must be maintained between these cores, which is achieved via infrequent inter-core barriers amongst the groups of C_{yz}^p cores working on the same portion of the C_t^p-way split in the T-direction.

3.4 SMT Threading and Lattice Traversal

Within each core we treat the available SMT threads as a grid of threads with dimensions $S_y \times S_z$. Our lattice traversal looping strategy for any given thread is as follows: First, the core index c_i and SMT coordinates (s_y, s_z) of the thread are computed from the thread index tid using the relation: $tid = s_y + S_y(s_z + S_z c_i)$. The threads then loop through all the phases. Threads with core index $c_i \geq C^p_{yz} C^p_t$ for the current phase p, have no work to do and can continue to the next one. The core indices of the remaining threads are then divided into coordinates (c_{yz}, c_t) for the current phase using the relation $c_i = c_{yz} + C^p_{yz} c_t$. Using c_{yz} and the phase index p, the block to be processed by the core is identified and the starting value of $t = c_t(L_t/C^p_t)$ for the core is computed.

The thread then streams through its range of T values in the block, and for each value of t it scans through the XYZ volume. The local YZ plane of the block is split between the SMT threads on the core. Loops over the Z dimension are carried out with a stride of S_z. As each SMT thread processes a $soa \times ngy$ block, looping is done in units of spinor-blocks of length soa in X, and in increments of ngy S_y in Y respectively. The innermost loop can then use the information from the loop indices and the origins computed to index the neighbouring spinor-blocks to be read, the output spinor-block to be written, and to identify the neighbours of the successive spinor-block for L2 cache prefetching. On Xeon Phi, the optimal tile size was $B_y = B_z = 4$ which gave the best tradeoff between cache efficiency and redundant memory traffic on the edges of the blocks.

3.5 Linear Algebra Routines

In order to implement a Conjugate Gradients solver, one needs several vector linear algebra operations. These are essentially streaming operations, requiring few floating point operations and are heavily bound by memory bandwidth. If there is a chance to increase reuse between them it is worth fusing several successive kernels, for example in a situation where one can compute a residual vector, and compute its norm at the same time. We coded these operations in vector intrinsics with explicit software prefetching. We found we can achieve different levels of throughput depending on how many threads are used per core. We show in table 1 the various kernels used along with the (possibly fused) operations they perform, and as an illustration, the memory bandwidths achieved by the kernels from a particular timing measurement. We see that in most kernels one or two threads per core performed optimally. In our timing runs we have auto-tuned the number of threads in our linear algebra kernels for efficiency.

3.6 Multi-node Considerations

It is typical to parallelize LQCD problems onto multiple nodes to increase performance. Further, memory limitations can also force the calculation onto multiple nodes. Consequently we parallelized our code onto multiple Xeon Phis (and also SNB-EPs). Since our vectorization scheme mixes X and Y directions, we communicate between nodes only in the T and Z directions. The multi-node implementation overlaps computation of the body with the communication of the faces which are projected using the $(1 \pm \gamma_\mu)$

Table 1. Vector linear algebra kernels and the operations they perform. Here x, y, r, p, q are lattice spinors, α and ρ are scalars, and in a Conjugate Gradients solver we have that $q = M^{\dagger}Mp$. We also quote memory bandwidths observed in these kernels as a function of threads per core from a single timing measurement on a Xeon Phi 7110P device in a node of the Endeavor cluster, rounded to the nearest GB/sec.

Routine Name	operation	B/W 1 thread (GB/s)	B/W 2 threads (GB/s)	B/W 3 threads (GB/s)	B/W 4 threads (GB/s)
`aypx2`	$y \rightarrow \alpha y + x$	170	175	167	161
`xmyNorm2`	$r \rightarrow x - y$ $\rho \rightarrow \|\|r\|\|^2$	112	158	150	149
`norm2`	$\rho \rightarrow \|\|r\|\|^2$	154	157	157	157
`rmammpNorm2rxpap`	$r \rightarrow r - \alpha q$ $\rho \rightarrow \|\|r\|\|^2$ $x \rightarrow x + \alpha p$	175	159	143	135
`copy`	$x \rightarrow y$	133	154	156	152

operations in Eq.1 into buffers prior to sending. Once the body computation and communications are complete, the received faces are multiplied appropriately with gauge links, and their contribution to Dslash is accumulated as is common in LQCD implementations (e.g. [9, 10]).

A novel feature of the Xeon Phi architecture and software ecosystem, is that native implementations can make direct MPI [11] message passing calls, freeing up the user from orchestrating data transfers between host and the co-processor as would be needed in an offload model. This allows the programmer to treat a cluster of KNCs as a regular cluster of homogenous MPI nodes and thus improves the ease of programming. Our code uses the QMP [12] message passing layer over MPI.

On the other hand, although communication across the KNCs using MPI directly is optimized for latency, the achievable peak bandwidth is quite low due to hardware issues unrelated to Xeon Phi. As we scale LQCD in a cluster, the boundaries that are exchanged with the neighbors vary in size between 256KB and several MBs for large problems such as one would schedule on Xeon Phi-s, and for these message sizes the bottleneck is typically the communication bandwidth.

There are different bandwidth characteristics between different endpoints in a multi-node Xeon-Phi platform and we have developed a reverse communication MPI proxy that exercises the fastest path, similar to the design proposed in [13]. We use this proxy to handle the bandwidth limited nearest neighbour communications via the host. We assign a CPU core to process requests from local KNCs, for extracting the data from local KNC memory to host memory via DMA and for sending the data to the destination CPU. Similarly, at the destination, a CPU core receives the data and copies it from host memory to KNC memory. The whole process is performed in a pipelined manner by splitting the application data into several small chunks. The chunk sizes for a given application message are also chosen dynamically since smaller chunk sizes can amortize

the startup overheads but at the cost of lower bandwidth while larger chunk sizes give good bandwidth but may expose startup overheads. We use a memory mapped request and response queue model [14] to control message handshakes between the CPU and Xeon Phi. This proxy is lightweight and does not consume significant CPU resources.

4 Hardware Setup and Experiments

Our numerical experiments consist of two kinds of measurements: the performance of our Dslash operator and the performance of the Conjugate Gradients solver. We have also measured single node performance of both, for comparison, using SNB-EP CPUs as well as NVIDIA Tesla® Kepler™ K20 GPUs, specifically K20m models (referred to as K20m or Kepler K20m from here on).

Two different kinds of Xeon Phi systems were used: nodes from the Endeavor Xeon Phi Cluster operated by Intel and nodes from from the Jefferson Lab (JLab) 12m cluster. An Endeavor cluster node is comprised of dual socket Intel Xeon E5-2670 (SNB-EP) CPUs with 8 cores per socket running at 2.6 GHz, and two Xeon Phi co-processors with 61 cores each running at 1.1 GHz. The Xeon Phi has SKU B1PRQ-7110P (7110P from here on), using B1 stepping silicon, running Intel MPSS version 2.1.3552-1. The 7110P has 7936 MB of GDDR memory running at 2.75GHz. The nodes are connected with an FDR Infiniband interconnect. For compilation we used Intel Composer XE version 13.0.0 and Intel MPI version 4.1.0.027. Nodes of the JLab 12m cluster contain dual socket Xeon E5-2650 CPUs running at 2.0 GHz. They contain four Xeon Phi 5110P co-processors (5110P from here on), each of which has 60 cores running at 1.053GHz. The 12m nodes run MPSS version 2.1.4346-16 (Gold) and are connected by FDR Infiniband. The JLab 12m nodes run CentOS 6.2. The Xeon Phi nodes were booted with icache snooping turned off, and we utilized large memory pages. The last core on each Xeon Phi is reserved for system functions and was not used in our tests.

Our Kepler K20m measurements were made on a node of the JLab 12k cluster. The base nodes of this system (chassis, motherboard, CPU, fabric) are the same as for the 12m cluster node described previously. We used the gcc-4.4.6 and CUDA Toolkit v5.0 for compilation, and ran using version 304.54 of the CUDA driver. Our reference measurements used the publicly available QUDA [6] software package for lattice QCD on GPUs[1] in a pure single precision mode. We have also verified our measured performance using an NVIDIA Tesla® Kepler™ K20c model GPU and found the results to be identical to the K20m results within experimental fluctuations.

We have chosen 5 volumes on which to run our timing tests. We were driven in our choice by the spatial sizes of 24^3, 32^3, 40^3 and 48^3 sites for the first four of these, aiming to make the temporal direction as large as we could fit on the device, in order to mimic a capacity mode of operation, where as few nodes as possible are used to perform a calculation. However due to memory limitations we had to vary the T-extent and in one case the Z extent. The volumes thus chosen were $24^3 \times 128$, $32^3 \times 128$, $40^3 \times 96$ and $48^2 \times 24 \times 64$ sites respectively. Our fifth volume, of $32 \times 40 \times 24 \times 96$ was chosen

[1] We used git commit-ID: 541c66ba1a0eca11eb555dc8de6686cd54383c6c, master branch, Feb-04, 2013, available from `https://github.com/lattice/quda`

to allow maximum performance on 60 cores without having to split the time direction over the cores. Using 4×4 blocks in the Y and Z directions, this volume can ideally use a single phase with C^1_{yz} of 60 cores and $C^1_t = 1$.

5 Single-Node Results

Our single node measurements for Wilson Dslash are shown in figure 4. We show both the case when gauge fields are compressed and when they are uncompressed. We show performance for all 4 systems considered. The different colored bars correspond to different volumes.

On our Xeon Phi platforms we see performances between 270 - 296 GFLOPS on our 5110P part and between 295 - 320 GFLOPS on our 7110P part when 2 row compression is enabled. We note that in the $V = 32 \times 40 \times 24 \times 96$ case, performance hits 320 GFLOPS on the 7110P, but recall that this is an 'ideal volume' for this device. In contrast, on the NVIDIA K20m devices the performances we have seen for Dslash are in the region of 250 GFLOPS with compression.

We note that a dual socket Xeon E5-2680 (2.6GHz) CPU can sustain performances that are just under half the speed of a the Xeon Phi 5110 – roughly 120 GFLOPS in the case of the SNB-EP vs roughly 280 GFLOPS in the case of the Xeon Phi 5110 with compression enabled.

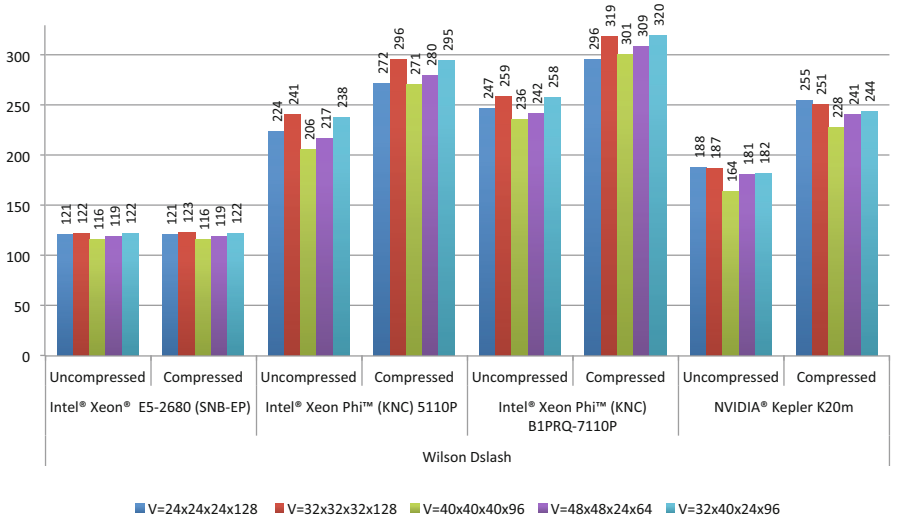


Fig. 4. Performance of Wilson Dslash for various volumes, for Xeon E5 (Sandy Bridge), Xeon-Phi 5110P, XeonPhi 7110P and NVIDIA K20m, Vertical axis shows performance in GFLOPS, rounded to the nearest GFLOPS

The performances of the Conjugate Gradients algorithm are shown in figure 5. We consider the same volumes as for the Dslash. Here we find that for uncompressed gauge fields the 5110P system slightly outperforms the K20m and with compression the 5110P and the K20m are very similar with the 5110P being faster on some volumes and the K20m being faster on others. The 7110P system is slightly faster than both the 5110P and K20m. Again, we observe roughly a factor of 2 in performance between the dual socket SNB-EP system and the Xeon Phi 5110P.

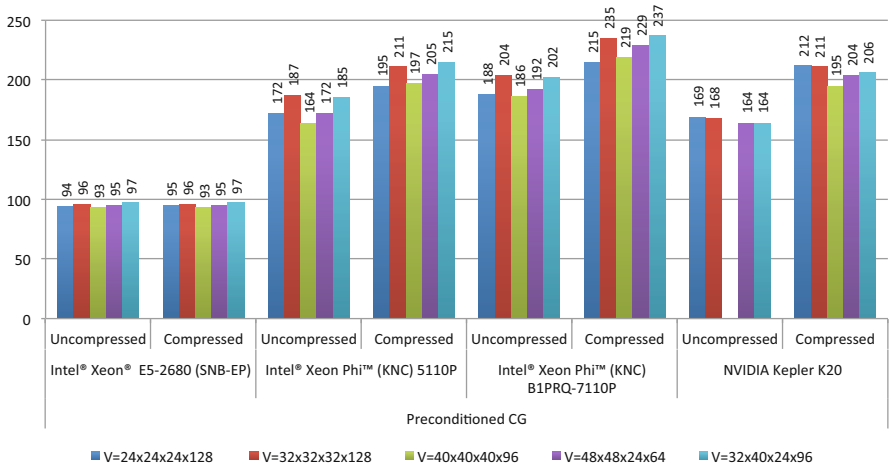


Fig. 5. Performance of Conjugate Gradient solver for various volumes, for Xeon E5 (Sandy Bridge), XeonPhi 5110P, XeonPhi 7110P and NVIDIA K20m. Vertical Axis shows performance in GFLOPS

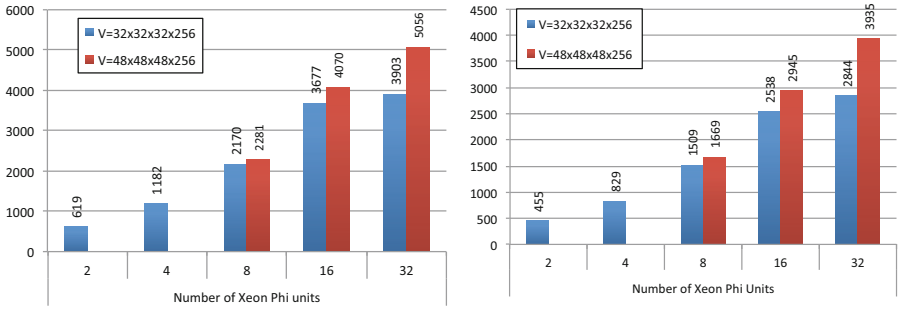
Finally we note that compression does not appear to help on the SNB-EP system, which sustains roughly 120 GFLOPS in the Dslash in both cases with and without compression. This is due to a temporary shortcoming in our implementation which explicitly stores the third row of the gauge fields. While the compressed implementation does not explicitly use this row, it is still prefetched by the hardware prefetcher of the SNB-EP resulting in the same amount of memory traffic as the uncompressed case and hence no speedup over that case is observed. Reducing the dimension of the rows from 3 to 2 in the `SU3MatrixBlock` datatype in Fig. 1 increased the performance on our $V = 24^3 \times 128$ problem to 146 GFLOPS in a test harness and we will integrate this change into our main code as future work.

To understand these results let us consider the following: with perfect spinor reuse where all but one of the neighbouring spinors are in cache, using gauge compression (but not counting the extra flops it incurs) the memory-bound performance of Wilson-Dslash is $\frac{1320}{4(24\times2+12\times8)/B_m}$ GFLOPS [5]. Here 1320 is the naive floating point operation count per site, and the numerical factors in the denominator count the amount of data required by the operation per site in bytes: one input and one output spinor comprised of 24 `floats` each, and 8 gauge link matrices comprised of 12 `floats` each

when compressed. The remaining factor of 4 is the size of a `float` in bytes and B_m is the memory bandwidth in GB/sec. Memory bandwidth can be measured using streaming kernels such as `aypx2` in Tab. 1, which is similar to the STREAMS triad benchmark. We see from Tab. 1 that the `aypx2` and `copy` kernels achieve bandwidths of 161 GB/sec and 152 GB/sec respectively, when using 4 threads per core. With $B_m = 161$ GB/sec, the maximum expected Dslash performance is ∼369 GFLOPS. The deviation from this idealized model is due in part to synchronization overheads and in part to our inability to use all the cores for certain problem sizes, although our blocking scheme attempts to maximize the number of cores used. These additional factors are mildest for the $32 \times 40 \times 24 \times 96$ site volume on the 7110P part where, as a result, we achieve 320 GFLOPS which is 87% of the available performance. On the SNB-EP, memory bandwidth is about half that of KNC, while on the K20m it is comparable to KNC. Hence the SNB-EP runs at roughly half the speed of, while the NVIDIA K20m performs similarly to KNC.

6 Multi-node Results

Our multi-node experiments were carried out on up to 32 nodes of the Endeavor cluster described in sec. 4. In our tests we used one KNC processor per cluster node. Figure 6 shows the strong scaling performance of our code on lattices of size $V = 32^3 \times 256$ and $V = 48^3 \times 256$ sites for both the Dslash operator and the Conjugate Gradients algorithm.



(a) (b)

Fig. 6. Strong Scaling multi-node Performance on the Endeavor cluster: (a) Wilson Dslash, (b) Conjugate Gradients. The vertical axis shows the performance in GFLOPS rounded to the nearest GFLOPS.

The performances in Fig. 6 show nearly linear strong scaling up to 16 nodes on both problem sizes. However, the scaling seems to top out when going to 32 nodes in the case of the $32^3 \times 256$ site problem, and we can see strong scaling effetcs on the $48^3 \times 256$ problem also, where scaling efficiency is reduced to 62% for Dslash. The two problems

sustain 3.9 TFLOPS and 5 TFLOPS respectively on 32 nodes. The reduction in scaling is due to the local problem decreasing to a sufficiently small size, that the overhead of packing, communicating and processing the boundaries is exposed.

In turn, the Conjugate Gradients algorithm sustained a performance of up to 2.8 TFLOPS for the $32^3 \times 256$ lattice and 3.9 TFLOPS for the $48^3 \times 256$ lattice. We also see that for the $48^3 \times 256$ volume, on 32 nodes, the CG solver has a performance of roughly $0.78\times$ that of the Dslash, or roughly a 22% slowdown. This slowdown is due partly to the heavily memory bandwidth bound vector-vector linear algebra in the solver and to synchronization overheads incurred in global reduction operations (global sums) performed across the MPI cluster. Although beyond the scope of this work, we expect we could improve our solver performance using a variant of pipelined CG [15].

7 Related Work

There are several implementations of LQCD Dslash implementations in the literature, many of them targeting novel architectures (of the time of their writing) such as GPUs. Optimizing for the Xeon architecture is described by us in [5], whereas recent efforts to optimize for BlueGene/Q architecture are presented in [10, 16]. GPU implementations using CUDA are presented in [6, 17], while OpenCL versions are in [18, 19]. Autotuned blocking schemes for Wilson Dslash have been investigated on GPUs in [20].

Historically the implementations on various architectures are too numerous to list. We will content ourselves with mentioning the implementation of [21] on the QCDSP supercomputer and of [22] on the BlueGene/L, since these contributions both won Gordon Bell prizes for cost effective supercomputing in 1998 and 2006.

Code generators for writing efficient low-level code are described in [23] and [24].

Reverse offload style communications are described in [13] for the Road Runner Supercomputer, and in [25] for GPU applications.

8 Conclusion

We have detailed our approach to implementing the Wilson Dslash operator for Intel Xeon Phi, and have presented performance results for both single and multi-node settings. Our single node Dslash operator sustains $\sim$280 GFLOPS and over 300 GFLOPS in single precision on Xeon Phi 5110P and 7110P devices respectively. Our multi-node implementation has been run on up to 32 Xeon Phi devices and has been strong scaled up to 5 TFLOPS for Wilson-Dslash, and up to 3.9 TFLOPS for the CG solver, both in single precision, on a lattice of $48^3 \times 256$ sites.

We compared our single node results with performances from the QUDA library on NVIDIA K20m GPUs. We ran QUDA in pure single precision mode to have like for like tests. We found that our code on Xeon Phi had higher performance on the Dslash operator in single precision than the K20m, but that the performance in Conjugate Gradients was similar between K20m and the Xeon Phi 5110P. The Xeon Phi 7110P system in turn performed a little faster than both the 5110P and the K20m. We note, however, that the QUDA library can gain additional performance on the GPUs than what is quoted

here by employing 16-bit precision in mixed precision solvers. We are considering implementing this approach as potential future work.

To achieve these performances required attention to expose parallelism on vector instruction, SMT thread and core levels on a single KNC as well as the use of cache-blocking techniques. To achieve the multi-node performances we used a reverse communication proxy. In order to utilize fully the vector capabilities of the architecture and to mimimize memory latency through software prefetching, we wrote a simple code-generator. A notable outcome of our work is an infrastructure which could be re-targeted to other vector formats. In particular, we sustained excellent performance on SNB-EPs ($\approx$ 120 GFLOPS single precision in the Dslash) simply by re-targetting the code-generator to emit AVX intrinsics, and re-tuning our blocking factors. This demonstrates a performance portability aspect of our infrastructure.

Our future work will include improving our blocking strategy, further optimization of our code for multiple nodes, implementing other formulations of LQCD, and investigating the potential of a hybrid code using both the host CPU and the Xeon Phi(s).

Acknowledgements. We thank Dr Steven Gottlieb for careful reading of this manuscript and helpful discussion. Partial support for this work was provided through the Scientific Discovery through Advanced Computing (SciDAC) program funded by U.S. Department of Energy, Office of Science, Offices of Advanced Scientific Computing Research, Nuclear Physics and High Energy Physics. We gratefully acknowledge the use of computing facilities of the US National Computational Facility for Lattice Gauge Theory (12k and 12m clusters at Jefferson Lab) and within Intel Corporation (Endeavor cluster). Notice: Authored by Jefferson Science Associates, LLC under U.S. DOE Contract No. DE-AC05-06OR23177. The U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce this manuscript for U.S. Government purposes.

References

1. Hestenes, M.R., Stiefel, E.: Methods of Conjugate Gradients for Solving Linear Systems. Journal of Research of the National Bureau of Standards 49(6), 409–436 (1952)
2. Creutz, M.: Quarks, Gluons and Lattices. Cambridge Monographs on Mathematical Physics, 169 p. Univ. Pr., Cambridge (1983)
3. Wilson, K.G.: Quarks and Strings on a Lattice. In: Zichichi, A. (ed.) New Phenomena in Subnuclear Physics, p. 69. Plenum Press, New York (1975)
4. van der Vorst, H.A.: Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems. SIAM Journal on Scientific and Statistical Computing 13(2), 631–644 (1992)
5. Smelyanskiy, M., Vaidyanathan, K., Choi, J., Joó, B., Chhugani, J., Clark, M.A., Dubey, P.: High-performance lattice QCD for multi-core based parallel systems using a cache-friendly hybrid threaded-MPI approach. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2011, pp. 69:1–69:11 (2011)
6. Clark, M.A., Babich, R., Barros, K., Brower, R.C., Rebbi, C.: Solving Lattice QCD systems of equations using mixed precision solvers on GPUs. Comput. Phys. Commun. 181, 1517–1528 (2010)

7. OpenMP Architecture Review Board: OpenMP Application Program Interface (2011)
8. Nguyen, A.D., Satish, N., Chhugani, J., Kim, C., Dubey, P.: 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. In: SC, pp. 1–13 (2010)
9. Babich, R., Clark, M.A., Joó, B.: Parallelizing the QUDA Library for Multi-GPU Calculations in Lattice Quantum Chromodynamics. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2010, pp. 1–11 (2010)
10. Boyle, P.A.: The BlueGene/Q supercomputer. PoS LATTICE 2012, 020 (2012)
11. MPI: A Message-Passing Interface Standard (March 1994)
12. Joó, B.: SciDAC-2 software infrastructure for lattice QCD. Journal of Physics: Conference Series 78(1), 012034 (2007)
13. Pakin, S., Lang, M., Kerbyson, D.J.: The reverse-acceleration model for programming petascale hybrid systems. IBM Journal of Research and Development 53(5), 8:1–8:15 (2009)
14. Heinecke, A., et al.: Design and Implementation of the Linpack Benchmark for Single and Multi-Node Systems Based on Intel(R) Xeon Phi(TM) Coprocessor. In: Proceedings of IPDPS Conference (2013)
15. Strzodka, R., Göddeke, D.: Pipelined mixed precision algorithms on FPGAs for fast and accurate PDE solvers from low precision components. In: IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2006), pp. 259–268 (April 2006)
16. Doi, J.: Peta-scale lattice quantum chromodynamics on a blue gene/Q supercomputer. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, pp. 1–45. IEEE Computer Society Press, Los Alamitos (2012)
17. Alexandru, A., Lujan, M., Pelissier, C., Gamari, B., Lee, F.X.: Efficient implementation of the overlap operator on multi-GPUs (2011)
18. Kowalski, A., Shen, X.: Implementing the Dslash Operator in OpenCL. College of William and Mary Technical Report (2010)
19. Bach, M., Lindenstruth, V., Philipsen, O., Pinke, C.: Lattice QCD based on OpenCL (2012)
20. Clark, M.A., Babich, R.: High-efficiency lattice QCD computations on the fermi architecture. In: Innovative Parallel Computing (InPar), pp. 1–9 (May 2012)
21. Chen, D., et al.: QCDSP machines: design, performance and cost. In: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (CDROM), Supercomputing 1998, pp. 1–6. IEEE Computer Society, Washington, DC (1998)
22. Vranas, P., et al.: The BlueGene/L supercomputer and quantum ChromoDynamics. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, SC 2006. ACM, New York (2006)
23. Boyle, P.A.: The BAGEL assembler generation library. Computer Physics Communications 180(12), 2739–2748 (2009) 40 YEARS OF CPC: A celebratory issue focused on quality software for high performance, grid and novel computing architectures
24. Pochinsky, A.: Writing efficient QCD code made simpler: QA(0). PoS LATTICE 2008, 040 (2008)
25. Chen, J., Watson, W., Mao, W.: GMH: A Message Passing Toolkit for GPU Clusters. In: 2010 IEEE 16th International Conference on Parallel and Distributed Systems (ICPADS), pp. 35–42 (December 2010)

Towards Addressing CPU-Intensive Seismological Applications in Europe*

Michele Carpené[1,**,***], Iraklis A. Klampanos[2,***], Siew Hoon Leong[3], Emanuele Casarotti[4], Peter Danecek[4], Graziella Ferini[1], André Gemünd[5], Amrey Krause[6], Lion Krischer[7], Federica Magnoni[4], Marek Simon[7], Alessandro Spinuso[8], Luca Trani[8], Malcolm Atkinson[2], Giovanni Erbacci[1], Anton Frank[3], Heiner Igel[7], Andreas Rietbrock[10], Horst Schwichtenberg[5], and Jean-Pierre Vilotte[9]

[1] CINECA, Bologna, Italy
m.carpen@cineca.it
[2] University of Edinburgh, School of Informatics, UK
[3] Leibniz Supercomputing Centre (LRZ), Garching, Germany
[4] Istituto Nazionale di Geofisica e Vulcanologia (INGV), Rome, Italy
[5] Fraunhofer Institute for Algorithms and Scientific Computing SCAI, Germany
[6] University of Edinburgh, Edinburgh Parallel Computing Centre (EPCC), UK
[7] Ludwig-Maximilianis-University, Department of Earth and Environmental Sciences, Germany
[8] The Royal Netherlands Meteorological Institute (KNMI), Netherlands
[9] Institut de Physique du Globe de Paris (IPGP), France
[10] University of Liverpool, Department of Earth, Ocean and Ecological Sciences, UK

Abstract. Advanced application environments for seismic analysis help geoscientists to execute complex simulations to predict the behaviour of a geophysical system and potential surface observations. At the same time data collected from seismic stations must be processed comparing recorded signals with predictions. The EU-funded project VERCE (`http://verce.eu/`) aims to enable specific seismological use-cases and, on the basis of requirements elicited from the seismology community, provide a service-oriented infrastructure to deal with such challenges. In this paper we present VERCE's architecture, in particular relating to forward and inverse modelling of Earth models and how the, largely file-based, HPC model can be combined with data streaming operations to enhance the scalability of experiments. We posit that the integration of services and HPC resources in an open, collaborative environment is an essential medium for the advancement of sciences of critical importance, such as seismology.

1 Introduction

Today's advanced seismology research aims at analysing the Earth's structure, composition and properties, largely by building geophysical models able to simulate earthquake

* VERCE is funded by the 7th Framework Programme of the European Commission, grant agreement number 283543.
** Corresponding author.
*** M. Carpené and I. A. Klampanos contributed equally to this paper.

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 55–66, 2013.

dynamics, from rupture to seismic radiation. Such research is essential for a multitude of applications, for instance to predict the consequences of seismic phenomena. The stakeholders for this kind of research activities are mainly government agencies interested in the management of national security aspects.

Seismology communities are typically focused on the following important topics: first, to obtain as much information as possible about seismic events, collecting data produced by seismic stations; second to build consistent geophysical wave propagation models derived from such data. The latter allows us to understand both Earth's surface and mantle characteristics, as well as to generate realistic mathematical models that are useful in estimating the effects of future seismic events. In order for this to be realised, such geophysical models are typically compared against real measurements obtained from seismic stations, in an iterative process. Such scientific computations involve solving complex differential equations over potentially large sets of data, and are therefore intractable, unless we rely on advanced, high-performance facilities, thus creating the conditions for the scalability and sustainability of experimental runs.

To address the data-access issue, end-users must be provided with an adequate data infrastructure, able to guarantee continuous availability of storage resources as well as data management tools and metadata support. Such infrastructure must also ensure that data access is effortless and transparent. EU funded projects such as EUDAT[1] are currently addressing some of these issues, exploring options and technologies within the context of a collaborative, data-oriented network[1].

As a major contributor to the e-science environment of the European Plate Observing System EPOS[2], and in collaboration with other initiatives, such as EUDAT, VERCE provides a service-oriented architecture delivering workflow tools and software services, bringing HPC and HTC/data-intensive facilities and related scientific archives under a single umbrella. VERCE is also actively following the progress of projects like MAPPER[3] and DRIHMS[4], which deal with multi-scale applications (such as hydrometeorology) on European infrastructures, *e.g.* PRACE[5] and EGI[6]. Although the scientific problems differ, the infrastructural challenges faced by these projects are similar to VERCE.

Focusing on the integration of the VERCE platform with HPC facilities, in this paper we: (1) Present the VERCE project, its goals and the challenges it is designed to address for the seismology community; (2) Present the proposed architecture, discussing core design decisions mainly related to HPC integration and (3) Discuss the software packages VERCE employs to meet its targets, including authentication and authorisation (AA), mechanisms for transparent job submission, integrating scientific solvers and merging streaming and file-based computational models.

In the next section we present the workflow platform of choice and the scientific use-cases that drive this ongoing effort, focusing on HPC-related scientific requirements.

[1] http://eudat.eu
[2] http://www.epos-eu.org
[3] http://www.mapper-project.eu
[4] http://www.drihms.eu
[5] http://www.prace-ri.eu
[6] http://www.egi.eu

In Section 3 we present the VERCE architecture, its core modules and components. In Section 4 we discuss technologies involved and how these are orchestrated towards addressing seismology computational challenges. Section 5 concludes this paper, briefly discussing ongoing and future work.

2 Background and Rationale

The work of the modern scientist involves managing and working with large datasets, typically distributed at remote sites, and accessible via different transfer protocols, security mechanisms, etc. Performing experiments and scientific analysis with these data may generate more data, which in turn have to be managed, further analysed and frequently be made available and shared within scientific communities, or more broadly with the world. At the same time, scientists have access to increasingly powerful computing facilities, *e.g.* PRACE and EGI, often away from their workplace, designed to address different needs and computing requirements. In the majority of cases, the laborious tasks of transferring and managing data and scientific codes is being orchestrated manually by the scientist. We posit that the current state of affairs is hardly manageable and, more importantly, it hinders scientists from making full use of the data and computing facilities they have at their disposal for scientific discovery.

Focusing on seismology, VERCE aims to provide a computation platform to the modern scientist, enabling them to make use of data as well as computing resources in a unified, efficient and tractable way. Further, in order to enhance the reproducibility of the exploration of models and the analysis of observational and simulation data, we require improvements in the automatic capture of relevant provenance information. This provenance information will also be used to improve processing methods, including making them more robust and efficient.

2.1 Dispel Scientific Workflows

VERCE addresses the above requirements by integrating HPC, data-intensive and storage resources through a workflow-enabled software platform built around the Dispel language. Dispel is a typed, streaming workflow specification language, one of the major achievements of the ADMIRE project[7][2,3].

At the core of Dispel modelling are logical units of computation referred to as *processing elements* (PEs). PEs can be organised in packages according to their functionality and can be shared and made available throughout the VERCE platform. PEs can be *abstract* or *concrete* (*i.e.* carrying associated implementations in conventional programming languages) as well as they can be combined in *composite* PEs to express arbitrarily complex functions. Workflows in Dispel are described by defining data flows (streams) passing through PEs. Workflows are logical descriptions of the computation to be carried out and are compiled, optimised and enacted by appropriate Dispel processing services, also referred to as Dispel gateways.

Using such a workflow language potentially allows data-intensive engineers and scientists to express scientific workflows at an abstraction layer closer to their cognitive

[7] http://www.admire-project.eu

level, thus focusing on the science rather than on complex technical issues. The VERCE architecture comprises a number of distributed Dispel gateways (*i.e.* components able to parse and enact Dispel workflows), each of which dealing with a number of known local resources. Dispel gateways are connected in a network of peers, orchestrating the deployment and execution of workflows in an efficient and tractable way.

VERCE innovates by combining the streaming model of Dispel with the file-based model of modern HPC in order to provide seamless execution of seismology-related workflows. In Section 4 we discuss some of the technologies and middleware integrated in VERCE.

Related Workflow Systems. A number of workflow systems have been proposed and are currently actively used, such as Pegasus[8], Taverna[9] and Kepler[10]. Such workflow systems typically offer complete solutions for workflow processing, data and resource management and monitoring, targeted at specific scientific communities. They come with graphical workflow editors and science gateways. A review of workflow management systems is outside the scope of this paper, however the interested reader can find more information in [4, Chapter 2]. On the other hand, Dispel is a fine-grained workflow specification language able to describe arbitrary workflows. It is currently supported by OGSA-DAI, which acts as its data and execution engine, but it is also translatable to other underlying execution engines and workflow systems, a direction we aim to explore within the lifetime of VERCE. Dispel is VERCE's workflow language of choice because of its stream-based design, thus providing maximum flexibility for addressing the requirements of the seismological community.

2.2 Use Cases

Based on computing requirements, on the size and form of scientific data, consumed and produced, and on current trends in seismology research, the development of the VERCE platform is driven by the following two broad use-cases:

Forward Modelling and Inversion. Many workflows in seismology are centred around the problem of simulating 3D wave propagation for earthquakes in Earth models. The forward modelling and inversion use-case entails a data-fitting procedure of seismic observations by calculating complete 3D wavefields, comparing observations with synthetic seismograms, and iteratively updating Earth models through adjoin techniques.

From a computational perspective, this use-case is characterised by: (1) the generation of synthetic seismograms as well as the inversion process use solvers (software applications) that are computationally intensive and are thus most efficient when run on HPC resources, *e.g.* PRACE. (2) Input data-sizes are typically smaller than in the cross-correlation use-case, however, depending on the experiment, output data can range from

[8] http://pegasus.isi.edu
[9] http://www.taverna.org.uk
[10] http://kepler-project.org

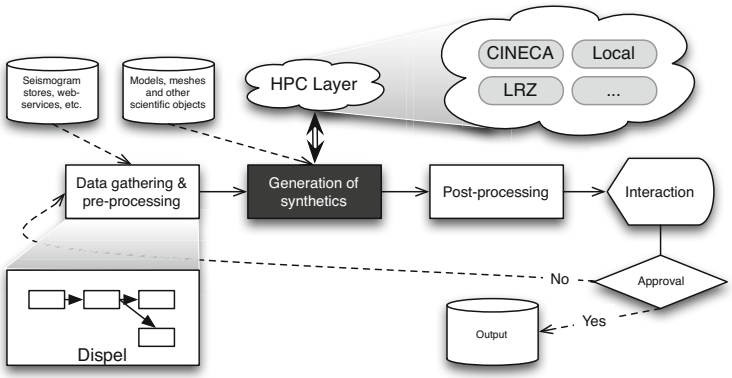


Fig. 1. An overview of the Dispel workflow implemented in VERCE for the computation of synthetic seismograms. The processing element responsible for HPC integration is emphasised.

tens of GBs to, potentially, tens of TBs, thus requiring efficient input/output (I/O) support. (3) Scientists typically need to interact during the execution of the workflow, when the workflow would need to seemingly pause.

The forward modelling and inversion use-case is inherently HPC-centric, not least because of current scientific practices. The technological requirements for addressing it form the basis of the design presented in this paper.

Ambient Noise Cross-Correlation. The calculation of cross-correlation of ambient noise, which is observed at a number of stations over arbitrary periods of time, constitutes a second VERCE use-case. The computational requirements of this use-case can be seen as primarily being *data-intensive*, due to the challenges in gathering and managing multiple, potentially large, data traces as well as due to the desired integration of the activity to local computational resources – therefore allowing for fine-grained tweaking of the methods involved. This use-case is introduced for completeness and is not within the scope of this paper.

In the remaining sections we focus on the *forward modelling and inversion* use-case, and in particular on technology integration that allows for triggering component execution on arbitrary and remote HPC facilities from within Dispel streaming workflows.

2.3 Forward Modelling and Misfit Calculation

Figure 1 depicts an overview of the forward modelling and inversion use-case. This part is representative of the complete use-case in that it combines data-intensive elements (in VERCE expressed in Dispel) with HPC computation. Dispel typically executes on locally administered VERCE resources using a streaming model, *i.e.* without generating files but rather pushing results along queues onto subsequent processing elements[11]. Clearly, the presence of a job submission processing element implies the

[11] In this context we use the term "processing element" loosely – each such processing element may consist of a number of actual Dispel processing elements.

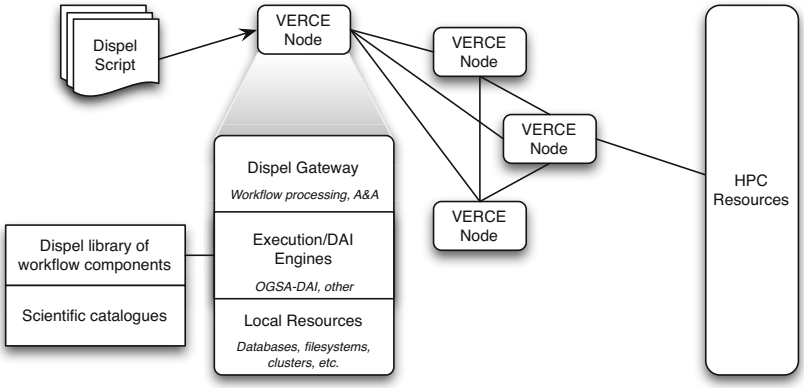


Fig. 2. Architectural overview of the VERCE platform

de-serialisation of the stream into files. These files will form the input files for the processing to take place at the HPC site. After the synthetics generation process has concluded on the remote HPC site, the submission processing element will be responsible for serialising output files and continuing further, in streaming mode.

3 Architecture Overview

The forward modelling and inversion use-case represents a complex workflow that requires occasional human interaction based on which Earth models are improved. Many of the steps are repetitive and can be automated. Pre-processing and post-processing steps can take advantage of institutional resources of scientists or HTC resources, EGI, in the case of big dataset. HPC resources from PRACE can be leveraged upon for computation-intensive steps. The VERCE architecture (Figure 3) is designed so that it achieves transparent coordination of all middleware components and applications offered by existing e-Infrastructures and required by users. Exceeding technical details are hidden from end-users. A science portal will be provided to handle complex authentication and authorisation issues and varying working environments on different e-Infrastructures – currently this is work in progress. In overview, Dispel gateways deal with parsing Dispel workflows and expanding them into actual, enactable graphs. They further orchestrate the execution of the workflow by forwarding parts of these graphs to appropriate resources for deployment and execution. Dispel gateways are also responsible for resolving and managing data and other resources required by the workflow specifications. The management of locally known resources is achieved through services such as OGSA-DAI, however the resolution and orchestration of foreign resources can take place through negotiation between Dispel gateways.

Execution of simulation code on HPC machines is coordinated by the Dispel gateway, which submits jobs using Globus (to LRZ SuperMUC machine) and UNICORE (to CINECA PLX), through the JSAGA API[12]. Following submission, the HPC application

[12] `http://grid.in2p3.fr/jsaga`

codes (*e.g.* Specfem3D and SeisSol) are executed and output files are retrieved and serialised before they are tunnelled back to the Dispel workflow for further processing. It is important to highlight that these files, which mark the transitional stream/file points in the workflow, will need to be stored (before serialisation) as close as possible to the location Dispel processing takes place. Finally, when results of computation are made available, these can be converted to files again and stored into appropriate VERCE data archives, from where they can be accessed and pulled back for future computations, if desired.

Before job execution can commence, input files, appropriate to the solver used, are generated by the gateway, starting from a set of meshes and Earth-model properties stored in the VERCE repository. The gateway, using appropriate processing elements according to the solver of choice, assembles and forwards only input files and commands for job execution to UNICORE and Globus, while the HPC application codes are expected to have already been staged in the HPC.

Results produced from simulations can be considered as new datasets to be used by other modelling applications or data analysis tools, allowing for further experimentation and/or scientific replication and validation, as required.

3.1 Abstraction through the Use of Processing Elements

The Dispel gateway recognises processing elements (PEs) previously registered with the system. Processing elements can be viewed as atomic, logical units for the execution of individual tasks. For this discussion we only consider three of the most important components for HPC integration (Figure 1): (1) Part of the first processing unit's (pre-processing) work is to produce input files for the simulation, starting with accessing appropriate Earth models and meshes in relevant repositories; (2) The second, core HPC PE (also analysed in Figure 3) is designed to submit jobs to external HPC resources (*e.g.* to PLX), passing generated input files as arguments; (3) The third, part of the workflow performs post-processing as well as stage-in and stage-out operations, retrieving output files and optionally storing them to VERCE's archive, before proceeding with additional stream-based computations.

It is important to note that the components of Figure 1 are conceptual entities and can be implemented and enacted in different ways.

Figure 3 shows the interaction between the core HPC PE and the job submission middleware (UNICORE and Globus Toolkit) employed in the first VERCE prototype. There are also other PEs, forming distributed programming libraries, each dealing with a specific type of stream analysis task; however their compilation and management are not within the scope of this paper.

3.2 Data Management

While offering long-term persistency and archiving capabilities is not within the scope of VERCE – for such functionality we collaborate and investigate how best to integrate our platform with other projects and initiatives, such as EUDAT and NERIES/EIDA[13],

[13] `http://www.neries-eu.org`

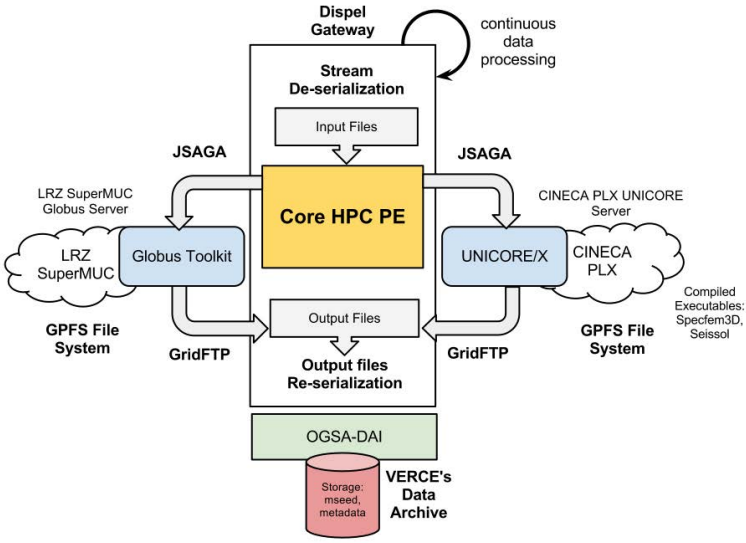


Fig. 3. Schema of interaction between the Core HPC PE and middleware components

the federated data archive of the EPOS seismology community – mid-term persistency and managing metadata pertaining to their scientific characteristics is a clear requirement. The solution currently under consideration within VERCE is a combination of iRODS[14], a distributed file management system, and a metadata store and management service based on MonetDB[15].

According to this design each VERCE scientific site maintains an iRODS store located within its internal network as well as an installation of the MonetDB metadata service. The VERCE iRODS sites can be interconnected and a number of replication and other policies could be implemented in the near future through iRODS rules and micro-services. By continuously monitoring its local iRODS store, the metadata service will be extracting seismological metadata (such as station, geographical location, channel, sampling rate, etc.) and indexing stored files against their corresponding metadata. Through the use of these technologies, the VERCE Dispel gateways will be able to lookup and stream data based on searches against scientific metadata at minimal cost to the researcher. At the same time, by replicating important scientific data across sites, we ensure redundancy as well as proximity to computational resources.

An Example Scenario. As a means of providing further understanding with respect to data handling and the size of transfers in the HPC use-case of VERCE, let us consider a hypothetical scenario. A researcher has the task to generate synthetics, based on

[14] `https://www.irods.org`
[15] `http://www.monetdb.org`

some Earth model. Consequently, the researcher has to calculate the misfit between the generated synthetics and real data, a pivotal step for model tuning and improvement.

For generating synthetics for, say, 50 seismological events, the corresponding input data (including the scientific model to be used, the event data themselves as well as other relevant information) is of the order of tens to a few hundreds of MBs. After a pre-processing phase, to take place on a local cluster in streaming fashion via Dispel, this data would need to be serialised and transferred to an HPC site for the main task of the generation of synthetics. The generated synthetics can range from hundreds of MBs, in the case of 2D wave-fields, to hundreds of GBs, in the case of high-resolution 4D (3D in time) wave-fields.

For the final step, the generated data will need to be transferred to a cluster, where the misfit calculation would take place. Supposing that the misfit will be calculated based on 50 seismic stations during a time-window of $1hr$, the additional data that would need to be streamed from the VERCE data-management service to the cluster would be of the order of a few tens of MBs. The outcome of this computation would be of a negligible size, of the order of KBs, and it would be stored on the local data iRODS file-system, allowing it to be replicated to other sites, not least the site from which the experiment was initiated.

4 Technology Description

In this section we present and discuss software and middleware components currently used in VERCE as well as our approach to interoperability and testing.

4.1 Applications

Each of VERCE's software packages has first been proposed by VERCE partners as a result of a prioritisation and analysis of the use-cases (introduced in Section 2.2), *i.e.* pilot applications, data-management tools and specific data sets, and then validated in a testing/monitoring campaign to spot and resolve possible issues relating to functionality and performance.

Proper installation procedures have been identified on each contributed computational resource (*i.e.* HPC, Grid or private/institutional) of the VERCE testbed and for each component. Compatibility and performance of the components have also been assessed as part of this procedure. Highly scalable simulation codes such as Specfem3D and SeisSol are found to be best compatible with HPC resources and have therefore been deployed both on CINECA and LRZ HPC facilities, while analysis tools such as ObsPy are better suited for institutional or privately-managed resources.

Specfem3D Specfem3D[16] is one of the solvers used in forward and inverse simulations to compute appropriate wavefields in two or three dimensions. In particular, the package simulates seismic wave propagation at the local or regional scale based on the spectral-element method (SEM). The SEM is a continuous Galerkin[5,6] technique which can

[16] `http://www.geodynamics.org/cig/software/specfem3d`

accurately handle very distorted mesh elements. The package, available under a GPL license, is widely used within the seismological community and has won several awards. All the software is written in Fortran2003 and employs parallel programming based on the Message Passing Interface (MPI).

SeisSol SeisSol[17] simulation software mimics seismic wave propagation in realistic media with complex geometry. The code is able to incorporate complex geological models and accounts for a variety of geophysical processes affecting seismic wave propagation, such as strong material heterogeneities, viscoelastic attenuation and anisotropy. The code is based on the ADER-Discontinuous Galerkin[7,8] method and has the unique property of achieving arbitrarily high approximation orders for the solution of the governing partial differential equations in space and time using three-dimensional tetrahedral meshes. It is written in Fortran and, similar to Specfem3D, uses parallel programming based on the Message Passing Interface (MPI).

ObsPy ObsPy[18] is an open-source Python framework for processing seismological data that includes typical processing routines, parsers for common file formats, etc.

4.2 The Middleware

While application environments, such as Specfem3D, are essential tools allowing to perform advanced simulations, they do not include binding components to manage data across resource groups, nor high-level APIs to perform job submission transparently across platforms. In order to execute complex workflows transparently it is necessary to introduce an additional software layer to our architecture. Behind interfaces and tools exposed to any HTC or HPC infrastructure lies the grid middleware: a set of software services that enable end-users to access distributed computing and data resources. Middleware tools, such as gLite, Globus Toolkit and UNICORE, allow geoscientists and other researchers to execute jobs and to collect output scientific results.

Here we list the main middleware tools employed in VERCE.

- Globus Toolkit[19] is a fundamental enabling technology for the "Grid" letting people share computing power, databases, and other tools securely online across corporate, institutional, and geographic boundaries without sacrificing local autonomy.
- gLite[20], born as part of the EGEE Project, provides a framework, as an integrated set of components, to build Grid applications and enable resource sharing.
- UNICORE6[21] (Uniform Interface to Computing Resources) offers a ready-to-run Job submission system, including client and server software. UNICORE makes distributed computing and data resources available in a seamless and secure way in intranets and well as over the Internet.

[17] http://www.geophysik.uni-muenchen.de/~kaeser/SeisSol
[18] https://github.com/obspy/obspy/wiki
[19] http://www.globus.org/toolkit/
[20] http://glite.cern.ch
[21] http://www.unicore.eu

– GridFTP[22] (Grid File Transfer Protocol) The Globus GridFTP server and client tools and libraries make up for a robust product suite designed to move large amounts of data faster, more securely, and more reliably than standard FTP.

4.3 Interoperability Testing

Regarding VERCE's HPC-related activities (addressing the forward simulation and inversion use-case), the first interoperability tests executed by submitting jobs to UNICORE and Globus using JSAGA APIs. In order to verify the middleware interoperability, job submission tests were performed using a test Dispel gateway residing on Edinburgh's Data-Intensive Machine[23], an official private VERCE resource. Sample jobs were submitted to PLX and SuperMUC. Input files were transferred from the gateway to HPC frontend nodes. The jobs were submitted to the UNICORE/Globus by using JSAGA. The detailed procedures were described in the Dispel scripts. Finally, output files were transferred back to the Dispel gateway via GridFTP before they were serialised for further stream-based processing.

Specifically in the case of PLX, after job submission the job identifier is returned from UNICORE to the gateway, in order for the gateway to be able to further retrieve output files by concatenating the identifier with the PLX file system's base path. Both PLX and SuperMUC are provided with access to the CINECA/LRZ GPFS, where job output files are stored. Once output files have been produced, the gateway finally moves them to its local (or locally managed) working directory using GridFTP.

Authentication is performed through X509 certificates both for Globus and UNICORE. This authentication process relies upon proxy certificates, since proxies are used by default in Globus. As a consequence, the UNICORE Gateway has been re-configured to support *proxy mode* so as to recognise the credentials.

Having designed and implemented the core functionality across the Dispel-execution engine-middleware stack, we are now implementing more realistic and larger, in data volume, test-cases for further testing.

5 Conclusions

At the end of the first year of the VERCE project, we are able to provide a concrete proposal for a novel HPC platform for seismology applications and a first prototype implementation. Thanks to VERCE's platform, geoscientists will be able to execute complex high-scale simulations of seismic events in a way which will improve our knowledge about the Earth's subsoil and predict effects and damages caused by earthquakes. This first prototype has been designed through extensive collaboration with a diverse team of researchers, geoscientists and technologists, coming together to define a service-oriented architecture able to offer tools for complex simulations execution as well as transparent access to resources.

The testbed deployed is according to proposed methodology and is expected to facilitate the modelling of complex geophysical systems through a continuous loop of

[22] http://www.globus.org/toolkit/data/gridftp

[23] http://research.nesc.ac.uk/files/report.eps

forward and inverse simulations. The methodology being addressed consists of a first phase, where the seismic wave propagation is simulated, starting from the geophysical model, and onto a second phase where misfit calculation and subsequent inversion leads to the improvement of models, iteratively. VERCE's infrastructure facilitates this closed-loop process by exploiting appropriate middleware tools. These tools allow for jobs execution on remote HPC resources, collection of results and performing continuous processing on data streams.

The design and development of the VERCE infrastructure is ongoing work. Having specified and implemented a first prototype to perform the core desired functionality, we are now actively working on additional required components. These include a scientific gateway/portal for end-users to interact and submit their experiments, processes and solutions to deliver a Dispel library of PEs for seismology tasks as well as integration with data-oriented initiatives and projects in order to achieve sustainability in managing data and meta-data.

References

1. Lecarpentier, D.: Towards a collaborative data infrastructure for science. iSGTW, International Science Grid This Week (2012)
2. Martin, P., Yaikhom, G.: Definition of the DISPEL Language. In: Atkinson, M., Baxter, R., Brezany, P., Corcho, O., Galea, M., van Hemert, J., Parsons, M., Snelling, D. (eds.) The DATA BONANZA: Improving Knowledge Discovery for Science, Engineering and Business, pp. 203–236. John Wiley & Sons Ltd. (April 2013)
3. Atkinson, M., Galea, M., Liew, C.S., Martin, P.: ADMIRE D2.9 — final report on the admire architecture, with an assessment and proposals for its development (2011), http://admire-project.eu
4. Liew, C.S.: Optimisation of the Enactment of Fine-Grained Distributed Data-Intensive Workflows. PhD thesis, School of Informatics, University of Edinburgh (2012)
5. Tromp, J., Liu, D.K., Spectral-element, Q.: adjoint methods in seismology. Communications in Computational Physics 3(1-32) (2008)
6. Peter, D., Komatitsch, D., Luo, Y., Martin, R., Goff, N.L., Casarotti, E., Loher, P.L., Magnoni, F., Liu, Q., Blitz, C., Nissen-Meyer, T., Basini, P., Tromp, J.: Forward and adjoint simulations of seismic wave propagation on fully unstructured hexahedral meshes. Geophys. J. Int. 186(2), 721–739 (2011)
7. Cockburn, B., Karniadakis, G., Shu, C.: Discontinuous galerkin methods, theory, computation and applications. LNCSE, vol. 11. Springer, Berlin (2000)
8. De la Puente, J., Käser, M., Dumbser, M., Igel, H.: An arbitrary high a order discontinuous galerkin method for elastic waves on unstructured meshes. IV. Anisotropy. Geophys. J. Int. 169(3), 1210–1228 (2007)

Leading Edge Hybrid Multi-GPU Algorithms for Generalized Eigenproblems in Electronic Structure Calculations*

Azzam Haidar[1], Raffaele Solcà[4], Mark Gates[1], Stanimire Tomov[1], Thomas Schulthess[4,5], and Jack Dongarra[1,2,3]

[1] University of Tennessee Knoxville
[2] Oak Ridge National Laboratory
[3] University of Manchester
[4] Institut for Theoretical Physics, ETH Zurich
[5] Swiss National Supercomputer Center

Abstract. Today's high computational demands from engineering fields and complex hardware development make it necessary to develop and optimize new algorithms toward achieving high performance and good scalability on the next generation of computers. The enormous gap between the high-performance capabilities of GPUs and the slow interconnect between them has made the development of numerical software that is scalable across multiple GPUs extremely challenging. We describe and analyze a successful methodology to address the challenges—starting from our algorithm design, kernel optimization and tuning, to our programming model—in the development of a scalable high-performance generalized eigenvalue solver in the context of electronic structure calculations in materials science applications. We developed a set of leading edge dense linear algebra algorithms, as part of a generalized eigensolver, featuring fine grained memory aware kernels, a task based approach and hybrid execution/scheduling. The goal of the new design is to increase the computational intensity of the major compute kernels and to reduce synchronization and data transfers between GPUs. We report the performance impact on the generalized eigensolver when different fractions of eigenvectors are needed. The algorithm described provides an enormous performance boost compared to current GPU-based solutions, and performance comparable to state-of-the-art distributed solutions, using a single node with multiple GPUs.

1 Introduction

In the context of electronic structure problems in material science and chemistry, the solution of the generalized Hermitian-definite eigenvalue problem is the most expensive task, dominating the entire computation [4, 20, 26]. In parallel electronic structure codes, many independent eigenvalue problems must be

* The authors would like to thank the National Science Foundation, the Department of Energy, NVIDIA, and MathWorks for supporting this research effort.

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 67–80, 2013.

solved, allowing each problem to be solved independently on a different node. In this work we are thus interested in dense eigensolvers, and in particular, for generalized Hermitian-definite problems of the form

$$Ax = \lambda Bx, \tag{1}$$

where A is a dense Hermitian matrix and B is Hermitian positive definite. Solving (1) requires the development of a number of routines. First, the matrix B is decomposed using a Cholesky factorization into $B = LL^H$, where ^{H} denotes conjugate-transpose. The resulting L factors are used to transform (1) to a standard Hermitian eigenproblem $\tilde{A}z = \lambda z$, where $\tilde{A} = L^{-1}AL^{-H}$. After solving the standard Hermitian eigenproblem, the eigenvectors X of the generalized problem (1) are then computed by backsolving with the Cholesky factor, $X = L^{-H}Z$. To solve the standard Hermitian (symmetric) eigenproblem of the form $\tilde{A}z = \lambda z$, finding its eigenvalues Λ and eigenvectors Z such that $\tilde{A} = Z\Lambda Z^H$, the standard strategy follows three steps [1,12,24]. First, reduce the matrix to a tridiagonal matrix T using an orthogonal transformation Q such that $\tilde{A} = QTQ^H$ (called the "reduction phase"). Note that when a two-sided orthogonal transformation is applied to generate T, the eigenvalues of the tridiagonal matrix are the same as those of the original matrix. Second, compute eigenpairs (Λ, E) of the tridiagonal matrix (called the "solution phase"). Third, back transform eigenvectors of the tridiagonal matrix to eigenvectors of the original matrix, $Z = QE$ (called the "back transformation phase"). All of these steps are computationally expensive, so we will develop an efficient multi-GPU implementation of each step.

2 Related Work

Solving the generalized eigenvalue problem is an active research field. Recently many researchers have been interested in this area and have developed various strategies, with a number of software implementations. The robust and conventional software LAPACK [3] and ScaLAPACK [7] are for shared-memory and distributed-memory systems, respectively. Recent work on symmetric eigenvalue problems has concentrated on accelerating separate components of the solvers, and in particular, the reduction to tridiagonal form, which is the most time consuming phase, and also the eigensolver. A new type of algorithm that challenges the standard one-stage reduction algorithms has been introduced. The idea behind this new technique is to split the reduction phase into two or more stages, recasting expensive memory-bound operations that occur during the panel factorization into compute-bound operations. One of the first uses of a two-stage reduction occurred in the context of out-of-core solvers for generalized symmetric eigenvalue problems [13]. Then, a multi-stage method was used to reduce a matrix to tridiagonal, bidiagonal and Hessenberg forms [21]. Consequently, a framework called Successive Band Reduction (SBR) was developed [5,6]. A multi-stage approach has also been applied to the Hessenberg reduction [18,19]. Tile algorithms have also recently seen a rekindled interest when applied to the two-stage tridiagonal [15,23] and bidiagonal reductions [22]. Their first stage is

implemented using high performance kernels and asynchronous execution while the second stage is implemented based on cache-aware kernels and a task coalescing technique [15]. Recently, a distributed-memory eigensolver library called ELPA [4] was developed for electronic structure codes. ELPA is similar to ScaLAPACK and does not support GPUs. It includes one-stage and two-stage tridiagonalizations, the corresponding eigenvector transformation, and a modified divide and conquer routine that can compute the entire eigenspace or a portion of it. These approaches, in contrast to our own, are not for hybrid GPU-CPU systems.

With the emergence of high-bandwidth, high-performance GPUs, memory-bound and compute-bound operations can be accelerated by an order of magnitude or more. Tomov et al. [29, 30] presented a hybrid CPU-GPU implementation for the one stage reduction algorithms, which take advantage of the high-bandwidth of the GPU by offloading the expensive Level 2 BLAS operations to the GPU. Dong et al. [9] extended this to multi-GPUs. Haidar et al. [16] developed a two-stage approach for multicore and a single GPU. The main thrust of the work presented here is the extension of this two-stage approach to multi-GPUs.

3 Main Contributions

Besides the software development efforts that we investigate to accomplish an efficient implementation, we highlight three main contributions related to the algorithm's design:

- **Fine grained memory aware and computationally intense tasks.** Our approach to efficient hardware use and parallelism relies on splitting the computation into tasks that either increase computational intensity or reduce data movement. Two main issues should be taken into consideration here. First, the task splitting and determination of granularity is essential for obtaining high performance. Second, the data distribution among the CPUs and GPUs should also be taken into consideration to minimize communication and achieve good performance.
- **Hybrid multi CPU-GPU execution.** Along with the computation splitting, a hybrid multi CPU-GPU implementation combined with task scheduling is an indispensable ingredient for obtaining high performance algorithms. We map computational tasks to the strengths of heterogeneous hardware components and overlap computation on GPUs with computation on CPUs.
- **A hierarchical multi-GPU communication model,** which optimizes communication for multi-GPUs and can be applied in general, beyond the scope of the algorithms developed.

4 Hybrid Multi GPU-CPU Algorithm

In this section we describe our multi CPU-GPU algorithm, presenting a detailed study to explain how we achieve good performance while dealing with a heterogeneous system. To make our description fruitfully interesting and clear we will

also describe implementation issues and give performance results of each kernel's multi-GPU implementation. Before developing our kernels, we should pay attention to the communication schema that our algorithm will use, as communication is an important component that affects any multi-GPU implementation.

4.1 Hierarchical Communication Model

As GPUs are well known for their high-performance capabilities and the relatively slow interconnect between them, reducing communication or overlapping communication with computation is critical in order to maximize the time GPUs spend in compute-intensive kernels, and minimize the time spent only in communication. To address the increase in communication when a large number of GPUs are used together, we developed a hierarchical communication model. Each PCIe switch connecting GPUs is viewed as a node in a distributed system, with one GPU in each node assigned to be the master. Within each node, GPUs communicate locally in a "free GPU-GPU mode" between the master GPU and other GPUs; between two nodes, the master GPUs communicate together directly. This hierarchical communication model is easily adaptable to a distributed environment, where communication between master GPUs of different nodes should be done via the CPU using MPI.

4.2 Transformation from Generalized to Standard Eigenvalue

As described above, the transformation from a generalized to a standard eigenvalue problem consists, first, in performing the multi-GPU Cholesky factorization of B. We refer the reader to [31] for a detailed description of our multi-GPU Cholesky implementation. Then, the resulting factor L is used to compute $\tilde{A} = L^{-1}AL^{-H}$. This operation is equivalent to the xHEGST function of the LAPACK library. We split this operation into three phases: (1) partially compute a panel A_i (blue portion of Figure 1a), then (2) use it to update the trailing matrix $A_{i+1:n}$ (red and green portion) by a xHER2K, and finally (3) continue the computation of the panel A_i (blue portion). Our multi-GPU algorithm distributes the matrix A in a 1D column block cyclic distribution, thus each GPU owns many blocks of A.

To optimize the code, phase 3 is delayed to the end of the computation, since its final result is not needed by any of the subsequent steps $i+1, i+2, \ldots$, while the GPU that owns it also owns other blocks involved in the update phase 2, so computing phase 3 at step i may delay the computation. To further reduce the synchronization between steps, during the update phase 2, the GPU that owns the next panel, A_{i+1}, will prioritize it and update it first, perform its partial computation (phase 1), broadcast it to other GPUs, and then continue the update of phase 2. In this way, while GPUs are updating (phase 2) at step i, they will receive the next panel A_{i+1} required to perform the next update of step $i+1$. Finally, once all updates are done, the GPUs will compute the remaining phase 3 computation of their blocks independently, without requiring any further communication.

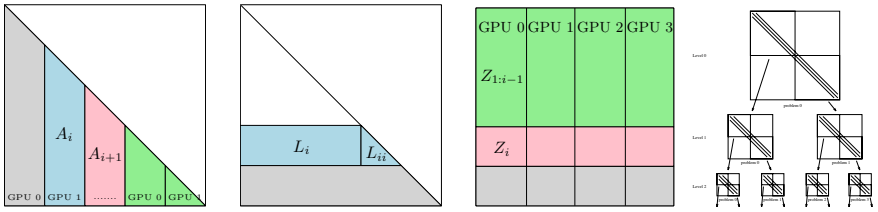


Fig. 1. (a) The multi-GPU xHEGST algorithm. (b) and (c) The i^{th} step of the blocked xTRSM. (d) the divide and conquer partitioning tree.

When the eigenvectors X are requested, the back transformation operation $X = L^{-H}Z$ needs to be performed. This operation can be performed in a parallel fashion where L^{-H} is applied independently to each vector of Z. Thus, if we split the matrix Z between the GPUs using a 1-D column block cyclic distribution, as depicted in Figure 1c, while the stored factor L is in block rows as shown in Figure 1b, then this operation can be performed independently by each GPU as follows. Each block row of the matrix L is broadcast over the GPUs (e.g., the blue block of Figure 1b). Once received, each GPU perform two operations. First, it computes $Z_i = L_{ii}^{-H} Z_i$ (red portion of Figure 1c), and then it updates $Z_{1:i-1} = Z_{1:i-1} - L_i^H Z_i$ (green portion). During these two operations the CPU will broadcast the next L_{i+1} to all the GPUs, so as to overlap the copy and the computation. The idea is to minimize the communication and overlap it as much as possible. We represent in Figure 2a the speedup obtained over the 1-GPU implementation by this multi-GPU implementation of this kernel when we vary the matrix size from 2000 to 40000, and also as we increase the number of GPUs from 2 to 8 GPUs. We see that this kernel asymptotically reaches near-perfect scaling.

4.3 Hybrid Multi CPU-GPU Tridiagonal Reduction

Due to its computational complexity and data access patterns, the tridiagonal reduction phase is the most challenging to develop, both algorithmically and implementation-wise. There are two algorithmic approaches — the standard one-stage approach from LAPACK [2], where block Householder transformations are used to directly reduce the dense matrix to tridiagonal form, and a newer two-stage (or more) approach, where block Householder transformations are used to first reduce the matrix to band form, and a second, bulge chasing stage is used to reduce the band matrix to tridiagonal [15]. The one-stage approach is well known to be memory bound as it relies on symmetric matrix-vector multiplications (50% of the flops).

The two-stage approach overcomes the memory-bound limitations of the one-stage. The reduction to band is done very efficiently using Level 3 BLAS. In particular, the dense matrix is first spread among the GPUs in a 1-D column block-cyclic distribution. The panel that must be factored at each iteration is

sent and factored on the CPU. The result is broadcast back to the GPUs and used for their local updates. A look ahead techniques is used — that is, the next panel to be factored is updated first and sent to the CPU for factorization while the GPUs complete the rest of their updates. This allows us to overlap CPU and GPU work. Moreover, all communication required during the update process is performed in an efficient manner using our hierarchical model.

Figure 2b shows the performance and scalability of the multi-GPU reduction to band. We see that this implementation of the two-stage approach provides good scalability. For two and three GPUs the scalability is perfect, while for four GPUs it approaches perfect scaling for a large matrix. For larger number of GPUs, it would require larger matrices to be able to reach the asymptotic perfect behavior. For example, if we split a matrix of size 20K over 8 GPUs, then each GPU will hold a small portion of size 20K×2.5K, which is not enough to perform intensive operations. The second stage that reduces the band matrix to tridiagonal is done on the multicore host using a multi-threaded bulge chasing implementation. Further detail can be found in Haidar et al. [14]. We observe that this multi-GPU implementation of the reduction to tridiagonal provides a jump in the performance compared to its one stage multi-GPU counterpart, being approximately four times faster than the one-stage approach. We will not detail more on this as the purpose of this paper is the overall generalized eigensolver problem.

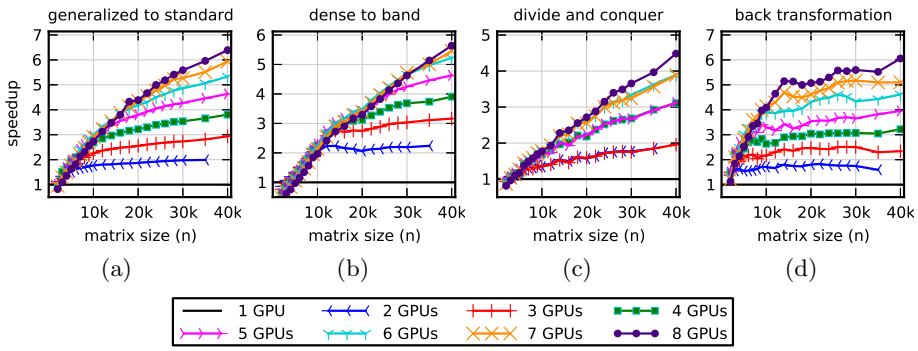


Fig. 2. Speedup relative to one GPU for major components of generalized eigensolver

4.4 Flexible Multi-GPU Divide and Conquer Algorithm

Introduced by Cuppen [8], the divide and conquer (D&C) algorithm computes the eigenvalues of the tridiagonal matrix T. Many serial and parallel Cuppen-based eigensolver implementations for shared and distributed memory have been proposed in the past [10, 11, 17, 25, 27, 28]. The overall D&C approach consists in splitting the problem into two subproblems (son nodes) representing a rank-one modification. Each of these subproblems is an independent problem without any data dependencies with the other subproblems. This process is repeated recursively, constructing a binary tree where the bottom nodes will have two

independent sons of small size considered as two *simple* eigenvalue problems. Then, on each parent node, merge the two subproblems (left and right son) which are defined by a rank-one modification of a diagonal matrix, and proceed to the next level in a bottom-up fashion.

To illustrate how our multi-GPU implementation is designed, let us describe it for two subproblems. Let the matrix T of size n be split into two subproblems, T_1 of size n_1 and T_2 of size $n_2 = n - n_1$, as described in (2). Let the eigensolution of those two sons be given by $T_1 = \tilde{E}_1 \tilde{\Lambda}_1 \tilde{E}_1^T$ and $T_2 = \tilde{E}_2 \tilde{\Lambda}_2 \tilde{E}_2^T$, where $(\tilde{\Lambda}_i, \tilde{E}_i)$, $i = 1, 2$ are the eigenvalues and eigenvectors pair of T_i. Assuming that $(\tilde{E}_0, \tilde{\Lambda}_0)$ are the eigenpairs solution of the system inside the bracket of (2), then $\Lambda = \tilde{\Lambda}_0$ and $E = \tilde{E}_i \tilde{E}_0$ are the eigenpairs of T.

$$\begin{aligned} T = \begin{pmatrix} T_1 & 0 \\ 0 & T_2 \end{pmatrix} + \rho \mathbf{v}\mathbf{v}^T &= \begin{pmatrix} \tilde{E}_1 & 0 \\ 0 & \tilde{E}_2 \end{pmatrix} \left\{ \begin{pmatrix} \tilde{\Lambda}_1 & 0 \\ 0 & \tilde{\Lambda}_2 \end{pmatrix} + \rho \mathbf{u}\mathbf{u}^T \right\} \begin{pmatrix} \tilde{E}_1 & 0 \\ 0 & \tilde{E}_2 \end{pmatrix}^T \\ &= \begin{pmatrix} \tilde{E}_1 & 0 \\ 0 & \tilde{E}_2 \end{pmatrix} \left(\tilde{E}_0 \tilde{\Lambda}_0 \tilde{E}_0{}^T \right) \begin{pmatrix} \tilde{E}_1 & 0 \\ 0 & \tilde{E}_2 \end{pmatrix}^T = E \Lambda E^T \end{aligned} \tag{2}$$

To find the eigensolution of each rank-one modified system M requires solving its secular equation. This is a memory bound process that requires only $\mathcal{O}(n^2)$ operations, so in our implementation we keep this computation on the CPU side, while the GPUs perform the multiplication of the intermediate eigenvector matrices $\tilde{E}_i$.

The independent parallelism generated by the D&C approach allows us to distribute each division over half of the GPUs recursively. For example, half of the GPUs will compute the upper part of the eigenvectors (involving E_1), and the rest the lower part (involving E_2). This imposes a constraint on the number of GPUs to be multiple of 2. The implementation could be generalized to deal with any number of GPUs, but based on the way the eigenvectors are generated we don't expect this would give more improvement. We plot in Figure 2c the speedup of this kernel for various matrix sizes when increasing the number of GPUs. When the computing intensive kernels are performed on the GPUs, it reduces the time to compute such expensive operations, and thus the time required to solve the memory bound secular equation becomes dominant, so we can expect that the performance of this kernel to have limited scalability. Nonetheless, the obtained scalability remains very attractive. We also modified the algorithm such that when only a portion of the eigenvectors are required, the multiplication is done with only this portion, reducing the total amount of computation.

4.5 Back Transformation

In this section, we discuss the application of the Householder reflectors generated from the two stages of the reduction to tridiagonal. The first stage reduces the original Hermitian matrix $\tilde{A}$ to a band matrix by applying a two-sided transformation to $\tilde{A}$ such that $\tilde{A} = Q_1 S Q_1^H$. Similarly, the second stage (bulge chasing) reduces the band matrix S to tridiagonal by applying the transformation from

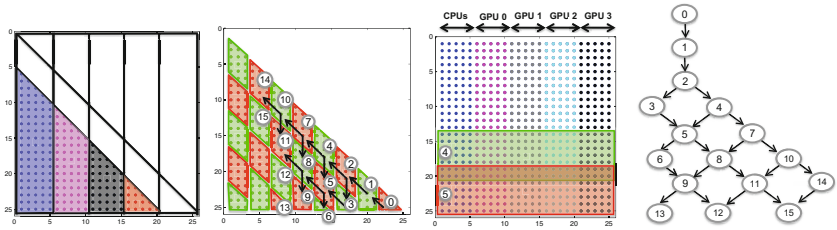


Fig. 3. (a) Tiling of V_1, (b) Blocking technique to apply V_2, (c) Distribution of the eigenvectors matrix that create independent fashion of applying Q_2 which increase locality per core, (d) Portion of the DAG showing the dependency of the V's of V_2

both the left and the right side to S such that $S = Q_2 T Q_2^H$. Thus, when the eigenvectors matrix Z of $\tilde{A}$ are requested, the eigenvectors matrix E resulting from the eigensolver needs to be updated from the left by the Householder reflectors generated during the reduction phase, according to

$$Z = Q_1 Q_2 E = (I - V_1 t_1 V_1^H)(I - V_2 t_2 V_2^H) E, \tag{3}$$

where (V_1, t_1) and (V_2, t_2) represent the Householder reflectors generated during the reduction stages one and two, respectively.

The application of the V_2 reflectors is not as simple as the application of V_1, and requires special attention. We represent the V_2 in Figure 3b. Note that these reflectors represent the annihilation of the band matrix, and thus each is of length nb, where nb is the bandwidth size. A naive implementation would take each reflector and apply it to the matrix E. Such an implementation is memory bound, relying on BLAS 2 operations and thus gives poor performance. However, if we want to group them to take advantage of the efficiency of BLAS 3 operations, we must pay attention to the overlap between them and that their application must follow the specific dependency order of the bulge chasing procedure in which they were created. Let us give an example that explain those issues. For sweep i (e.g., the column at position S(i,i):S(i,i+nb)), its annihilation creates a set of k Householder reflectors v_i^k, each of length nb represented in column i of the matrix V_2 depicted in Figure 3b. Similarly, the ones related to sweep $i+1$ are those presented in column $i+1$. They are shifted one element down compared to those of sweep i. After analyzing the dependencies of the bulge chasing procedure as explained by the example above, we notice that we can group the reflectors v_i^k from sweep i with those from sweep i+1, i+2,..., $i+l$ to apply them together using a blocked technique according to the diamond shape region as defined in Figure 3b. While each of those diamonds is considered as one block, their application needs to follow the dependency order. For example, applying the green block 4 and the red block 5 of the V_2's in Figure 3b modifies the green block row 4 and the red block row 5, respectively, of the eigenvector matrix E drawn in Figure 3c, where we can easily observe the overlapped region. According

to the chasing order, block 4 needs to be applied before block 5. We have drawn a sample of those dependencies by the arrows in Figure 3b. For clarity, we also represented them by the DAG in Figure 3d. A little effort studying the pattern of dependencies of this DAG leads us to the conclusion that designing an algorithm based on such schema provides a very limited number of parallel and pipelined tasks. Despite all of those constraints, a nice feature to create efficiency is that, if we design our parallelism based on the matrix E, where we split E by block column over the number of cores/GPUs as shown in Figure 3c, then we can apply each diamond block independently to each portion of E. Moreover, this way does not require any data communication between GPUs. The overlap between each application of V's as described above increase the cache reuse. For the CPUs, we also define the size of the block of E in a way to fit more than one region of it in the L2 cache level to increase locality. We implemented a new kernel to deal with these diamonds to increase the cache reuse and overlap the communication to the GPUs. These blocks are broadcast to the GPUs in a look ahead fashion, meaning that when GPUs are using the block V_2^m to update their portion of E, the CPU broadcasts the next V_2^{m+1} in a overlapped technique.

The application of V_1 to the resulting matrix from above, $G = (I - V_2 T_2 V_2^T)E$, can be done easily. First, there is no overlap between the different V_1's. Second, they can be blocked as shown in Figure 3a. Thus their application is computing intensive and involves efficient BLAS 3 kernels. Using the same parallelism design, the V_1 can be applied independently to each block column of Figure 3c, which are now the block of G and thus the distribution remains the same. This operation does not require any communication between GPUs or between the previous kernel (apply V_2) and the current kernel (apply V_1). Similarly, each block of V_1's is broadcast over the GPUs in a look ahead fashion. This implementation is independent and very suitable for parallel and heterogeneous implementation, especially when communication is expensive. Speedup results for matrices raising from 2000 to 40000 when varying the number of GPUs are presented in Figure 2d, showing a very good speedup is obtained.

5 Experimental Results

The eigensolver presented in this paper was tested on an experimental machine offering a dual-socket six-core Intel Xeon 5675 running at 3.07 GHz, with 48 GB of main system memory and 8 NVIDIA Fermi M2090 GPUs. We tested the distributed memory libraries on a tightly coupled computing cluster system, offering nodes based on the dual-socket six-core Intel Xeon 5650 processor architecture, running at 2.6 GHz, with 24 GB of main system memory per node.

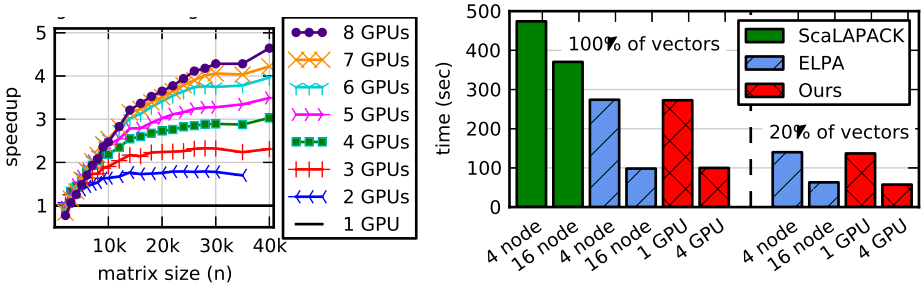
5.1 Accuracy Analysis

We mention that the only difference, numerically, between our algorithm and the LAPACK algorithm is that we use a two-stage algorithm for the reduction to tridiagonal. The reduction to tridiagonal relies on the Householder elimination,

which has been proved to be backward stable and accurate, and hence we expect the same accuracy as LAPACK. Our implementation of the divide and conquer is based on the main Cuppen algorithm and is exactly the same as the one implemented in the LAPACK library. All of our experiments obtained the same order of accuracy as that computed by the LAPACK solver, using the metrics described in the LAPACK Users' Guide [2].

5.2 Performance Scalability

We performed an extensive study with a large number of experimental tests to give the reader as much information as possible. We computed the eigenpairs of the generalized eigenvalue problem, varying the size of matrices from 2000 to 40000 and varying from 1 to 8 GPUs. Figure 4a shows the speedup obtained by our hybrid multi-GPU symmetric generalized eigenvalue solver as compared to its one GPU implementation. As expected, the speedup performance obtained has a similar trend to the ones presented above in Figure 4 for each kernel of the algorithm. Strong scalability is observed as, for a fixed matrix size, when we increase the number of GPUs the time should decrease linearly. The results show a very good scalability for our implementation on such an heterogeneous hybrid system with a huge computing intensive component (8 GPUs) connected to it. We can see that although some kernels of this algorithm are strictly multicore CPU implementations (the bulge chasing and the secular equation solver), our hybrid multi-GPU implementation provides a very attractive scalability. On four GPUs, our approach is able to run three times faster than on one GPU, which is considered to be very good scalability. On a larger number of GPUs, it requires large matrices to be able to see the asymptotic scaling behavior.



(a) Generalized eigensolver scalability (b) Performance comparison

Fig. 4. (a) Speedup relative to one GPU for generalized eigensolver. (b) Time comparison of various implementations to solve $n = 20000$ generalized eigenvalue problem, computing either 100% of eigenvectors (on left) or only 20% of eigenvectors (on right).

5.3 Performance Results

We compare the performance of our hybrid multi-GPU eigensolver against the optimized state-of-the-art numerical linear algebra library ScaLAPACK, and the ELPA library, using both the one stage and the two stage reduction. We ran tests for ELPA and ScaLAPACK for various problem sizes, while varying the number of processors from 24 to 240. The results reported here are the best results achieved. In our tests we found that both one stage and two stage ELPA implementations show good scalability compared to ScaLAPACK; the required time decreases three times when increasing the number of processors from 48 to 192. We also found that the two-stage approach of the ELPA solver was faster than its one-stage approach for any number of nodes and any percentage of the eigenvectors requested, and so we omit results of the one stage of ELPA from our graphs. Figure 4b shows the time needed for all the solvers to find the solution of the generalized eigenvalue problem for a matrix of size 20000. For ScaLAPACK and ELPA, we give results on four nodes, each dual-socket six-core (48 processors), and also on 16 nodes (192 processors), which have reasonable time compared to our multi-GPU solver. These distributed and GPU-based systems have comparable peak matrix-matrix multiply performance: 48 processors has 460 Gflop/s peak, compared to 500 Gflop/s using 12 cores plus one GPU, while 192 processors has 1.9 Tflop/s peak, compared to 1.6 Tflop/s using 12 cores plus four GPUs. Comparing to our approach, we can see that, to solve the generalized eigenvalue problem computing all (100%) of its eigenpairs, the time needed by ScaLAPACK on 192 processors and by the ELPA implementation on 48 processors is very close to the time needed by our hybrid multi-GPU solver running with only one GPU. Similarly, the time needed by the ELPA solver running on 192 processors is similar to the time needed by our multi-GPU solver running on only four Fermi GPUs. This behavior has also been observed for other matrices size, in particular, for a matrix of size 30000, the ELPA solver running on 192 cores requires 327 seconds to find all of its eigenpairs, while our hybrid multi-GPU solver running with four Fermi GPUs needs 314 seconds. These results show that with only a small numbers of devices, an efficient multi-GPU implementation can achieve as much speed as one of the best solvers on 192 processors.

As many applications need only a portion of the eigenvectors, we also present in Figure 4b the comparison with the ELPA solver when only 20% of the eigenvectors are needed. The trend shown here is again similar to the performance shown when all the eigenpairs are computed. For example, for matrices of size 20000 and 30000, the ELPA solver running on 192 processors requires 63 seconds and 218 seconds respectively, while our solver running on four GPUs requires 57 seconds and 174 seconds respectively. We note here that when a fraction of the eigenvectors are computed, both our approach and the two stage ELPA are significantly faster than the one stage approach implemented in either ELPA or ScaLAPACK.

5.4 Real Electronic Structure Application

We did preliminary experiments in the context of a real electronic structure application. We performed a ground state computation for the Eu_6C_{60} compound with the density functional method. The density functional calculations was done with a new prototype of a linearized augmented plane wave (LAPW) library []. During each iteration a generalized eigenvalue problem of size 25383 in double complex precision, requiring the 1335 eigenvectors with the lowest eigenvalues, has to be solved. We note that this latter consists of 67% of the total iteration time when solved on a distributed system and could be decreased down to around 20% when solved with our multi-GPU implementations meaning that the time per iteration could be speeded up 3 to 4 times.

The experiment has been done on a cluster of Intel Xeon E5-2670 (Sandy Bridge) 2.6 GHz processors. The time required to perform each iteration using the ScaLapack library is 787 and 263 seconds using 64 and 256 MPI processes, respectively. Of that, 525 and 175 seconds, respectively, are spent solving the generalized eigenvalue problem (67% of the time per iteration). Using only one node (consisting of 16 processors of the same type) with 4 Nvidia K20 GPUs, we were able to reduce the time of the generalized eigensolver to 145 seconds, which will provide a huge boost. The total number of iterations needed depends on the compound and on the mixer used, and usually it varies between 20 and 100 iterations.

6 Conclusions and Future Directions

We demonstrated that it is possible to develop efficient and scalable algorithms for heterogeneous systems with an enormous gap between their computing power and interconnection bandwidth. Our hybrid multi-CPU-GPU implementation demonstrated very promising results in terms of performance as well as in terms of scalability on heterogeneous architecture systems. It has been extensively tested using different matrix types and many parallel configuration against other well-known generalized symmetric eigenvalue solvers. The performance obtained is very encouraging. These results show the impact of our work on applications, especially the field of electronic structure computations where a large number of dense generalized eigenvalue problem need to be solved in the solution of Schrödinger equation, thus the choice of a suitable method is of great importance. We believe that these techniques will only increase in relevance for upcoming architectures. We plan to further study the implementation of multi-GPU algorithms in a distributed computing environment.

References

1. Aasen, J.O.: On the reduction of a symmetric matrix to tridiagonal form. BIT 11, 233–242 (1971)
2. Anderson, E., Bai, Z., Bischof, C., Blackford, L.S., Demmel, J.W., Dongarra, J.J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.: LAPACK Users' Guide. SIAM, Philadelphia (1992), `http://www.netlib.org/lapack/lug/`

3. Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Croz, J.D., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D.: LAPACK Users' Guide, 3rd edn. Society for Industrial and Applied Mathematics, Philadelphia (1999)
4. Auckenthaler, T., Blum, V., Bungartz, H.J., Huckle, T., Johanni, R., Krämer, L., Lang, B., Lederer, H., Willems, P.R.: Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations. Parallel Comput. 37(12), 783–794 (2011)
5. Bientinesi, P., Igual, F.D., Kressner, D., Quintana-Ortí, E.S.: Reduction to condensed forms for symmetric eigenvalue problems on multi-core architectures. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2009, Part I. LNCS, vol. 6067, pp. 387–395. Springer, Heidelberg (2010)
6. Bischof, C.H., Lang, B., Sun, X.: Algorithm 807: The SBR Toolbox—software for successive band reduction. ACM Transactions on Mathematical Software 26(4), 602–616 (2000)
7. Blackford, L.S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R.C.: ScaLAPACK Users' Guide. Society for Industrial and Applied Mathematics, Philadelphia (1997)
8. Cuppen, J.J.M.: A divide and conquer method for the symmetric eigenproblem. Numer. Math. 36, 177–195 (1981)
9. Dong, T., Dongarra, J., Schulthess, T., Solca, R., Tomov, S., Yamazaki, I.: Matrix-vector multiplication and tridiagonalization of a dense symmetric matrix on multiple GPUs and its application to symmetric eigenvalue problems. Parallel Comput. (July 2012) (submitted)
10. Dongarra, J.J., Sorensen, D.C.: A fully parallel algorithm for the symmetric eigenvalue problem. SIAM J. Sci. Statist. Comput. 8, s139–s154 (1987)
11. Gates, K., Arbenz, P.: Parallel divide and conquer algorithms for the symmetric tridiagonal eigenproblem (1994)
12. Golub, G.H., Loan, C.F.V.: Matrix Computations, 2nd edn. The Johns Hopkins University Press, Baltimore (1989)
13. Grimes, R.G., Simon, H.D.: Solution of large, dense symmetric generalized eigenvalue problems using secondary storage. ACM Transactions on Mathematical Software 14, 241–256 (1988)
14. Haidar, A., Gates, M., Tomov, S., Dongarra, J.: Toward a scalable multi-gpu eigensolver via compute-intensive kernels and efficient communication. In: ICS 2013: 27th International Conference on Supercomputing, Eugene, Oregon, USA, June 10-14 (submitted, 2013)
15. Haidar, A., Ltaief, H., Dongarra, J.: Parallel reduction to condensed forms for symmetric eigenvalue problems using aggregated fine-grained and memory-aware kernels. In: SC 2011: International Conference for High Performance Computing, Networking, Storage and Analysis, Seattle, WA, USA, November 12-18 (2011)
16. Haidar, A., Tomov, S., Dongarra, J., Solca, R., Schulthess, T.: A novel hybrid CPU-GPU generalized eigensolver for electronic structure calculations based on fine grained memory aware tasks. International Journal of High Performance Computing Applications (September 2012) (accepted)
17. Ipsen, L.C.F., Jessup, E.R.: Solving the symmetric tridiagonal eigenvalues problem on the hypercube. SIAM J. Sci. Stat. Comput. 11, 203–229 (1990)
18. Kågström, B., Kressner, D., Quintana-Orti, E., Quintana-Orti, G.: Blocked Algorithms for the Reduction to Hessenberg-Triangular Form Revisited. BIT Numerical Mathematics 48, 563–584 (2008)

19. Karlsson, L., Kågström, B.: Parallel two-stage reduction to Hessenberg form using dynamic scheduling on shared-memory architectures. Parallel Computing (2011), doi:10.1016/j.parco.2011.05.001
20. Kent, P.: Computational challenges of large-scale, long-time, first-principles molecular dynamics. Journal of Physics: Conference Series 125(1), 012058 (2008)
21. Lang, B.: Efficient eigenvalue and singular value computations on shared memory machines. Parallel Computing 25(7), 845–860 (1999)
22. Ltaief, H., Luszczek, P., Dongarra, J.: High Performance Bidiagonal Reduction using Tile Algorithms on Homogeneous Multicore Architectures. In: ACM TOMS (2011) (accepted)
23. Luszczek, P., Ltaief, H., Dongarra, J.: Two-stage tridiagonal reduction for dense symmetric matrices using tile algorithms on multicore architectures. In: IPDPS 2011: IEEE International Parallel and Distributed Processing Symposium, Anchorage, Alaska, USA, May 16-20 (2011)
24. Parlett, B.N.: The Symmetric Eigenvalue Problem. Prentice-Hall, Englewood Cliffs (1980)
25. Rutter, J., Rutter, J.D.: A serial implementation of cuppen's divide and conquer algorithm for the symmetric eigenvalue problem (1994)
26. Singh, D.J.: Planewaves, Pseudopotentials, and the LAPW Method. Kluwer, Boston (1994)
27. Sorensen, D.C., Tang, P.T.P.: On the orthogonality of eigenvectors computed by divide-and-conquer techniques. SIAM J. Numer. Anal. 28(6), 1752–1775 (1991)
28. Tisseur, F., Dongarra, J.: Parallelizing the divide and conquer algorithm for the symmetric tridiagonal eigenvalue problem on distributed memory architectures. SIAM J. SCI. Comput. 20, 2223–2236 (1998)
29. Tomov, S., Nath, R., Dongarra, J.: Accelerating the reduction to upper Hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing. Parallel Comput 36(12), 645–654 (2010)
30. Vomel, C., Tomov, S., Dongarra, J.: Divide and conquer on hybrid GPU-accelerated multicore systems. SIAM Journal on Scientific Computing 34(2), C70–C82 (2012)
31. Yamazaki, I., Tomov, S., Dongarra, J.: One-sided dense matrix factorizations on a multicore with multiple gpu accelerators. In: Proc. of ICCS 2012, Procedia CS, vol. 9, pp. 37–46 (2012)

Heterogeneous Programming and Optimization of Gyrokinetic Toroidal Code and Large-Scale Performance Test on TH-1A

Xiangfei Meng[1], Xiaoqian Zhu[2], Peng Wang[3], Yang Zhao[1], Xin Liu[2], Bao Zhang[4], Yong Xiao[5], Wenlu Zhang[6], and Zhihong Lin[7]

[1] National Supercomputer Center in Tianjin, Tianjin, China
{mengxf,zhaoyang}@nscc-tj.gov.cn
[2] School of Computer, National University of Defense Technology, Changsha, China
zhu_xiaoqian@sina.com, liuxin@nscc-tj.gov.cn
[3] NVIDIA, Beijing, China
penwang@nvidia.com
[4] Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an, China
123zhangbao456@163.com
[5] Institute for Fusion Theory and Simulation, Zhejiang University, Hangzhou, China
yxiao@zju.edu.cn
[6] CAS Key Laboratory of Plasma Physics, University of Science and Technology of China, Hefei, Anhui, China
wzhang@iphy.ac.cn
[7] Department of Physics and Astronomy, University of California, Irvine, California, USA
zhihongl@uci.edu

Abstract. In this work, we discuss the porting to the GPU platform of the latest production version of the Gyrokinetic Torodial Code (GTC), which is a petascale fusion simulation code using particle-in-cell method. New GPU parallel algorithms have been designed for the particle push and shift operations. The GPU version of the GTC code was benchmarked on up to 3072 nodes of the Tianhe-1A supercomputer, which shows about 2x–3x overall speedup comparing NVIDIA M2050 GPUs to Intel Xeon X5670 CPUs. Strong and weak scaling studies have been performed using actual production simulation parameters, providing insights into GTC's scalability and bottlenecks on large GPU supercomputers.

Keywords: particle-in-cell, hybrid programming, TH-1A.

1 Introduction

The global gyrokinetic toroidal code (GTC) [1] is a massively parallel particle-in-cell code for first-principles, integrated simulations of the burning plasma experiments such as the International Thermonuclear Experimental Reactor (ITER) [2], the crucial next step in the quest for the fusion energy. The GTC code has

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 81–96, 2013.

grown over the years from a single-developer code to a prominent code being developed by an international collaboration with many users and contributors from the magnetic fusion energy and high performance computing communities. GTC is the key production code for the multi-institutional U.S. Department of Energy (DOE) Scientific Discovery through Advanced Computing (SciDAC) project, Gyrokinetic Simulation of Energetic Particle Turbulence and Transport (GSEP) Center, and the National Special Research Program of China for ITER. GTC is currently maintained and developed by an international team of core developers who have the commit privilege, and receives contributions through the proxies of core developers from collaborators worldwide [3].

The GPU version of the GTC code reported in this paper, GTC-GPU, has the same physics capability and parallel scalability of the current production version of GTC originally written in Fortran. GTC continuously pushes the frontiers of both physics capabilities and high performance computing. At present, GTC is the only fusion code capable of integrated simulations of key physical processes that underlie the confinement properties of fusion plasmas, including micro-turbulence [4–7], energetic particles instabilities [8–11], magnetohydrodynamic modes [12], and radio-frequency heating and current drive.

GTC is the first fusion code to reach the teraflop in 2001 on the Seaborg computer at NERSC and the petaflop in 2008 on the jaguar computer at ORNL in production simulations, and to fully utilize the computing power provided by CPU and GPU heterogeneous architecture on Tianhe-1A.

There have been some other recent works on porting PIC (particle-in-cell) codes with simplified physics models to GPU for proof of principles. Decyk et al. [13] discussed a new 2D PIC code with data structures optimized for GPU. Burau et al. [14] discussed the PIConGPU code, a GPU PIC code developed for fast-response simulations of laser-plasma interaction. Stanchev et al. [15] focused on optimizing the particle-to-grid interpolation kernel. Rossinelli et al. [16] focused on optimizing the grid-to-particle interpolation kernel. In contrast, the GTC-GPU reported in this paper is a current production version and the benchmark simulations use actual physics simula-tion parameters [6,7].

The most relevant to the current work is the work of Madduri et al. [17,18], who discussed the porting of an earlier version of GTC to GPU. They concluded that GPU was slower than CPU for their version of GTC, which only include kinetic ions with adiabatic electrons. However, we use the latest version of GTC including new important features for a realistic turbulence simulation, such as kinetic electrons and general geometry. We also employ a set of realistic experiment parameters suitable for simulating plasma turbulence containing both kinetic ions and kinetic electrons [6,7]. As a result, our version contains more routines, especially the computing-intensive module for the electron physics. We also designed new GPU parallel algorithms for the PUSH and SHIFT operations, which are the two most dominant operations in our profiling. The GTC-GPU shows a 2x–3x overall speedup comparing NVIDIA M2050 GPUs to Intel Xeon X5670 CPUs on up to 3072 nodes of the Tianhe-1A supercomputer. Preliminary test run of the GTC-GPU on a small number of nodes of the Titan at ORNL shows a similar speedup.

2 Experimental Platform: Tianhe-1A

All the benchmark runs in the following sections were performed on the Tianhe-1A supercomputer. In this section, we introduce the Tianhe-1A supercomputer which is a hybrid massively parallel processing (MPP) system with CPUs and GPUs. Tianhe-1A is the host computer system of National Supercomputer Center in Tianjin (NSCC-TJ).

The hardware of Tianhe-1A consists of five components: computing system, service system, communication system, I/O storage system, monitoring and diagnostic system. Its software consists of operating system, compiling system, parallel programming environment and the scientific visualization system. Tianhe-1A includes 7168 computing nodes. Currently, each computing node is equipped with two Intel Xeon X5670 CPUs (2.93 GHz, six-core) and one NVIDIA Tesla "Fermi" M2050 GPU (1.15 GHz, 448 CUDA cores). The GPU offers 3 GB GDDR5 memory on board, with the bus width of 384 bits and peak bandwidth of 148 GB/s. The total memory of Tianhe-1A is 262 TB, and the disk capacity is 4 PB. All the nodes are connected via a fat tree network for data exchange. This communication network is constructed by high-radix Network Routing Chips (NRC) and high-speed Network Interface Chips (NIC). The theoretical peak performance of Tianhe-1A was 4.7 PFlops, and its LINPACK test result reached 2.566 PFlops. Moreover, the power dissipation at full load is 4.04 MW and the power efficiency is about 635.1 MFlops/W, which was ranked the fourth highest according to the Green 500 list released in 2010. The whole system power of Tianhe-1A is 4.04 MW with 7168 nodes. So the average power consumption per node is 564 W, which has 2 CPUs and 1 GPU. Using the single CPU power 95 W and single GPU power 200 W, we can then roughly estimate the per node power of using only the CPU or the GPU by subtracting the unused processor power from the total power data.

3 GPU Acceleration

To simulate electron turbulence, we need many subcycles to track the fast electron motion [6,7]. As a result, typically the electron push routine PUSHE takes about 50–70% of the total time while electron shift routine SHIFTE takes about 10–20% of the total time. So those two routines were our focus in the GPU porting so far. We used the CUDA C programming language for the GPU port [19]. The strategy of incremental migration to GPU also allow physics users to immediately utilize the new GPU acceleration while further upgrades of physics capabilities and computing power are continuously implemented in the code. This ensures that the GTC-GPU always remains as the current production version shared by all developers and users.

3.1 Pushe

There are two time-consuming loops in PUSHE. The first loop gathers fields at grid point to every electron's positions (electric fields in electrostatic case

and electromagnetic fields in electromagnetic case). The second loop updates electrons' positions based on the gathered fields. Those two loops were mapped to two kernels in the CUDA version: *gather_fields* and *update_gcpos*.

The parallelization scheme and optimization of those two kernels are actually very similar so we will focus on discussing the *gather_fields* kernel below.

The field gathering loop is highly parallel: the calculations of every particle are completely independent of other particles. Thus a straightforward one thread per particle scheme was used to parallelize the loop. As a result, the CUDA porting of the gather loop is fairly straightforward: just replace the loop statement over all electrons by a thread index calculation and then put all the loop body into a CUDA kernel, which won't need much change itself.

```
e1=e1+wt00*(wz0*gradphi(1,0,ij)+wz1*gradphi(1,1,ij))
e2=e2+wt00*(wz0*gradphi(2,0,ij)+wz1*gradphi(2,1,ij))
e3=e3+wt00*(wz0*gradphi(3,0,ij)+wz1*gradphi(3,1,ij))
e4=e4+wt00*(wz0*phit(0,ij)+wz1*phit(1,ij))
```

→

```
float4 tmp1 = tex1Dfetch(texGradphiPhit, ij*2);
float4 tmp2 = tex1Dfetch(texGradphiPhit, ij*2+1);
e1 += wt00*(wz0*tmp1.x + wz1*tmp1.w);
e2 += wt00*(wz0*tmp1.y + wz1*tmp2.x);
e3 += wt00*(wz0*tmp1.z + wz1*tmp2.y);
e4 += wt00*(wz0*tmp2.z + wz1*tmp2.w);
```

Fig. 1. Field gathering code in the original CPU (left) and CUDA version using texture prefetch (right)

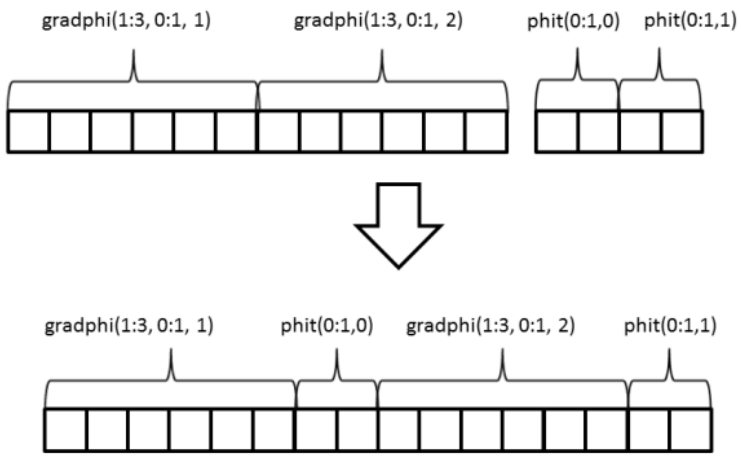


Fig. 2. Reorganizing gradphi and phit arrays into a single array

In this initial CUDA version, kernel profiling showed that the bottleneck is in the reading of fields, part of which is shown in Fig. 1. In this code segment, each

particle gathers the *gradphi* and *phit* fields surrounding its hosting cells. As neighboring particles in the particle array do not necessarily belong to the same cell, this gathering operation is potentially highly uncoalesced, which may lead to a lot of wasted memory transactions on GPU [19]. Furthermore, the linear memory layout of GPU's L1 cache is not a good fit to the higher dimensional locality nature of 3D particle position. GPU has a special cache called texture which is optimized for 2D spatial locality. As a result, binding the *gradphi* and *phit* arrays to texture will improve the performance by about 20%. However, if just doing this code change, the texture cache hit rate turns out to be fairly low: about 8%. Analyzing the field gathering code suggests that if both arrays can be prefetched to the texture cache, the cache hit rate may be significantly increased. To achieve this, *gradphi* and *phit* arrays are first reorganized into a single array (Fig. 2) so that the data needed for a thread is laid out continuously in memory. Second, as each gathering operation needs to load 6 floats in *gradphi* and 2 floats in *phit*, the reorganized array was bound to a float4 texture. In this way, 2 float4 texture fetch to 2 float4 register variables would be enough for all the data a field gathering operation needs. This texture prefetch technique increased the texture cache hit rate from 8% to 35%, which led to 3x kernel performance increase. The final kernel is memory-bound and achieves about 65% of the peak memory band-width. It worth commenting that, as the *gather_fields* kernel is called many times every time *gradphi* and *phit* are updated, the data reorganization overhead is negligible.

Another key optimization technique in PUSHE is minimizing CPU-GPU data transfer. As a complicated plasma physics code, several dozen arrays are needed for the PUSHE calculation. If all the arrays were transferred from CPU to GPU at the beginning of PUSHE and back at the end, CPU-GPU data transfer time would totally dominate the PUSHE time. To avoid this, careful analysis of the data flow in PUSHE was performed. Based on the analysis, there are several optimizations one can do to minimize CPU-GPU data transfers. First, for temporary arrays that are used only inside PUSHE, they can be allocated, used and deallocated directly on the GPU. Second, some temporary arrays in the CPU version are used only within a single kernel. In this case, they can be replaced by register variables, instead of explicitly allocated. Third, there are some arrays whose values are initialized at simulation initialization time and will never change during the whole simulation, e.g. arrays describing grid geometries. Thus those arrays were declared as static variables in PUSHE and data are transferred from CPU to GPU only when calling the routine for the first time. After those optimizations, as can been seen in the next section, CPU-GPU data transfer time will not be a significant fraction.

3.2 Shifte

SHIFTE contains both computation and MPI part. The computation part contains two major steps:

1. Figure out which particles need to be transferred outside of the current MPI process and then copying outgoing particles to separate send buffers.

2. Fill in the "holes" in the particle array left by outgoing particles so that particles are always stored continuously in memory.

The two computational part is actually not very expensive on CPU: both steps basically needs only a single *O(*N*)* transversal of the particle array. The operation itself has also nontrivial sequential part: only after all the previous outgoing particles are processed, one can know where to put the next outgoing particle. Thus it's expected that the porting of SHIFTE to GPU is nontrivial and probably won't see the significant kernel speedup as compared to PUSHE due to the low compute intensity and nontrivial parallelism.

However, porting SHIFTE to GPU is important for the overall performance of the GPU code. If SHIFTE stays on CPU, one needs to transfer all the particle positions to CPU as we don't know which particles are outgoing. This was actually our first implementation. It turns out that in this case data transfer will be a significant overhead. On the other hand, if one can figure out the outgoing particles on GPU, then only outgoing particles need to be transferred back to CPU, which is typically a small fraction. Thus if SHIFTE was not ported to GPU, it will be much slower than the CPU version. So the major purpose of porting SHIFTE is really to get a GPU version that is not slower than the CPU version so SHIFTE will not become a bottleneck.

Flagging which particles are outgoing is straightforward to do in parallel. One can just assign one thread per particle for this calculation. After this step, a flag array is produced whose elements will be one if the corresponding particle is outgoing and zero otherwise. However, writing outgoing particles to a continuous buffer is nontrivial to implement in parallel: the key difficulty is that each outgoing particle needs to know the number of outgoing particles are before itself, which will be used as its output position in the send buffer. However, the number of outgoing particles is determined during runtime. So the key is to figure out a parallel algorithm for finding out the number of outgoing particles before each outgoing particles. Scan is the usual parallel primitive to solve this problem [20]. If one performs an exclusive scan of the flagging array, the corresponding scan result will be just the output positions in send buffer. This was used as the first version in our GPU SHIFTE implementation. However, this version turns out to be about 2x slower than the CPU version. Profiling shows that the bottleneck is in the scan operations as expected. A parallel scan needs to traverse the particle array many times [20]. So the GPU version turns out to be slower than the CPU version since the CPU version only needs to traverse the array once.

To overcome this problem, we adapted the hierarchical scan solution to stream compaction problem [21]. This approach can avoid the need to traverse the particle array multiple times. The basic idea is dividing the operation into 3 steps:

1. First the __ballot warp vote function is combined with the intrinsic integer instruction __popc to implement a very fast intra-warp scan routine. Then the position array is divided to groups of 32 in size (the same size as a CUDA warp). Next the intra-warp scan routine is applied to each group and it writes the number of outgoing particles of each group into an *outgoing_count* array.

2. Performing a scan of the *outgoing_count* array, writing the results to a *global_offset* array. The key now is to realize that the i-th element of the *global_offset* array now stores the number of outgoing particles before the i-th group.
3. Applying the intra-warp scan routine to each group again to calculate the intra-warp offset. Finally, adding the *global_offset* to the intra-warp offset would give the global output positions of each outgoing particles.

Note that in step two, as the *global_offset* array is only 1/32 of the size of the particle array, this scan will be much faster than our first scan-based implementation, which is the key for the higher performance. It is also worth mentioning that as the outgoing particles are divided into left and right directions, these steps need to be performed for both the left and right outgoing particles simultaneously. The hole filling part of SHIFTE is handled in a similar scan-based way, so it won't be discussed in detail here.

4 Results and Analysis

In this section, we present the performance of GTC on Tianhe-1A.

4.1 Medium Scale Benchmark Problem

We carried out the performance study using the typical collisionless trapped electron (CTEM) turbulence [6,7] parameters in a production run for a medium size tokamak, $\alpha = 250\rho_i$, where α is the tokamak minor radius and ρ_i is the ion gyroradius. There are in total 2 billion ions and 0.83 billion electrons for this medium size problem.

For this benchmark, we launch 128 MPI processes on 128 nodes of Tianhe-1A. In the CPU case, each MPI process launches 6 OpenMP threads to utilize all the 6 cores of a single CPU within a node. In the GPU case, each MPI process still launches 6 OpenMP threads for the CPU computation while uses the M2050 within each node to accelerate the electron calculation.

Table 1 shows the total time ("loop") and profile of different modules in GTC. The first two columns show the time and fraction of the CPU version using 128 CPUs while the next two columns show the corresponding numbers for the 128 CPUs + 128 GPUs runs. The last column reports the overall speedup, as well as speedup of the modules that has been ported to GPU. The GTC GPU version gets about 3.1x speedup comparing 128 CPUs + 128 GPUs to 128 CPUs. The results demonstrate that our GPU acceleration of GTC can deliver significant application performance increases. The GPU accelerated routine PUSHE actually got about 8x speedup. It is just because of Amdahl's law the overall speedup is lower.

To understand in more details the GPU performance, Table 2 shows the profile of the GPU PUSHE routine. In this table, "h2d" refers to the CPU-GPU transfer time; "d2h" refers to the GPU-CPU transfer time; "init" refers to memory allocation, free and initialization time; "cpu" refers to computations that

Table 1. Profile of the CPU and GPU version for 128 MPI processes run

	128 CPUs (second)	%	128 CPUs + 128 GPUs (second)	%	Speedup
loop	522.25	100	168.28	100	3.1
field	0.63	0.12	0.66	0.39	
ion	54.4	10.42	54.6	32.45	
shifte	84.6	16.20	51.8	30.78	1.6
pushe	365.4	69.97	44	26.15	8.3
poisson	4.4	0.84	4.4	2.61	
electron other	12	2.30	12	7.13	
diagnosis	0.82	0.16	0.82	0.49	

Table 2. Profile of the GPU version PUSHE for 128 MPI processes run

	elapsed time (second)	%
total	43.99	100
kernel	32.5	73.9
h2d	2.75	6.3
d2h	3.16	7.2
init	1.11	2.5
cpu	4.35	9.9
mpi	0.12	0.3

are still performed on CPU. It can be seen that about 74% of the PUSHE time is spent in the kernel computations. This shows that PUSHE spends most of its running time doing actual computations instead of communication and other overheads. Within the kernel time, 50% of the time is spent in the *gather_fields* kernel and 36% of the time is spent in the *update_pos* kernel. So those two kernels will be the primary target for future optimizations. About 13.5% of the PUSHE time is spent in CPU-GPU data transfer. This shows that our optimizations in minimizing data transfer were successful and it is not a bottleneck of the GPU implementation.

For SHIFTE, with the new hierarchical scan method, the GPU version performance is about 1.6x of the CPU version. As our initial purpose of porting SHIFTE is to make it not slower than the CPU version so we can avoid the expensive data-transfer cost, this result exceeded our initial goal.

4.2 Strong Scaling Results

For the strong scaling study, we use the same medium size problem as in section 4.1, but vary the number of cores in the simulation.

As Tianhe-1A has 2 CPUs and 1 GPU within each compute node, 2 sets of strong scaling studies were performed. The first set uses 1 CPU and 1 GPU within a node (1 CPU + 1 GPU) while the other set uses 2 CPUs and 1 GPU within a node (2 CPUs + 1 GPU). We think both sets are interesting because the first set is the typical way of discussing GPU acceleration result while the second set would be the practical benefit that a Tianhe-1A user will get.

For both sets, the number of MPI processes launched is equal to the number of nodes used. Six OpenMP threads were launched for the first set while twelve OpenMP threads were launched for the second set.

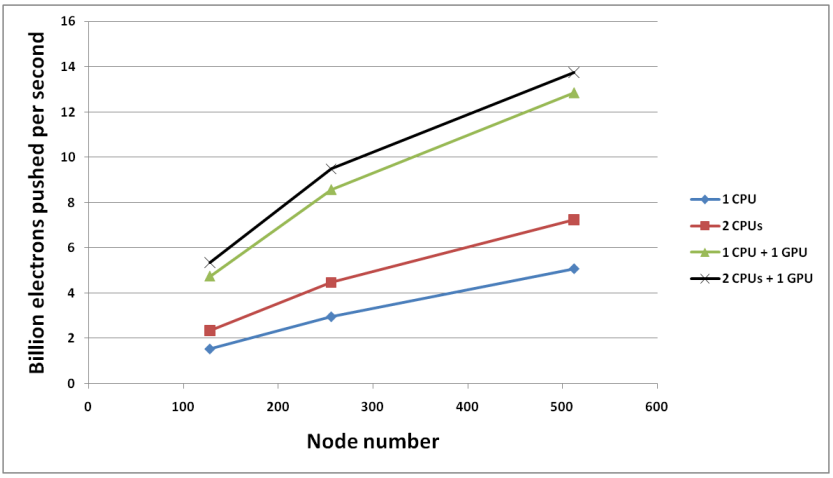


Fig. 3. Performance of CPU and GPU GTC on Tianhe-1A. From bottom to top, the lines correspond to each nodes use 1 CPU, 2 CPUs, 1 CPU + 1 GPU and 2 CPUs + 1 GPU

Fig. 3 shows the total time for those runs. The x-axis shows node number while the y-axis shows billion electrons pushed per second. It is worth commenting that the reason strong scaling experiment used up to 512 nodes is because weak scaling is more relevant in practices than the strong scaling since we typically use more cores for simulations of a larger problem size (i.e., roughly constant wall-clock time for each simulation). For the modest problem size in the strong scaling plot of Fig. 3, typically less than 512 nodes are used in the production runs. It can be seen that for a fixed node count, if comparing the CPU version performance in those two sets, going from 1 CPU to 2 CPUs in each node leads to about 1.5x speedup. On the other hand, the differences in the GPU performance are much smaller because most of the application time is spent on GPU so additional CPU computation power won't help much in improving overall performance.

Fig. 4 shows the GPU speedup factor in those two sets. One thing to notice is that the speedup factor will decrease at large node counts: e.g. from 3.1x at 128

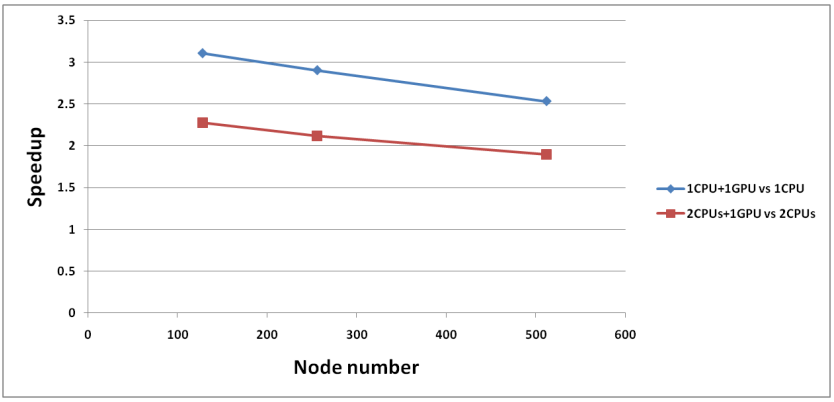


Fig. 4. GTC GPU speedup on Tianhe-1A. The blue line is comparing 1 CPU + 1 GPU to 1 CPU per node while the red line is comparing 2 CPUs + 1 GPU to 2 CPUs per node.

Table 3. Profile of the CPU and GPU version for 512 MPI processes run

	512 CPUs (second)	%	512 CPUs + 512 GPUs (second)	%	Speedup
loop	158.09	100	62.05	100	2.5
field	0.63	0.40	0.69	1.11	
ion	16.3	10.31	16.2	26.11	
shifte	31.7	20.05	17.5	28.20	1.8
pushe	94.3	59.65	12.5	20.15	7.5
poisson	4.9	3.10	4.9	7.90	
electron other	10	6.33	10	16.12	
diagnosis	0.26	0.16	0.26	0.42	

nodes to 2.5x at 512 nodes for the 1 CPU + 1 GPU set. There are two possibilities causing this phenomenon. First, as a strong scaling study, at large node count the problem size for each MPI process will decrease. A smaller computational problem may lead to a lower GPU routine speedup. Second, the remaining CPU part doesn't scale well so the total time spent in GPU-accelerated routines will decrease at a large node count. As a result, the GPU speedup factor also decreases because of Amdahl's law.

To find out which is the case, the profiling details of the 512 nodes case are shown in Table 3. It can be seen that with the smaller problem per MPI processes, the PUSHE speedup indeed decreases from 8.3x to 7.5x. However, larger node also leads to more computation in SHIFTE and correspondingly SHIFTE speedup increases from 1.6x to 1.8x. Those two effects slightly offset each other. As a result, GPU acceleration in PUSHE + SHIFTE drops from 4.7x

to 4.2x. On the other hand, the percentage of PUSHE + SHIFTE decreases from 86.3% to 80%. This is mainly because the Poisson and electron charge deposition part do not scale as well as PUSHE. With those data, we can do two thought experiments. First, if PUSHE + SHIFTE was still 86.3% of the total time and GPU speedup is 4.2x, the overall GPU speedup would be 2.9x. Second, if GPU speedup was still 4.7x and PUSHE + SHIFTE is 80% of the total time, the overall GPU speedup would be 2.7x. This analysis shows that *for strong scaling, both reduced GPU acceleration and CPU scaling contribute to the reduced GPU speedup at large node counts but the CPU scaling plays a bigger role.*

Energy efficiency is one of the most important issues in current and future supercomputer systems. So in addition to absolute performance, performance per watt would also be an interesting metric. Using the method discussed in the end of section 2, the performance per W of different runs can be estimated and plotted in Fig. 5.

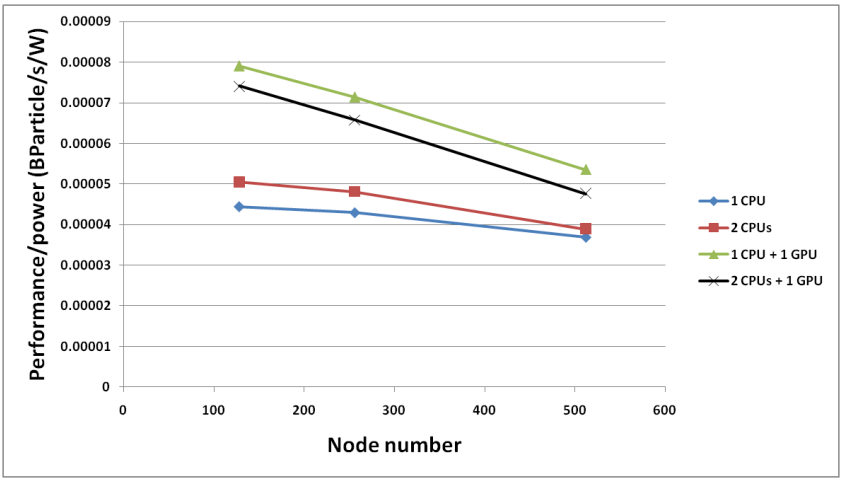


Fig. 5. Performance/W for the strong scaling runs

Fig.5 shows the interesting fact that while 2 CPUs + 1 GPU has the highest performance, 1 CPU + 1 GPU actually has the highest energy efficiency. This is easy to understand. The GPU is more energy efficient for GTC's computations. So using one more less energy-efficient CPU will decrease the overall energy efficiency.

4.3 Weak Scaling Results

It is even more important to test the weak scaling of the GTC code, since more cores are usually desired for a larger size problem in production simulation.

For weak scaling study on Tianhe-1A, we still use the two sets of comparison methods in the strong scaling study, i.e. comparing 1 CPU to 1 CPU + 1 GPU and comparing 2 CPUs to 2 CPUs + 1 GPU.

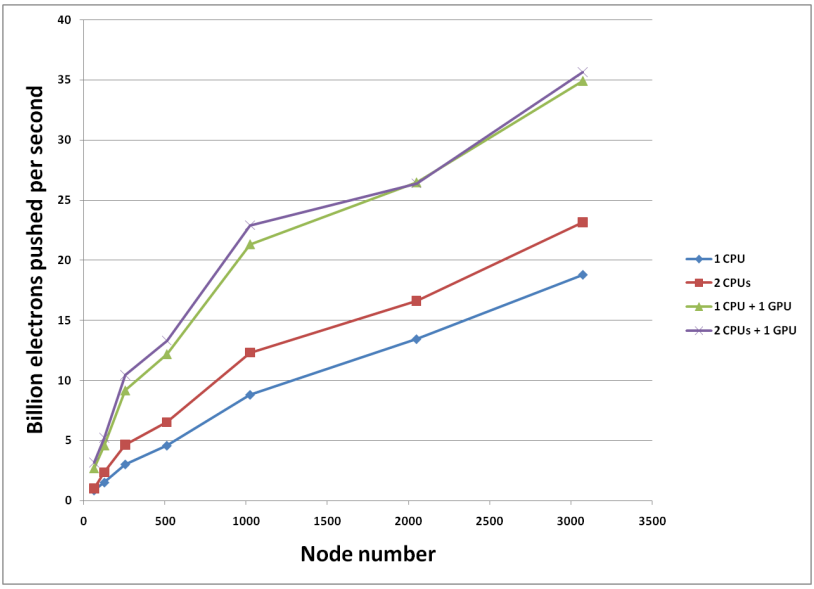


Fig. 6. GTC weak scaling performance on Tianhe-1A

Fig. 6 shows the GTC performance in this weak scaling study while Fig. 7 shows the speedup factor. Fig. 6 shows that in the CPU code, using 2 CPUs within a node will give a significant performance increase compared to using just 1 CPU. However, for the GPU version, 2 CPUs does not give any significant performance increase compared to the 1 CPU case. This is because most of the computation is now in the GPU and the additional CPU power does not lead to further performance gains. As a result, the speedup factor is lower in the second comparison set as can be seen in Fig. 7. In either case, it can be seen that using the GPU in each Tianhe-1A's node, large scale GTC simulations can get a 1.5–2x performance gain even at large node counts. This demonstrates the advantage of using GPU for large scale fusion simulations on Tianhe-1A.

Fig.7 also shows that the GPU speedup decreases with increasing node count. Similar to our analysis in the strong scaling case, Table 4 shows the detailed profiling for the 1 CPU vs 1 GPU set at 3072 nodes. From the speedup column, it can be seen that the GPU speedup factor is about the same for PUSHE, as expected from a weak scaling study. The speedup factor for SHIFTE decreases from 1.6x to 1.2x. This is primarily because MPI communication takes a larger percentage of SHIFTE time at larger node count and thus the effect of GPU

acceleration is smaller. The CPU profiling result shows that PUSHE + SHIFTE decreases from 86% at 128 nodes to 61%, which leads to a lower GPU speedup because of Amdahl's law. Thus *for weak scaling, CPU scaling is the main reason for the reduced GPU speedup at a large node count.*

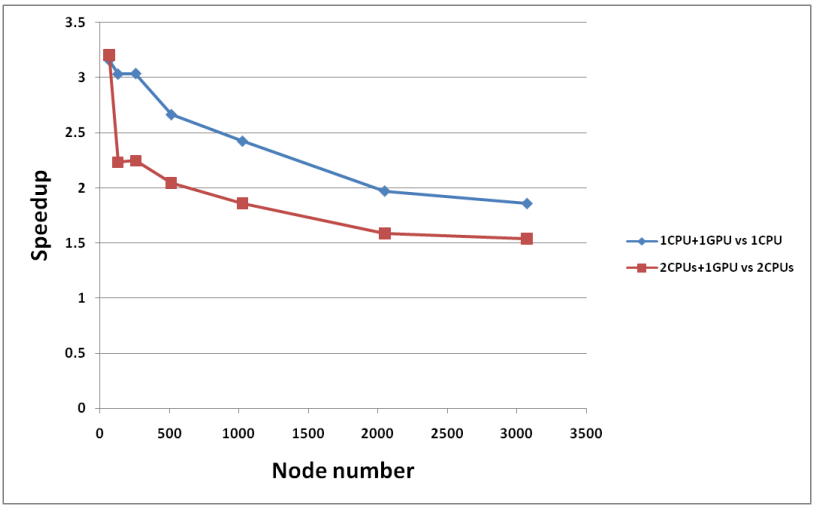


Fig. 7. GTC weak scaling speedup on Tianhe-1A

Table 4. Profile of the CPU and GPU version for 3072 MPI processes run

	3072 CPUs (second)	%	3072 CPUs + 3072 GPUs (second)	%	Speedup
loop	699.5	100	375.4	100	1.9
field	9.3	1.33	8.9	2.37	
ion	79.5	11.37	79.3	21.12	
shifte	67.3	9.62	55	14.65	1.2
pushe	359.5	51.39	44.2	11.77	8.1
poisson	53	7.58	53	14.12	
electron other	98.3	14.05	102.8	27.38	
diagnosis	32.6	4.66	32.2	8.58	

Similar to the strong scaling case, we also plot the performance per watt of the weak scaling runs in Fig.8.

Similar to the strong scaling case, Fig.8 also shows that 1 CPU + 1 GPU would be the most energy efficient way to run GTC simulations.

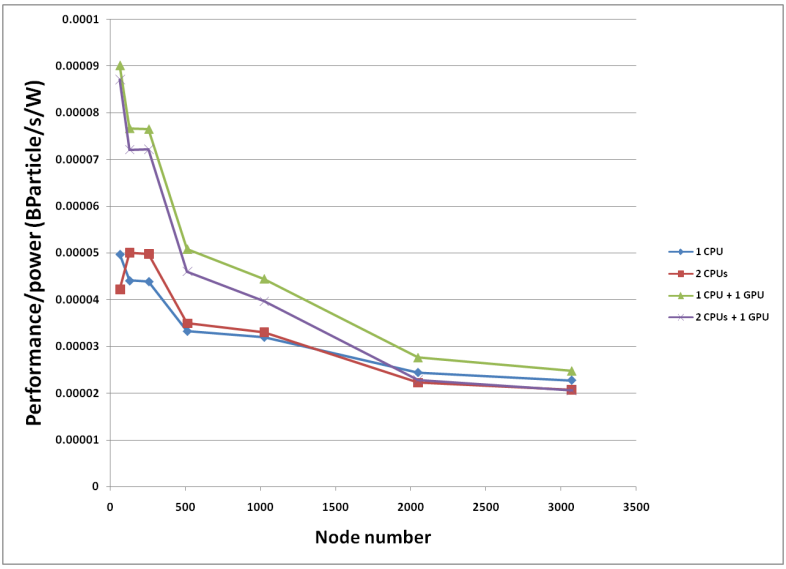


Fig. 8. Performance per Watt of the weak scaling runs

5 Conclusions and Discussions

As far as we know, this is the first work to demonstrate the advantage of GPU for large-scale production fusion simulations. We showed that the texture cache specific to the GPU architecture, combined with some data reorganization, is particularly suitable for the data locality pattern of the PUSH operations, which led to significant kernel speedup. We also presented new hierarchical scan-based implementation of the SHIFT operation on GPU, which is faster than the CPU version.

To further advance our understanding of the microphysics of burning plasma, larger scale simulations will be needed whose computational requirements far exceed the capabilities of today's petascale machines. This work shows that GPU has the great potential to enable those next generation large scale fusion simulations. Our experience with GTC-GPU demonstrates again the implications of the large-scale heterogeneous cluster computing: both the CPU and GPU parts need to get good performance and scaling in order to get good overall application performance and scaling.

As for future development, we have seen that the primary reason for reduced GPU speedup at large node counts is the CPU scaling. Thus further work is needed to improve the performance and scaling of the other CPU part. Furthermore, after electron part is ported to GPU, the next most time-consuming module is ion, which will be our next porting target. Finally, with OpenACC and CUDA Fortran becoming mature recently, we are also evaluating reporting PUSHE using OpenACC and CUDA Fortran. As the main difficulty of porting to

GPU is choosing the proper parallelization scheme and optimization techniques, which are solved in this work, using OpenACC and CUDA Fortran will be just applying those techniques with a different syntax. So we expect the performance to be basically the same as the CUDA C version as reported in this paper. However, OpenACC and CUDA Fortran should make the GPU code easier to maintain as GTC's CPU code is Fortran-based.

Acknowledgments. This research is supported by National Magnetic Confinement Fusion Science Program under grant NO. 2009GB105000 and the National Natural Science Foundation of China under grant NO. 11105033. Thanks to Jinghua Feng, Bin Xu of National Supercomputer Center in Tianjin, they help us solve some problems when we do large-scale test on TH-1A.

References

1. Lin, Z., Hahm, T.S., Lee, W.W., Tang, W.M., White, R.B.: Turbulent Transport Reduction by Zonal Flows: Massively Parallel Simulations. Science 281, 1835 (1998)
2. `http://www.iter.org`
3. `http://phoenix.ps.uci.edu/GTC`
4. Lin, Z., Holod, I., Chen, L., Diamond, P.H., Hahm, T.S., Ethier, S.: Wave-particle decorrelation and transport of anisotropic turbulence in collisionless plasmas. Phys. Rev. Lett. 99, 265003 (2007)
5. Zhang, W., Lin, Z., Chen, L.: Transport of Energetic Particles by Microturbulence in Magnetized Plasmas. Phys. Rev. Lett. 101, 095001 (2008)
6. Xiao, Y., Lin, Z.: Turbulent transport of trapped electron modes in collisionless plasmas. Phys. Rev. Lett. 103, 085004 (2009)
7. Xiao, Y., Lin, Z.: Convective motion in collisionless trapped electron mode turbulence. Phys. Plasmas 18, 110703 (2011)
8. Holod, I., Zhang, W.L., Xiao, Y., Lin, Z.: Electromagnetic formulation of global gyrokinetic particle simulation in toroidal geometry. Phys. Plasmas 16, 122307 (2009)
9. Zhang, H.S., Lin, Z., Holod, I., Wang, X., Xiao, Y., Zhang, W.L.: Gyrokinetic particle simulation of beta-induced Alfven eigenmode. Phys. Plasmas 17, 112505 (2010)
10. Zhang, W., Holod, I., Lin, Z., Xiao, Y.: Global Gyrokinetic Particle Simulation of Toroidal Alfven Eigenmodes Excited by Antenna and Fast Ions. Phys. Plasmas 19, 022507 (2012)
11. Deng, W., Lin, Z., Holod, I., Wang, Z., Xiao, Y., Zhang, H.: Linear properties of reversed shear Alfven eigenmodes in DIII-D tokamak. Nuclear Fusion 52, 043002 (2012)
12. Deng, W., Lin, Z., Holod, I.: Gyrokinetic simulation model for kinetic magnetohydrodynamic processes in magnetized plasmas. Nuclear Fusion 52, 023005 (2012)
13. Decyk, V.K., Singh, T.V.: Adaptable particle-in-cell algorithms for graphical processing units. Computer Physics Communications 182(3), 641–648 (2011)
14. Burau, H., Widera, R., Honig, W., Juckeland, G., Debus, A., Kluge, T., Schramm, U., Cowan, T.E., Sauerbrey, R., Bussmann, M.: PIConGPU: A Fully Relativistic Particle-in-Cell Code for a GPU Cluster. IEEE Transaction on Plasma Science 38(10), 2831–2839 (2010)

15. Stantchev, G., Dorland, W., Gumerov, N.: Fast parallel particle-to-grid interpolation for plasma PIC simulations on the GPU. Journal of Parallel and Distributed Computing 68(10), 1339–1349 (2008)
16. Rossinelli, D., Conti, C., Koumoutsakos, P.: Mesh-particle interpolations on graphics processing units and multicore central processing units. Philosophical Transactions of the Royal Society 369, 2164–2175 (2011)
17. Madduri, K., Ibrahim, K.Z., Williams, S., Im, E.J., Ethier, S., Shalf, J., Oliker, L.: Gyrokinetic toroidal simulations on leading multi- and manycore HPC systems. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (2011)
18. Madduri, K., Im, E.J., Ibrahim, K.Z., Williams, S., Ethier, S., Oliker, L.: Gyrokinetic particle-in-cell optimization on emerging multi- and manycore platforms. Parallel Computing 37(9), 501–520 (2011)
19. NVIDIA Corporation, CUDA Programming Guide. In: CUDA Development Toolkit (2011)
20. Sengupta, S., Harris, M., Zhang, Y., Owens, J.: Scan Primitives for GPU Computing. In: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware (2007)
21. Billeter, M., Olsson, O., Assarsson, U.: Efficient Stream Compaction on Wide SIMD Many-Core Architectures. In: High Performance Graphics (2010)

Achieving Efficient Strong Scaling with PETSc Using Hybrid MPI/OpenMP Optimisation

Michael Lange[1,⋆], Gerard Gorman[1], Michèle Weiland[2], Lawrence Mitchell[2], and James Southern[3]

[1] Applied Modelling and Computation Group, Imperial College London, London, UK
{michael.lange,g.gorman}@imperial.ac.uk
http://amcg.ese.ic.ac.uk
[2] EPCC, The University of Edinburgh, Edinburgh, UK
[3] Fujitsu Laboratories of Europe Ltd., Hayes, Middlesex, UK

Abstract. The increasing number of processing elements and decreasing memory to core ratio in modern high-performance platforms makes efficient strong scaling a key requirement for numerical algorithms. In order to achieve efficient scalability on massively parallel systems scientific software must evolve across the entire stack to exploit the multiple levels of parallelism exposed in modern architectures. In this paper we demonstrate the use of hybrid MPI/OpenMP parallelisation to optimise parallel sparse matrix-vector multiplication in PETSc, a widely used scientific library for the scalable solution of partial differential equations. Using large matrices generated by Fluidity, an open source CFD application code which uses PETSc as its linear solver engine, we evaluate the effect of explicit communication overlap using task-based parallelism and show how to further improve performance by explicitly load balancing threads within MPI processes. We demonstrate a significant speedup over the pure-MPI mode and efficient strong scaling of sparse matrix-vector multiplication on Fujitsu PRIMEHPC FX10 and Cray XE6 systems.

Keywords: PETSc, Hybrid MPI/OpenMP, strong scaling, task-based parallelism, hierarchical load balancing, sparse matrix-vector multiply.

1 Introduction

Recent development in High Performance Computing (HPC) architectures has been driven by a clear trend towards greater numbers of lower power cores and a decreasing memory to core ratio. Numerical algorithms and scientific software have to adapt to these changes to efficiently utilise the available memory and network bandwidth. Hybrid programming techniques, where shared memory programming is combined with inter-node message passing, can be used to exploit

⋆ The work presented here was funded by Fujitsu Laboratories of Europe Ltd. and the European Commission in FP7 as part of the APOS-EU project (grant agreement 277481).

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 97–108, 2013.

the multiple levels of parallelism inherent in modern architectures in order to achieve sustainable scalability on massively parallel systems.

In this paper we describe the addition of OpenMP thread parallelism to the Portable Extensible Toolkit for Scientific Computation (PETSc) [5, 6]. PETSc is a widely used library for the scalable solution of partial differential equations and is often used as a key component of large scientific applications. Sparse matrix-vector multiplication (spMVM) is by far the most computationally expensive component of sparse iterative linear solvers [13]. Therefore we focus on optimising spMVM within PETSc using hybrid programming techniques and evaluate strong scalability on large numbers of compute nodes. We demonstrate that using task-based parallelism to hide communication latency can provide significant speedups over naive OpenMP parallelisation. Further, explicit thread-level load balancing can be used to give additional increases in performance, resulting in significantly improved scalability over pure-MPI implementations in the strong scaling limit.

The matrices used for benchmarking our implementation are extracted from the open source, general-purpose, multi-phase computational fluid dynamics (CFD) code Fluidity [2]. Fluidity solves the Navier-Stokes equations and accompanying field equations on arbitrarily unstructured finite-element meshes. It is used in areas including geophysical fluid dynamics, computational fluid dynamics and ocean modelling [10].

1.1 Sparse Matrix-Vector Multiplication

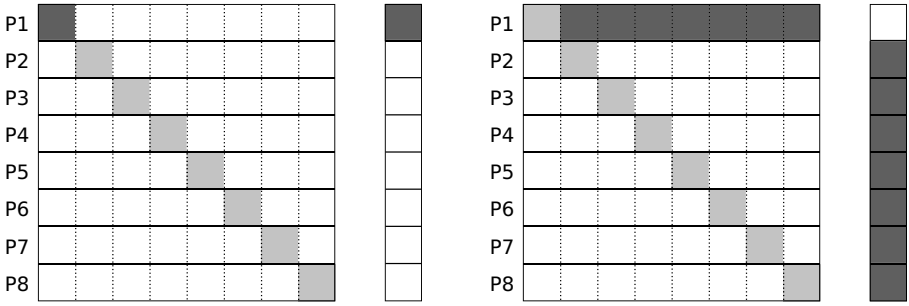
PETSc offers a wide range of high-level components required for linear algebra, such as linear and non-linear solvers as well as preconditioners. These are based on a suite of parallel data structures which implement basic vector and matrix operations. The most computationally expensive operation for solvers and preconditioners alike is the multiplication of sparse matrices with an input vector.

PETSc represents distributed MPI matrices by dividing them into diagonal and off-diagonal parts, which on each process are stored as sequential matrices. The diagonal sub-matrix hereby corresponds to the part of the input vector that is stored locally by the process. As a consequence of this storage strategy, as shown in Fig. 1, the matrix-vector multiplication is implemented in two phases:

- First, each process multiplies its diagonal sub-matrix with the local elements of the input vector, while vector elements that reside off-process are gathered into the local memory of the executing process.
- Off-diagonal matrix elements are then multiplied with the formerly remote vector elements and added to the partial solution.

1.2 Related Work

Sparse matrix multiplication is one of the most heavily used kernels in scientific computing and has therefore received attention from several groups [7, 9, 11, 15].



(a) Diagonal matrix elements are multiplied with the local part of the vector while remote vector elements are gathered.

(b) The off-diagonal sub-matrix is then multiplied with a local copy of the gathered vector elements and added to the partial solution.

Fig. 1. Parallel sparse matrix-vector multiplication using 8 MPI processes

Multiple storage formats, optimisation strategies and even auto-tuning frameworks exist to improve spMVM performance on a wide range of multi-core architectures [15]. On modern HPC architectures hybrid programming methods are being investigated to better utilise the hierarchical hardware design by reducing communication needs, memory consumption and improved load balance [11]. In particular, task-based threading methods have been highlighted by several researchers, where dedicated threads can be used to overlap MPI communication with local work [11, 13, 14].

2 Hybrid MPI/OpenMP Parallelism

Multi-core processors are now ubiquitous in HPC and programmers are effectively presented with two levels of parallelism: inside a compute node, cores share a contiguous memory address space and they can exchange information by directly manipulating this memory space; between nodes, distributed memory parallelism is most commonly implemented using explicit message passing via MPI. Exposing and expressing both intra- and inter-node parallelism can be achieved using a hybrid programming approach.

One motivation for moving away from MPI-only parallelised applications is given by memory limitations. While the number of cores is steadily increasing in modern HPC architectures, the memory available to each core is decreasing [11]. By exploiting thread-level parallelism, the same number of cores can be utilised within a single node while reducing the MPI memory footprint [4]. For scientific applications based on domain decomposition, reducing the MPI process granularity also reduces data replication due to halos or ghost cells.

Performance gains may also be expected from using fewer MPI processes, since it not only reduces communication overheads, but also improves the load balance between individual processes [11, 13]. However, reducing process-level

imbalance may have a negative effect on the load balance among threads, which in turn can be compensated for by node-level scheduling strategies, as discussed in Sec. 2.3.

2.1 NUMA Architecture

Non-Uniform Memory Access (NUMA) refers to multiprocessor systems whose memory is divided into multiple memory nodes. This architecture was designed to overcome the scalability limits of the symmetric multiprocessing (SMP) architecture. However, this hierarchical memory model for multi-core processors means that it takes longer for a process or thread to access some parts of the memory than others.

It is therefore important to consider data locality in threaded applications, since regular off-domain memory access can be detrimental to the performance of already memory-bound applications. In order to minimise bus contention a parallel *first touch* memory initialisation is often used on NUMA architectures to bind data to the memory bank that is closest to the core subsequently using the data block [11]. In addition, thread and process pinning is required to optimise memory utilisation for all bandwidth-bound algorithms.

When multiplying sparse matrices a master-only approach is most often used to parallelise the local computation steps using threads (see Sec. 1.1). However, threaded spMVM across multiple NUMA domains requires random but frequent off-domain memory access to fetch input vector elements. In order to avoid the high-latencies associated with off-domain data fetch NUMA domains can be treated as single address spaces connected by multiple MPI tasks within a compute node. This approach restricts threads to accessing a single NUMA domain as demonstrated in Sec. 4.1.

2.2 MPI-Communication Overlap

As described in Sec. 1.1, PETSc splits parallel spMVM into two phases in order to allow the multiplication of the diagonal submatrix to be overlapped with the MPI communication required to fetch off-core vector elements. Nevertheless, Schubert et al. [13] showed that few MPI implementations provide truly asynchronous communication and significant performance gains can be achieved by using *task-based* threading, where a single thread is dedicated to actively perform the localisation of global vector elements. This approach not only overlaps MPI transfer latencies with computation but also hides any sequential overhead incurred from moving data to and from the required MPI buffer space.

Task-based threading stands in contrast to traditional *vector-based* threading, where all threads share the computational load evenly. In order to utilise the *task-based* variant the thread-parallel section needs to be lifted to enclose the vector scatter-gather operation. This prohibits the use of OpenMP `parallel for` pragmas to distribute the local row-wise computation among threads and requires the explicit computation of thread partition boundaries.

2.3 Thread-Level Load Balance

Traditional *vector-based* threading with OpenMP divides the number of matrix rows approximately evenly among threads by applying `parallel for` pragmas to the outer loop. This, however, ignores the fact that individual rows may incur varying amounts of computational work, creating a potential load imbalance within individual thread groups. Instead, thread-level load balance may be improved statically by dividing the number of non-zeros approximately equally between threads, as pointed out by Williams et al. [15].

It is important to note that the matrix stencil does not change during the solve. Thus, an explicit thread partitioning scheme may be computed after the matrix has been assembled and cached with the matrix object. This turns the load balance optimisation into a one-off cost, allowing, in principle, the use of load balancing schemes of arbitrary complexity.

The method used in this paper starts with an initial greedy allocation, where each worker thread receives a block of continuous rows. This is followed by an iterative local diffusion algorithm, which further balances the number of non-zeros allocated to each thread, This procedure balances the thread-level work load and memory bandwidth requirement according to floating point operations required for the solution.

3 Benchmark

The matrices used for benchmarking the hybrid MPI/OpenMP implementations have been generated by Fluidity from a global baroclinic ocean simulation, which is representative of a range of three-dimensional multi-scale oceanographic problems [10]. The unstructured mesh is based on two-dimensional high-resolution coastline data that is extruded vertically using constant spacing. By changing the vertical resolution of the extruded mesh the size of the problem can be scaled linearly, allowing a controlled quasi-linear increase in work load for the extracted matrices.

The benchmark matrices used in this work are pressure field solves extracted after five timesteps. The resulting matrices are solved using the Conjugate Gradient method with a Jacobi preconditioner and the number of iterations was limited to $10,000$.

3.1 Cray XE6

One of the benchmarking systems used for the work presented here is HECToR, a Cray XE6 based on the AMD Opteron 6200 Interlagos processor series and Crays Gemini interconnect [1]. The Interlagos compute nodes are based on two AMD Bulldozer processors, each with 16 cores at 2.3 GHz paired into two modules and a peak memory bandwidth of 51.2 GB/s. Each module has its own associated memory bank, resulting in four separate memory nodes per compute node [8].

3.2 Fujitsu PRIMEHPC FX10

The second benchmarking system available to us is a 96-node Fujitsu PRIMEHPC FX10 system [3]. The PRIMEHPC FX10 is a UMA (Uniform Memory Access) architecture based on the SPARC64 IXfx processor. A single compute node has 16 cores at 1.848 GHz and a peak memory bandwidth of 85 GB/s.

4 Results

In this section we evaluate the parallel performance of the different hybrid spMVM approaches detailed in Sec. 2. Since hybrid programming offers a complex set of choices on how to utilise a given hardware set, we start our investigation by analysing various process-to-thread ratios for fixed numbers of cores. This provides insights into the resource utilisation of each algorithm and provides an estimate for the best hybrid configuration to be used during the subsequent strong scalability study on large numbers of compute nodes.

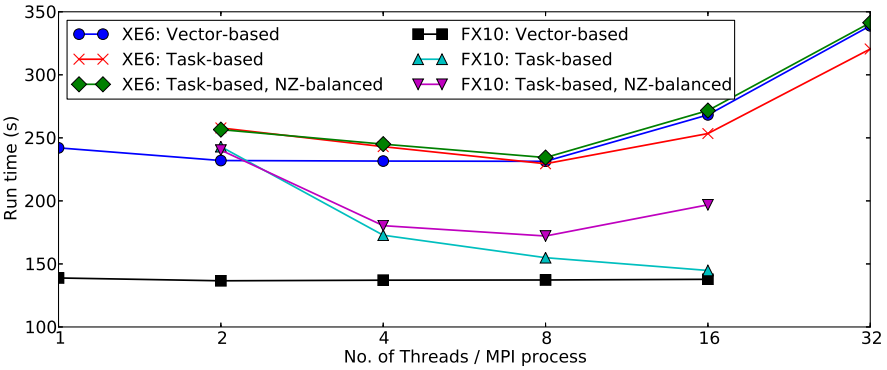
4.1 Hardware Utilisation

Figure 2 shows the performance of varying hybrid process-thread combinations on the Cray XE6 and Fujitsu PRIMEHPC FX10 systems. The left-most entry of the vector-based configuration constitutes the MPI-only baseline configuration. OpenMP overheads have been verified to be negligible for the given problem size using microbenchmarks [12].

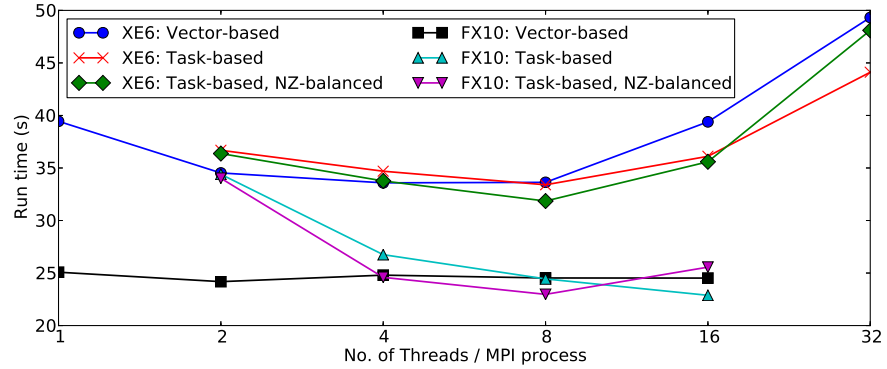
On the XE6, using only a small number of compute nodes (Fig. 2a and 2b), the task-based algorithms with and without explicit thread-level load balancing perform best when running 8 threads wrapped by 4 MPI processes per node. This correlates with NUMA alignment, where threads are used only inside individual NUMA domains and MPI tasks connect separate memory nodes. A significant performance reduction can then be observed with 16 and 32 threads per process, which coincides with NUMA traffic being incurred due to fetching input vector elements (see Sec. 2.1).

However, using 4096 cores (128 XE6 nodes, Fig. 2c), the task-based mode without explicit load balancing seems to defy the slowdown due to NUMA traffic when using 16 and 32 threads per process. We can conclude that the algorithm is now bound by memory bandwidth rather than latency. In contrast, the thread-balancing mode still experiences a latency slowdown, but exhibits superior performance with a NUMA-aligned configuration. This is due to an imbalance in vector elements required by each thread due to the explicit thread-balancing, which aggravates the algorithm's sensitivity to memory latency.

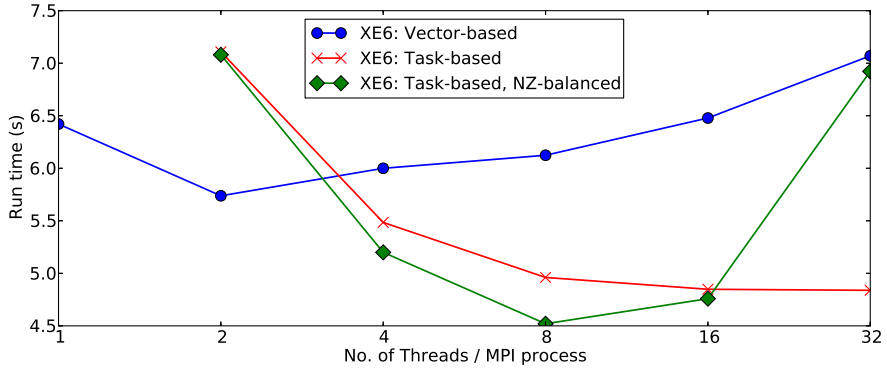
Furthermore, both task-based modes significantly outperform the vector-based threading approach on 4096 cores, demonstrating the performance loss due to MPI communication overheads. Although vector-based threading provides better performance on small numbers of cores due to having an extra worker thread, on large numbers of compute nodes the approach struggles to utilise the given



(a) 128 cores



(b) 1024 cores



(c) 4096 cores

Fig. 2. Matrix multiplication run times on a fixed number of cores with varying thread-to-process ratios. The left most value represents a close approximation to MPI-only performance. Native compilers were used on both architectures.

memory bandwidth with an increasing number of threads. As shown in Fig. 2c, performance is greatest with only two threads per process, indicating that the algorithm's performance is communication-bound.

On the PRIMEHPC FX10 system, we observe similar scaling properties and resource limitations with an increasing number of processing cores for all three algorithms. Although the test system used for this work was limited to 1536 cores, we can, therefore, infer an estimate of the the scaling behaviour of the PRIMEHPC FX10 architecture for large scale systems.

The key difference to the XE6 is that PRIMEHPC FX10 is a UMA architecture, and therefore does not incur memory latency penalties due to using multiple memory nodes per thread group. This can be observed in Fig. 2a where, in contrast to the XE6, the task-based mode without thread balancing improves performance steadily with increasing numbers of threads per node. However, the same memory latency limitation on small numbers of cores affects the thread-balancing mode.

On 1024 cores (64 PRIMEHPC FX10 nodes, Fig. 2b), the profiles exhibit properties similar to the 4096-core XE6 results. The vector-based mode is limited by inter-process communication and performs best with two threads per process, while the overall best performance is achieved by the thread-balancing approach using eight threads per process.

4.2 Strong Scaling

In this section we analyse the strong scalability of the described hybrid algorithms on the Cray XE6 system and compare their performance to a pure-MPI approach. All hybrid modes were run using four MPI processes per compute node with eight threads each in order to prevent NUMA traffic due to input vector elements (see Sec. 2.1).

The matrix used in Fig. 3 has 13,491,933 degrees-of-freedom (DoF) and 371,102,769 non-zero elements and was generated by a parallel Fluidity simulation decomposed into 1024 sub-domains. For the hybrid modes the number of MPI processes used in the strong limit therefore matches the number of processes used during the original decomposition. For more than 1024 cores, however, the pure-MPI mode uses more processes than the matrix was originally optimised for, resulting in a potential slowdown due to load imbalance. Therefore, an equivalent matrix which has been optimised for 8192 MPI processes has also been included in the benchmark (dashed line).

At the low end of the scaling curve no significant performance differences can be noted. For more than 512 cores (16 XE6 nodes) the task-based hybrid methods show a better scalability over the vector-based approach. The thread-balancing implementation hereby performs best, maintaining a nearly constant parallel efficiency of $> 88\%$ between 512 and 2048 cores, and even experiences slightly super linear scaling between 1024 and 2048 cores.

On the same matrix, the pure-MPI performance decreases significantly faster than the hybrid algorithms for more than 512 cores (16 XE6 nodes). The equivalent MPI runs using a more finely decomposed matrix, on the other hand, closely

match the performance of the task-based mode without thread-balancing up to 2048 cores. However, in the strong limit the thread-balancing mode outperforms the optimised MPI runs.

Furthermore, between 2048 and 4096 cores (64 and 128 XE6 nodes) we observe strong super linear scaling for both task-based methods. Since the final runtime in the strong limit is below 4 seconds, we can deduce that scalability ceases at this point due to a lack of computational work and that the super linear scaling effects are due to favourable cache effects.

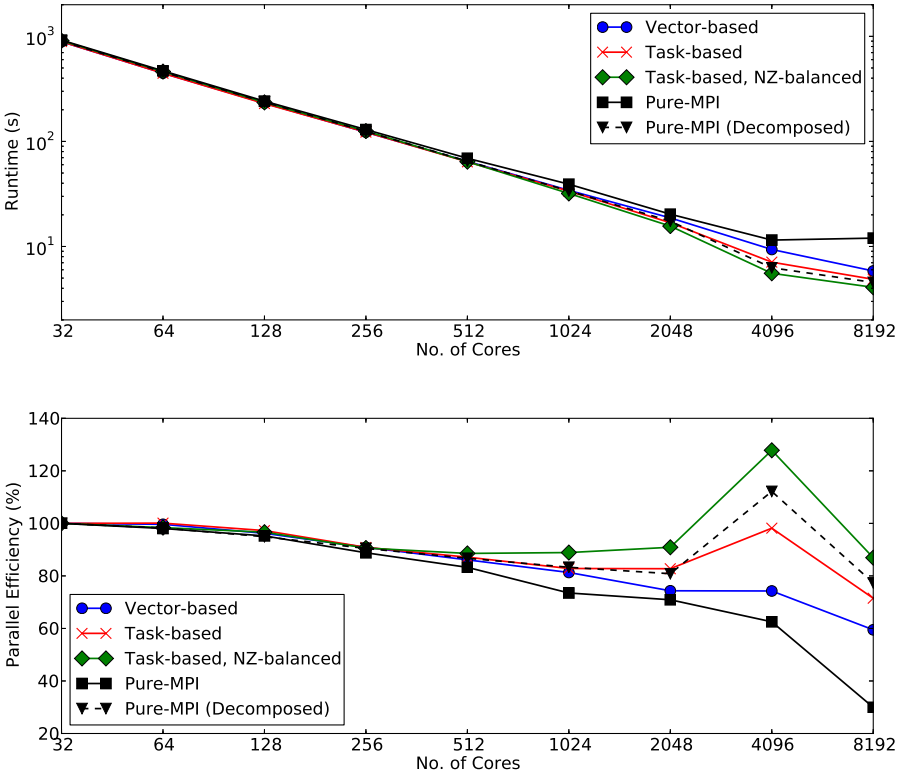


Fig. 3. Strong scaling results for the pressure matrix on up to 256 XE6 nodes (8192 cores). All hybrid modes use 4 MPI ranks per node and 8 threads per rank.

Fig. 4 shows scalability on up to 32,768 cores (1024 XE6 nodes) when the workload of the matrix multiplication is increased by a factor of 4 by changing the vertical extrusion of the parent mesh (see Sec. 3). This matrix has 52,040,313 DoF and 1,462,610,289 non-zeros and is based on a 4096-domain partitioning. The results follow the same general trend, with significant differences in performance observable in the strong end of the scalability curve. The pure-MPI performance starts to deteriorate earlier and the super linear scaling in the high end is more pronounced for all approaches.

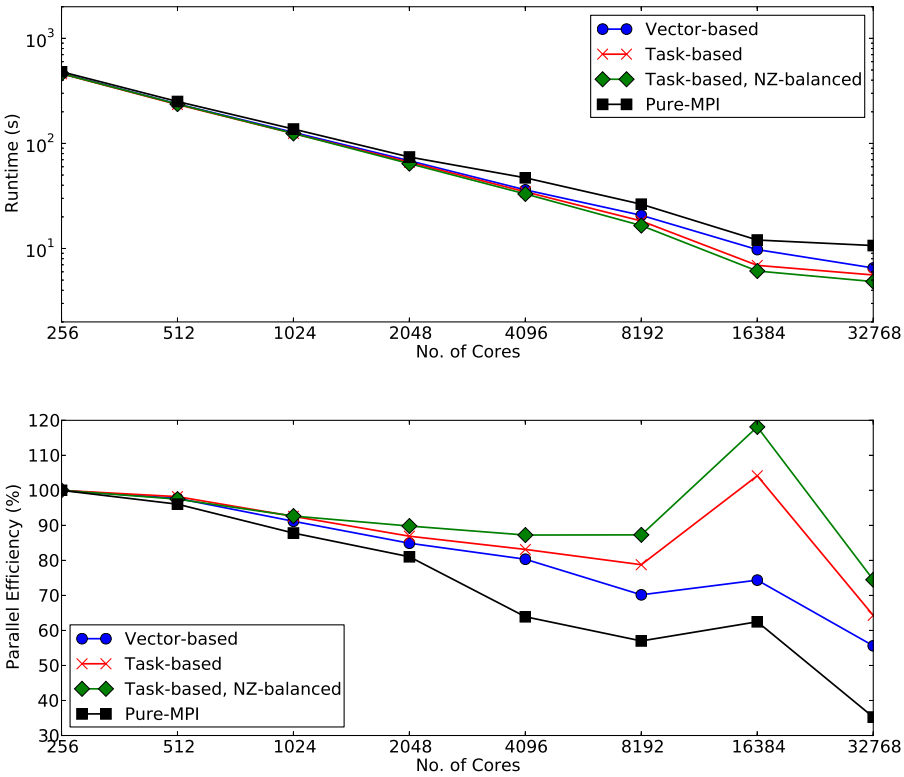


Fig. 4. Strong scaling results for a larger pressure matrix on up to 1024 XE6 nodes (32768 cores). All hybrid modes use 4 MPI ranks per node and 8 threads per rank. Runs with less than 256 cores (8 XE6 nodes) have been omitted due to insufficient memory per MPI process.

5 Summary and Discussion

In this paper we present an analysis of the scaling properties of sparse matrix-vector multiplication using a hybrid MPI/OpenMP extension to the PETSc library. We compare hybrid vector-based and task-based algorithms with a pure-MPI variant using large matrices generated by Fluidity. We describe an extension to the traditional task-based approach, where the load balance among threads is optimised a-priori according to the number of non-zeros in each row.

The thread-balancing extension is shown to give superior performance when scaled to large numbers of compute nodes on a Cray XE6 system and on moderate numbers of nodes of a Fujitsu PRIMEHPC FX10 system. The algorithm achieves this by improving the memory bandwidth utilisation within a given compute node and by hiding MPI communication latency. This comes at the cost of increased memory latency effects on small numbers of cores, since the

algorithm creates an imbalance in input vector elements per thread. However, once the main resource limitation of the algorithm shifts to memory bandwidth the thread-balancing approach can improve performance significantly.

Furthermore, the thread-balancing approach enhances one of the fundamental advantages of hybrid programming: By reducing the number of MPI processes the inherent load imbalance among processes is reduced at the expense of load imbalance among threads. This is desirable, however, since we can deal with the thread imbalance explicitly by caching an optimised thread partitioning with the matrix. As a result, this approach improves work load balance and memory bandwidth utilisation at the compute node level in order to increase overall performance.

References

[1] Cray XE6 system (March 2013), http://www.cray.com/Products/Computing/XE/Specifications/Specifications-XE6.aspx

[2] Fluidity Manual. Applied Modelling and Computation Group, Department of Earth Science and Engineering, South Kensington Campus, Imperial College London, London, SW7 2AZ, UK, version 4.1.8.2 edn. (March 2013), http://launchpad.net/fluidity/4.1/4.1.8.2/+download/fluidity-manual-4.1.8.2.pdf

[3] Fujitsu PRIMEHPC FX10 (March 2013), http://www.fujitsu.com/global/services/solutions/tc/hpc/products/primehpc/spec/

[4] Balaji, P., Buntinas, D., Goodell, D., Gropp, W., Kumar, S., Lusk, E., Thakur, R., Träff, J.L.: MPI on a Million Processors. In: Ropo, M., Westerholm, J., Dongarra, J. (eds.) PVM/MPI. LNCS, vol. 5759, pp. 20–30. Springer, Heidelberg (2009)

[5] Balay, S., Brown, J., Buschelman, K., Eijkhout, V., Gropp, W.D., Kaushik, D., Knepley, M.G., McInnes, L.C., Smith, B.F., Zhang, H.: PETSc users manual. Tech. Rep. ANL-95/11 - Revision 3.3, Argonne National Laboratory (2012)

[6] Balay, S., Gropp, W.D., McInnes, L.C., Smith, B.F.: Efficient management of parallelism in object oriented numerical software libraries. In: Arge, E., Bruaset, A.M., Langtangen, H.P. (eds.) Modern Software Tools in Scientific Computing, pp. 163–202. Birkhäuser Press (1997)

[7] Bell, N., Garland, M.: Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC 2009, pp. 18:1–18:11. ACM, New York (2009)

[8] Butler, M., Barnes, L., Sarma, D.D., Gelinas, B.: Bulldozer: An approach to multithreaded compute performance. IEEE Micro 31(2), 6–15 (2011)

[9] Goumas, G., Kourtis, K., Anastopoulos, N., Karakasis, V., Koziris, N.: Performance evaluation of the sparse matrix-vector multiplication on modern architectures. The Journal of Supercomputing 50, 36–77 (2009)

[10] Piggott, M.D., Gorman, G.J., Pain, C.C., Allison, P.A., Candy, A.S., Martin, B.T., Wells, M.R.: A new computational framework for multi-scale ocean modelling based on adapting unstructured meshes. International Journal for Numerical Methods in Fluids 56(8), 1003–1015 (2008)

[11] Rabenseifner, R., Hager, G., Jost, G.: Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. In: 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, pp. 427–436 (2009)
[12] Reid, F.J.L., Bull, J.M.: OpenMP microbenchmarks version 2.0. In: European Workshop on OpenMP, EWOMP (2004)
[13] Schubert, G., Fehske, H., Hager, G., Wellein, G.: Hybrid-parallel sparse matrix-vector multiplication with explicit communication overlap on current multicore-based systems. Parallel Processing Letters 21(3), 339–358 (2011)
[14] Wellein, G., Hager, G., Basermann, A., Fehske, H.: Fast sparse matrix-vector multiplication for teraflop/s computers. In: Palma, J.M.L.M., Sousa, A.A., Dongarra, J., Hernández, V. (eds.) VECPAR 2002. LNCS, vol. 2565, pp. 287–301. Springer, Heidelberg (2003)
[15] Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., Demmel, J.: Optimization of sparse matrix-vector multiplication on emerging multicore platforms. Parallel Computing 35(3), 178–194 (2009)

Designing Scalable Graph500 Benchmark with Hybrid MPI+OpenSHMEM Programming Models

Jithin Jose[1], Sreeram Potluri[1], Karen Tomko[2], and Dhabaleswar K. Panda[1]

[1] Department of Computer Science and Engineering, The Ohio State University,
{jose,potluri,panda}@cse.ohio-state.edu
[2] Ohio Supercomputer Center, Columbus, OH, USA
ktomko@osc.edu

Abstract. MPI has been the de-facto programming model for scientific parallel applications. However, it is hard to extract the maximum performance for irregular data-driven applications using MPI. The Partitioned Global Address Space (PGAS) programming models present an alternative approach to improve programmability. The lower overhead in one-sided communication and the global view of data in PGAS models have the potential to increase the performance at scale. In this study, we take up 'Concurrent Search' kernel of Graph500 — a highly data driven irregular benchmark — and redesign it using both MPI and OpenSHMEM constructs. We also implement load balancing in Graph500. Our performance evaluations using MVAPICH2-X (Unified MPI+PGAS Communication Runtime over InfiniBand) indicate a 59% reduction in execution time for the hybrid design, compared to the best performing MPI based design at 8,192 cores.

Keywords: Graph500, MPI, OpenSHMEM, Hybrid.

1 Introduction

Most traditional High Performance Computing (HPC) applications and current petascale applications are written using the Message Passing Interface (MPI) programming model. For instance, all of the HPC best practice case studies presented by the HPC Advisory Council [6] are MPI applications. However, it can be very difficult to use MPI and maintain performance for applications with irregular and dynamic communication patterns [1]. The emerging Partitioned Global Address Space (PGAS) programming models, such as OpenSHMEM [14] and Unified Parallel C (UPC) [23], present a flexible way for these applications to express parallelism using one-sided communication semantics and a global view of data. However, PGAS models are still emerging, and it is unlikely that entire applications will be re-written with these models. Instead, it is more likely that applications will continue to be written with MPI as the primary model, but parts of them will be re-designed with newer models, resulting in hybrid MPI+PGAS designs. Such designs can leverage the best features from each

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 109–124, 2013.

model. The Exascale roadmap identifies the hybrid model as the 'practical' way of programming exascale systems [5].

The Graph500 [20] benchmark is designed to represent the subclass of data intensive and irregular applications that use graph algorithm-based processing methods. Many emerging application areas in bioinformatics and life sciences, social networking, data mining, and security/intelligence rely on graph algorithmic methods. Earlier researchers [18,21] have indicated scalability limitations of the MPI-based Graph500 implementations. This leads to a broad challenge - *Can a high performance and scalable Graph500 benchmark be designed using MPI and PGAS models?*

In this study, we perform detailed profiling and evaluation of Graph500 and expose major performance bottlenecks, such as overhead due to MPI_Test and lack of computation-communication overlap. We redesign the Concurrent Search kernel of Graph500 benchmark using MPI and OpenSHMEM constructs resulting in a 'hybrid' benchmark. We use OpenSHMEM one-sided data movement and atomic routines, in addition to advanced MPI-3 features in our design. Further, we enhance our design with load balancing. Performance evaluations using MVAPICH2-X [13] (a unified MPI+PGAS communication runtime over InfiniBand) shows reduction in Graph500 traversal time by 59%, compared to the best performing MPI design at 8,192 cores. Further, the hybrid design performs 8 times better than the MPI design having the same communication pattern and volume. Our scalability analysis reveals that the hybrid design demonstrates good scaling (both weak and strong).

We also evaluate our hybrid design using separate communication runtimes (GASNet [2] for OpenSHMEM and MVAPICH2-X for MPI), which emphasizes the need for a unified communication runtime. To the best of our knowledge this is the first hybrid design of Graph500 using MPI and OpenSHMEM.

The following statements detail the main contributions of this study:

1. Identifying major bottlenecks in the MPI-based implementation of Graph500 Concurrent Search kernel by detailed profiling and analysis
2. Identifying critical design challenges for efficient one-sided communication with maximum computation-communication overlap and addressing them
3. Designing a scalable and high performance hybrid Graph500 benchmark with MPI and OpenSHMEM constructs
4. Designing an efficient load balancing for Concurrent Search kernel using PGAS constructs
5. In-depth performance evaluation and scalability analysis of the hybrid design

The rest of the paper is organized as follows. Section 2 provides a high level overview of the Graph500 benchmark, OpenSHMEM and the MVAPICH2-X unified communication runtime. In Section 3, we discuss the MPI based Graph500 implementation and expose major bottlenecks. In Section 4, we discuss the design challenges and present the hybrid MPI+OpenSHMEM design of Graph500. Section 5 presents the performance evaluations of the hybrid design. Finally, we discuss our future work and conclude in Section 7.

2 Background

2.1 Graph500 Benchmark

The Graph500 Benchmark [20] consists of three comprehensive benchmarks to address application kernels: Concurrent Search, Single Source Shortest Path and Maximal Independent Set. We focus on the Concurrent Search benchmark in this study, which is fundamentally a Breadth First Search (BFS) traversal of the graph. The Concurrent Search benchmark consists of three phases (termed, sub-kernels, in the benchmark specification). The 'Graph Construction' sub-kernel constructs graph in Compressed Sparse Row (CSR) format. The second sub-kernel is the actual 'Breadth-First-Search'. The final sub-kernel validates the BFS traversal. The Graph500 problem size is represented using Scale and Edge Factor. Scale is logarithm base two of the number of vertices; and, edge-factor is the ratio of the graph's edge count to its vertex count. Thus Scale = N and and Edge factor = M indicates a graph with 2^N vertices and 2^N * M edges.

2.2 OpenSHMEM

SHMEM (SHared MEMory) [17] is a library-based approach to realize the PGAS model and it offers one-sided, point-to-point communication operations, along with collective and synchronization primitives. There are several implementations of the SHMEM model that are customized for different platforms. However, these implementations are not portable. OpenSHMEM [14] aims to create a new, open specification to standardize the SHMEM model to achieve performance, programmability, and portability.

2.3 MVAPICH2-X Unified Communication Runtime

MVAPICH2-X [13] provides a unified high-performance runtime that supports both MPI and PGAS programming models on InfiniBand clusters. It enables developers to port parts of large MPI applications that are suited for the PGAS programming model. This minimizes the development overheads that have been a substantial deterrent in porting MPI applications to PGAS models. The unified runtime also delivers superior performance compared to using separate MPI and PGAS libraries by optimizing use of network and memory resources [9,8]. MVAPICH2-X is derived from the popular MVAPICH2 library and inherits all the features for performance and scalability of MPI communication.

3 Bottlenecks in Graph500 MPI Version

Graph500 provides four MPI based reference implementations: MPI_Simple, MPI_CSR (Replicated Compressed Sparse Row), MPI_CSC (Replicated Compressed Sparse Column), and MPI_OneSided. All of these implementations use the level synchronized BFS traversal algorithm [18].

The MPI_Simple implementation is listed in Algorithm 1. In this implementation, each MPI process maintains two queues, `CurrQueue` and `NewQueue`, and

Algorithm 1. MPI_Simple BFS Traversal

```
1: pred[root] ← 0, all_done ← 0
2: Enqueue(CurrQueue, root)
3: {Procedure: HandleReceive}
4: if rcv_count = 0 then
5:    all_done ← all_done + 1
6: else if received then
7:    for each received edge (u,v) do
8:       if visited[v] = 0 then
9:          visited[v] ← 1, pred[v] ← u
10:         Enqueue(NewQueue, v)
11:      end if
12:   end for
13: end if
14: {End Procedure}
15: while true do
16:   while CurrQueue not empty do
17:      for all vertex u in CurrQueue do
18:         HandleReceive()
19:         u ← Dequeue(CurrQueue)
20:         owner ← find_owner()
21:         if owner = me then
22:            visited[v] ← 1, pred[v] ← u
23:            Enqueue(NewQueue, v)
24:         else
25:            Send (u,v) to owner
26:         end if
27:      end for
28:   end while
29:   Send empty messages to all others
30:   while all_done ≠ N − 1 do
31:      HandleReceive()
32:   end while
33:   AllReduce NewQueue.length
34:   if NewQueue is empty in all processes then
35:      break
36:   end if
37:   Swap(CurrQueue, NewQueue)
38:   NewQueue ← empty
39:   all_done ← 0
40: end while
```

two arrays — `pred` and `visited` — to store predecessor information and to track whether or not each vertex has been visited. The vertices are evenly distributed

among participating processes, and only the owner process has complete information regarding adjacency list, visited array, and predecessor array for owned nodes. Initially, the 'root' vertex is inserted into `CurrQueue` by the owner process of 'root'. An iteration of the main while loop (lines 15 to 40) corresponds to a level in the BFS traversal. In each level, the adjacent vertices of all the vertices in `CurrQueue` are discovered. Newly discovered edges are coalesced and sent to the corresponding owner processes using MPI_Send. Each process periodically checks for incoming data using MPI_Test / MPI_Recv and processes it (indicated as HandleReceive). Unvisited vertices in the incoming data packet are added to `NewQueue`. After processing vertices in `CurrQueue`, every process sends an empty message to all of the others to indicate end-of-level and waits until it receives empty messages from all other processes. After this 'implicit' barrier, an MPI_AllReduce (sum) is performed over the size of `NewQueue`. A non-zero sum indicates that there exists at least one process that has to process vertices in the next level. In this case, `NewQueue` and `CurrQueue` are swapped and the loop is repeated; otherwise, the algorithm ends.

Table 1. Time Spent in MPI Routines for MPI_Simple Implementation

MPI Routine	Total Time (us)
MPI_GetCount	1.5
MPI_IRecv	20.4
MPI_ISend	109.0
MPI_AllReduce	258.0
MPI_Test	1100.0
Total BFS Time	2040.0

The following are the three main bottlenecks in this implementation.
Overhead in Send/Recv Communication Model: Even though non-blocking MPI_ISend and MPI_IRecv are used, a lot of CPU cycles are consumed for the actual communication. To analyze this, we profiled MPI calls in the MPI_Simple implementation for 128 processes (Scale=26). The results shown in Table 1 indicate that more than 50% of total BFS time is spent in the MPI_Test call.
Implicit Linear Barrier: The implicit barrier at the end of each level is linear in nature. At a large scale, this linear barrier can cause significant overheads.
Lack of Overlap: Even though the MPI_Simple implementation processes received edges while MPI_ISend/ MPI_IRecv calls are in progress, the actual overlap of computation-communication is low. CPU cycles spent in the MPI communication library reduce the effective computation-communication overlap.

The MPI_CSR and MPI_CSC versions employ slightly different algorithms. In MPI_CSR, each process holds the entire graph data as a bit array. Similarly in MPI_CSC, each process has information as to whether any vertex exists in current level queue. These versions do not communicate during the level, but do an MPI_AllGather at the end of each level [18]. The MPI_OneSided version uses MPI one-sided operations for implementing BFS. The current implementation of MPI_OneSided exhibits poor performance and is excluded from our evaluations.

Since the proposed hybrid design is based on MPI_Simple, the implementation details of the other versions are not discussed in detail. However, we compare our proposed design with MPI_Simple, MPI_CSR, and MPI_CSC versions.

4 Design and Implementation

Redesigning an MPI benchmark into a hybrid MPI+Open-SHMEM version is not trivial. A common fallacy is that replacing MPI communication routines with one-sided routines and converting the data into global arrays will just improve the performance. However, in order to extract the best performance and scalability, a meticulous design which can attain maximum communication computation overlap and reduce communication overheads is imperative. We list the challenges in designing an efficient and scalable hybrid Graph500 benchmark, discuss how we overcome these challenges, and then present our design.

4.1 Design Challenges

Co-ordination between Sender and Receiver Processes: In one-sided communication semantics, the receiver process is not involved in the data transfer. So, how does the receiver process know whether the data packet has arrived? Further, how does it make sure that the entire packet has arrived and when can it start processing the data?
Co-ordination between Multiple Sender Processes: Given the irregular nature of Graph500, how do multiple sender processes coordinate access to the target data buffer while using one-sided operations? Keeping separate receive buffers for each sender process or using locks will limit scalability [10].
Coalescing and Optimal Data Transfer Size: Since the underlying communication operations are different for MPI send/receive and Remote Direct Memory Access (RDMA) based one-sided semantics, the optimal coalescing size needs to be determined for each.
Memory Scalability: While using one-sided operations for communication, sufficient buffer space must be allocated for remote memory operations. If the buffer requirement increases linearly with scale, the application will not scale. Reusing buffer requires extra synchronization between the sender and receiver processes and might incur additional overheads. Thus, optimal receive buffer size has to be determined.
Synchronization at the End of Each Level: We discussed in the previous section that the linear barrier causes scalability limitations. Using an MPI_Barrier or shmem_barrier (implemented using tree-based algorithms) can improve scalability, but it limits the computation-communication overlap during the barrier. Thus, it is critical to achieve synchronization without compromising computation - communication overlap.
Load Imbalance: Because of the level synchronization algorithm, any skew in the computation among processes results in a higher synchronization time. Thus, it is imperative to analyze the load imbalance and reduce it as much as possible.

4.2 Detailed Design

We keep the same level-synchronized BFS algorithm in our hybrid version. Thus, the communication pattern and volume in both the MPI_Simple and the Hybrid versions are the same. We follow a flat execution model [4] for the hybrid design i.e., MPI rank and OpenSHMEM rank of the hybrid program are kept the same. **Communication Using One-Sided Routines:** In the Hybrid design, we use OpenSHMEM one-sided routine (`shmem_put`) for communication. During initialization, every process allocates a globally shared buffer (allocated using `shmalloc`) called `receive_buffer`. Any process can read/write, from/to this buffer using OpenSHMEM routines. The size of the receive buffer is calculated based on the number of local edges that the process owns. Therefore the size is not dependent on the system scale and does not impose scalability concerns. Like the MPI_Simple design, the hybrid design also employs coalescing of edges. **Co-ordination Using Fetch-Add Atomic Operation:** We use the OpenSHMEM atomic fetch-add (`shmem_fadd`) operation for coordinating between sender and receiver, as well as between multiple senders. The `shmem_fadd` atomically updates (add) the remote data and the previous value is returned. Each process maintains a globally shared variable `receive_index`, for its `receive_buffer`. Whenever a remote process wants to write data to the `receive_buffer`, it first atomically fetch-adds the `receive_index` with the write data size. After the execution of `shmem_fadd`, the remote process owns the region in `receive_buffer` and can safely write the data. Thus, an atomic fetch-add followed by a put operation achieves synchronization between sender and receiver and also between multiple senders and receiver. The `receive_index` is reset at the end of each level so that the `receive_buffer` is reused at each level.

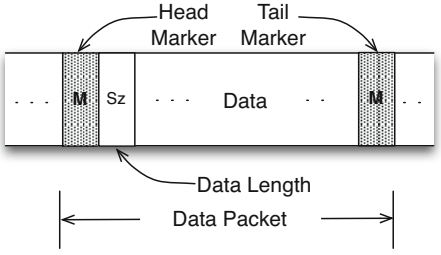
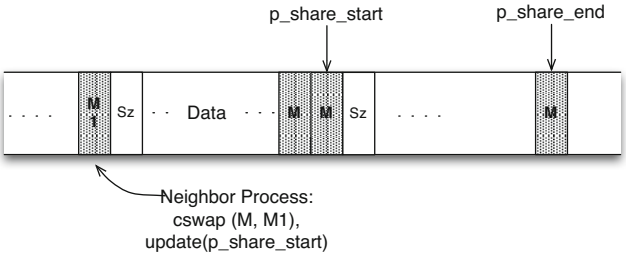


Fig. 1. Buffer Structure for Polling

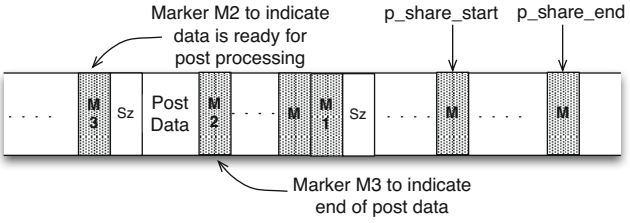
Buffer Structure for Computation-Communication Overlap: Since the receiver process is not notified when data arrives, we introduced the buffer structure shown in Figure 1; remote processes write into `receive_buffer` in this format. The head and tail markers (M) indicate the beginning and end of the buffer, and the size flag (Sz) indicates the data size. The receiver process can poll on the head marker for incoming data. If the head marker is present, it will check whether the tail marker is set at an offset indicated by data size. If both markers are set, then the receiver can start processing the data. The markers are cleared after processing so that they do not signal false data arrival in the next level.

Level Synchronization Using Non-blocking Barrier: We use an MPI-3 non-blocking barrier for level synchronization in our hybrid design. This allows each process to enter the barrier and still continue to receive and process edges. **Load Balancing:** Processing of edge `(u,v)` includes updating the `pred` array, adding into `NewQueue`, and updating the `visited` array if the vertex `v` has not been visited yet. Data structures — predecessor array (`pred`), bitwise visited array (`visited`), and `NewQueue` — are kept local to the owner process. Therefore, for remote process to share the workload, it should first get the data block for processing, and update the data structures in a mutually exclusive manner. These dependencies limit the scope for load balancing in Graph500. As a result, we restrict load balancing within a node. The `shmem_ptr` routine in OpenSHMEM allows processes within a node to access globally shared memory by direct memory loads and stores. We exploit this feature in OpenSHMEM to implement load balancing. In our hybrid design, the `pred`, `visited` and `NewQueue` data structures are allocated in shared memory.

(a) Overloaded process exposes work



(b) One of the idle neighbor processes picks up data packet



(c) Neighbor process puts back data for post processing

Fig. 2. Load Balancing

When a process identifies that it has more work than a predefined threshold, it exposes a portion of its `receive_buffer`. If the process that exposed work becomes idle, it can re-acquire work from its shared region. The exposed region

is identified using shared variables — `p_share_start` and `p_share_stop`. Idle processes grab a portion of the exposed `receive_buffer` in a mutually exclusive manner. We considered two design alternatives for this — using OpenSHMEM atomic routines and using shared memory mutex.

Figure 2 illustrates mutual exclusion using OpenSHMEM atomics. An idle process does a compare-swap, `shmem_cswap`, operation at the location indexed by `p_share_start`, that conditionally swaps markers from head marker (M) to M1. A successful operation indicates that the idle process has gotten the chunk, and it atomically updates (using `shmem_add`) `p_share_start`. Since OpenSHMEM atomic routines are implemented over RDMA atomics, this design relies on the Network Interface Controller for mutual exclusion. In the shared-memory mutex-based design, every process exposes a mutex that is allocated in a globally shared region. An idle process locks the corresponding mutex of an overloaded process to grab a portion of its `receive_buffer`.

The update operations in processing, which must be mutually exclusive with those of owner processes, will further slow down the overloaded process. So, we define processing as checking the `visited` array and removing all edge information, that has already been visited. Only unvisited vertices are kept for post processing by the owner process. This is indicated as 'Post Data' in Figure 2(c). The end of post data is denoted by marker M2. After preparing the post data, a special marker M3 is set at the beginning of the data packet to indicate data is ready for post-processing. The owner process keeps track of shared work and finally processes the post-data.

5 Experimental Evaluation

5.1 Experiment Setup

We used two clusters for performance evaluations.

Cluster A: This cluster (TACC Stampede [19]) is equipped with compute nodes with Intel Sandybridge series of processors using Xeon dual eight-core sockets, operating at 2.70 GHz with 32 GB RAM. Each node is equipped with MT4099 FDR ConnectX HCAs (54 Gbps data rate) with PCI-Ex Gen3 interfaces. The operating system used is CentOS release 6.3, with kernel version 2.6.32-279.el6 and OpenFabrics version 1.5.4.1. Even though this system has large number of cores, we were able to gain access to only 8,192 cores for running experiments for this paper.

Cluster B: This cluster consists of 144 compute nodes with Intel Westmere series of processors using Xeon Dual quad-core processor nodes operating at 2.67 GHz with 12 GB RAM. Each node is equipped with MT26428 QDR ConnectX HCAs (32 Gbps data rate) with PCI-Ex Gen2 interfaces. The operating system used is Red Hat Enterprise Linux Server release 6.3 (Santiago), with kernel version 2.6.32-71.el6 and OpenFabrics version 1.5.3-3.

We used Graph500 v2.1.4 in our experiment evaluations. We used OpenSHMEM (v1.0d) [14] over GASNet (v1.20.0) [2] and MVAPICH2-X OpenSHMEM (v1.9a2) [13] as OpenSHMEM stacks. In all our Graph500 experiments, we kept the Edge Factor as 16.

Tuning Optimal Parameters: We tuned the optimal coalescing size for sending edges in the MPI_Simple and hybrid designs. Edges are represented as two `int64_t` elements (16 bytes). A coalescing size of 256 indicates that the edge information is sent in 4,096 byte (256*16) packets. We measured the BFS traversal times for different coalescing sizes. The optimal coalescing size identified for both *MPI_Simple* and *Hybrid* is 1,024 (16 KB), on Clusters A and B.

MPI_CSR and MPI_CSC versions are compute-intensive and rely on OpenMP threads to exploit parallelism [18]. We tuned *MPI_CSR* and *MPI_CSC* configurations with respect to the number of processes per node and the number of OpenMP threads per process. A configuration with eight processes per node and two OpenMP threads per process performed best for '16-core per node' Cluster A. Similarly, configuration with four process per node and two OpenMP threads per process performed best on '8-core per node' Cluster B. We used these optimal values for all of our experiments. The MPI_Simple and Hybrid versions are executed using one process per core configurations.

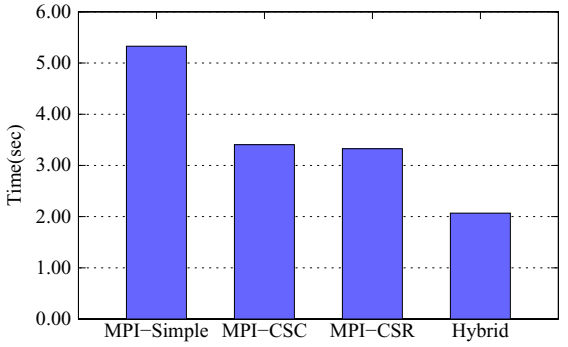
5.2 Performance Evaluation

Figure 3 presents the performance results of 'Hybrid' MPI+OpenSHMEM versions and compares our design with pure MPI based versions. These graphs report the execution time for BFS traversal. We present results with 1,024 and 2,048 cores in Figures 3(a) and 3(b), respectively. These experiments were run on Cluster A. It can be observed from the figure that the 'MPI_CSC' and 'MPI_CSR' versions perform better than the 'MPI_Simple' version. But the 'Hybrid' version outperforms all of the MPI versions. With 2,048 cores, the time taken by the best performing MPI version ('MPI_CSR') was 3.13 seconds, where as the hybrid version took just 1.67 seconds. This is about 47% reduction in execution time.

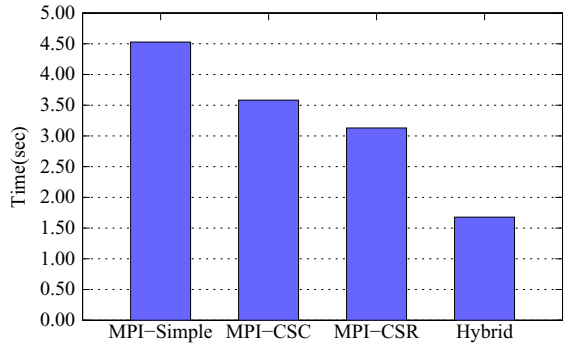
The communication pattern and volume in 'MPI_Simple' and 'Hybrid' versions are identical. It is the lower overhead, associated with one-sided communication calls in OpenSHMEM and efficient communication-computation overlap in the hybrid design, that resulted in better performance. In effect, the 'Hybrid' design reduced the total execution time from 4.5 seconds to 1.67 seconds.

Performance Evaluation - Separate Runtimes vs. Unified Communication Runtime: In this section, we compare the performance of the Hybrid design executing over separate runtimes for MPI and OpenSHMEM, versus a unified communication runtime supporting both models.

Results are presented in Figure 4. The former is denoted as 'Hybrid_GASNet' and the latter is denoted as 'Hybrid_MV2X'. The experiment was conducted on Cluster B. We used a graph with scale = 26 for this experiment. The figure demonstrates that the Hybrid_MV2X version performs significantly better than Hybrid_GASNet. For 1,024 processes, Hybrid-_GASNet took 22.8 seconds and Hybrid_MV2X took just 0.58 seconds. The difference in performance is due to the following reasons: The OpenSHMEM implementation over GASNet lacks efficient atomic routine implementation compared to MVAPICH2-X. The hybrid design relies on the atomic `fetch-add` operation for acquiring a data region at the destination process. For each data transfer, there exists a `fetch-add`



(a) 1,024 Cores



(b) 2,048 Cores

Fig. 3. Performance Evaluation (Scale = 29)

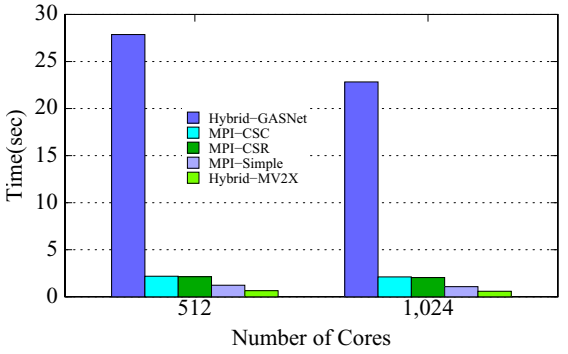


Fig. 4. Performance Comparison with OpenSHMEM over GASNet (Scale=26)

operation. The second reason is the overhead caused by executing two runtimes and extra network resource usage [8].

Interestingly, if the hybrid design is evaluated using separate runtimes, then the observed performance is quite low. This could mislead researchers to draw incorrect conclusions about the potential of hybrid designs. On the other hand, a unified communication runtime enables a true comparison between programming models, rather than comparing their runtimes. Note also, that at scale=26 as shown in Figure 4 MPI_Simple performs better than the MPI_CSR and MPI_CSC versions unlike what is shown in Figure 3 for a larger graph.

5.3 Evaluation of Load Balancing

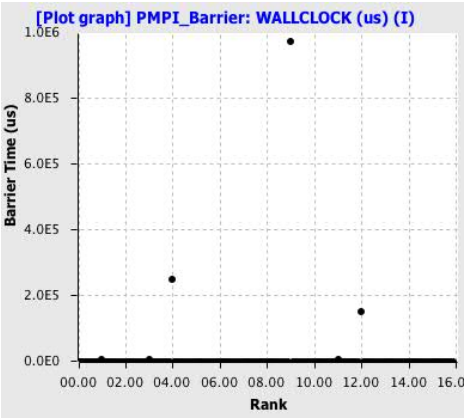
In Section 4.2, we discussed two design alternatives for load balancing - using pthread_mutex and using RDMA atomics. Our evaluations indicated similar performance results for both alternatives. Further, the mutex based design blocks on the lock; however if compare-swap fails, the process can proceed immediately. Because of this, we chose the non-CPU intensive RDMA atomics based scheme in our design. To measure the impact of load balancing in Graph500, we inserted an `MPI_Barrier` at the end of each level and measured the time for barrier at each process. This barrier time represents the load imbalance.

We executed this modified benchmark on Cluster A and measured the total barrier time at each process in a node using HPCToolKit [7]. Figure 5(a) presents these results. We can see that three of the processes take higher time, while the other processes finish barrier almost immediately. This indicates that those three processes have comparatively lesser work. We enabled the work sharing and evaluated load imbalance using the same experiment. The results shown in Figure 5(b) indicate that the work is more balanced now. However, even with work sharing enabled, we observed very little improvement in overall performance (Figure 8). To investigate this further, we measured the total amount of work at each level. Figure 6 presents this data. The results indicate that the amount of work imbalance is less and all the participating processes are busy operating on their own data. Further, the dependency on the vertex owner for processing a vertex and the higher cost for sharing work across nodes, limits the scope for work sharing in Graph500.

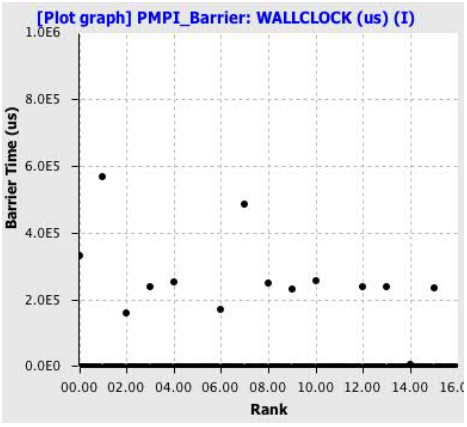
5.4 Scalability Analysis

Weak scalability results are presented in Figure 7(b). In this experiment, we kept a constant problem size per processor core as Scale=26 per 1,024 cores. As discussed in Section 2, with every step increase in Scale, the problem size doubles. Thus, we doubled the system size with every step in problem size. We can observe that the hybrid version of Graph500 achieves better weak scaling results. Results at larger scale indicate that the hybrid design imposes no overheads with increase in system size.

Figure 8 demonstrates the BFS traversal time with 8,192 processor cores. At this scale, the traversal time for MPI_Simple, MPI_CSC and MPI_CSR versions are 8.32, 2.83, and 2.68 seconds, respectively. Traversal time for the hybrid version with and without work sharing is 1.10 and 1.12 seconds. Hybrid design



(a) Without Work Sharing



(b) With Work Sharing

Fig. 5. Effect of Work Sharing

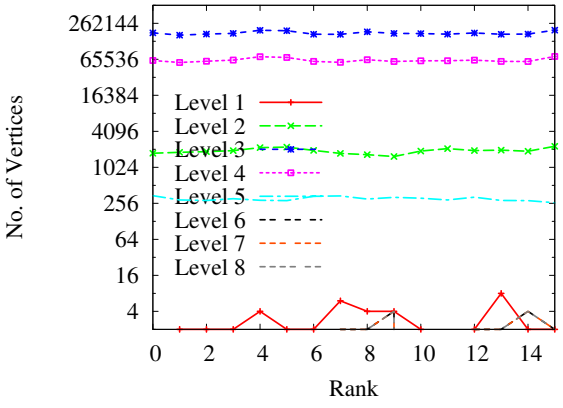
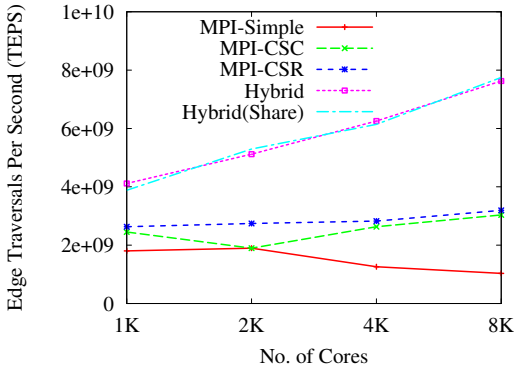
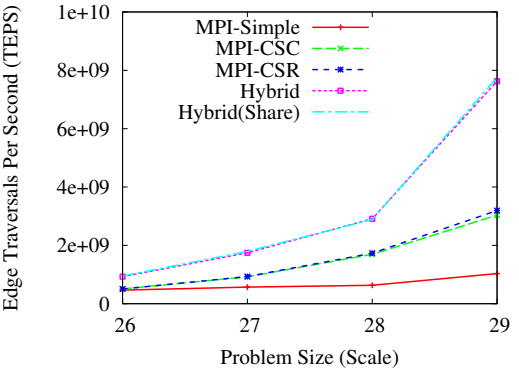


Fig. 6. Per Level Work Distribution

We present strong and weak scalability evaluation results and finally present the performance results at 8,192 cores. These experiments were executed on Cluster A. Figure 7(a) depicts the strong scalability results. In this experiment, we kept the constant problem size (Scale = 29) and varied the scale of the system from 1,024 cores to 8,192 cores. For each scale, we measured and reported the number of edge traversals per second (TEPS). The performance results indicate that the hybrid design exhibits very good strong scalability. For MPI versions, the performance does not increase much as the system size increases because of higher communication overheads at larger scales.



(a) Strong Scaling (Scale=29)



(b) Weak Scaling

Fig. 7. Scalability Results

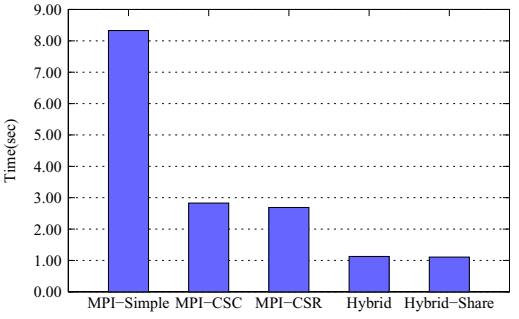


Fig. 8. Performance Evaluation with 8,192 Processes (Scale = 29)

achieves around 59% reduction in BFS traversal time over the best performing MPI version. As mentioned earlier, the communication volume and pattern is exactly the same in both the MPI_Simple and the hybrid designs. Thus, the hybrid design is able to reduce the overall execution time from 8.32 to 1.10 seconds, which is about 8X improvement in performance.

6 Related Work

Dinan et.al proposed different hybrid program execution models [4]. Jose et.al proposed a Unified Communication Runtime [10,8] in MVAPICH2-X for supporting hybrid MPI+UPC and MPI+OpenSHMEM models. Studies [16,15] discuss converting MPI applications to PGAS models, but focus only on changing the communication to one-sided semantics. Suzumura et.al studied Graph500 performance characteristics in detail [18]. Several algorithmic enhancements for Graph500 were also proposed [22,21,10]. Work sharing in PGAS models also have been studied [3,12], but these case studies did not have the dependency on owner process data as in the Graph500 case.

7 Conclusion and Future Work

We presented a detailed analysis of existing MPI based Graph500 implementation and exposed critical bottlenecks and presented a scalable and high performance design using MPI and OpenSHMEM constructs. Performance evaluations using MVAPICH2-X Unified Runtime show a reduction in Graph500 traversal time by 59%, compared to the best performing MPI design at 8,192 cores. Scalability analysis indicates that the hybrid design demonstrates good strong and weak scaling characteristics. At this scale, hybrid design performs 8X better than the MPI design which has the same communication pattern and volume. Our evaluations with a unified runtime and with separate runtimes highlight the need for a unified communication runtime for hybrid programming models.

Our intention is not to compare MPI and PGAS models and declare that one is better than the other. MPI also supports one-sided communication semantics using `MPI_Put / MPI_Get` routines and enables shared memory access using `MPI_Alloc_mem`. MPI-3 [11] has proposed several rich features and extensions. The global shared address space abstraction provided by PGAS models improves productivity. Our aim in this study is to show the potential benefits of a hybrid programming model that combines the best features of both.

We plan to continue working along these directions. We plan to evaluate our design at larger scale. We would also like to redesign real world MPI applications using hybrid constructs and showcase the benefits. Further, we would like to enhance our load balancing scheme and evaluate it with applications.

Acknowledgment. This work is supported in part by National Science Foundation grants #OCI-0926691, #OCI-1148371 and #CCF-1213084. It used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575.

References

1. Basumallik, A., Eigenmann, R.: Optimizing Irregular Shared-memory Applications for Distributed-memory Systems. In: Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2006 (2006)
2. Bonachea, D.: GASNet Specification v1.1. Tech. Rep. UCB/CSD-02-1207, U. C. Berkeley (2008)
3. Dinan, J., Larkins, D.B., Sadayappan, P., Krishnamoorthy, S., Nieplocha, J.: Scalable Work Stealing. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (2009)
4. Dinan, J., Balaji, P., Lusk, E., Sadayappan, P., Thakur, R.: Hybrid Parallel Programming with MPI and Unified Parallel C. In: Proceedings of the 7th ACM International Conference on Computing Frontiers, CF 2010 (2010)
5. Dongarra, J., Beckman, P., Moore, T., Patrick, Aerts, e.a.: The International Exascale Software Project Roadmap. Int. J. High Perform. Comput. Appl. 25(1), 3–60 (2011), http://dx.doi.org/10.1177/1094342010391989
6. HPC Advisory Council, http://www.hpcadvisorycouncil.com
7. HPCToolkit, http://hpctoolkit.org/
8. Jose, J., Kandalla, K., Luo, M., Panda, D.: Supporting Hybrid MPI and OpenSHMEM over InfiniBand: Design and Performance Evaluation. In: 2012 41st International Conference on Parallel Processing, ICPP (2012)
9. Jose, J., Luo, M., Sur, S., Panda, D.K.: Unifying UPC and MPI Runtimes: Experience with MVAPICH. In: PGAS (2010)
10. Jose, J., Potluri, S., Luo, M., Sur, S., Panda, D.K.: UPC Queues for Scalable Graph Traversals: Design and Evaluation on InfiniBand Clusters. In: PGAS (2011)
11. Message Passing Interface Forum, http://www.mpi-forum.org/
12. Min, S.J., Iancu, C., Yelick, K.: Hierarchical Work Stealing on Manycore Clusters. In: PGAS (2011)
13. MVAPICH2-X: Unified MPI+PGAS Communication Runtime over OpenFabrics/Gen2 for Exascale Systems, http://mvapich.cse.ohio-state.edu/
14. OpenSHMEM, http://openshmem.org/
15. Preissl, R., Shalf, J., Wichmann, N., Long, B., Ethier, S.: Advanced Communication Techniques for Gyrokinetic Fusion Applications on Ultra-Scale Platforms. In: PGAS (2011)
16. Shan, H., Austin, B., Wright, N.J., Strohmaier, E., Shalf, J., Yelick, K.: Accelerating Applications at Scale Using One-Sided Communication. In: PGAS (2012)
17. Silicon Graphics International.: SHMEM API for Parallel Programming, http://www.shmem.org/
18. Suzumura, T., Ueno, K., Sato, H., Fujisawa, K., Matsuoka, S.: Performance Characteristics of Graph500 on Large-scale Distributed Environment. In: 2011 IEEE International Symposium on Workload Characterization, IISWC (2011)
19. TACC Stampede Cluster, http://www.xsede.org/resources/overview
20. The Graph500, http://www.graph500.org
21. Ueno, K., Suzumura, T.: 2D Partitioning Based Graph Search for the Graph500 Benchmark. In: 2012 IEEE 26th International on Parallel and Distributed Processing Symposium Workshops PhD Forum, IPDPSW (2012)
22. Ueno, K., Suzumura, T.: Highly Scalable Graph Search for the Graph500 Benchmark. In: Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2012 (2012)
23. UPC Consortium: UPC Language Specifications, v1.2. Tech. Rep. LBNL-59208, Lawrence Berkeley National Lab (2005)

On the GPU Performance of 3D Stencil Computations Implemented in OpenCL

Huayou Su[1,2,3], Nan Wu[1,2], Mei Wen[1], Chunyuan Zhang[1], and Xing Cai[2,3]

[1] School of Computer Science, National University of Defense Technology, Changsha, Hunan 410073, China
{shyou,nanwu,meiwen,cyzhang}@nudt.edu.cn
[2] Simula Research Laboratory, P.O. Box 134, 1325 Lysaker, Norway
xingca@simula.no
[3] Department of Informatics, University of Oslo, P.O. Box 1080 Blindern, 0316 Oslo, Norway

Abstract. Aiming at a close examination of the OpenCL performance myth, we study in this paper OpenCL implementations of several representative 3D stencil computations. It is found that typical optimization techniques such as array padding, plane sweeping and chunking give similar performance boosts to the OpenCL implementations, as those obtained in corresponding CUDA programs. The key to good performance lies in maximizing the use of on-chip resources of a GPU, same for both OpenCL and CUDA programming. In most cases, the achieved FLOPS rates on NVIDIA's Fermi and Kepler GPUs are fully comparable between the two programming alternatives. For four typical 3D stencil computations, the performance of the OpenCL implementations is on average 9% and 2% faster than that of the CUDA counterparts on GTX590 and Tesla K20, respectively. At the moment, the only clear advantage of CUDA programming for stencil computations arises from CUDA's ability of using the read-only data cache on NVIDIA's Kepler GPUs. The skepticism about OpenCL's GPU performance thus seems unjustified for 3D stencil computations.

Keywords: GPU programming, OpenCL, CUDA, stencil computations.

1 Introduction

Despite being hardware neutral and cross-platform portable, the OpenCL [1] programming standard has so far not enjoyed its expected popularity. There seems to be a skepticism about OpenCL's performance on particularly the NVIDIA GPU architectures, where CUDA is currently the dominating standard of programming. Although there has been some research (such as [2,3,4,5]) on the worthiness of OpenCL in comparison with CUDA, we believe that more rigorous comparisons are needed to fully investigate this subject.

This paper has thus chosen a specific domain of scientific computing, namely stencil computations. These computations lie in the heart of many simulations that involve structured computational meshes. The achieved efficiency of stencil

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 125–135, 2013.

computations closely reflects the quality of software implementation and optimization, boiling down to whether the targeted hardware architecture is effectively used. A thorough comparison between OpenCL and CUDA programming for 3D stencil computations, in respect of both programmability and performance on GPUs, can therefore tell us whether the prejudice against OpenCL is justifiable. We will report numerical experiments done on NVIDIA's Fermi and Kepler architectures. In the same process, we also want to show how typical performance optimization techniques can be implemented using OpenCL. This can be useful for novice OpenCL programmers.

2 Stencil Computations

The computations involved in a stencil method are in form of repeatedly sweeping through a structured computational mesh. During each sweep, calculation at one mesh point depends on a fixed number of neighboring mesh points, which form a computational stencil. Typical stencil shapes in 3D are *e.g.* 7-point, 13-point, 19-point, and 27-point. The obtainable efficiency of a stencil computation depends not only on the shape of the stencil, but also on the number of involved input and output arrays, in addition to the number of floating-point operations.

We have chosen to look at the following four representative examples of 3D stencil computations (the same as in [6]):

$$u^{\text{new}}_{i,j,k} = \alpha u_{i,j,k} + \beta \left(u_{i\pm1,j,k} + u_{i,j\pm1,k} + u_{i,j,k\pm1}\right), \qquad \text{(7PT-1)}$$

$$u^{\text{new}}_{i,j,k} = \alpha r_{i,j,k} + \beta \left(u_{i\pm1,j,k} + u_{i,j\pm1,k} + u_{i,j,k\pm1}\right), \qquad \text{(7PT-2)}$$

$$\begin{aligned} u^{\text{new}}_{i,j,k} = {} & \alpha \left[r_{i,j,k} + \beta \left(u_{i\pm1,j,k} + u_{i,j\pm1,k} + u_{i,j,k\pm1}\right)\right. \\ & \left. + \left(u_{i\pm1,j\pm1,k} + u_{i\pm1,j,k\pm1} + u_{i,j\pm1,k\pm1}\right)\right], \end{aligned} \qquad \text{(19PT)}$$

$$\begin{aligned} u^{\text{new}}_{i,j,k} = {} & u_{i,j,k} + r_{i,j,k} + \\ & \alpha \left[\left(\kappa_{i+1,j,k} + \kappa_{i,j,k}\right)\left(u_{i+1,j,k} - u_{i,j,k}\right)\right. \\ & - \left(\kappa_{i,j,k} + \kappa_{i-1,j,k}\right)\left(u_{i,j,k} - u_{i-1,j,k}\right) \\ & + \left(\kappa_{i,j+1,k} + \kappa_{i,j,k}\right)\left(u_{i,j+1,k} - u_{i,j,k}\right) \\ & - \left(\kappa_{i,j,k} + \kappa_{i,j-1,k}\right)\left(u_{i,j,k} - u_{i,j-1,k}\right) \\ & + \left(\kappa_{i,j,k+1} + \kappa_{i,j,k}\right)\left(u_{i,j,k+1} - u_{i,j,k}\right) \\ & \left. - \left(\kappa_{i,j,k} + \kappa_{i,j,k-1}\right)\left(u_{i,j,k} - u_{i,j,k-1}\right)\right]. \end{aligned} \qquad \text{(7PT-3)}$$

In all the above four stencil computations, α and β are scalar constants. Moreover, values of u^{new} (output) and u (input) are assumed to be stored in two separate 3D arrays. Note that 7PT-1 differs from 7PT-2 in that the latter needs an additional input 3D array r. The 19PT stencil is often used to obtain better accuracy than the 7-point stencils, at the cost of more floating-point operations and a larger footprint of the stencil. The computations used in 7PT-3 typically arise from solving a diffusion equation with a variable coefficient $\kappa(x, y, z)$, whose values are assumed to be stored in another 3D array. Table 1 gives a detailed comparison of these stencil computations.

Table 1. A detailed comparison of four 3D stencils

Stencil	# reads,writes	# operations	In,out arrays
7PT-1	7,1	2(∗), 6(+)	1,1
7PT-2	7,1	2(∗), 6(+)	2,1
19PT	19,1	2(∗), 18(+)	2,1
7PT-3	15,1	7(∗), 10(+), 9(−)	3,1

3 OpenCL Implementation and Optimizations

3.1 Basic Concepts in OpenCL Programming

The OpenCL framework [1] is made up of three parts: platform model, execution model and memory model. The platform model contains a host and one or several OpenCL devices, such as CPUs and GPUs. Each device has one or several compute units, and each compute unit includes one or several processing elements. An OpenCL program consists of two parts: a host program and one or several kernels. The host program executes on the host and is similar to a standard C program, apart from calling OpenCL APIs. The kernels execute on the devices according to the configuration of NDRange, which is the index space of the execution instance of the kernel code. The NDRange is in fact an N-dimensional grid of work-items—the smallest execution entities. Several work-items can be organized into a work-group, which is similar to a thread block in CUDA. The memory model of an OpenCL device is abstracted into four levels: global memory, constant memory, local memory and private memory. It should be noticed that the local memory in OpenCL is different from CUDA's local memory, but plays a similar role as CUDA's shared memory. Table 2 compares the different concepts used in OpenCL and CUDA.

Table 2. Comparing the different concepts used in OpenCL and CUDA

OpenCL	CUDA
Platform Model	*Hardware Model*
CPU+OpenCL devices	CPU+NVIDIA GPUs
Compute Units	Streaming Multiprocessors (SMs)
Processing Elements	CUDA cores
Execution Model	*Programming Model*
NDRange	grid index
work-group	thread block
work-item	thread
Memory Model	*Memory Model*
global memory	global memory
constant memory	constant memory
local memory	shared memory
private memory	local memory

3.2 Baseline OpenCL Implementation

Implementing an OpenCL kernel is very similar to CUDA programming. We will show in the following a baseline OpenCL kernel implementation for 7PT-1.

```
__kernel void stencil(__global double * device_u,
                      __global double * device_u_new,
                      const double alpha, const double beta,
                      int n_x, int n_y)
{
    int gid_x = get_global_id(0)+1;
    int gid_y = get_global_id(1)+1;
    int gid_z = get_global_id(2)+1;
    __global double (*in)[n_y][n_x];
    __global double (*out)[n_y][n_x];
    in  = (__global double (*)[n_y][n_x])device_u;
    out = (__global double (*)[n_y][n_x])device_u_new;
    out[gid_z][gid_y][gid_x]=(alpha*in[gid_z][gid_y][gid_x])+
                             beta*(in[gid_z][gid_y][gid_x-1]
                                  +in[gid_z][gid_y][gid_x+1]
                                  +in[gid_z][gid_y-1][gid_x]
                                  +in[gid_z][gid_y+1][gid_x]
                                  +in[gid_z-1][gid_y][gid_x]
                                  +in[gid_z+1][gid_y][gid_x]);
}
```

In the above baseline implementation, `device_u` and `device_u_new` are two flattened 1D arrays living on the device. The two integers `n_x` and `n_y` contain the x- and y-dimensions of the original 3D arrays, whose 3D images are reconstructed as `in` and `out` inside the kernel. Each OpenCL work-item will invoke the kernel to compute $u^{\text{new}}_{i,j,k}$ at a single mesh point, following the formula of 7PT-1. The corresponding 3D index (i, j, k) is conveniently acquired through OpenCL's standard `get_global_id` function. The resemblance between the OpenCL kernel and its CUDA counterpart is quite clear if we remember that an OpenCL work-item is the same as a thread in CUDA.

3.3 Plane Sweeping in the z-Direction (OPT-1)

The OpenCL kernel of the baseline implementation lets each work-item compute only a single value $u^{\text{new}}_{i,j,k}$. If the z-dimension of an OpenCL work-group is by convention chosen as 1, this baseline implementation offers no possibility of data reuse in the z-direction. One approach to improving this inefficiency is to adopt the technique of plane sweeping, see *e.g.* [7]. That is, the kernel computes a column of points in the z-direction instead. A for-loop sweeping the z-direction must thus be introduced into this enhanced kernel, in exactly the same way as CUDA programming of this technique. Note that each iteration of this for-loop

touches three xy-planes (bottom, middle, top). Data reuse arises between two consecutive iterations, because some of the data on the middle- and top-planes of the current iteration can be stored in the registers, thus reusable by the bottom- and middle-planes of the next iteration. This type of data reuse will reduce the amount of data loaded from the device's global memory and the number of accesses to the on-chip memory.

3.4 Sharing Data on the xy-Plane (OPT-2)

We notice so far that the neighboring work-items within an OpenCL work-group formally load all their needed data from the device's global memory, although many data items on the xy-planes can be shared. In case the on-chip hardware cache is large enough and properly working, the global memory bandwidth will probably not be wasted. One approach to ensuring maximum use of the bandwidth is to use OpenCL's local memory, which is called on-chip shared memory in NVIDIA terminology. That is, each work-item only explicitly loads $u_{i,j,k}$ into OpenCL's local memory, while expecting that the four neighbors will respectively load the needed values of $u_{i-1,j,k}$, $u_{i+1,j,k}$, $u_{i,j-1,k}$, and $u_{i,j+1,k}$ also into the local memory.

To this purpose, the OpenCL kernel can add a new input argument `__local double *xy_plane`, inside which each work-item of a work-group will store its $u_{i,j,k}$ value that is loaded from the global memory. The size of the `xy_plane` array is prescribed in the host program, and this array is automatically allocated in OpenCL's local memory for each work-group, before the work-items start invoking the kernel. More info about using OpenCL's local memory can be found in [8,9]. There are two things that need to be remembered by the programmer. First, the work-items lying on the boundary of a work-group need to do additional loads from the global memory. Second, synchronization between all the work-items of a work-group has to be explicitly enforced by calling the `barrier(CLK_LOCAL_MEM_FENCE)` function. In comparison with CUDA programming, using OpenCL's local memory closely resembles using CUDA's shared memory.

3.5 Chunking in the y-Direction (OPT-3)

Chunking [10] is a technique that can be used to improve data reuse on the xy-plane. It follows the same principle behind plane sweeping in the z-direction. That is, each OpenCL work-item can be allowed to compute more than one mesh point on the xy-plane. However, letting each work-item compute several adjacent mesh points in the x-direction will destroy coalesced data loads that would have been issued by consecutive single-mesh-point work-items in the x-direction. Therefore, we consider chunking only in the y-direction. More specifically, each work-item can be made responsible for two or four adjacent mesh points in the y-direction.

4 Performance Study

This section will study the performance benefits provided by the three optimizations described in the preceding text. The resulting different OpenCL implementations were run on two NVIDIA GPUs, one of the Fermi architecture, the other of the Kepler architecture. The hardware specifications of the two GPUs are listed in Table 3.

Table 3. Hardware specifications of two NVIDIA GPUs, where the numbers for GTX590 are for one of its two GPU devices

GPU	GeForce GTX590	Tesla K20
# SMs, SPs	16, 512	13, 2496
Architecture	Fermi GF110	Kepler GK110
Compute capability	2.0	3.5
Clock (GHz)	1.26	0.71
Registers/SM	32768	65536
Local memory/SM	48 KB	48 KB
L1 Cache/SM	16 KB	16 KB
L2 Cache	768 KB	1280 KB
Peak DP rate	161.3 GFLOPs	1170 GFLOPs
Peak Bandwidth	165.9 GB/s	208 GB/s
Measured Bandwidth	142.94 GB/s	160.88 GB/s
Compiler	CUDA 5.0	CUDA 5.0

It should be noted that we only used one of the two devices available in a GTX590 GPU. It is also remarked that the L1 caches on the K20 GPU do not cache the data that are loaded from the global memory. Instead, a read-only data cache (48KB for each SM) has been introduced on the Kepler architecture, replacing much of the role played by L1 on Fermi.

The size of all the 3D stencil computations was fixed at 256^3, *i.e.*, the total number of mesh points was 258^3 including the physical boundary points. Moreover, all the 3D arrays were padded in the x-direction, so that the actual array dimension was $272 \times 258 \times 258$. All the following performance measurements will be in form of double-precision FLOPs rates, which were calculated based on the time usages of 100 sweeps over the 3D mesh.

4.1 Tuning Work-Group Size

It is well known that the size of an OpenCL work-group, the same as a thread block in CUDA, can have an impact on the achievable performance. We therefore first tried a few typical sizes of the work-group, such as $(16, 16, 1)$, $(32, 8, 1)$, etc., for the baseline implementation of the four stencil computations. It was found that, if the z-dimension of the work-group is kept as 1, the best-performing size of an OpenCL work-group was $(128, 4, 1)$ among our tests. This work-group size

was thus used in all the subsequent numerical experiments, except for OPT-2. It was also found that the performance of the baseline implementations would lose about 2%∼14%, if the 3D arrays were not padded in the x-direction, especially on the Fermi architecture.

4.2 Evaluating the Different Optimizations

Figure 1 shows the achieved GPU performance of different OpenCL implementations. It can be observed that OPT-1 and OPT-3 (chunk size as 2 in the y-direction) improved the performance of the baseline implementation considerably, for all the four types of 3D stencil computations. However, OPT-2 gave rather modest performance benefits on K20, and had even a negative performance impact on GTX590. This is because the L2 cache on K20 obviously has a quite good caching effect (even though the L1 cache is reserved for register spills and stack data), making the use of OpenCL's local memory not very important. On the Fermi architecture, the on-chip L1 cache implicitly ensures good data sharing on the xy-plane between the neighboring work-items, thus making the use of OpenCL's local memory unnecessary on Fermi. In addition, if-test and synchronization are introduced due to using local memory.

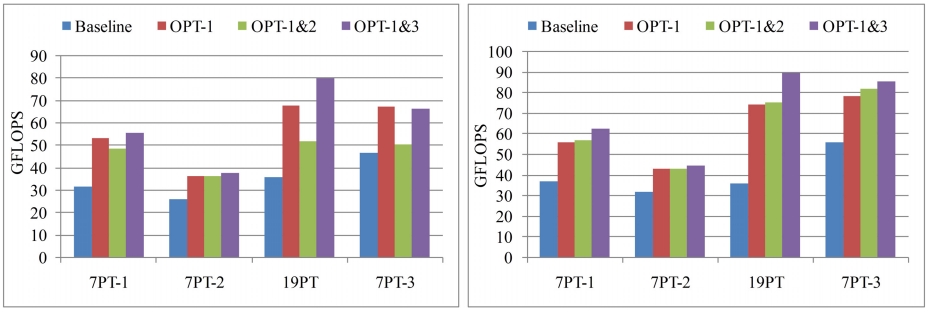


Fig. 1. Performance of different OpenCL implementations of four types of 3D stencil computations, measured on GTX590 (left) and K20 (right)

In comparison with existing works on OpenCL programming of stencil computations, our best performance of 7PT-1 obtained on the GTX590 GPU (using one of its two devices) was 55.5 GFLOPs. This is a considerable improvement over the OpenCL performance reported in [11], which used automated OpenCL code generation and time tiling optimization to achieve 28.7 GFLOPs on a GTX580 GPU for a similar 3D stencil computation that has 6 loads and 7 floating-point operations. (It should be remarked that one device on a GTX590 GPU is less powerful than a GTX580 GPU.) In [12], CUDA programming and on-chip texture memory were used, together with auto-tuning to find the best thread block configuration. For an identical stencil computation as 7PT-1, the authors of [12] obtained 55.2 GFLOPs on a GTX480 GPU, which is comparable with one device on a GTX590 GPU.

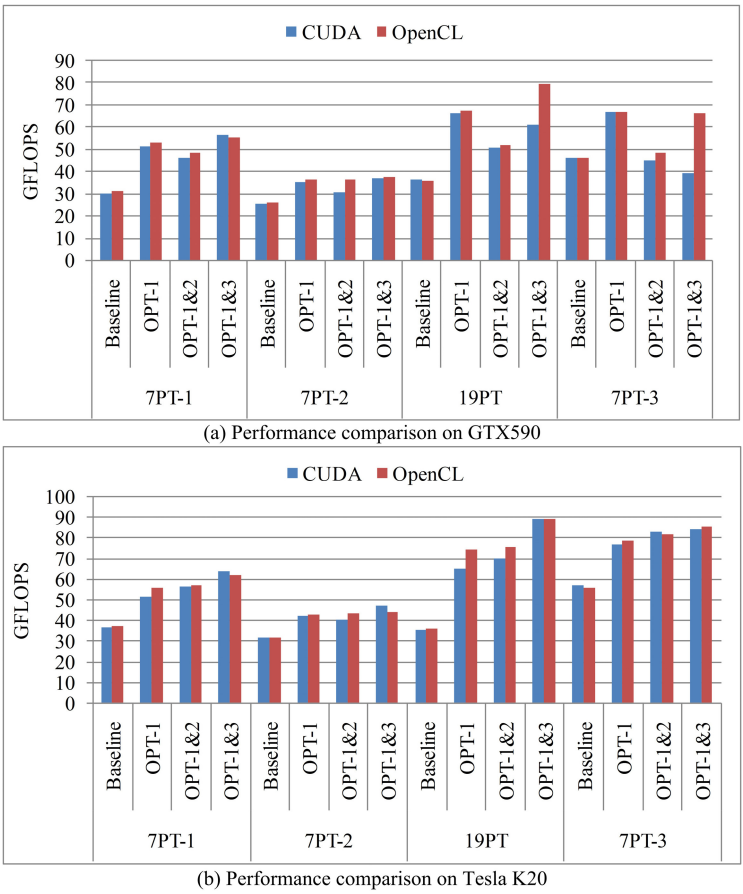


Fig. 2. A comparison between the achieved OpenCL and CUDA performance, obtained on GTX590 (top) and K20 (bottom)

4.3 Comparing OpenCL and CUDA Implementations

To investigate the performance myth of OpenCL in comparison with CUDA, we made matching baseline implementations in CUDA, together with implementing the same optimization techniques. A comparison between the OpenCL and CUDA performance obtained on GTX590 and K20, for all the four stencil computations, is shown in Figure 2. It is clear from the figure that OpenCL programming did not result in any inferior performance for the four representative stencil computations. The poorer CUDA performance associated with 19PT and 7PT-3 on GTX590, when OPT-1 optimization was combined with OPT-3, was due to the limit of maximum 63 registers that can be used by each CUDA thread, causing considerable register spills of the CUDA implementations. Another comment is that we tried for each CUDA implementation quite a few

different thread block sizes, and the achieved highest FLOPs rate is reported in Figure 2. The experimental results show that the OpenCL implementations are, on average, 9% and 2% faster than the CUDA counterparts on GTX590 and K20, respectively. The performance differences between the two programming models warrant further investigations in future. A last comment is that, using either programming model, the K20 GPU does not seem to have too many performance advantages over a single GTX590 GPU device. This is mainly due to their comparable memory bandwidths, as shown in Table 3. In other words, stencil computations depend much more on the memory bandwidth than on the theoretical peak floating-point rate.

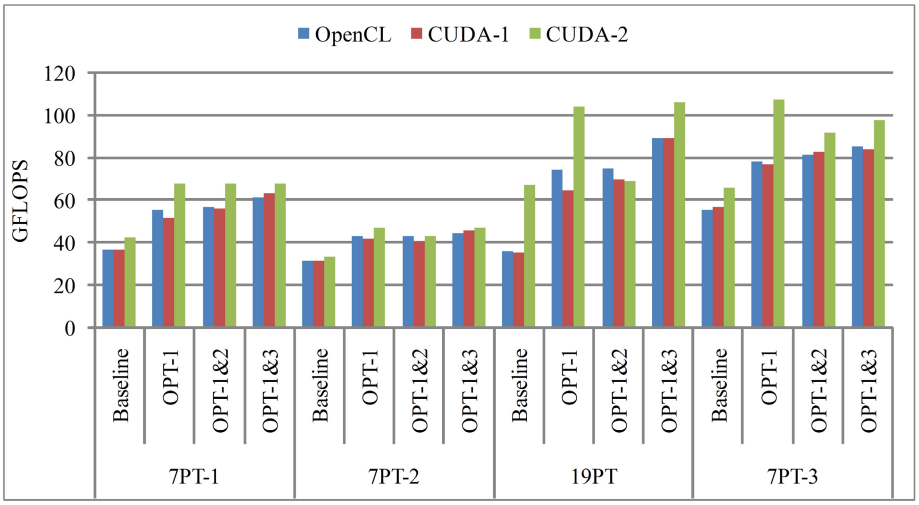


Fig. 3. A comparison between the achieved OpenCL and CUDA performance on K20, where the second set of CUDA implementations utilized Kepler's read-only data cache

We therefore state that the performance of OpenCL-programmed 3D stencil computations is fully comparable with that of CUDA programming. This statement assumes, of course, that same implementation and optimization strategies are used by the two programming standards. On the K20 GPU, however, CUDA programming has the possibility of using the newly introduced read-only data cache on the Kepler architecture. This new on-chip resource is unfortunately not touchable by the current OpenCL standard. If the read-only cache is used properly, CUDA programming will again have an upper hand, as illustrated in Figure 3. In this figure, CUDA-1 denotes best-performing CUDA implementations that did not use Kepler's read-only data cache, whereas CUDA-2 indicates that the read-only data cache was used.

5 Concluding Remarks

In this paper, three widely used techniques for enhancing 3D stencil computations on GPUs have been implemented in OpenCL and compared with the CUDA counterparts. There exist more similar performance-enhancing techniques. For example, Nguyen *et al.* proposed in [13] 3.5-D blocking to exploit the spatial and temporal data locality. A similar implementation was performed on 1D finite difference stencil computations in [14]. Yang *et al.* presented in [15] a hybrid circular queue method to speedup stencil computations on GPUs, by balancing the shared memory and register resources. Nevertheless, our findings show that OpenCL programming does not present any particular difficulties for implementing GPU code. Moreover, the OpenCL implementations have produced fully comparable performance benefits as those obtained by CUDA programming. In a way, our investigation has broken the skepticism about the OpenCL performance for 3D stencil computations. The only obvious advantage of adopting CUDA programming is associated with the read-only data cache on the Kepler architecture, which is currently out of reach for OpenCL programming.

To realize the true values of OpenCL programming, *i.e.*, hardware transparency and cross-platform portability, extensive future work is needed to investigate the performance portability of OpenCL stencil code between different GPUs, and eventually, between GPUs and CPUs. This type of investigation will follow the same spirit of [4,5,16], but with the focus being firmly placed on stencil computations. At the same time, automated code generation [11] and library-based methods [17] can be good candidates for simplifying OpenCL programming.

Acknowledgments. The authors gratefully acknowledge the support from the National Natural Science Foundation of China under NSFC No. 61033008, SRFDP No. 20104307110002, Innovation in Graduate School of NUDT under Nos. B100603, B120605 and CJ11-06-01, the FriNatek program of the Research Council of Norway under No. 214113. We would also like to thank the support from China Scholarship Council and the Simula School of Research and Innovation.

References

1. Khronos OpenCL Working Group: The OpenCL Specification (2011), `http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf`
2. Fang, J., Varbanescu, A., Sips, H.: A comprehensive performance comparison of CUDA and OpenCL. In: Proceedings of the 2011 International Conference on Parallel Processing, pp. 216–225. IEEE Computer Society Press (2011)
3. Karimi, K., Dickson, N., Hamze, F.: A performance comparison of CUDA and OpenCL (2010), `http://arxiv.org/ftp/arxiv/papers/1005/1005.2581.pdf`
4. Komatsu, K., Sato, K., Arai, Y., Koyama, K., Takizawa, H., Kobayashi, H.: Evaluating performance and portability of OpenCL programs. In: Proceedings of the Fifth International Workshop on Automatic Performance Tuning (iWAPT 2010). IEEE Computer Society Press (2010)

5. Du, P., Weber, R., Luszczek, P., Tomov, S., Peterson, G., Dongarra, J.: From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. Parallel Computing 38(8), 391–407 (2012)
6. Unat, D., Cai, X., Baden, S.: Mint: realizing CUDA performance in 3D stencil methods with annotated C. In: Proceedings of the 25th ACM International Conference on Supercomputing, pp. 214–224. ACM (2011)
7. Schäfer, A., Fey, D.: High performance stencil code algorithms for GPGPUs. In: Proceedings of the International Conference on Computational Science. Procedia Computer Science, vol. 4, pp. 2027–2036. Elsevier (2011)
8. NVIDIA: NVIDIA OpenCL Best Practices Guide (2009), http://developer.download.nvidia.com/compute/cuda/2_3/opencl/docs/NVIDIA_OpenCL_BestPracticesGuide.pdf
9. NVIDIA: NVIDIA OpenCL SDK code sample of 3D FDTD, http://developer.download.nvidia.com/compute/DevZone/OpenCL/Projects/oclFDTD3d.zip
10. Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., Yelick, K.: Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing. IEEE Computer Society Press (2008)
11. Holewinski, J., Pouchet, L.N., Sadayappan, P.: High-performance code generation for stencil computations on GPU architectures. In: Proceedings of the 26th ACM International Conference on Supercomputing, pp. 311–320. ACM (2012)
12. Zhang, Y., Mueller, F.: Auto-generation and auto-tuning of 3D stencil codes on GPU clusters. In: Proceedings of the Tenth International Symposium on Code Generation and Optimization, pp. 155–164. ACM (2012)
13. Nguyen, A., Satish, N., Chhugani, J., Kim, C., Dubey, P.: 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE Computer Society Press (2010)
14. Zumbusch, G.: Tuning a finite difference computation for parallel vector processors. In: Proceedings of the 2012 11th International Symposium on Parallel and Distributed Computing, pp. 63–70. IEEE Computer Society Press (2012)
15. Yang, Y., Cui, H., Feng, X., Xue, J.: A hybrid circular queue method for iterative stencil computations on GPUs. Journal of Computer Science and Technology 27(1), 57–74 (2012)
16. Rul, S., Vandierendonck, H., D'Haene, J., De Bosschere, K.: An experimental study on performance portability of OpenCL kernels. In: Symposium on Application Accelerators in High Performance Computing, SAAHPC 2010 (2010)
17. Demidov, D.: VexCL: Vector expression template library for OpenCL (2013), http://www.codeproject.com/Articles/415058/VexCL-Vector-expression-template-library-for-OpenC

Improving Performance Portability in OpenCL Programs

Yao Zhang, Mark Sinclair II, and Andrew A. Chien

Department of Computer Science
University of Chicago
{yaozhang,msinclair94,achien}@cs.uchicago.edu

Abstract. We study the performance portability of OpenCL across diverse architectures including NVIDIA GPU, Intel Ivy Bridge CPU, and AMD Fusion APU. We present detailed performance analysis at assembly level on three exemplar OpenCL benchmarks: SGEMM, SpMV, and FFT. We also identify a number of tuning knobs that are critical to performance portability, including threads-data mapping, data layout, tiling size, data caching, and operation-specific factors. We further demonstrate that proper tuning could improve the OpenCL portable performance from the current 15% to a potential 67% of the state-of-the-art performance on the Ivy Bridge CPU. Finally, we evaluate the current OpenCL programming model, and propose a list of extensions that improve performance portability.

1 Introduction

The recent development of OpenCL [2] provide an open, portable C-based programming model for highly parallel processors. In contrast to NVIDIA's proprietary programming API CUDA [17], a primary goal of OpenCL is portability across a diverse set of computing devices including CPUs, GPUs, and other accelerators [6]. Although the initial focus of OpenCL is to offer functional portability, performance portability is a critical feature for it to be widely adopted. However, it still remains unclear and lacks a systematic study on how performance-portable OpenCL is across multicores and GPUs, given that it is heavily influenced by a GPU-centric programming model, CUDA.

In this work, we study the performance portability of OpenCL programs (SGEMM, SpMV, and FFT) across diverse architectures including NVIDIA GPU, Intel Ivy Bridge CPU, and AMD Fusion CPU. The central questions we would like to answer are: (1) what is the gap between the portable performance of single-source OpenCL programs and the optimized performance of architecture-specific programs? (2) How much of this gap could be closed by certain tuning knobs that adapt OpenCL programs to diverse architectures? And what are those tuning knobs? (3) How should the OpenCL programming interface be extended to better incorporate these tuning knobs?

With these questions in mind, we make the following contributions in this work. First, our study found that the portable performance of three OpenCL

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 136–150, 2013.

programs is poor, generally achieving a low percentage of peak performance (7.5%–40% of peak GFLOPS and 1.4%-40.8% of peak bandwidth). Second, we identify a list of tuning knobs including thread-data mapping, parallelism granularity, data layout transformation, data caching, and we demonstrate that they could improve the OpenCL portable performance from the current 15% to a potential 67% of the state-of-the-art performance. Third, we evaluate current Intel and AMD OpenCL CPU compilers, particular on their features of vectorization, multithreading, and thread aggregation. Fourth, we evaluate the OpenCL programming interface and propose potential extensions on parallelism and data abstractions for performance portability.

The rest of the paper is organized as follows. Section 2 describes our test platform and selected benchmarks. Section 3 presents our experiment results on the portable performance, as well as an evaluation of compiler quality. Section 4 identifies performance critical tuning knobs and demonstrates their performance impacts. Section 5 evaluates the OpenCL programming interface, and proposes a performance portable programming framework. Section 6 discusses related work. Section 7 summarizes and describes future work.

2 Experiment Setup

2.1 Test Platform

Our test processors include NVIDIA Tesla C2050 (Fermi), Intel Core i5 3570K CPU (Ivy Bridge), and AMD A8-3850 APU. Table 1 summarizes the specifications for these processors including the integrated GPUs. Our software platforms use NVIVIA CUDA 4.2 for Ubuntu 12.04, AMD Catalyst 12.4 driver for OpenCL 1.2 and Ubuntu 12.04, and Intel SDK for OpenCL 1.1 and Windows 7.

Table 1. Processor specifications

Processor	Cores	Vector width (32 bits)	Freq (GHz)	Peak GFLOPS	LLC size	Bandwidth (GB/s)
Fermi GPU	14	32 (ALUs)	1.15	1030.4	768 KB	144
APU CPU	4	4	2.9	92.8	4 MB	29.9
APU GPU	5	16×5 (VLIW)	0.6	480	256 KB	29.9
Ivy Bridge CPU	4	8	3.4	217.6	6 MB	25.6
Ivy Bridge GPU	16	8	1.15	166.4	6 MB	25.6

2.2 Benchmarks

We select three programs from the SHOC OpenCL benchmark suite [8] as our case studies: Single Precision General Matrix Multiply (SGEMM), Sparse Matrix Vector multiply (SpMV), and Fast Fourier Transform (FFT), which represent a range of easy to difficult, but computationally important benchmarks. Table 2 summarizes their computation characteristics and performance bottlenecks.

Table 2. Benchmark characteristics

Benchmarks	Compute complexity	Compute-to-memory ratio	Bottleneck
SGEMM	$O(N^3)$	$O(N)$	Compute-limited
SpMV	$O(N)$	$O(1)$	Bandwidth-limited
FFT	$O(NlogN)$	$O(logN)$	Compute-limited

SGEMM is a sub-routine in the Basic Linear Algebra Subprograms (BLAS) library, and its performance behaviors are representative of other level-3 matrix-matrix operations [12]. The SHOC program is based on the implementation developed by Volkov and Demmel [24]. The major improvement introduced by them is to use a block-based algorithm and an improved loop order so that only one input sub-matrix needs to be cached instead of two. Choosing an appropriate block size is critical to the SGEMM performance.

The SHOC SpMV routine is adapted from the version developed by Bell and Garland [4]. It is well known that SpMV is memory-bound, and a compact storage format is critical to its performance. Bell and Garland experimented with a number of compact formants and discovered the ELLPACK format [18] generally performs best on GPUs. This format could guarantee continuous memory access to matrix entries by adjacent threads and thus maximize the bandwidth utilization. We will use the ELLPACK format and its column-major and row-major variants for performance experiments.

The SHOC FFT routine is based on the version developed by Volkov and Kazian [25]. The program is hard-coded for processing many 512-point FFTs. The major optimization exploits the Cooley-Tukey algorithm [6] to decompose a 512-point FFT to many 8-point FFTs and process them by individual threads in registers, instead of processing a large 512-point FFT by many threads collectively in the slower on-chip scratchpad memory. A vectorization-friendly data layout and efficient twiddle factor calculation are critical performance factors.

3 OpenCL Portable Performance

In this section, we study the portable performance of the three OpenCL benchmarks for a diverse set of processors. We will also investigate the causes of the gap between the portable performance and optimized performance.

3.1 SGEMM

Figure 1a shows the nomalized SGEMM performance. The benchmark is not tailored to any of our test processors, as it was originally written in CUDA for NVIDIA G80/GT200 GPUs [24], and later ported to OpenCL in SHOC. Still, it reaches 40% of the peak Tesla C2050 (Fermi) performance, which is much higher

than that of other processors, but lower than the reported 60% for the previous generation GT200 GPU [24]. Since the Fermi GPU significantly increases the hardware resources of ALUs, registers and scratchpad memory, it may need a larger sub-matrix size to achieve a higher GPU utilization.

The major inefficiency of the APU GPU (integrated) comes from the low utilization of its VLIW ALUs. Our examination of the assembly code reveals that only an average of 2.4 slots of the total 5 VLIW slots are used, and only 38% of the total dynamic instructions are compute instructions, which together bound the performance to $\frac{2.4}{5} \times 38\% = 18.4\%$, close to our measured 14.6%. We also test a SGEMM program in the AMD APP SDK, which achieves a performance of 329 GFLOPS, 68.5% of the peak, thanks to its customized vector operations. The low performance of the APU CPU is mainly due to two factors: (1) the AMD CPU compiler does not support vectorization, and (2) the expensive context switching between threads at synchronization points.

The Intel CPU compiler does a better job on supporting vectorization and thread aggregation (serialize threads to avoid unnecessary thread synchronizations), and thus achieves a higher percentage of the peak performance (13.5% vs. 7.5% for the AMD CPU compiler). The OpenCL program uses a column-major data layout, which favors GPUs by preserving the inter-thread locality, instead of the intra-thread locality favored by the CPUs. This is why the Ivy Bridge CPU performance decreases for larger matrices, which demand better locality to reduce the bandwidth requirement.

3.2 SpMV

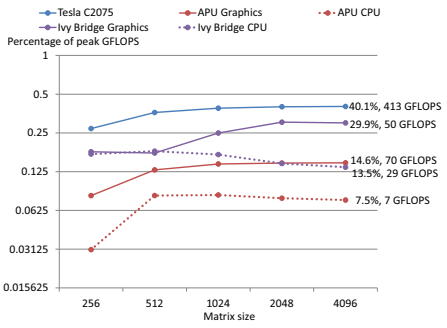
We generate our random test sparse matrices of various sizes, with 1% non-zeros. Figure 1b shows the SpMV performance and bandwidth utilization, which we define as the ratio of the effective bandwidth to the peak bandwidth. The effective bandwidth is calculated as

$$\frac{3 \times NumberNonzeros \times 4Bytes}{ProgramRuntime}$$

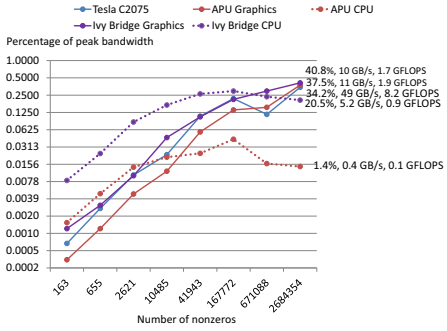
where 3 is the number of reads for a matrix entry, a column index, and a vector entry, and 4 bytes is the size for a 32-bit floating point or integer value. The performance is memory-latency-bound for small matrices, and gradually becomes bandwidth-bound as the matrix size increases. Due to the random access to vector entries, the bandwidth utilization is low on all processors. The Ivy Bridge CPU performance is higher than the integrated GPU performance for smaller matrices, mainly thanks to the L1–L2 cache. However, because of the poor locality of the column-major data layout on the CPU, the CPU performance drops as input matrix becomes too big to fit into the cache; the Ivy Bridge integrated GPU performance is not affected, because the program uses a GPU-friendly column-major data layout. The APU processor shows a similar performance trend.

3.3 FFT

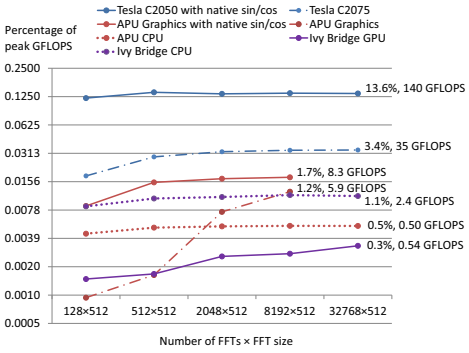
We use $5Nlog_2N$ (radix-2 FFT arithmetic complexity) as the number of flops for a N-point FFT, as a convenient and normalized way to compare the performance of different FFT algorithms, as suggested in the paper by Volkov and Kazian [25]. Figure 1c shows the normalized FFT performance. Both Tesla C2050 and APU integrated GPU support native sin/cos instructions, which enables a speedup of $2-8\times$ over the program using software-emulated sin/cos calculations. The program defines a data type for complex numbers and uses a array-of-structure (AoS) layout with real and imaginary parts of complex numbers stored in an interleaved way. In this SHOC FFT program, this interleaved layout is fine for GPUs, but unfriendly for CPUs, because the CPU vectorization is at the butterfly level, instead of the 8-point-FFT level in the GPU case. Both AMD CPU and Ivy Bridge CPU show low performance. The Intel OpenCL CPU compiler chooses not to vectorize the code, because of the interleaved data layout, while the AMD compiler does not support vectorization yet.



(a) SGEMM performance normalized to peak GFLOPS.



(b) SpMV effective bandwidth normalized to peak bandwidth.



(c) FFT performance normalized to peak GFLOPS.

Fig. 1. SGEMM, SpMV, and FFT performance normalized to the peak performance of each processor

3.4 Summary

For the three OpenCL benchmarks across multicores and GPUs, the portable performance is generally low (7.5%–40% of peak GFLOPS and 1.4%-40.8% of peak bandwidth). In addition to compiler support, we identified the major causes of low performance, related to submatrix size, thread-data mapping, data layout, and native sin/cos support. Next, we show how portable performance can be significantly improved by tuning.

4 Performance Tuning

In this section, we will discuss the motivation and methodology of performance tuning, and show it could significantly improve the portable performance.

4.1 Problem Statement

Multicores and GPUs have different architecture features and configurations, and thus demand different program optimizations. However, current single-source OpenCL programs lack the ability to adapt. A number of architecture features impact programmers' tuning decisions on thread-data mapping, data layout, tiling size, and so on. These features include core types (complex ILP cores vs. simple highly-threaded cores), vectorization style (explicit SIMD vs. implicit SIMT [17]), core count, vector width, cache types (hardware vs. programmer-managed), cache size (kilobytes vs. megabytes), and bandwidth. The goal of performance tuning is to make the best choices to map a program to architecture features.

4.2 Methodology

We adopt a systematic approach for performance tuning by summarizing all potential program optimization aspects as tuning knobs, which form a high dimensional optimization space. We explore this optimization space by experimenting with the settings of these tuning knobs. Table 3 summarizes all the tuning knobs and their settings for three benchmarks. In the following subsections, we will explain why these tuning knobs might be critical performance factors.

Tiling Size. Tiling is an effective technique used by block-based algorithms (e.g. SGEMM) to increase the cache reuse and thus the compute-to-memory ratio for bandwidth-limited programs. An optimal tiling size depends on multiple architecture features including bandwidth, cache size, core count, vector width, and processor frequency. A perfect size will balance between the effective cache reuse (large tiles preferred) and sufficient parallelism (small tiles preferred). GPUs usually prefer smaller tiling sizes because of limited on-chip scrachpad memory and massively parallel hardware, while CPUs often prefer bigger tiling sizes because of large cache and fewer hardware parallelism.

Data Layout. Data layout plays an important role in program performance optimization. GPU threads are lightweight and work in a tightly coupled, synchronized fashion in thread groups. It is highly desirable for adjacent threads in a group to access adjacent data in memory to maximize the bandwidth utilization. As a result, GPUs usually favor column-major data layout for inter-thread data locality. On the other hand, CPU threads are more independent and have a larger working set. They usually favor row-major data layout for intra-thread locality, and a column-major layout will result in inefficient strided memory access. Both CPUs and GPUs favor the SoA layout for vectorized operations. There are tricky exceptions where the AoS layout could be efficient on GPUs, and the SHOC FFT benchmark is an example, as discussed in Section 3.3.

Caching and Prefetching. GPUs use programmer-managed scratchpad memory (as well as hardware L1 and L2 caches for post-Fermi GPUs) for data caching, while CPUs use hardware-managed cache. For CPUs, OpenCL currently simply treats arrays in scratchpad memory as the ones in external memory, in order to ensure the program correctness. The extra loads and stores to such emulated scrachpad memory for CPUs may have a performance penalty. However, the prefetching effect of such loads can help the performance, as shown in Section 4.3.

Thread Data Mapping. CPUs and GPUs have a two-level parallelism structure with cores and vector units. However, many applications show multiple levels of parallelism (e.g. SGEMM has three levels of parallelism: sub-matrix, row, and element). It is desirable to have the ability to flexibly map the cross-level parallelism to the two-level architectures to maximize the hardware utilization. Another option is to choose between blocked and interleaved thread-data mapping. GPUs prefer interleaved mapping (adjacent data mapped to adjacent threads) for intra-thread locality, while CPUs prefer blocked mapping (adjacent data are mapped to a single thread) for intra-thread locality, because CPU threads are more independent and often have their own L1/L2 cache.

Operation-Specific Tuning. Different generations of CPUs and GPUs may support a different set of hardware intrinsic instructions such as trigonometric, logarithmic, and thread coordination (atomic, sync, fence, etc.) operations. To avoid expensive software emulation, programs should try minimizing the use of intrinsic functions for the architectures that do not support them.

4.3 Performance Tuning Results

In this section, we will present our performance tuning results. Although we use the Intel CPU for experiments, we expect similar tuning benefits on other processors. To experiment with different optimizations, we port the baseline OpenCL programs to OpenMP, so that we can control the mapping of parallelism to the hardware, and if vectorization or multithreading is applied. For all experiments, we use Intel Core i5 3570K CPU (Ivy Bridge) and Intel C++ Compiler XE 13.0, which supports both AVX vectorization and OpenMP multithreading. All experiments are performed with single precision floating point arithmetic.

Table 3. Tuning knobs and their settings for three benchmarks

Benchmark	Tuning knob	Setting
SGEMM	Tile size	2×2
		.....
		128×128
	Data layout	Row-major
		Col-major
	Prefetching/caching	Enabled
		Disabled
SpMV	Thread-data mapping	Interleaved
		Blocked
	Data layout	Row-major
		Col-major
FFT	Share sin/cos calculations	Enabled
		Disabled
	Data layout	SoA
		AoS

SGEMM *Tiling size.* Figure 2a compares the performance of the original OpenCL program, the ported OpenMP program with tunable sub-matrix size, and the Intel MKL 10.2 SGEMM routine. Our ported OpenMP program is auto-vectorized by the Intel compiler using 8-wide AVX instructions on 32-bit floating point numbers. An optimal sub-matrix size of 64×64 doubles the performance of the original OpenCL program which has a hard-coded sub-matrix size of 16×16.

Caching and Prefetching. On the CPU, the use of scratchpad memory (which caches one input sub-matrix) is emulated by external memory. We first thought the extra copies to and from such emulated scrachpad memory would have a performance penalty. However, it turns out it even provides a slight performance improvement, most likely due to the prefetching effect of such extra load.

Data Layout. We experiment with both the column-major and row-major layout. Although the column-major format introduces strided memory access and is bad for cache performance, the two-dimensional block-based algorithm minimizes its performance penalty by caching the sub-matrices and making better use of the cacheline data brought in by each strided access. With a sub-matrix size of 16×16, a row of 16 32-bit values could use the entire cacheline of 64 KB brought in by a strided access. As a result, the row-major layout only offers slight performance advantage over the column-major layout.

Comparing with the State-of-the-Art Performance. Our tuned performance is is still low compared with the MKL routine (Figure 2a). By comparing their assembly code, we find that the vectorized section of our ported OpenMP program is not as efficient as that of the MLK routine, in that it requires two extra data shuffling and replacement instructions per multiply-add.

SpMV *Data Layout.* We experiment with both the row-major and column-major ELLPACK format. Using four threads and the row-major layout, we achieve the best performance, which is $7.4\times$ higher than that of the SHOC benchmark as shown in Figure 2b. Although SpMV is a memory-bound program, we find that the performance scales well with the number of threads for large inputs.

This is most likely because more threads are able to use the L1 and L2 caches in more cores and thus reduce the bandwidth requirement. For small matrix sizes, data layout and multithreading do not help with the performance, because the program is memory-latency-bound and all data could fit into cache. Data layout or multithreading starts to make a performance difference at the point where the total matrix data and index size($2 \times 41943 Nonzeros \times 4 Bytes/1024 = 327 KB$) exceeds the 256 KB of L2 cache.

Thread Data Mapping. We experiment with interleaved and blocked thread-row mapping schemes for one random matrix plus four matrices from a collection of sparse matrices from various scientific and engineering applications, which is also used in prior work by others [7,4]. The blocked mapping is always faster than the interleaved mapping by 7% to 21% (Figure 2c).

Comparing with the State-of-the-Art Performance. The Intel MKL library does not support ELLPACK. We test its SpMV routine in compressed sparse row (CSR) format. It has a high startup cost for small matrices and is 33% slower than our row-major ELLPACK program for the largest matrix (Figure 2b).

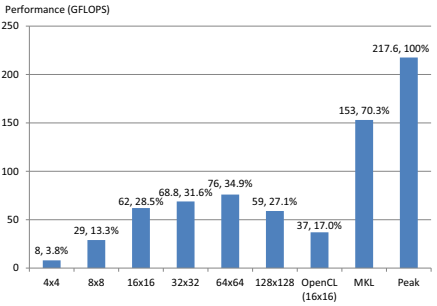
FFT *Data Layout.* We experiment with both the SoA and AoS layout. As shown in Figure 2d, the ported program with the same AoS data layout achieves comparable performance to that of the OpenCL FFT benchmark. The auto-vectorized program by the Intel compiler even hurts the performance. However, the SoA layout is suitable for vectorization and makes the program 2× faster than the OpenCL benchmark.

Operation-Specific Tuning. Calculating twiddle factors consumes a considerable amount of compute time, as the Intel CPU does not have hardware arithmetic for sin/cos functions. The Cooley-Tukey algorithm decomposes a 512-point FFT to 8 64-point FFTs, which share the same set of twiddle factors. Based on this observation, we further improve the program efficiency by cutting 8 times of redundant twiddle factor calculations to one time per 8 64-point FFTs. This speeds up the program by another 68% in addition to the data layout optimization.

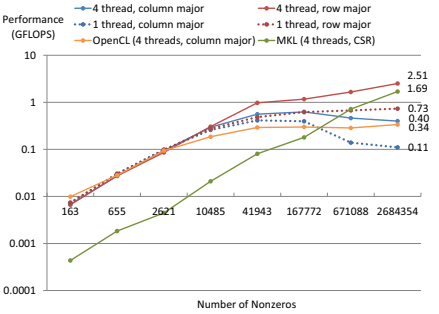
Comparing with the State-of-the-Art Performance Our tuned program is still 50% slower than the state-of-art FFTW library. There are two improvement oppurtunities. First, we could speedup the twiddle factor calcuation by using lookup tables and the symmetric property of sin/cos functions. Second, currently only half of the 8-wide AVX vector unit are utilized, limited by the four concurrent butterflies in the radix-8 algorithm. A radix-16 algorithm will fully utilize the vector unit. Another option is to vectorize over 8 FFTs rather than over butterflies in a single FFT. This optition is used on the GPU, thanks to convinient data shuffling provided by the crossbar interconnect of scrachpad memory. On CPUs, however, this will involve major data shuffling overhead.

4.4 Summary

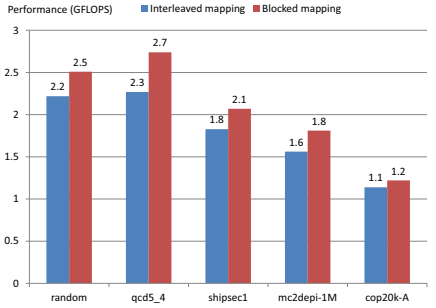
We have explored the performance optimization space formed by various tuning knobs, and demonstrated a large room of performance tuning for three benchmarks. Table 4 summarizes all the optimial setting of these tuning knobs for



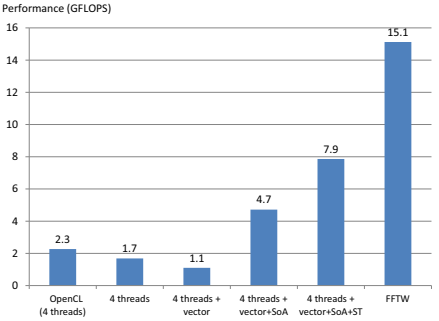
(a) SGEMM with tunable submatrix tile size. Percentage of peak performance.



(b) SpMV for row- and column-major layouts, w/o multithreading. Blocked thread-row mapping.



(c) SpMV with interleaved and blocked thread-row mapping for various matrices.



(d) FFT with various optimizations. SoA: structure of arrays. ST: shared twiddle factors.

Fig. 2. Performance of SGEMM, SpMV, and FFT, with tuning knobs incorporated on the Ivy Bridge CPU

Table 4. Summary of the optimal settings of tuning knobs for the Ivy Bridge CPU. The improvement is compared with the baseline OpenCL programs.

Programs	Optimal knob settings	Improvement
SGEMM	64×64 tile size	2×
	Row-major layout	Insignificant
	Prefetching/caching	Insignificant
	Total	2×
SpMV	Row-major (small input)	Insignificant
	Row-major (larger input)	6.2×
	Blocked thread mapping	1.2×
	Total	7.4×
FFT	SoA layout	2.0×
	Minimize expensive sin/cos	1.7×
	Total	3.4×
Average	Total	4.3×

the Ivy Bridge CPU. On average, the optimized programs perform 4.3× faster. Another major observation is that the same optimizations may have drastically different performance impacts for different programs and input sizes. For example, the SpMV performance is much more sensitive to the row-major layout optimization than the SGEMM performance.

One primary goal of this paper is to investigate the gap between the "current" portable performance of single-source OpenCL programs and the performance of state-of-the-art programs with architecture-specific optimizations. We also want to quantify how much of this gap could be closed by "potential" portable performance, which is the performance achieved with our tuning knobs incorporated (summarized in Table 4). Figure 3 shows the current and potential portable performance of SGEMM, SpMV, and FFT on the Ivy Bridge CPU, normalized against the performance of today's state-of-the-art programs (the MKL SGEMM routine, FFTW, and our ELLPACK SpMV routine, which outperforms the MKL CSR SpMV routine). On average, by incorporating tuning knobs to program, the OpenCL portable performance could be improved by more than 4×, from 15% to a 67% of the state-of-the-art performance.

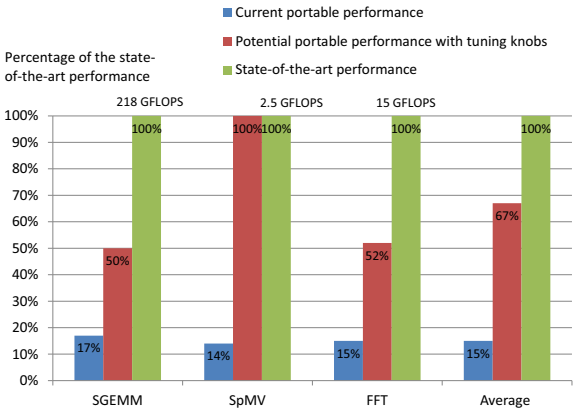


Fig. 3. Current and potential portable performance on the Ivy Bridge CPU. Performance is normalized against the state-of-the-art performance results.

5 Programming Model Implications

In this section, we evaluate current OpenCL programming interface, and propose extensions towards a performance-portable programming framework.

Although OpenCL provides functional portability across multicores and GPUs, its performance portability is poor (Figure 3). We have demonstrated a set of tuning knobs could significantly improve the portable performance. To incorporate these tuning knobs, however, the OpenCL programming interface needs to be raised to a more abstract level. In particular, we propose the following extensions.

First, the mapping between threads and data is currently specified by programmers in a two-level parallelism hierarchy with work-items (threads) and work-groups (groups of threads), where work-groups are mapped to cores and work-items to vector lanes. This two-level parallelism model limits the possibility to tune thread-data mapping across multiple levels of parallelism. The current model also requires programmers to specify a fixed parallelism granularity (the amount of work per work-item is fixed by the specified total number of work-items, although the OpenCL runtime could select a work-group size if not provided) and implies interleaved thread-data mapping, which is not favorable to CPUs. To support tunable thread-data mapping, we propose the notion of logical threads with more levels of parallelism hierarchy and logical dimensions, so that they could be re-mapped and re-sized to match today's two-level (core and vector) processors in an adaptive way.

Second, OpenCL currently does not support any data layout abstraction and requires programmers to specify a fixed data layout. However, multicores and GPUs favor different data layout between row major and column major, and between structure-of-arrays and array-of-structures. To solve this problem, OpenCL needs to introduce some form of data abstraction, which decouples data structure and content from the data layout, and allows programmers to write generic code without specifying an architecture-specific layout. Examples of such decoupling include layout specifiers in UPC [5] and data distributive directives in HPF [15].

Third, OpenCL currently does not have an abstract way to use or not use scrachpad memory. Although OpenCL programs could explicitly manage scratchpad memory on GPUs, such programs do not naturally fit into CPUs with hardware cache. As a result, the OpenCL compiler uses external memory as an emulated scratchpad memory for CPUs, which may cause a performance penalty. OpenCL needs to introduce a simple switch for using either programmer-managed or hardware-managed cache, such as the cache directive in OpenACC [1].

A higher-level programming interface allows a larger tuning space, but is still not sufficient. To support architecture-specific performance tuning, we need to build a more intelligent compiler and runtime framework. As we have noted previously, there is not a one-size-fit-all recipe on how to turn those tuning knobs, and the same optimizations may show totally different performance impacts for different programs and inputs. Therefore, such tuning should be analyzed on a case-by-case basis and could potentially be guided by recently proposed performance models [3,13,26]. Application-specific GPU performance autotuning has also been recently explored with success [7,9,16]. These studies, as well as ours in this paper, still require considerable manual work of programmers, and we expect a higher-level programming interface with model-guided tuning will be an important direction for future research.

6 Related Work

There have been quite a few studies on the portability of GPU programming models [10,11,14,19,20,21,22,23]. However, the previous work focus mainly on

architecture-specific optimizations for OpenCL programs. The contribution of this work is a methodically designed performance portability study that covers both compute-limited and bandwidth-limited benchmarks, systematically summarizes common tuning knobs, and discusses their programming model implications.

MCUDA is one of the pioneering work to compile CUDA programs to a CPU architecture [22]. The loop fission technique is used to convert explicitly synchronized fine-grained parallel GPU programs to implicitly synchronized coarse-grained multi-threaded CPU programs. Potential optimizations including data layout transformation, optional use of shared memory, and flexible thread-data mapping are not explored in the paper.

Du et al. [10] did an interesting study on portable performance of vendor-specific SGEMM kernels across NVIDIA and ATI GPUs. Only compute-bound dense matrix kernels and one tuning parameter (tiling size) are investigated in their proposed autotuning infrastructure. Similar studies [19,23] explored other tuning parameters including caching, vectorization, and thread block size. Seo et al [20] also noticed the limited performance portability of OpenCL, but did not further investigate its causes.

Shen et al. [21] showed properly tuned OpenCL programs could achieve comparable performance to OpenMP versions. Various tuning aspects including data layouts and parallelism granularity are explored. Fang et al. [11] and Komatsu et al. [14] respectively compare the performance of OpenCL and CUDA, and reach a similar conclusion that the performance of OpenCL programs is comparable to those of CUDA if optimized appropriately. Both studies call for future autotuning research to adapt OpenCL programs to various processors.

7 Conclusions and Future Work

We have identified major tuning knobs for performance portable programming, and demonstrated that they could improve the OpenCL portable performance from the current 15% to a potential 67% of the state-of-the-art performance. We also evaluated the current OpenCL compilers and programming model, and made a list of proposals towards a performance portable programming framework. We believe these results will inform and inspire more thinkings on the design and evolution of OpenCL and other emergining higher-level programming models.

Future directions of this work include: (1) study the performance portability of irregular programs with data-dependent control flows and memory access patterns, (2) investigate the feasibility of incorporating a performance model to compiler or runtime for model-based tuning, and (3) extend this study to other architectures such as Intel's Xeon Phi and other emerging programming APIs such as OpenACC.

Acknowledgments. Thanks to the editors and anonymous reviewers for their helpful suggestions, which improved the final version of this paper. This work was

supported in part by the National Science Foundation under awards NSF OCI-1057921 and the Defense Advanced Research Projects Agency under PERFECT program award HR0011-13-2-0014. The contents of this report do not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred.

References

1. The OpenACC application programming interface 1.0 (November 2011), http://www.openacc-standard.org/
2. The OpenCL specification 1.2 (November 2011), http://www.khronos.org/registry/cl/
3. Baghsorkhi, S.S., Delahaye, M., Patel, S.J., Gropp, W.D., Hwu, W.W.: An adaptive performance modeling tool for GPU architectures. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2010), pp. 105–114 (January 2010)
4. Bell, N., Garland, M.: Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: Proceedings of the 2009 ACM/IEEE Conference on Supercomputing (SC 2009), pp. 18:1–18:11 (November 2009)
5. Carlson, W., Draper, J., Culler, D., Yelick, K., Brooks, E., Warren, K.: Introduction to UPC and language specification. Center for Computing Sciences, Institute for Defense Analyses (1999)
6. Chien, A.A., Snavely, A., Gahagan, M.: 10x10: A general-purpose architectural approach to heterogeneity and energy efficiency. Procedia CS 4, 1987–1996 (2011)
7. Choi, J.W., Singh, A., Vuduc, R.W.: Model-driven autotuning of sparse matrix-vector multiply on GPUs. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2010), pp. 115–126 (January 2010)
8. Danalis, A., Marin, G., McCurdy, C., Meredith, J.S., Roth, P.C., Spafford, K., Tipparaju, V., Vetter, J.S.: The scalable heterogeneous computing (SHOC) benchmark suite. In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU 2010), pp. 63–74. ACM, New York (2010)
9. Davidson, A., Zhang, Y., Owens, J.D.: An auto-tuned method for solving large tridiagonal systems on the GPU. In: Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium, pp. 956–965 (May 2011)
10. Du, P., Weber, R., Luszczek, P., Tomov, S., Peterson, G., Dongarra, J.: From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. Parallel Comput. 38(8), 391–407 (2012)
11. Fang, J., Varbanescu, A.L., Sips, H.: A comprehensive performance comparison of CUDA and OpenCL. In: Proceedings of the 2011 International Conference on Parallel Processing (ICPP 2011), pp. 216–225. IEEE Computer Society, Washington, DC (2011)
12. Goto, K., Van De Geijn, R.: High-performance implementation of the level-3 BLAS. ACM Trans. Math. Softw. 35(1), 4:1–4:14 (2008)
13. Hong, S., Kim, H.: An integrated GPU power and performance model. In: Proceedings of the 37th International Symposium on Computer Architecture (ISCA 2010), pp. 280–289 (2010)
14. Komatsu, K., Sato, K., Arai, Y., Koyama, K., Takizawa, H., Kobayashi, H.: Evaluating performance and portability of OpenCL programs. In: The Fifth International Workshop on Automatic Performance Tuning (June 2010)

15. Loveman, D.: High performance Fortran. IEEE Parallel & Distributed Technology: Systems & Applications 1(1), 25–42 (1993)
16. Meng, J., Skadron, K.: Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In: Proceedings of the 23rd International Conference on Supercomputing (ICS 2009), pp. 256–265 (June 2009)
17. NVIDIA Corporation. NVIDIA CUDA compute unified device architecture, programming guide 5.0 (October 2012), http://developer.nvidia.com/
18. Rice, J.R., Boisvert, R.F.: Solving Elliptic Problems using ELLPACK. Springer-Verlag New York, Inc. (1984)
19. Rul, S., Vandierendonck, H., D'Haene, J., De Bosschere, K.: An experimental study on performance portability of OpenCL kernels. In: 2010 Symposium on Application Accelerators in High Performance Computing, p. 3 (2010)
20. Seo, S., Jo, G., Lee, J.: Performance characterization of the NAS parallel benchmarks in OpenCL. In: 2011 IEEE International Symposium on Workload Characterization (IISWC 2011), pp. 137–148 (November 2011)
21. Shen, J., Fang, J., Sips, H., Varbanescu, A.: Performance gaps between OpenMP and OpenCL for multi-core CPUs. In: 2012 41st International Conference on Parallel Processing Workshops (ICPPW 2012), pp. 116–125 (September 2012)
22. Stratton, J.A., Stone, S.S., Hwu, W.-m.W.: MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. In: Amaral, J.N. (ed.) LCPC 2008. LNCS, vol. 5335, pp. 16–30. Springer, Heidelberg (2008)
23. Thoman, P., Kofler, K., Studt, H., Thomson, J., Fahringer, T.: Automatic openCL device characterization: Guiding optimized kernel design. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part II. LNCS, vol. 6853, pp. 438–452. Springer, Heidelberg (2011)
24. Volkov, V., Demmel, J.W.: Benchmarking GPUs to tune dense linear algebra. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC 2008), pp. 31:1–31:11 (November 2008)
25. Volkov, V., Kazian, B.: Fitting FFT onto the G80 architecture (May 2008), http://www.cs.berkeley.edu/ kubitron/courses/cs258-S08/projects/reports/project6_report.pdf/
26. Zhang, Y., Owens, J.D.: A quantitative performance analysis model for GPU architectures. In: Proceedings of the 17th IEEE International Symposium on High-Performance Computer Architecture (HPCA 17), pp. 382–393 (February 2011)

Auto-tuning of Sparse Matrix-Vector Multiplication on Graphics Processors

Walid Abu-Sufah[1,2] and Asma Abdel Karim[2]

[1] University of Illinois at Urbana-Champaign, Urbana, Illinois, USA
abusufah@illinois.edu
[2] The University of Jordan, Amman, Jordan
asma_abdelkarim@hotmail.com

Abstract. We present a heuristics-based auto-tuner for sparse matrix-vector multiplication (SpMV) on GPUs. For a given sparse matrix, our framework delivers a high performance SpMV kernel which combines the use of the most effective storage format and tuned parameters of the corresponding code targeting the underlying GPU architecture. 250 matrices from 23 application areas are used to develop heuristics which prune the auto-tuning search space. For performance evaluation, we use 59 matrices from 12 application areas and different NVIDIA GPUs. The maximum speedup of our framework delivered kernels over NVIDIA library kernels is 7x. For most matrices, the performance of the kernels delivered by our framework is within 1% of the kernels found using exhaustive search. Compared to exhaustive search auto-tuning, our framework can be more than one order of magnitude faster.

Keywords: SpMV, GPUs, Auto-tuning, sparse linear algebra, CUDA.

1 Introduction

Sparse matrix-vector multiplication (SpMV) operations dominate the performance of numerous applications in scientific and engineering computing, financial and economic modeling, information retrieval, and others. SpMV operations are performance bottleneck in iterative methods for solving large scale linear systems, eigenvalue problems, and least squares problems. In SpMV, the operation y=A*x+y is performed, where A is a sparse matrix and x, y are dense vectors. Vectors x and y provide the only opportunities for data reuse since elements of matrix A are used only once. SpMV is a memory bound operation because of the indirect and irregular memory accesses introduced by the special schemes used to store sparse matrices and the low computational intensity caused by the lack of data reuse.

With peak performance of more than 1Teraflop in double precision and peak memory bandwidth in excess of 192 GB/s, GPUs are becoming ideal to use in accelerating memory-intensive kernels like SpMV. In order to exploit the computing potential of GPUs, programmers mostly use NVIDIA's Compute Unified Device Architecture (CUDA) parallel programming model and software platform

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 151–164, 2013.

(CUDA C and CUDA FORTRAN), C++ Accelerated Massive Parallelism (C++ AMP), OpenACC API, or the Open Compute Language (OpenCL) [1,2,3,4,5]. However, the time and effort required to program an optimized routine or an application kernel(s) are significant [6].

Auto-tuning has been used extensively to automatically generate high performance numerical libraries and applications' kernels for single and multi-core processors [7,8,9,10]. Like multi-core processors, GPUs offer complex and diverse architectural features which require nontrivial optimization strategies that often change from one chip generation to the next [6]. For SpMV, our recent study and others show that different storage schemes achieve the best GPU performance for different matrix sparsity patterns [11,12]. Moreover, selecting the best storage scheme for a given matrix is often not a simple task. Hence, auto-tuning SpMV requires automating both the selection of a storage scheme and code optimizations that make the most of the underlying GPU.

Recent efforts to auto-tune SpMV on GPUs do not address automating the selection of the best matrix storage scheme [13,14]. Code optimizations are limited to implementations using specific storage schemes which are effective only for certain types of matrices [15]. Another drawback is the expensive use of exhaustive search auto-tuning.

In this paper, we present a heuristics-based SpMV auto-tuning framework for NVIDIA GPUs. Given a sparse matrix and a GPU, our framework chooses the best storage scheme and selects values for parameters of the code of a corresponding kernel to obtain best performance. We consider seven SpMV storage schemes/kernels; diagonal, ELLPACK, coordinate, CSR, hybrid [16], Blocked ELLPACK [17], and Blocked Transpose Jagged Diagonal [12]. SpMV kernels and our framework are coded using CUDA 4.0.

To develop search space pruning heuristics, we conduct experiments using 250 matrices from the University of Florida sparse matrix collection [18]. The matrices are from 23 application areas and span a wide spectrum of sparsity patterns, matrix dimensions, and number of nonzeros. The heuristics are two types. The first prunes the candidate storage schemes. Pruning decisions are made by comparing computed values of sparse matrices characterization parameters to threshold values that are determined empirically. The second type of heuristics prunes the search space for parameter values of SpMV codes. The heuristics are based on either matrix characteristics or the architecture features of the GPU or both. These heuristics are developed through analysis and observations based on experiments using the set of 250 matrices. For both types of heuristics, our experimentation starts using a set of 50 matrices. Heuristics threshold values are computed before the sample of matrices is incremented by 50 matrices and the experiments are repeated to fine tune the computed threshold values. This process is repeated until no change is detected in the computed threshold values for three consecutive 50 matrices increments of the matrices sample size.

We use 59 other matrices and two GPUs to evaluate our framework. Executing on the Fermi class Tesla M2070 and compared to exhaustive search, we speedup auto-tuning by more than 10x for 19 matrices. The speedup is greater than 1.5x

for all matrices. Performance of the delivered kernels is within 1% of exhaustive search kernels for most matrices. The maximum speedup of delivered kernels over the measured performance of NVIDIA library kernels is 7x. Executing on an older generation GPU, the Tesla S1070, auto-tuning speedup is greater than 10x for 27 matrices. The speedup is greater than 1.50x for most matrices. Performance of delivered kernels is within 1% of the performance of exhaustive search kernels for 44 matrices and is within 6% for all matrices. Delivered kernels outperform the measured performance of NVIDIA library kernels with up to 3.3x speedup.

2 Related Work

Auto-tuning SpMV for multicore processors and GPUs received considerable attention in literature. Williams and co-workers developed a SpMV auto-tuning framework for multi-core architectures [9]. Their auto-tuning framework uses a combination of heuristics and exhaustive search to set the optimizations parameters. The optimizations were incorporated in the Optimized Sparse Kernel Interface (OSKI) to form (pOSKI) [8,10]. On top of pOSKI, an MPI-layer was built to support distributed-memory architectures forming MPI-pOSKI.

For GPUs, El-Zein and Rendell [13] tested the effect of different implementation options on the performance of SpMV using the CSR storage scheme. The tested options use different memory types, different data types and different assignments of rows to threads/thread blocks. Based on experiments using 735 matrices, El-Zein and Rendell proposed a two-level decision tree that attempts to select the optimal CSR kernel implementation for a matrix. Performance achieved by their module was within 6% of the performance of the best implementation.

Grewe and Lokhmotov [14] presented a framework consisting of a high level representation for describing sparse matrix storage schemes and a compiler that generates SpMV CUDA or OpenCL code. The framework allows for applying several optimizations to the generated code. They evaluated their framework using the 14 matrices in [9] and the storage schemes CSR, DIA, ELL, HYB [16], blocked ELLPACK [17], and sliced ELLPACK [19]. Experimental results on an NVIDIA Tesla S1070 and an ATI Radeon HD 5970 showed that the optimized code has similar or better performance than manually-tuned code.

In order to achieve the best performance of the BELLPACK kernel, Choi and co-workers proposed a framework that models the architectural features of GPUs to automatically choose optimal blocking parameters [17]. Their framework was capable of finding the best implementation with a median error of 15% when compared to exhaustive search.

Guo and co-workers proposed an empirical performance model combined with a partitioning framework that finds the most appropriate storage scheme for each partition of a matrix for the best execution time of SpMV [15]. Their performance model predicts the execution time of SpMV kernels ELL, CSR and HYB on each partition based on the architectural features of a GPU and the characteristics of the partition. The total estimated execution time is the summation of

each partition estimated execution time. Their framework also auto-tunes CUDA code parameters. The framework exhaustively searches parameters values during the first iteration of a computation and uses the parameters with the best performance for the remaining iterations. Their auto-tuning framework achieves 222%, 197% and 33% performance improvements over CSR (vector), ELL and the HYB kernels on an NVIDIA's GeForce GTX 295.

None of the auto-tuning SpMV approaches discussed above deal with automating the selection of the matrix storage scheme. The framework in this paper automates both the selection of the storage scheme and the tuning of the corresponding kernel's parameters. Unlike the exhaustive search framework of Guo and co-workers our framework uses heuristics to speed up auto-tuning.

3 Auto-tuning SpMV

3.1 The Search Space

Sparse matrices use special data structures that store only the nonzero elements to eliminate redundant computations and storage. For SpMV, it is rarely obvious which storage scheme is the most effective for a given matrix [12,16]. Hence it is necessary to automate the process of selecting the best storage scheme for a given matrix. Our study considers the following storage schemes/SpMV kernels described in details elsewhere: DIA, ELLPACK, CSR (vector), COO, and HYB [16], BELLPACK [17], and BTJAD [12].

DIA, ELLPACK and BELLPACK are optimized for matrices with structured sparsity patterns. DIA is suited for matrices whose nonzero elements are concentrated along diagonals. ELLPACK is suited for matrices with nonzero elements per row that do not deviate much from the average. BELLPACK works best for matrices with dense block substructures. CSR and COO work best for highly unstructured sparse matrices since the amount of storage is always proportional to the number of nonzero elements. HYB stores rows of the matrix with close number of nonzero elements per row using ELLPACK and the remaining rows using COO. BTJAD is a blocked version of the Transpose Jagged Diagonal format (TJAD)[20]. In BTJAD, rows of the matrix are divided into blocks and each block is stored using TJAD. The resulting rows are called transpose jagged diagonals (TJDs). The main optimization made in BTJAD is the ability to load and reuse x vector elements in registers.

We performed experiments that measure the performance of these kernels and relate them to matrices characteristics using 250 matrices. These matrices were selected to cover a wide spectrum of sparse matrices dimensions (tens of thousands to millions), number of nonzero elements (hundreds of thousands to tens of millions), variations in rows lengths (highly non-uniform to highly uniform), and sparsity patterns (diagonal, dense sub-blocks, and random). Our experiments show that neither CSR nor COO performs best for any subset of the tested matrices. For any matrix BTJAD always outperforms CSR and HYB always outperforms COO. When the number of rows per block is set to 1, the

BTJAD storage scheme reduces to CSR with the source vector reordered for each row and hence with coalesced accesses to the source vector. As the number of rows per block (BLOCKYDIM) increases, the BTJAD storage scheme benefits from register reuse. Consequently, BTJAD outperforms CSR for all matrices. COO is known for its insensitivity to variations in row lengths. For matrices with highly variable row lengths, HYB stores most of or the entire matrix using COO achieving equivalent or better performance than COO. Based on these conclusions, we exclude both CSR and COO from the search space.

In addition to the diverse matrix sparsity patterns, the complexity and variation in architectural features of GPUs make auto-tuning of kernels parameters necessary. Table 1 lists the kernels parameters that are tuned by our framework. In the table, the term "kernel block size" refers to the number of threads per thread block in the launched kernel, whereas the term "matrix block size" refers to the size of matrix blocks in BTJAD and BELLPACK. In BTJAD, either half a warp or full warp is assigned per matrix block. In addition, each thread reuses loaded elements of the source vector (x) in its registers. Hence, the number of registers used to load these elements is a code parameter.

Table 1. Auto-tuned kernels parameters

Kernel	Kernel Block Size	Number of Launched Threads	Threads per Matrix Block	Matrix Block Size	Number of x Registers
DIA	X				
ELL	X				
HYB	X	X			
BELLPACK	X			X	
BTJAD	X	X	X	X	X

3.2 Heuristics-Based Auto-tuning

Using heuristics, our framework first identifies the set of candidate storage schemes for an input matrix. Then the framework tunes the parameters of the kernels optimized for the candidate storage schemes and executed on a certain GPU. To identify the best kernel, the framework uses a combination of timing of the execution of each candidate kernel and heuristics to prune the search space of kernel code parameters. Figure 1 shows an abstraction of the workflow of our framework. A detailed discussion of the heuristics used by our framework can be found in [21].

Pruning the Search Space of Storage Schemes. We develop heuristics which use computed threshold values of matrix characterization parameters to

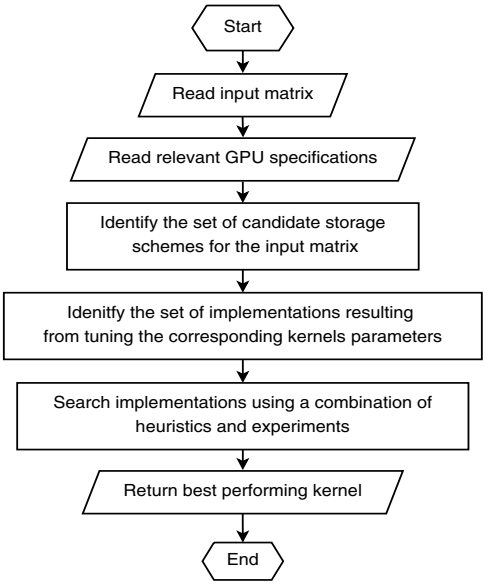


Fig. 1. An abstraction of the auto-tuning workflow

exclude a storage scheme. Threshold values are set based on analysis of experimental results using our 250 matrices. We started our experiments by setting the threshold values based on kernels' performance and characterization parameters for a set of 50 matrices. Threshold values were then fine tuned based on the kernels' performance and characterization parameters after adding another set of 50 matrices to the sample. When repeating this process three times by adding 3 sets of 50 matrices each, we found that threshold values which were set based on experimenting with the first 100 matrices needed no further tuning. Therefore, our auto-tuner uses these threshold values which were obtained and validated using our testing sample of 250 matrices. Figure 2 shows a flow chart of how our heuristics prune the storage schemes search space.

Our experiments show that HYB produces the best performance for matrices with highly non-uniform rows lengths. We considered the use of several different parameters to identify matrices which are best stored using HYB. These parameters were formed using different statistical values of nonzero elements rows lengths (i.e. average, standard deviation, coefficient of variation, and maximum). In addition, we also considered the possibility of defining metrics which describe how the matrix storage was divided into ELLPACK and COO. Of the different parameters, we use the coefficient of variation of nonzero elements row lengths to compare dispersion of row lengths for matrices with different average row lengths. For a given matrix, this is computed by dividing the standard deviation of nonzero elements row lengths by their average. Our experiments show that this parameter distinguishes matrices where nonzero elements row lengths largely deviate from the average since they have large standard deviation values

relative to their average rows lengths. Our experiments also revealed that HYB performs best for matrices with coefficients of variation greater than 20. Hence our framework selects the HYB storage scheme for matrices with coefficient of variation values greater than 20.

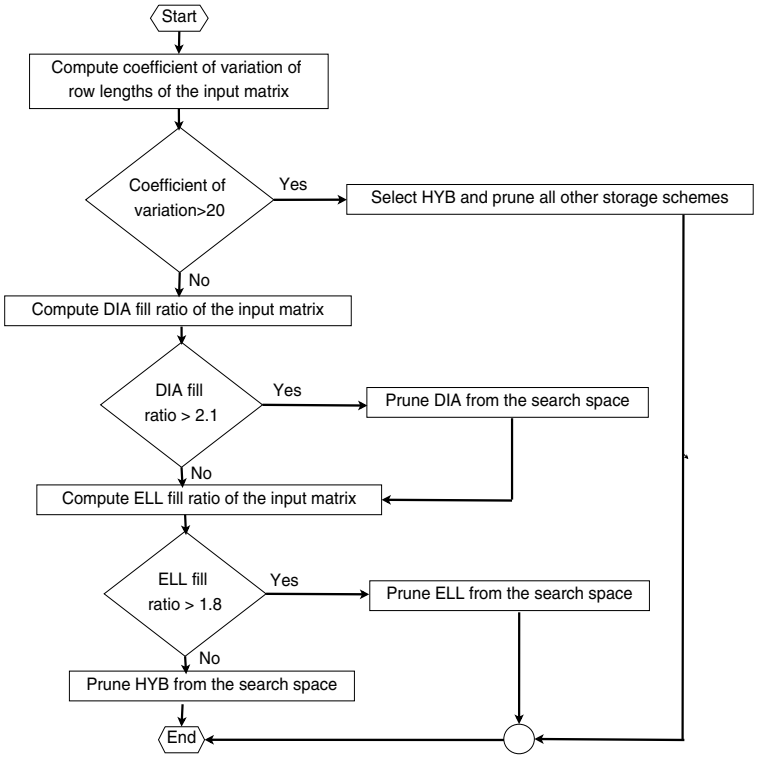


Fig. 2. Using heuristics to prune storage schemes

The performance of DIA and ELLPACK deteriorates as the amount of required zero padding increases. In order to identify when to prune these storage schemes we identified the fill ratio as the parameter that reflects the amount of zero padding [16,22]. It is computed by dividing the number of stored elements by the number of nonzero elements of the matrix. Higher fill ratios indicate more explicitly stored zero elements and consequently lower suitability of the storage format for the given matrix. Our experiments show that DIA does not produce the best performance for any matrix with a DIA fill ratio greater than 2.1 and ELL does not produce the best performance for any matrix with ELL fill ratio greater than 1.8. We observed that for matrices with uniform row lengths HYB stores most of or the entire matrix using ELL, achieving equivalent or lower performance than ELL. Consequently for matrices with ELL fill ratios less than 1.8, we prune HYB.

Pruning the Search Space of Kernels Parameters. For storage schemes that are not pruned, we heuristically or analytically prune implementations that are predicted not to produce the best performance. For DIA, ELL and HYB, we prune thread block sizes that do not maximize the number of threads scheduled per GPU streaming multiprocessor, SM. For BTJAD the matrix block size and number of x registers affect the number of threads scheduled per SM [12]. Our framework prunes all combinations of these parameters that yield a number of scheduled threads per SM which is less than 1/4 of the allowed maximum since our experiments show that they always produce worse performance than other combinations.

For the COO portion of HYB and for BTJAD, the kernel should specify the number of threads to be launched on kernel invocation. Our framework limits the number of launched threads values to those with which the best performance is always achieved based on observations from experiments using our set of 250 matrices.

In order to prune the search space of the blocking parameters values for BELLPACK, we introduce the computation of the BELLPACK fill ratio. BELLPACK performance deteriorates for matrices that do not have dense block substructures. The fill ratio of BELLPACK is computed for each combination of the blocking parameters, discussed in [17], for the given matrix. Based on our experiments, our auto-tuner prunes all blocking parameters values that yield BELLPACK fill ratios larger than 3.

For BTJAD, we develop heuristics to prune the search space of the number of assigned threads per matrix block (warp or half warp), registers used to load x vector elements, and matrix block sizes. Tuning the number of x registers is dependent on the number of nonzero elements in the formed TJDs. Assigning a smaller number of registers means more kernel iterations are needed to load the x vector elements and perform the multiplication. On the other hand, using more registers to load elements from the x vector than required means that some of the registers will not be used. Our framework starts with the smallest number of registers in the search space, which is 2 registers, and continues the search by increasing the number of registers in each implementation. The search stops when performance decreases for two consecutive values for the number of registers. Our code computes the maximum number of registers to be considered for loading x elements, which is for the case when one block is scheduled per SM, based on the total number of registers available in the GPU per SM, the currently considered thread block size, and number of other registers used in the kernel code. The total number of registers available in the GPU per SM is read using the function cudaGetDeviceProperties; a part of the CUDA API. Using the command ptxas, we find the fixed number of registers which is used in the kernel code in addition to registers used to load x vector elements.

For matrices with large numbers of rows, our framework starts with the largest matrix block size in the search space testing possible block sizes in descending order. On the other hand, for matrices with small number of rows, the kernel launches as many threads as needed by the matrix blocks. For such matrices our

framework starts the search with the smallest matrix block size testing possible block sizes in ascending order. In both cases, the framework stops the search when performance decreases for two consecutive values of matrix block size. This heuristic was verified empirically using our 250 matrices.

4 Performance Evaluation

4.1 Experimental Setup

To evaluate our framework we use 59 matrices, a Tesla S1070, and a Fermi class Tesla M2070. We select these two systems in order to test and validate our auto-tuner performance using different generations of NVIDIA GPUs (i.e. Tesla T10 of the S1070 and Tesla T20 of the M2070). The matrices are selected from 12 application areas to include a wide spectrum of sparsity patterns. In addition, the matrices vary in their dimensions and number of nonzero elements. For each matrix, we compare the GFLOP/s of the kernel delivered by our auto-tuning framework and the highest GFLOP/s we measured for NVIDIA's library kernels. In both cases we measure the GFLOP/s as the average obtained over 1000 trials or 3.0 seconds of execution time, whichever is less. The measured time does not include time required to transfer data from the host to the device and back. For a particular kernel and input matrix, the GFLOP/s is computed by dividing twice the number of nonzero elements in the matrix by the measured execution time. We also compare heuristic-based and exhaustive search auto-tuning in terms of the GFLOP/s of the delivered kernels and the auto-tuning times.

4.2 Experimental Results

Speedup over NVIDIAs Library Best Performing Kernels. Our framework delivers kernels that outperform NVIDIA's library best performing kernels for 36 and 40 matrices using the Tesla S1070 and the M2070, respectively. Figure 3 shows the speedup results for these matrices. Using the Tesla S1070, the speedup ranges between 1.2x and 4.2x with an average of 2x. Using the M2070, the speedup ranges between 1.2x and 7x with an average of 2.2x. For the majority of these matrices, our auto-tuner chooses BTJAD and auto-tunes its kernel presented in [12]. Our framework achieves speedup values greater than 2 when it chooses and auto-tunes BTJAD. These results are very close to those reported in [12] where a comparison between the BTJAD implementation found using manual exhaustive search and NVIDIA's kernels is provided. For 7 matrices, our auto-tuner chooses BELLPACK and auto-tunes its kernel. For 2 matrices (1 and 5), our auto-tuner chooses a storage scheme from the set discussed in [16] and auto-tunes the corresponding NVIDIA kernel. For both matrices, it achieves minor speedups by auto-tuning parameters of the best performing kernel of NVIDIA's library.

Executing on the M2070, the performance of both BTJAD and NVIDIA's library kernels improves. Depending on the distribution of nonzero elements in

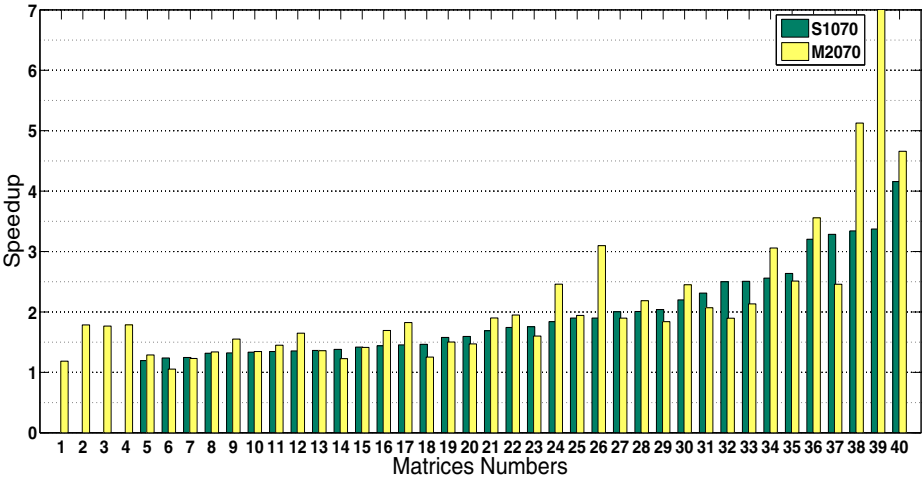


Fig. 3. The speedup achieved by our auto-tuner delivered kernels over the best performing NVIDIA kernels

the input sparse matrix, BTJAD can benefit significantly from a larger register file and shared memory. The M2070 has twice the number of registers and three times the size of shared memory available on the S1070. This explains the greater speedup values achieved using the M2070 for some matrices. For some other matrices, the speedup using the S1070 is greater. This is because the improvement in the performance of BTJAD is less compared to the improvement of the best performing kernel of NVIDIA's library when executing on the M2070.

For matrices not shown in Fig. 3, our auto-tuner chooses a storage scheme from the set discussed by Bell and Garland in [16] and it auto-tunes the corresponding NVIDIA kernel. The selected parameters values turn out to be identical to those chosen by NVIDIA's developers. This is why kernels delivered by the auto-tuner for these matrices have identical performance to the best performing kernels in NVIDIA's library.

Performance Comparison to Exhaustive Search Auto-tuning. We compare the performance of kernels delivered by heuristic-based and exhaustive search auto-tuning. Figure 4 shows the distribution of the performance differences as % of the performance of the exhaustive search kernel for the 59 matrices. The performance of heuristic-based kernels is always at least 93% of the performance of the exhaustive search kernels. The performance of the kernels delivered by our framework is at least 99% of the exhaustive search kernels for 37 and 44 matrices on the S1070 and M2070, respectively.

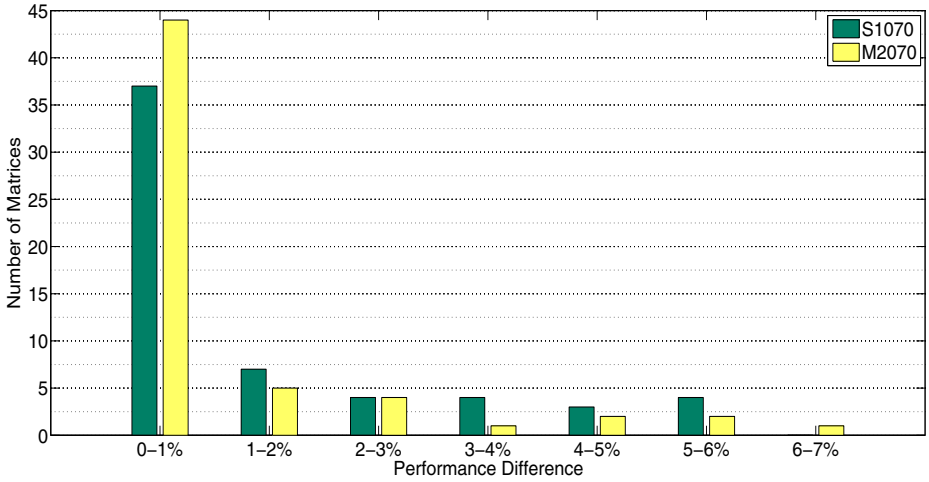


Fig. 4. Histogram of the performance difference between the kernel delivered by our auto-tuner and the kernel found using exhaustive search auto-tuning as % of the performance of the exhaustive search kernel

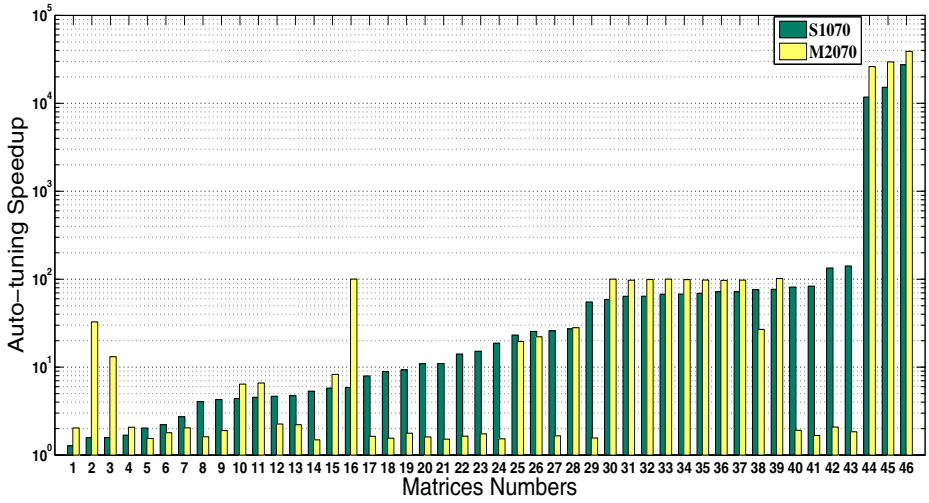


Fig. 5. Auto-tuning speedup due to using heuristics instead of exhaustive search

Auto-tuning Speedup. For an input matrix and a GPU, we define the auto-tuning speedup as the ratio of the exhaustive search to the heuristics-based auto-tuning times. For 46 matrices, Figure 5 shows that the auto-tuning speedup is 2x for at least one of the GPUs. For the S1070 the speedup is greater than 50x for 18 matrices and greater than 10x for 27 matrices. The speedup is greater than

4x for 39 matrices, and at least 2x for 42 matrices. The speedup is greater than 1.5x for all 59 matrices except 3. For the M2070 the speedup is greater than 90x for 13 matrices and greater than 10x for 19 matrices. The speedup is at least 2x for 28 matrices and greater than 1.5x for all 59 matrices. For some matrices, the significant difference in auto-tuning speedup using the two GPUs is due to the randomness in the way BTJAD heuristics prune its search space. For 3 matrices and both GPUs, the auto-tuning speedup is more than ten thousand times. For these matrices, our auto-tuning framework selects HYB. This leads to significant savings in the search time because we avoid the very expensive consideration of BELLPACK and BTJAD. Storing each of these matrices using blocked schemes is very time-consuming.

5 Conclusions and Future Work

We presented a heuristics-based auto-tuning framework for SpMV on GPUs. Our framework generates an optimized SpMV kernel for a given sparse matrix and GPU by selecting the best storage scheme and automatically tuning parameters of the corresponding kernel. The kernels considered by our framework are the NVIIDIA's library kernels, BELLPACK and the BTJAD kernels [17,12]. We demonstrated the performance improvement of our framework delivered kernels over NVIDIA's library kernels using two different generations of NVIDIA GPU architectures. Our framework delivers kernels with GFLOP/s very close to those delivered by exhaustive search kernels. However, our framework reduces the auto-tuning time significantly.

Pruning the BELLPACK storage scheme would speedup the auto-tuning process significantly but this requires computing the fill ratio of the matrix which is done after storing it using BELLPACK. Storing a matrix in the BELLPACK format requires prohibitively long time for matrices with unstructured sparsity patterns. Our future work includes investigating the validity of using an estimation of the BELLPACK fill ratio based on sampling of the sparse matrix instead of computing the exact fill ratio [22]. We also plan to investigate the possibility of integrating the model-driven framework proposed in [17] for choosing the best blocking parameters when BELLPACK is selected. In addition, our future work includes investigating the implementation and auto-tuning of SpMV using multiple GPUs with MPI across GPUs.

Acknowledgments. This work was supported in part by the University of Jordan. It used the AC cluster [23] which was operated by the Innovative Systems Laboratory (ISL) at the National Center for Supercomputing Applications (NCSA) of the University of Illinois. The cluster was funded by NSF SCI 05-25308 and CNS 05-51665 grants along with generous donations of hardware from NVIDIA, Nallatech, and AMD. This work also used the LinkSCEEM CyTera cluster and was supported by the LinkSCEEM-2 project, funded by the European Commission under the 7th Framework Programme through Capacities Research Infrastructure, INFRA-2010-1.2.3 Virtual Research Communities,

Combination of Collaborative Project and Coordination and Support Actions (CP-CSA) under grant agreement no RI-2616000.

References

1. NVIDIA Corporation. NVIDIA CUDA C Programming Guide (version 5.0), http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf
2. The Portland Group Corporation. CUDA FORTRAN Programming Guide and Reference (version 12.10), http://www.pgroup.com/doc/pgicudaforug.pdf
3. Microsoft Corporation. C++ Accelerated Massive Parallelism Overview, Visual Studio (2012), http://msdn.microsoft.com/en-us/library/vstudio/hh265137.aspx
4. OpenACC Corporation. OpenACC Home, http://www.openacc-standard.org
5. Khronos Group. The OpenCL Specification (version 1.2), http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf
6. Stratton, J.A., Anssari, N., Rodrigues, C., Sung, I., Obeid, N., Chang, L., Liu, G., Hwu, W.: Optimization and Architecture Effects on GPU Computing Workload Performance. In: Proc. 2012 Innovative Parallel Computing: Foundations and Applications of GPU, Manycore, and Heterogeneous Systems, San Jose, CA, pp. 1–10 (2012)
7. Whaley, R., Petitet, A., Dongarra, J.: Automated empirical optimizations of software and the ATLAS project. Parallel Computing 27(1-2), 3–25 (2000)
8. Vuduc, R., Demmel, J., Yelick, K.: OSKI: A library of automatically tuned sparse matrix kernels. In: Proceedings of SciDAC 2005, San Francisco, CA, USA. Journal of Physics: Conference Series (2005)
9. Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., Demmel, J.: Optimization of Sparse Matrix-vector Multiplication on Emerging Multicore Platforms. In: 2007 ACM/IEEE Conference on Supercomputing, New York, USA (2007)
10. Jain, A.: pOSKI: An Extensible Auto-tuning Framework to Perform Optimized SpMV on Multi-core Architectures. Master's thesis, University of California, Berkeley, CA, USA (2008)
11. Williams, S., Bell, N., Choi, J.W., Garland, M., Oliker, L., Vuduc, R.: Sparse Matrix-Vector Multiplication on Multicore and Accelerators. In: Dongarra, J., Bader, D.A., Kurzak, J. (eds.) Scientific Computing with Multicore and Accelerators, pp. 83–109. CRC Press (2010)
12. Abu-Sufah, W., Abdel Karim, A.: An effective approach for implementing sparse matrix-vector multiplication on graphics processing units. In: Proceedings of the 14th IEEE International Conference on High Performance Computing and Communications, Liverpool, UK, pp. 453–460 (2012)
13. El-Zein, A., Rendell, A.: From Sparse Matrix to Optimal GPU CUDA Sparse Matrix Vector Product Implementation. In: 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, Melbourne, Australia, pp. 808–813 (2010)
14. Grewe, D., Lokhmotov, A.: Automatically Generating and Tuning GPU Code for Sparse Matrix-Vector Multiplication from a High-Level Representation. In: Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units, New York, USA (2011)

15. Guo, P., Huang, H., Chen, Q., Wang, L., Lee, E.-J., Chen, P.: A Model-Driven Partitioning and Auto-Tuning Integrated Framework for Sparse Matrix-Vector Multiplication on GPUs. In: Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery. ACM, NY (2011)
16. Bell, N., Garland, M.: Implementing Sparse Matrix-vector Multiplication on Throughput-oriented Processors. In: Proceedings of the 2009 ACM/IEEE Conference on Supercomputing, NY, USA, pp. 1–11 (2009)
17. Choi, J., Singh, A., Vuduc, R.: Model-driven Autotuning of Sparse Matrix-vector Multiply on GPUs. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 37–48. ACM, Bangalore (2010)
18. Davis, T.: The University of Florida Sparse Matrix Collection. NA Digest, 97(23), `http://www.cise.ufl.edu/research/sparse/matrices/`
19. Monakov, A., Lokhmotov, A., Avetisyan, A.: Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures. In: Patt, Y.N., Foglia, P., Duesterwald, E., Faraboschi, P., Martorell, X. (eds.) HiPEAC 2010. LNCS, vol. 5952, pp. 111–125. Springer, Heidelberg (2010)
20. Montagne, E., Ekambaram, A.: An Optimal Storage Format for Sparse Matrices. Information Processing Letters 90(2), 87–92 (2004)
21. Abdel Karim, A.: Auto-tuning Data Structures of Sparse Matrix-Vector Multiplicaqtion on Graphics Processing Units. Masters thesis, University of Jordan, Amman, Jordan (2012)
22. Vuduc, R.: Automatic Performance Tuning of Sparse Matrix Kernels. Ph.D. thesis, University of California, Berkeley, Berkeley, CA, USA (2003)
23. Kindratenko, V., Enos, J., Shi, G., Showerman, M., Arnold, G., Stone, J., Phillips, J., Hwu, W.: GPU Clusters for High-Performance Computing. In: Proc. Workshop on Parallel Programming on Accelerator Clusters, IEEE International Conference on Cluster Computing (2009)

A Simple Concept for the Performance Analysis of Cluster-Computing

Heinz Kredel[1], Sabine Richling[2], Jan Philipp Kruse[3], Erich Strohmaier[4], and Hans-Günther Kruse[1]

[1] IT-Center, University of Mannheim, Germany
{kredel,kruse}@rz.uni-mannheim.de
[2] IT-Center, University of Heidelberg, Germany
richling@urz.uni-heidelberg.de
[3] Institute of Geosciences, Goethe University Frankfurt, Germany
jp.kruse@gmail.com
[4] Future Technology Group, Lawrence Berkeley National Laboratory, Berkeley
estrohmaier@lbl.gov

Abstract. There seems to be a lack of reliable thumb rules to estimate the size and performance of clusters with respect to applications. Since modern cluster architecture is based on multi-cores we follow a concept derived by S. Williams et. al. for the analysis of such systems. The performance is described by the dimensionless speed-up in dependence on important hardware and application parameters. The hardware parameters are the number and the theoretical performance of each processing unit and the bandwidth of the network. The application parameters are the total number of operations performed on a number of bytes and the total number of bytes communicated between the processing units. In order to test our theoretical concept we apply our model to the scalar product of vectors, matrix multiplication, Linpack and the TOP500-list.

Keywords: performance model, performance analysis, compute clusters, roofline model.

1 Introduction

In the last century the methods of performance analysis of computers and their networks have been developed with very sophisticated mathematical tools.

Within the framework we presented in earlier publications [1, 2, 3] some adequate results in performance prediction by a classical stochastic approach. But most applications in High Performance Computing (HPC) are straightforward and the stochastic method seems not to be the appropriate tool. This seems to be true for web applications [24] too.

Further it seems that a detailed analysis of systems is mostly a waist of time. The performance of systems changes rapidly due to hardware development rendering available analysis methods unpractical and slow to adapt. There is a lack of reliable thumb rules to estimate the size and the performance of cluster systems or server farms. The classical concepts developed for mainframes and

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 165–180, 2013.

time-sharing systems (as discussed for example in [1]) are not appropriate for clusters with more than 100.000 cores or for very big server farms.

Considering all this, we propose a simple and transparent approach in analyzing the performance by simple mathematics – inspired by the ideas of S. Williams et al. [4] for the roofline model of multi-cores. In the roofline model an upper bound for the performance is modeled by the maximum of the peak flop rate, and the product of streaming memory bandwidth times the flop to byte ratio. In our approach we describe the performance by the dimensionless speed-up expressed with the help of the product of the measured network bandwidth times the flop to exchanged bytes ratio. Compared to the laws of Amdahl [5] and Gustafson [6], we try to incorporate important characteristics of the application and the hardware. Further we can obtain very easily the scaling in the number of cores, the problem size or the bandwidth of the internal network. With this approach it could be possible to find a transparent and homogenous method of modeling the whole cluster with all its elements – from cores and multi-cores to nodes and regions of nodes.

Related work is discussed in Sect. 2 and the assumptions underlying our model are described in Sect. 3. In Sect. 4 we present our performance model which is based on ideas presented in [11, 10] and derive general equations for the performance and for the speed-up. In Sect. 5 we take account of the parameters of the applications and show how to apply the resulting equations to specific applications. The application of our model to real systems is discussed in Sect. 6. Finally Sect. 7 draws some conclusions.

2 Related Work

In the field of cluster and parallel computing there exists a lot of interesting analysis related to our approach.

Hockney proposed some strategies to describe the performance of parallel systems. In an earlier paper and book [7, 8] he classified vector and parallel machines by computing the overall performance depending on the vector lengths n, $n_{1/2}$, the computational force or intensity f, and some hardware characteristics like the number of processors p, the single processor performance, and the communication. In general our proposed methods are similar but they are based directly on a cluster and multi-core architecture, not yet available in the 1980s. Furthermore Hockney focused on the discussion of benchmark runtimes and performance. The concept of dimensionless parameters was not considered at that time. A second and later work [9] searched for some analogies to Fluid Dynamics in order to explain performance and computational similarity [9] by introducing dimensionless parameters δ_{1-3} (comparable to theg Reynold Number). He picked up his earlier ideas on the performance based on computation and communication phases and discussed the speed-up as a dimensionless variable. In principle our argumentation is close to his guidelines, but there is a big difference in the number of dimensionless parameters and the integration of hardware and software characteristics.

Numrich has built up some smart theories in performance analysis. The first one [16] is based on an approach by Newtons classical mechanics. In an abstract computational space, the actions of the software correspond to the masses and the computational intensity corresponds to the forces of mechanics. The principle of least action determines the dynamics by the shape of the potential function. With the computational action S it is possible to derive a metric space for programs. But it seems not easy to integrate the architecture of a parallel system, the complex application software and communication. Hence this approach is not successful in order to describe the performance by one intuitive and dimensionless parameter. Apart from the fact, that classical mechanics and performance of modern computer systems must not follow the same laws stringently, one of the most doubtful assumptions is the use of the energy conservation theorem, which is true in physical time-invariant system but is not necessarily true in the world of computer systems. We are not sure that a purely formal analogy is sufficient to describe systems correctly. The other theory of Numrich is based on the dimension analysis [17] and the Pi-theorem of Buckingham [18]. The argumentation seems to be very abstract and not close to practical experience, but nevertheless they give useful hints for algorithm and hardware design. In addition one finds some interesting examples and model validations. But in summary it seems far from our goal to find simple thumb rules for cluster computing.

Luszczek and Dongarra [14] developed a very comprehensive model for the Linpack in High Performance Computing (HPC) [12] without looking inside the software. They use a simple polynomial of 3rd order in the matrix size n for determining the runtime t. All parameters of the system are hidden in the coefficients of this polynomial. Their values can be taken from experiments by varying n and measuring the corresponding runtime. An explicit and simple dependence on the number of processing elements, the computational intensity, or the network bandwidth is not easy to extract, but it is an excellent work to learn more about Linpack and ScaLApack [13].

Our earlier work [2, 3] used an elaborate stochastic approach which we can now avoid. Moreover we now have a communication term of the application for more realistic analysis. The earlier papers were focused on the performance of two distributed clusters coupled by an InfiniBand network and the only application was Linpack. The communication within the cluster was neglected completely and there was no way to treat inhomogeneous systems.

3 Assumptions and Goals

In this section we will determine our goals and describe the necessary assumptions of our methodology. The goals or guidelines are

- simplicity and transparency in the modeling process,
- few and directly available parameters to describe the characteristics of hardware and software of cluster systems,
- speed-up as a dimensionless metric in order to describe the dependencies on the number of cluster elements and the problem size,

- to find the size of a cluster which assures a reasonable or optimal runtime of a given application,
- validation of our approach by some empirical experiments with standard application.

We need to bear in mind that we cannot consider all hardware and communication topologies nor all application software parameters. Therefore we choose a model with p processing elements or cores (called PUs), which have on the multi-core level only one link to the communication network. There are no dependencies between the computations on different PUs, the effects of synchronization are neglected. Each PU is characterized by the theoretical performance which is measured in floating point operations per second (flop), memory operations remain out of consideration. The network is characterized by the bandwidth only (measured in GBytes/sec), the effects of topologies are neglected, too. The influence of the latency we have studied in earlier publications [2, 3] with a more elaborate stochastic model, where it turned out, that the importance of this parameter is negligible for our purpose. Furthermore we want to avoid more free parameters, although Numrich in his work [18] has shown how to hide latency in a dimensionless parameter. We believe that latency is partly included in our additive term for the communication time. The good qualitative agreement of our results with the measured data is a confirmation for this strategy. Regarding the application software we introduce three parameters, the number of operations, the number of bytes used by the operations and the number of bytes exchanged by the PUs, without any protocol overhead. The exchange parameter emphasizes the difference between two earlier approaches [15, 11], since it allows to integrate the communication on the level of the actual algorithm or application.

Our theory about the performance of cluster computing is very general and includes heterogenous systems, too. This is important if we want to analyze regions with different multi-core nodes. But in a first step we apply our methods only to homogenous clusters, the most common systems in use today. The presented modeling ignores memory and cache behavior of the application code which is an important aspect in the performance analysis of a cluster. But in a first step we are regarding the cluster on the level of the nodes, as black boxes, only. In a second and future step we want to include the analysis of multi-cores in agreement with the approaches of [4]. Our approach is open for non-numerical applications too, like data intensive computing [23] or Web-applications [24].

4 Modeling

According to the previous section we propose a model with p processing units (PUs), connected by a network with a bandwidth b_c [GByte/sec]. The theoretical peak performance $l^{\text{peak}}_{k=1,\ldots,p}$ of each PU is measured in [Gflops/sec], the application is described by the total number of operations $\#op$ on $\#b$ bytes. In the original setting of the roofline model there is also the theoretical performance (in Gflop/sec) but instead of our network bandwidth, the memory bandwidth (measured by Stream BW) is considered. The operations are distributed on the PUs ($k = 1, \ldots, p$) according to

$$o_k = \omega_k \cdot \#op, \quad d_k = \delta_k \cdot \#b \quad \text{with} \quad \sum_{k=1}^{p} \omega_k = \sum_{k=1}^{p} \delta_k = 1 \tag{1}$$

The homogeneous case with $\omega_k = 1/p$, $\delta_k = 1/p$ as well as $l_k^{\text{peak}} = l^{\text{peak}}$ for $k = 1, \ldots, p$ will be discussed in detail in Sect. 5. The primary performance quantity of the system is the total time t to process the load $(\#op, \#b)$. Derived quantities are the performance[1] l as a function of the number of PUs

$$l(p) = \frac{\#op}{t} \tag{2}$$

and the speed-up

$$S = \frac{l(p)}{l(1)}, \tag{3}$$

which we want to express as a function of $(\#op, \#b)$, (ω_k, δ_k), and (l_k^{peak}, b_c). If the $k = 1, \ldots, p$ PUs work sequentially, each with a time $t_k(o_k, d_k)$, it follows

$$t = \sum_{k=1}^{p} t_k(o_k, d_k). \tag{4}$$

We skip this rather uninteresting case and concentrate on parallel working PUs only. Then, the total execution time t^r for the operations is

$$t^r = \max\{\, t_1(o_1, d_1), \ldots, t_n(o_p, d_p) \,\}. \tag{5}$$

In addition, there is a communication time t^c, which can be estimated by the total number of bytes $\#x$ to be exchanged and the aggregate bandwidth b_c between the PUs,

$$t^c \simeq \frac{\#x}{b_c}. \tag{6}$$

This additive term is in agreement with a lot of approaches like Hockney [9] or Numrich [18]. But in contrast to these works we have a different argumentation for the latency – see Sect. 3. For the rest of this section let $k = k_{\max}$ be the cluster element (PU), which determines the maximum of (5), then it follows

$$t \simeq t^r + t^c = t_k(o_k, d_k) + \frac{\#x}{b_c}. \tag{7}$$

The time t_k can be determined similar to (6),

$$t_k \simeq \frac{o_k}{l_k} \geq \frac{o_k}{l_k^{\text{peak}}}. \tag{8}$$

[1] We use l instead of P to avoid confusion with p.

With (1) we get

$$t \geq \omega_k \cdot \frac{\#op}{l_k^{\text{peak}}} + \frac{\#x}{b_c} = \omega_k \cdot \frac{\#op}{l_k^{\text{peak}}} \cdot \left(1 + \frac{l_k^{\text{peak}}}{b_c} \cdot \frac{\#b}{\omega_k \#op} \cdot \frac{\#x}{\#b}\right). \tag{9}$$

With this, the performance l is bounded by

$$l \leq \frac{l_k^{\text{peak}}}{\omega_k} \cdot \frac{x_k}{1 + x_k} \leq \frac{l_k^{\text{peak}}}{\omega_k}, \tag{10}$$

where we defined

$$\begin{aligned} &x_k = \omega_k \cdot \tfrac{a}{a_k^*} \cdot r \quad \text{with} \\ &a = \tfrac{\#op}{\#b}, \quad a_k^* = \tfrac{l_k^{\text{peak}}}{b_c}, \quad r = \tfrac{\#b}{\#x}. \end{aligned} \tag{11}$$

The performance behavior of the system "application plus hardware" is described by a dimensionless parameter x_k. Like other analysis we observe that the ratios $\frac{a}{a^*}$ and $r = \frac{\#b}{\#x}$ have an important influence on the behavior of the speed-up. The parameter a is the computational force or intensity of the software [17] and a^* of the hardware. Only for $a > a^*$ we expect significant performance values, but more important is $r = \frac{\#b}{\#x}$. Decreasing $\#x$ by a better algorithm increases r and results in a higher speed-up. In the case $\#x = 0$, no communication or all communication is parallel, we get the ideal linear speed-up (see 14: if $\#x \to 0$ then $r \to \infty$ and $S \to p$). In principle our procedure is in agreement with the dimension analysis by Hockney [9] and Numrich [18], but we try to avoid a lot of theoretical overhead by using our variable x. Its derivation and meaning comes from practice and it is calculated easily.

For the calculation of the speed-up we need the performance of the whole application $(\#op, \#b)$ on a single PU. Without loss of generality, we can choose PU $k = k_{\max}$ of (5) and with $\omega_k = 1$ we receive for the speed-up

$$S = \frac{l_k(\omega_k < 1)}{l_k(\omega_k = 1)} = \frac{1 + x_k(\omega_k = 1)}{1 + \omega_k \cdot x_k(\omega_k = 1)}, \tag{12}$$

where $x_k = \omega_k \cdot x_k(\omega_k = 1)$ according to (11). We further factorize $x_k(\omega_k = 1)$ in dimensionless parameters

$$x_k(w_k = 1) = \frac{a}{a_k^*} r = a \frac{b_c}{l_k^{\text{peak}}} r = a \cdot \frac{b_c^0}{l_k^{\text{peak}}} \cdot \frac{b_c}{b_c^0} \cdot r \tag{13}$$

and define $\hat{x}_k = a \cdot (b_c^0 / l_k^{\text{peak}})$ and $z = b_c / b_c^0$. b_c^0 is an arbitrary reference bandwidth. Since $\omega_k \leq 1$ for all $k = 1, \ldots, p$, we write $\omega_k = \omega(k,p)/p$ with $\omega(k,p) \leq p$ and for the speed-up follows

$$S = \frac{1 + \hat{x}_k \cdot r \cdot z}{1 + \omega(k,p) \cdot \frac{\hat{x}_k \cdot r \cdot z}{p}}. \tag{14}$$

The speed-up (14) can be used to analyze both *strong scaling* (Amdahl) by fixing the problem size n (so fixing a) and increasing the number of processing units p

as well as *weak scaling* (Gustafson) by changing p to $p + \Delta p$ and increasing the problem size n (so increasing a). Both scalings can easily be observed in figures 1, 2 and 3.

In case of a homogenous cluster we have $\omega(k, p) = 1$ and the index k in (10 - 14) can be omitted ($\hat{x}_k = \hat{x}$ for all k). So the speed-up (14) becomes

$$S = \frac{1 + \hat{x} \cdot r \cdot z}{1 + \frac{\hat{x} \cdot r \cdot z}{p}}. \tag{15}$$

Hence we achieve our goal to describe the performance of the most modern cluster systems by a 4-tuple of dimensionless parameters ($\hat{x}$, r, z, p).

5 Application-Oriented Analysis of Speed-Up and Performance

In the previous section we avoided to discuss a concrete application and we only preferred to elaborate the dependence of the speed-up from performance l_k^{peak} ($k = 1, \ldots, p$), bandwidth b_c and the number p of PUs.

Now we try to discuss the application problem and for that reason we introduce the problem size n. Examples for n are the dimension of vectors in a scalar product or the dimension of matrices in linear equations. To characterize the chosen application we need three variables: $\#op$ (number of operations), $\#b$ (number of bytes required for the operations) and $\#x$ (number of bytes exchanged by the PUs). We assume that $\#op$ and $\#b$ depend only on the problem size n (not on p) and that means for the operational intensity $a = \#op/\#b = a(n)$ and for the dimensionless parameter $\hat{x}_k = a(n)/a_k^*$. The constant a_k^* is only defined by hardware characteristics. High values of $a(n)$ reflect operation-intensive applications which are able to exploit the theoretical performance of the system.

If we analyze the product $\hat{x}_k \cdot r \cdot z$, the dependence on the number p of PUs is concentrated in the ratio $r = \#b/\#x$. This parameter has a strong influence on the speed-up, and depends only on problem size n and the number p of PUs. But without any big loss of generality we can suggest

$$r(n, p) = \frac{c(n)}{d(p)} \tag{16}$$

where nominator and denominator are monotone increasing functions, for example polynomials or logarithmic terms. The details depend on the algorithm. We will show this for some applications later.

For further discussion we analyze the performance also in the case of a homogenous cluster as follows

$$l \leq \frac{l^{\text{peak}}}{\omega} \frac{x}{1 + x} = p\, l^{\text{peak}} \frac{x}{x + 1}, \tag{17}$$

with (13) $x = \hat{x} \cdot z \cdot r(n, p)/p$. For better insight we define $y = \hat{x} \cdot z$ and the performance results in

$$l \leq p\, l^{\text{peak}} \frac{x}{x + 1} = l^{\text{peak}} y \cdot \frac{r(n, p)}{1 + y \frac{r(n,p)}{p}} \tag{18}$$

We turn now to the determination of a lower and upper bound for the number p of PUs in the given setting of a homogenous cluster. Such bounds are interesting for the optimal sizing or using of parts of a cluster with maximal interconnect bandwidth. For a given application at least $p_{1/2}$ PUs are necessary to obtain half of the maximal performance and using more than $p_{\max}$ PUs is a waste of resources. In order to obtain a lower bound for the number of PUs we follow the approach of Hockney [7, 9] for a half performance PU number, which we call $p_{1/2}$. We conclude from (18) with half of the (peak) performance

$$\frac{1}{2}\, p\, l^{\text{peak}} = p_{1/2}\, l^{\text{peak}}\, \frac{x(p_{1/2})}{1 + x(p_{1/2})}. \tag{19}$$

This equation now fixes $p_{1/2}$ by the new equation

$$y \cdot r(n, p_{1/2}) = p_{1/2} \tag{20}$$

For an upper bound, i.e. the number of PUs at maximum performance $p_{\max}$, we need a little bit more algebra. If we determine $p_{\max}$ by the maximum of the performance l in (18)

$$\frac{dl}{dp} = l^{\text{peak}} y \frac{d}{dp} \left(\frac{r(n,p)}{1 + y\frac{r(n,p)}{p}} \right) = 0 \tag{21}$$

which results in

$$r'(p_{\max}) + y\frac{r^2(p_{\max})}{p_{\max}^2} = 0 \tag{22}$$

The further analysis depends strongly on the explicit form of $r(n, p)$. Going back to the assumption (16), which is true for the most applications, equation (22) reduces to

$$p_{\max}^2 \cdot d'(p_{\max}) = y = \hat{x} \cdot z \cdot c(n) \tag{23}$$

The choice of the interval $[p_{1/2}, p_{\max}]$ is not canonical. Sometimes it seems better to calculate a lower bound bigger than $p_{1/2}$ (for example $0.9\, p_{\max}$) and to set the interval to $[0.9\, p_{\max}, 1.1\, p_{\max}]$ – but we prefer Hockney's classical argumentation.

6 Application to Scalar-Product, Matrix Multiplication, Linpack and TOP500

Our more general discussions in Sect. 4 and 5 are theoretical and far from practice, if we do not test the results on real systems – as we have done already in previous works [2, 15].

We use homogeneous clusters from the bwGRiD, described in detail in [19]. For example each of the two sites at the Universities of Heidelberg and Mannheim consists of 140 nodes (8 cores per node) interconnected by InfiniBand. The only important data of the hardware (for our modeling procedure) are

$$\begin{aligned} l^{\text{peak}} &= 8 \quad \text{GFlop/sec for one core,} \\ b_c &= 1.5 \text{ GByte/sec node-to-node,} \\ b_c^0 &= 1.0 \text{ GByte/sec.} \end{aligned} \tag{24}$$

The chosen examples in the following three sections reflect some relevant cases of the general speed-up in (15) with assumption (18). An increasing speed-up without saturation demonstrates section 6.2, while sections 6.1 and 6.3 show a constant behavior in case of very big p-values (like Amdahl's law).

6.1 The Scalar-Product of Vectors

A very simple and easy to explain example is the scalar-product of two vectors $u = (u_1, \ldots, u_n), v = (v_1, \ldots, v_n)$ with n components $(u, v) = \sum_{k=1}^{n} u_k \cdot v_k$ computed on p PUs. On each PU we store two sub-vectors of length n/p, compute the the sub-product, store it in one PU and sum all sub-products finally. The initial distribution of the vectors is ignored, only the data exchanged during computation is counted by $\#x$. The data are

$$\begin{aligned} \#op &= 2\,n - 1 \simeq 2n \;\text{ if }\; n \gg 1 \\ \#b &= 2\,n\,w, \quad \#x = p\,w = 8p, \end{aligned} \tag{25}$$

where $w = 8$ [Byte]. With (25) it follows for the operational intensity $a = \frac{\#op}{\#b} = \frac{1}{8}$, $a^* = 8/1.5$, $z = 3/2$, $r(n,p) = \frac{\#b}{\#x} = \frac{2n}{p}$ and $x = \frac{6}{128} \cdot \frac{n}{p}$. The index $k = 1, \ldots, p$ from (11,13) can be omitted if we work with a homogenous cluster. The speed-up results in

$$S = \frac{1 + \frac{3}{64} \cdot \frac{n}{p}}{1 + \frac{3}{64} \cdot \frac{n}{p^2}} \tag{26}$$

and shows no significantly high values for lower problem sizes $n \leq 500.000$, see Fig. 1. For every configuration of p PUs we measured 20 runtimes and canceled the minimum and maximum. From the rest we computed the mean value t_m and the std-deviation s. In a second step we neglect values outside the interval $(t_m - s, t_m + s)$ and compute the final mean time-value, the speed-up and the the error-limits.

This is an important example with a very low computational intensity $a = 1/8$ which is even independent from problem size n and only driven by the bandwidth of the network. Calculating $p_{\max}$ by (22) we get $p_{\max} = \frac{1}{8}\sqrt{3n}$ for the used cluster, the broad behavior of S suggests an optimal region of PUs by the interval $[0.9 \cdot p_{\max}, 1.1 \cdot p_{\max}]$.

6.2 Analysis of Matrix-Matrix Multiplication

We look at the matrix multiplication $A^{n\times n} \cdot B^{n\times n} = C^{n\times n}$ on a $\sqrt{p} \cdot \sqrt{p}$ processor-grid of a homogenous cluster ($\omega(k,p) = 1$ in (14)) and choose an algorithm with a block-size of $\frac{n}{\sqrt{p}} \cdot \frac{n}{\sqrt{p}}$. In order to calculate one block in C we need to transfer $\sqrt{p} - 1$ blocks of A and B. Hence in summary we have to move $2(\sqrt{p} - 1) \cdot p$ matrix blocks. With (11, 13) our parameters are:

$$\begin{aligned} \#op &= 2n^3 - n^2 \simeq 2n^3, \\ \#b &= 2n^2 w, \\ \#x &= 2n^2\sqrt{p}(1 - \tfrac{1}{\sqrt{p}})w \simeq 2n^2 w\sqrt{p}, \end{aligned} \tag{27}$$

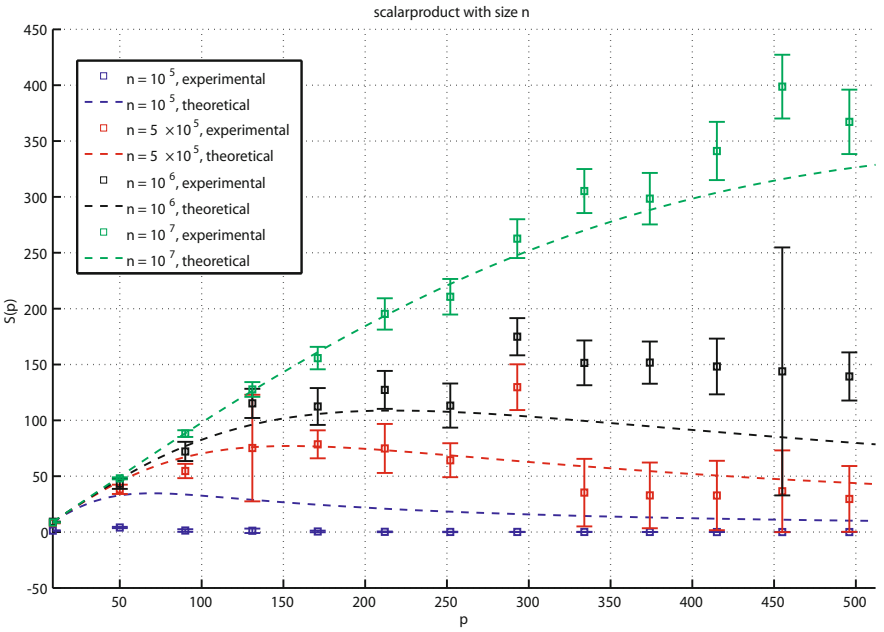


Fig. 1. Theoretical and empirical speed-up for the scalar product with four different problem sizes

$$\begin{aligned} a &= \frac{\#op}{\#b} = \frac{n}{w}, \\ r &= \frac{\#b}{\#x} = \frac{1}{\sqrt{p}}, \\ x &= \frac{a}{a^*} rz = \frac{z}{a^* w} \cdot \frac{n}{\sqrt{p}}. \end{aligned} \tag{28}$$

With the values of the available clusters (24) we derive $x = \frac{1.5}{8 \cdot 8} \cdot \frac{1}{8} \frac{1}{2} p \cdot \frac{n}{\sqrt{p}} = \frac{3}{2048} n \sqrt{p}$, where the factor $z = \frac{1}{8} \frac{1}{2} p$ represents the aggregate bandwidth. The aggregate bandwidth depends on the used number of cores per node sharing a single InfiniBand link. The factor 1/8 is for 8 cores per node, we also have runs with 4 cores per node. Fig. 2 shows the speed-up in the case of matrix multiplication.

According to (22) or (23) with $d(p) = \frac{1}{\sqrt{p}}$ we cannot find a real $p_{\max}$, since $d'(p) < 0$. The speed-up is increasing like $\sqrt{p}$ and we observe a good agreement with Numrich's theory.

6.3 Analysis of Linpack

This section covers the solving of big linear equations $Ax = b$ with Linpack - it is a short summary of our earlier and detailed analysis [3, 10]. The problem (matrix) size is $n = 10.000, 20.000, 40.000$. The relevant parameters of the

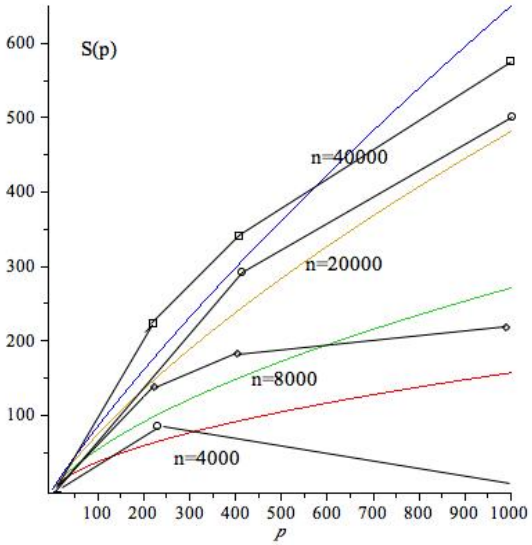


Fig. 2. Speed-up for matrix-matrix multiplication on a homogenous cluster, comparison of theoretical and measured values for 4 different problem sizes n

software are

$$\begin{aligned} \#op &= \tfrac{2}{3}n^3 \\ \#b &= 2n^2 \cdot w \\ \#x &= 3\alpha\left(1 + \tfrac{\log_2 p}{12}\right) n^2 \cdot w \end{aligned} \tag{29}$$

w represents the number of bytes per word ($w = 8$ Byte). With this and by (10-15) we are able to derive the dimensionless parameters

$$\hat{x} \cdot z \cdot r = \frac{n}{128}\frac{2}{3\alpha}\frac{1}{1 + \frac{\log_2 p}{12}} \tag{30}$$

which fix the performance or the speed-up with $\hat{x} \cdot z = (n/128)$ and $\omega = 1/p$. With the approximation of $1 + \frac{\log_2 p}{12} \sim 2$, which is reasonable for $p < 10.000$, and the value $\alpha = 1/3$ we get for the speed-up

$$S = \frac{1 + \hat{x} \cdot r \cdot z}{1 + \frac{\hat{x} \cdot r \cdot z}{p}} \sim \frac{1 + \frac{n}{128}}{1 + \frac{n}{128 \cdot p}}. \tag{31}$$

This result shows a good agreement with empirical data, see Fig. 3, in regions where we have measured data and for $\alpha = 1/3$. Smaller values of α yield better fits for low numbers of PUs p, which indicates that there is eventually a dependence on p contained in α. We will have to investigate this for larger problem sizes n.

From (20) and the above approximation for the logarithmic term follows

$$p_{1/2} = \frac{y}{3\alpha}, \tag{32}$$

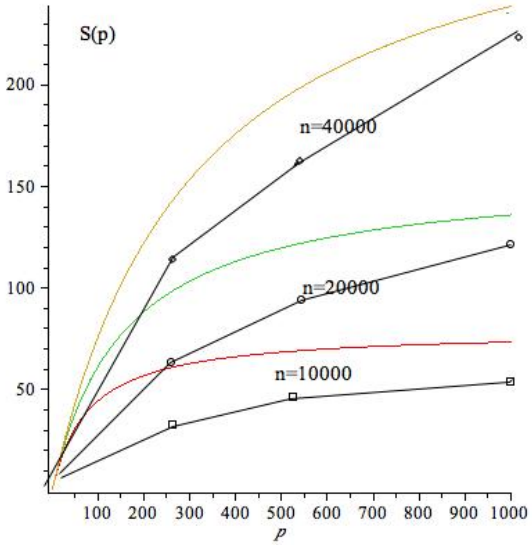


Fig. 3. Speed-up for Linpack solving $Ax = b$ on a homogeneous cluster, comparison of theoretical and measured values for three different problem sizes n

which gives a lower bound of $p_{1/2} = n/128$. For the upper bound we calculate from (23) $p_{\max} = (24 \cdot \ln 2/128) \cdot n = 24 \ln(2) p_{1/2}$. Choosing $n = 10.000$ it follows that reasonable performance is possible for p in the region $[80, 1300]$. The maximum performance which can be achieved from (22) and the approximation for $\log_2 p$ results in

$$l_{\max} = l^{\text{peak}} y \frac{r(n, p_{\max})}{1 + \frac{\alpha r(n, p_{\max})}{8 \ln 2}} \sim \frac{l^{\text{peak}} y}{3\alpha} \frac{9}{10}. \tag{33}$$

It follows $l_{\max} = (n/128) \cdot 8 \cdot (9/10) = 560$ GFlop/sec in the case $n = 10.000$. But we must strictly remark that this value is outside the available number of cores of bwGRiD. We measured 490 GFlop/sec for $p = 1024$ cores and so we did not reach the saturation point $p_{\max}$.

A detailed analysis shows a slowly decreasing behavior of the speed-up with $p > p_{\max}$, which is in agreement with Amdahl's law.

6.4 Application to TOP500

Analyzing systems of the TOP500 [20] we have the difficulty that we do not know the internal bandwidth, because this value is not presented in the list. Instead we calculate an effective bandwidth from the given values. For Linpack

$$y = \hat{x} \cdot z = a \cdot \frac{b_c}{l^{\text{peak}}} = \frac{\#op}{\#b} \cdot \frac{b_c}{l^{\text{peak}}} = \frac{n}{3w} \cdot \frac{b_c}{l^{\text{peak}}} \tag{34}$$

and from (33) follows with $\alpha = 1/3$

$$l_{\max} = \frac{n \cdot b_c}{3w} \frac{9}{10}. \tag{35}$$

Note that $l_{\max}$ is independent from l^{peak} and the number of cores p. For the TOP500 we know the matrix size n ($= N_{\max}$ in the list) for the measured maximum performance $l_{\max}$ ($= R_{\max}$ in the list). From this we calculate an effective bandwidth (with $w = 8$ Byte):

$$b_c^{\text{eff}} = \frac{R_{\max}}{N_{\max}} \cdot 3w \cdot \frac{10}{9}. \tag{36}$$

To verify this analysis we take the first 100 systems of the November 2011 and 2012 list and neglect all systems employing accelerator techniques and systems where $N_{\max}$ is missing. The results for b_c^{eff} are shown in Figs. 4 and 5 in red diamonds. The effective bandwidth b_c^{eff} correlates well with list rank. The values for b_c^{eff} are in a bandwidth range attainable with available interconnect technology.

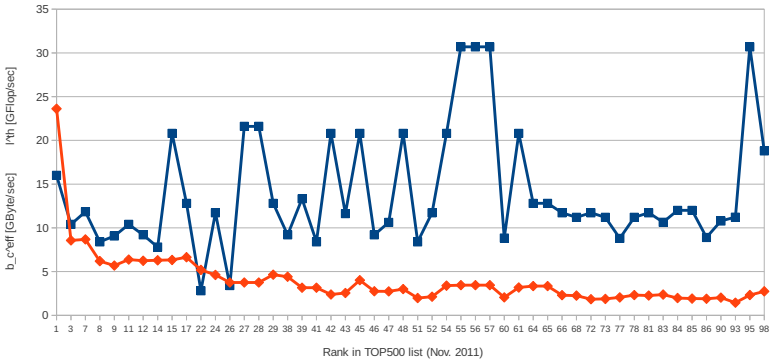


Fig. 4. Effective bandwidth b_c^{eff} (diamonds, red line) and theoretical peak performance l^{peak} (squares, blue line) against rank for selected TOP500 sites (November 2011)

To further check the accuracy of (35), we consider two examples of the November 2012 list. System number 19 is Hopper, NERSC, United States. Its Gemini network allows a maximum bandwidth of 9.375 GByte/sec, but only 5.8 GByte/sec for MPI applications [21]. Our value of 6.1 GByte/sec for b_c^{eff} is very close to the real value. The **K** computer, RIKEN, Japan, is the number 3 system. The Tofu network provides a bandwidth of 50 GByte/sec [22]. Assuming a reduced bandwidth for MPI applications, our calculated b_c^{eff} of 23.61 GByte/sec is comparable. This indicates that (35) is a good thumb rule to estimate the Linpack performance.

However, in general a direct comparison of b_c^{eff} with the actual bandwidth of a specific system is difficult. In our model we assume that the PUs are processor cores and that the bandwidth between each pair of PUs is b_c. But TOP500 systems are predominantly multi-core systems where several cores share the interconnect between the nodes. It depends on the application and the MPI library,

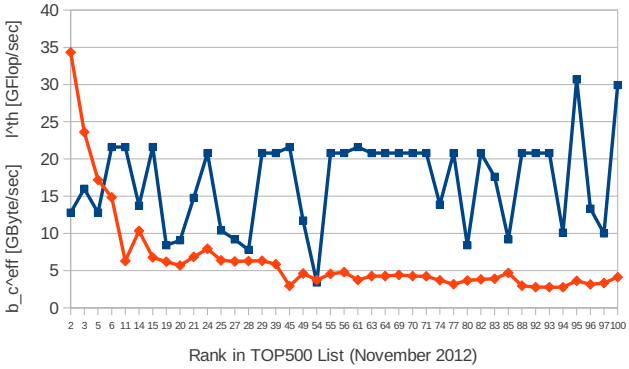


Fig. 5. Effective bandwidth b_c^{eff} (diamonds, red line) and theoretical peak performance l^{peak} (squares, blue line) against rank for selected TOP500 sites (November 2012)

whether cores communicate simultaneously between nodes and how cores communicate with cores on the same node. For the bwGRiD cluster, which is also a multi-core system, we found that 1/3 is a good value for the free parameter α, but for other node architectures and interconnect topologies a different α could be more suitable. It would be interesting to see how this observation compares to the situation within a node between the cores with a given internal communication bandwidth.

Figs. 4 and 5 also shows the theoretical performance of a single core calculated from $l^{\text{peak}} = R_{\text{peek}}/p_{\max}$ (blue squares), where $p_{\max}$ (= Cores in the list) is the number of available cores in a system. The lack of a correlation of l^{peak} with position in the list confirms that for Linpack type applications the actual value for l^{peak} is unimportant for the maximum performance of a system. This means that hardware and networking design, sizing, as well as the ability to adapt the Linpack algorithms and the MPI implementations to the new hardware are more important for a good Linpack performance than the advances in single core performance.

7 Conclusions

The analysis of homogeneous cluster systems without any stochastic tools via a variant of the roofline model has shown some simple and insightful results. We have described performance and speed-up by a few dimensionless scaling variables, which summarize all important hardware and software characteristics like the number of operations, amount of processed data, and amount of communicated data. Preliminary stages of this approach are discussed in [10, 11].

In this paper we applied our model to important classes of applications with different communication to computation ratios. Our speed-up formula (14) explicitly shows the dependence on the hardware (peak-performance of an PU

l^{peak}, the interconnect bandwidth b_c, the number of PUs p) and the application (problem size n, operational intensity $a(n)$ and the scaling behavior $r(n,p)$). Using this model we estimated the optimal number of PUs for a given hardware and application in the variable $p_{\max}$.

A detailed analysis, which focuses on the original roofline model for (inhomogeneous) multi-core nodes, for inhomogeneous clusters and with asymmetric load distribution will be considered in future investigations. In our approach this amounts to a substitution $l^{\text{peak}} \longrightarrow l^{\text{peak}} s(x_m)$, where $s(x_m)$ describes the characteristics of a multi-core CPU as a function of the dimensionless parameter x_m (similar to our parameter x above). In particular we will look at applications like sparse matrix-vector operations and Fast-Fourier-Transform (FFT).

In summary we have the following new contributions:

1. Integration of hardware and software characteristics in 1-3 dimensionless parameters.
2. Estimated predictions of the optimal size of a cluster for a given class of applications.
3. Estimated predictions of the maximal performance for a given application and system parameters.
4. The key performance indicator for a compute-cluster suitable to run TOP500 style applications has been identified to be the effective bandwidth.

Clearly it is not difficult to create more complex models. But do they offer more and rapid insights? Our answer is: no.

Acknowledgments. The authors thank J. Dongarra, Univ. of Tennessee for some valuable information on Linpack details. Thanks also to the anonymous referees for the helpful suggestions to improve the paper.

bwGRiD is part of the German D-Grid initiative and is funded by the Ministry for Education and Research (Bundesministerium für Bildung und Forschung), the Ministry for Science, Research and Arts, Baden-Württemberg (Ministerium für Wissenschaft, Forschung und Kunst Baden-Württemberg), and the universities of Baden-Württemberg.

References

[1] Kruse, H.G.: Leistungsbewertung bei Computer-Systemen. Springer (2009)
[2] Kredel, H., Kruse, H.-G., Richling, S.: Zur Leistung von verteilten, homogenen Clustern. PIK (2), 166–171 (2010); English summary, see section V in [3]
[3] Richling, S., Hau, S., Kredel, H., Kruse, H.-G.: Operating Two InfiniBand Grid Clusters over 28 km Distance. In: Proc. 3PGCIC 2010. IEEE (2010)
[4] Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multi-core architectures. Commun. ACM 52(4), 65–76 (2009)
[5] Amdahl, G.: Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In: AFIPS Conference Proceedings, vol. 30, pp. 483–485 (1967)
[6] Gustafson, J.: Reevaluating Amdahl's law. Commun. ACM 31(5), 532–533 (1988)

[7] Hockney, R.W.: Parametrization of computer performance. Parallel Computing 5(1-2), 97–103 (1987)
[8] Hockney, R.W., Jesshope, C.R.: Parallel Computers 2: architecture, programming and algorithms. Adam Hilger, Bristol (1988)
[9] Hockney, R.W.: Computational similarity. Concurrency – Practice and Experience 7(2), 147–166 (1995)
[10] Kredel, H., Kruse, H.-G., Richling, S.: Einige Überlegungen zur Leistung von Cluster-Computern. PIK (3), 207–211 (2012); For a partial English summary and extensions see section 3 in [11]
[11] Kredel, H., Kruse, H.G., Richling, S., Strohmaier, E.: Performance Analysis and Prediction for distributed homogenous clusters. In: Computer Science – Research and Development, Special Issue ISC 2012, Hamburg (May 2012)
[12] LinPack and HPL, Linear Algebra Package and High Performance LinPack, http://www.netlib.org/benchmark/hpl/ (accessed January 2012)
[13] Dongarra, J., et al.: ScaLAPack documentation, http://www.netlib.org/scalapack/slug/node112.html (accessed January 2012)
[14] Luszczek, P., Dongarra, J.: Reducing the Time to Tune Parallel Dense Linear Algebra Routines with Partial Execution and Performance Modeling. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2011, Part I. LNCS, vol. 7203, pp. 730–739. Springer, Heidelberg (2012)
[15] Richling, S., Hau, S., Kredel, H., Kruse, H.-G.: A Long-distance InfiniBand Interconnection between two Clusters in Production use. In: Proc. Supercomputing, November 12-18. IEEE (2011)
[16] Numrich, R.W.: Computational Force: A Unifying Concept for Scalability Analysis. In: Proc PARCO, pp. 107–112 (2007)
[17] Numrich, R.W.: A metric space for computer programs and the principle of computational least action. Journal of Supercomputing 43(3), 281–298 (2008)
[18] Numrich, R.W.: Computer performance analysis and the Pi Theorem. Comput. Sci. Res. Dev. (2010)
[19] bwGRiD, Member of the German D-Grid initiative, funded by the Ministry of Education and Research and the Ministry for Science, Research and Arts Baden-Württemberg, Universities of Baden-Württemberg, 2007-2010, 2007-2012, http://www.bw-grid.de/ (accessed December 2012)
[20] Meuer, H., Strohmaier, E., Dongarra, J., Simon, H.: Top 500 Supercomputer Sites, http://top500.org/ (accessed November 2012)
[21] Hopper Interconnect, http://www.nersc.gov/users/computational-systems/hopper/configuration/interconnect/ (accessed May 2012)
[22] Uno, A.: K computer system overview, http://www.fujitsu.com/downloads/TC/sc11/k-computer-system-overview-sc11.pdf (accessed May 2012)
[23] Graph 500 Steering Committee, Benchmarks for data intensive supercomputer applications, http://www.graph500.org/
[24] Kredel, H., Kruse, H.-G., Ott, I.: Performance analysis and performance modeling of Web-applications. In: Proc. 3PGCIC 2011, pp. 115–122. IEEE (2011)

Using Simulation to Validate Performance of MPI(-IO) Implementations

Julian Martin Kunkel

University of Hamburg
Bundesstraße 45a
20146 Hamburg
julian.martin.kunkel@informatik.uni-hamburg.de

Abstract. Parallel file systems and MPI implementations aim to exploit available hardware resources in order to achieve optimal performance. Since performance is influenced by many hardware and software factors, achieving optimal performance is a daunting task. For these reasons, optimized communication and I/O algorithms are still subject to research. While complexity of collective MPI operations is discussed in literature sometimes, theoretic assessment of the measurements is de facto non-existent. Instead, conducted analysis is typically limited to performance comparisons to previous algorithms.

However, observable performance is not only determined by the quality of an algorithm. At run-time performance could be degraded due to unexpected implementation issues and triggered hardware and software exceptions. By applying a model that resembles the system, simulation allows us to estimate the performance. With this approach, the non-function requirement for performance of an implementation can be validated and run-time inefficiencies can be localized.

In this paper we demonstrate how simulation can be applied to assess observed performance of collective MPI calls and parallel IO. PIOsimHD, an event-driven simulator, is applied to validate observed performance on our 10 node cluster. The simulator replays recorded application activity and point-to-point operations of collective operations. It also offers the option to record trace files for visual comparison to recorded behavior. With the innovative introspection into behavior, several bottlenecks in system and implementation are localized.

Keywords: Simulation, MPI-IO, Performance evaluation.

1 Introduction

Parallel file systems and MPI implementations aim to achieve optimal performance on all systems. The performance of communication and IO certainly depends on the hardware characteristics – the specific hardware configuration limits potential network throughput, computation power and available memory bandwidth. The selection of the optimal algorithm depends on the hardware characteristics, the network topology and application behavior. From a library's

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 181–195, 2013.

point of view, optimization can be done based on the parameters provided by the programmer. Typically, this includes the *memory datatype*, the *communicator*, *target/source rank* (for all-to-one or one-to-all operations), and the actual *amount of data* shipped with the call. Additionally, the process placement across the hardware resources is important. Therefore, MPI implementations offer several algorithms and they realize a rich variety of optimization strategies to gear algorithms towards the given system.

This adaption leads to better exploitation of available hardware resources and, ultimately, to better performance and thus application runtime. However, the interplay of hardware optimizations such as caches, the software optimizations offered by operating system and intermediate libraries result in complex behavior which make the selection of an optimal algorithm hard. With Open MPI, MPICH2, MVAPICH2, this complexity also leads to a diverse landscape of open source MPI implementations. Also, vendors and integrators offer their own proprietary solution.

Up to now, effectiveness of alternative algorithms is mainly demonstrated by comparing measured performance with performance of existing algorithms. However, observable performance is not only determined by the quality of an algorithm. At run-time performance could be degraded due to unexpected implementation issues and triggered hardware and software exceptions. Visualizing the real system activity helps analyzing the behavior and localizing regions that require most of the execution time. However, determining whether recorded activity is conducted optimally is not possible because it depends on platform and optimizations.

In this paper we propose a simulation driven systematical validation of MPI-IO performance. By applying a model that resembles the system, simulation approximates performance and, thus estimates performance of algorithms. The main contributions of the paper are 1) a performance study motivating integrated performance testing of MPI and 2) a discussion of a feasible implementation of such an approach. Without the power of simulation, many performance bottlenecks could not be found in our cluster.

This paper is organized as follows: In Section 2 an overview of the state-of-the-art is presented. In Section 3 the benefits of simulation to evaluate performance of MPI-IO implementations are described. A brief introduction to the simulator and the underlying hardware and software models is given in Section 4. Several experiments in Section 5 illustrate how theory aids to localize bottlenecks and to check for correctness. While the model is developed manually, this process could be automated to perform these steps automatically. Section 6 concludes the paper.

2 State of the Art

Many algorithms were proposed to optimize collective communication. They are either directly implemented in one of the MPI implementations, e.g. [1], or provided as an external library such as STAR-MPI [2] or Magpie [3].

To our knowledge, MPI libraries lack self-awareness. There is no implementation which takes the hardware characteristics into account while determining an algorithm for collective communication or I/O. While middleware implementations ship with tests for functionality they do not automatically detect hardware characteristics. Instead, a library is shipped with empirically chosen defaults, which might be determined for a completely different system than the system the library is deployed on. Tuning of these parameters is time consuming, thus, the defaults might achieve only a fraction of theoretical performance. Many of these parameters exist, for example, in Open MPI, the *Modular Component Architecture (MCA)* lists more than 250 parameters on a COST Beowulf cluster.

Although MPI implementations are not considering hardware characteristics, they have become increasingly aware of the communication topology and try to utilize shared-memory communication if possible which leads to SMP-aware collective algorithms. For example, the CARTO framework of Open MPI provides topological information.

As algorithms must be handcrafted towards the system – for instance for a BlueGene [4] – one major problem is to pick the best algorithm for a system. Several approaches have been developed that assist in determining the best algorithm and MPI configuration. The *Abstract Data and Communication Library (ADCL)* [5] uses historic knowledge during the application run. ADCL assumes a program performs operations iteratively – in the first few iterations ADCL evaluates a set of MPI functions to determine which one is best suited for the given problem, then this function is applied to subsequent invocations. Compared to ADCL, the *Self-Tuned Adaptive Routines for MPI Collective Operations (STAR-MPI)* provides a rich set of MPI implementations for collective operations by itself [2], for instance a set of 13 algorithms is supplied for `MPI_Alltoall()`.

For parallel I/O, the problem becomes even more complex since it depends on communication. For example, non-contiguous operations and collective calls have been defined in MPI-IO which lead to a classification of data access into four levels [6]. These levels are characterized by two orthogonal aspects: contiguous vs. non-contiguous data access, and independent vs. collective calls. Depending on the level, a different set of optimizations can be thought of, for example, two-phase I/O and multiphase-collective I/O [7] aim to improve collective non-contiguous access. An adaptive approach is introduced in [8], which automatically sets hints for collective I/O based on the access pattern, topology and the characteristics of the underlying file system.

Typically, evaluation of improved algorithms is conducted by comparing performance of existing algorithms with the new algorithm. This includes improvements in the communication submodules of MPI, e.g., in Nemesis [9], or completely new MPI implementations such as Open MPI[10]. Similarly, parallel I/O research demonstrates improvements by comparing observed performance. In most cases, a baseline of expected performance is not provided. This is mainly due to the complexity of determining these baselines. There are a few exceptions to this general observation, but theoretic considerations are restricted to simple cases. For example, in [11] upper bounds for performance are provided

based on the component throughput and latency. In many cases, very coarse estimates could be computed even for complex behavior, but these are not very tight. Development of an adequate mathematical equation for complex behavior is nontrivial. Simulation of the behavior is much easier.

There are many simulators for distributed systems, most focus on communication routines, for example, the Structural Simulation Toolkit (SST) [12], LogGOPSim [13] and Dimemas [14].

Some simulators can replay previously recorded MPI(-IO) activity inside the virtual environment. For example, trace information is altered in [15], then an MPI program replays the modified trace on the original machine, which automatically enforces causality between dependencies among processors. While this approach scales well, it is not possible to simulate other hardware configurations or to gain insights into MPI. LogGOPSim is a simulator for a class of analytical models of the $\log_x$P family [16]. It supports a simple network collision model. Dimemas reads trace files and applies an analytical model to individual and collective communication. Network collisions are modeled in an abstract way by limiting the maximum throughput which can occur at a given time over a central network infrastructure.

CODES [17] and PIOsimHD [18] target parallel I/O. Built on top of the Rensselaer Optimistic Simulation System (ROSS), CODES supports parallel discrete-event simulation of queuing models. It has been successfully applied to study the role of burst buffers in systems with 100k application processes and 120 PVFS file servers. In contrast to the introduced systems, PIOsimHD covers parallel I/O and allows replaying of recorded MPI traces on a high level of abstraction – commands are implemented in the simulator to react on system conditions. The event-driven nature of PIOsimHD allows localizing of network congestion and to evaluate I/O optimization on client, server or disk side. For example, an analysis of several I/O schedulers and collective I/O variants has been performed using PIOsimHD in [19]. While simulation has been used to evaluate what-if scenarios, to our knowledge it has not been used to systematically validate measured MPI-IO performance in order to identify hidden bottlenecks. Instead, complex simulation parameters are introduced and fitted to meet observations.

3 Using Simulation to Validate MPI-IO Performance

Simulation aids in validating MPI-IO performance in two ways: First, by comparing observed run-time and theoretical run-time estimate quantitatively, implementation issues and unexpected bottlenecks can be identified. This is especially useful for validation of complex operations such as collective operations. Second, a complex sequence of operations, such as the behavior of real applications, can be inspected visually and qualitatively compared to a simulated run of the application. Therewith, unexpected behavior of individual operations can be identified and assessed. For both scenarios, simulation parameters can be varied to study the impact of certain hardware characteristics, for example by turning off computation.

We propose systematic validation of performance achieved with MPI-IO functions using simulation. Imagine an MPI-IO implementation which does not only run functional tests after installation but also performance benchmarks and assesses the results. First, it could run simple point-to-point benchmarks and create a system model. Then complex benchmarks could be run and their performance could be assessed automatically for soundness. For example, a bi-directional message transfer of a large message with `MPI_Sendrecv()` should require the same time as a unidirectional communication. The basic system model is of interest for performance optimization by itself, as it illustrates expected communication overhead. An administrator could compare these performance characteristics with micro-benchmarks to ensure that the basic communication routines extract the performance as anticipated.

This becomes more interesting for complex operations, as their performance cannot be understood easily. If expected and observed performance diverge too much, the system should raise a warning. Then the administrator has a starting point for investigating performance degradation which could be due to MPI-internal overhead, kernel, or external libraries. A result telling the administrator MPI performance behaves as anticipated is valuable too, as it reduces the chance to experience unexpected performance loss in production. Finally, by determining hardware characteristics at installation time, these values could be used at run-time to determine well-suited collective algorithms without manual intervention and ultimately allow a self-aware MPI implementation.

To conduct such a validation, it is mandatory for the simulation to mimic the expected behavior of the experimental system. Thus, basic model parameters should have similar characteristics as the real system. Since simulation should help identifying inefficiencies, it is not constructive to mimic the real system perfectly as we could not spot differences and thus unexpected behavior. In both cases, it helps if activities of an application can be recorded and replayed by the simulator because this reduces the effort to validate the execution. By this means theoretically any MPI benchmark can be run and its results can be easily compared to our expectations. For later analysis, it is also useful if the simulator can create trace files which can be compared to real traces.

4 PIOsimHD

The goal of the sequential discrete event simulator PIOsimHD is to assist MPI-IO research and to foster understanding of performance factors in clusters. PIOsimHD performs a discrete event simulation and, if requested, stores the processing as trace files. It can also read activity from recorded trace files. HDTrace is an experimental tracing environment which also provides tools to instrument existing applications and to record activity of PVFS and MPI internal communication. Simulation results can then be visualized by Sunshot, which enables a comparison of the recorded process and file system activities and simulation results.

PIOsimHD offers a hardware model which reflects the common sense of a cluster computer: Several compute resources (CPUs) are hosted on a node which is connected to one or several networks via a network interface (NI). Arbitrary network topologies can be created. On each node one I/O server can be placed, each holding a cache layer which schedules operations, and an I/O-subsystem.

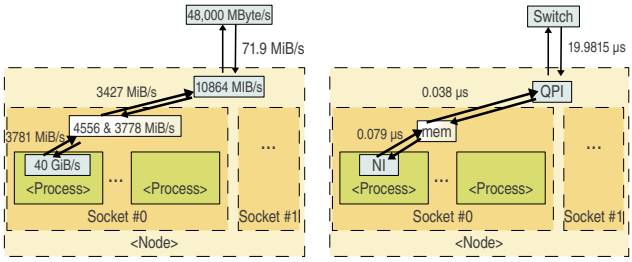
To cope with several levels of abstraction, a component can have several implementations. Component implementations are parameterized with certain characteristics. Usually, characteristics are provided in vendor specifications or obtained by benchmarking the existing system. The level of detail of the cluster hardware covers basic information describing an Ethernet based cluster. With the amount of memory and number of CPUs, a node offers shared resources for hosted processes. Each CPU processes a fixed number of instructions per second. The memory is used for caching I/O on the server side.

The simulator permits the user to create arbitrary network graphs representing store-and-forward systems. Network edges have a latency and a transfer rate. Network nodes have a maximum bandwidth to relay data. With the help of network components, memory access of communication can be simulated which permits modeling of local communication. A special node adds the local throughput as an additional parameter, which is used when two direct neighboring components of this network node exchange data. An example model of a dual-socket node is given in Figure 1. In this figure, throughput and latency of all network components are given as observed on our Intel Westmere cluster consisting of 10 nodes.

To utilize the network well, a network flow model was designed in which messages are fragmented into packets of a maximum size, which flow from source to target in a stream. When data is transferred from one component to another, the transmission of incoming data flows is continued. The maximum number of packets in flight for every stream is limited by the bandwidth-delay-product of the given link. While many concepts can be found in real systems, the data flow differs because this concept achieves the highest utilization of all network components for all streams, and it does not throw packets away.

A hard-disk as an *I/O-subsystem* is modeled by a sequential transfer rate, an average access time, track-to-track-seek time and RPM. Depending on the distance to the last byte accessed within a file, a disk will either perform no seek, will seek to the neighboring track or will apply the average access time. Access to other files always enforces an average seek.

An abstract parallel file system defines the interaction between client and server. The abstract model describes many parallel file systems because they work similarly. Clients and servers interact in a similar fashion to the PVFS model, but the concept is universal to most parallel file systems: File data is partitioned among all servers as defined by a selectable distribution function. To write data, a client requests a write operation from the server and then starts to transfer all data. File sizes are updated once a write operation finishes. Metadata operations are currently not considered since these depend on the specific file system. More details can be found in [18].



(a) Throughput. (b) Latency of edges.

Fig. 1. Network topology model for the working groups cluster. Throughput of intra-socket communication is slightly higher

4.1 Experiments

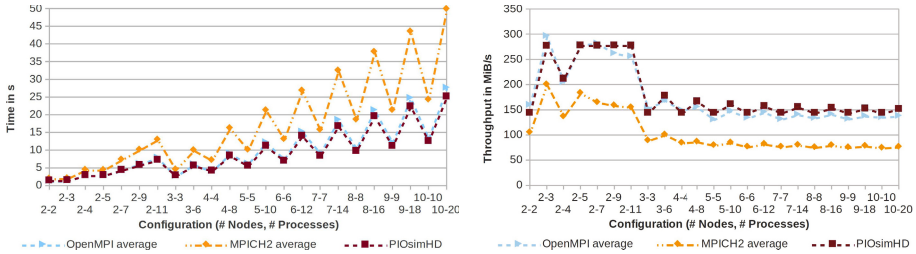
During the validation of the simulator, several unexpected bottlenecks in hardware and software could be identified. An excerpt of interesting results demonstrating the benefit of validating the soundness of observed MPI-IO performance is given in the following. Measurements are executed on our 10 node Ubuntu cluster; interconnected via. Gigabit-Ethernet, each node is equipped with two *Intel Xeon 5650* processors providing 12 cores for the experiments and 12 GByte of memory. Used software versions are: *Open MPI* 1.5.3, *MPICH2* 1.3.1, and Orangefs-2.8.3. The conducted validation is described in detail in [18].

To conduct complex validation runs, recorded activities are replayed in the simulator. Thus the same sequence of compute, network and I/O activity is executed. While a compute job takes the exact time as recorded in a validation run, execution time of parallel I/O and communication is computed by the simulator using the virtual file system and network models.

Parameterization. To parameterize the simulator for a validation run, the hardware characteristics must be determined. Throughput and latencies for the network links have been measured using MPI point-to-point operations (values are annotated in Fig. 1[1]). An HDD is characterized by a track-to-track seek time of 1.1 ms, an average seek time of 9 ms, a sequential transfer rate of 96 MiB/s and 7200 RPM. The hardware model uses the fast seek time for accesses to the same file to an offset which is within a window of 1 MiB to the last access.

Communication. Before analysis of collective results is conducted, a simple example of a suboptimal point-to-point communication pattern is given. In this experiment, each process exchanges a 100 MiB message with Rank 0 by calling `MPI_Sendrecv()` – the whole experiment is repeated 9 times. The average measured time is plotted for a variable number of nodes and processes in Figure 2a

[1] These values have been validated with network benchmarks such as Iperf. The issues with the network are discussed in Section 4.1.



(a) Time (b) Aggregated throughput

Fig. 2. Sequential data exchange of Rank 0 and all other processes using `MPI_Sendrecv()` for several configurations and 100 MiB messages

and the achieved throughput is illustrated in Figure 2b. It can be observed that PIOsimHD approximates Open MPI very well, but MPICH2 needs more time than anticipated by simulation. Without a simulation tool, the performance could be approximated manually, for example, using the following few considerations: Since our network allows bi-directional communication, about 140 MiB/s can be achieved over the single node hosting Rank 0, this performance can be seen for many configurations. However, already this simple pattern shows that computation is not this simple. Due to shared memory intra-node communication, processes hosted by the same node achieve much higher throughput. This can be observed in Figure 2b for Open MPI and PIOsimHD. Thus, while a manual computation of the expected throughput is possible, it is non-trivial. By comparing simulation results with the results of MPICH2, the unexpected slow-down become visible and could be subject for further investigation[2].

Examples for collective communication are given in Figure 3. Experiments with 10 KiB message transfers are repeated 10,000 times and for larger messages at least 9 times. Figures show the quartiles for the small messages to account for deviation, and minimum and average values for larger messages (typically, the slowest time is much higher than the average). In Figure 3a and Figure 3b it can be seen that `MPI_Allgather()` is well approximated by PIOsimHD, and thus observed performance is consistent with our theoretic expectations. In comparison, the intra-node algorithm of Open MPI achieves a better performance for configurations with 2 nodes. For small messages, Figure 3a shows much better times for configurations 4-8 and 8-16 than for other configurations. Without the simulation result, one question might be whether this behavior is due to the system's characteristics. Since the simulator executes the exact same communication pattern and results in similar performance, we can conclude the communication algorithm changes and leads to this behavior[3].

[2] Actually, our version of MPICH2 extracted the same performance numbers as for uni-directional communication.

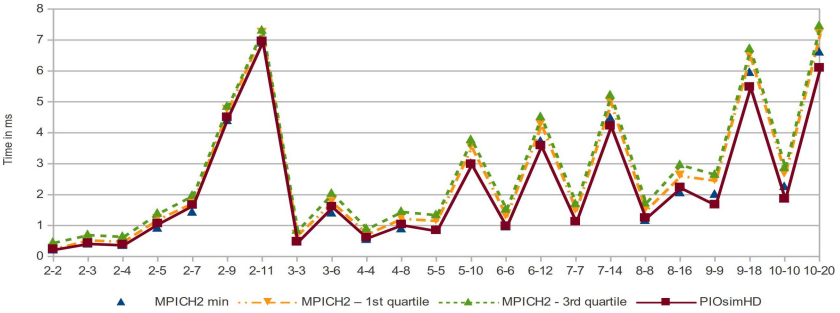
[3] Actually, the trace files can be inspected demonstrating correctness of this theory.

Similarly, an analysis of `MPI_Allreduce()` shows interesting behavior (see Figure 3c). While the measurements of a single configuration fluctuate much more, the simulation still recreates the overall pattern. The complexity of the analysis can be observed for larger messages in Figure 3d. While Open MPI shows a completely different behavior than MPICH2, none of the algorithms is optimal in all cases. In this example, PIOsimHD estimates better times than MPICH2, which reasons should be analyzed further. Thanks to simulation, the impact of certain factors can be studied. For example, the impact of computation has been investigated – as it turns out, the required time only improves slightly when recorded computation is not simulated.

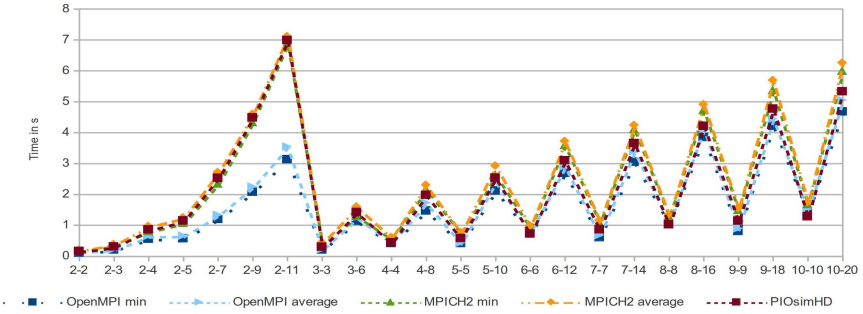
Visual Inspection of Application Behavior. Finally, a run of our Jacobi PDE solver is evaluated with the simulation. While the simulation recreates the overall pattern quite well (refer to [18]), communication in the final phase takes longer than anticipated. Timelines of the cleanup phase are given in Figure 4. Rank 0 receives selected lines of the PDE from all processes (users might inspect them to validate the run). Several data transfers need 0.2 ms although the sender and receiver is ready. Since only 400 KiB of data is transferred, a performance of only 2 MiB/s is achieved. This problem has been found by first comparing trace profiles, then a visual inspection of the individual communications has been performed. During the iterations, message exchange behaves as anticipated by the simulation. Without theoretic considerations, an assessment of the performance in terms of overall achieved time and the individual operations would be difficult. However, estimating run-time for an complex application is cumbersome.

Parallel I/O. With Parabench [20] the four levels of access have been investigated for several setups. Results for independent contiguous reads are given for a variable number of clients and servers in Figure 5. In these experiments, each client reads 100 KiB (and 100 MiB) records – a total of 1 GiB of data is accessed per client. Client records are distributed in round-robin over the logical file, i.e. the first record of the file is accessed by Rank 0, then by Rank 1 and so forth. Data is stored on tmpfs, thus there is no slow I/O device involved and performance is expected to be limited by the network. The simulator approximates performance for 100 MiB records well as shown in Figure 5a. However, it overestimates performance of 100 KiB records significantly as illustrated in Figure 5b. Since the model uses measured latency and bandwidth as characteristics, these hardware factors cannot be the reason. A detailed analysis revealed timing effects in the real system leading to congestion of individual servers while most servers are idle. Once requests of multiple clients are pending on a single server, all clients must wait for data stored on this server but since the server multiplexes the NIC, data transfers of all responses take longer. With a slight variation in the simulation characteristics, these effects can also be reproduced in-silicon.

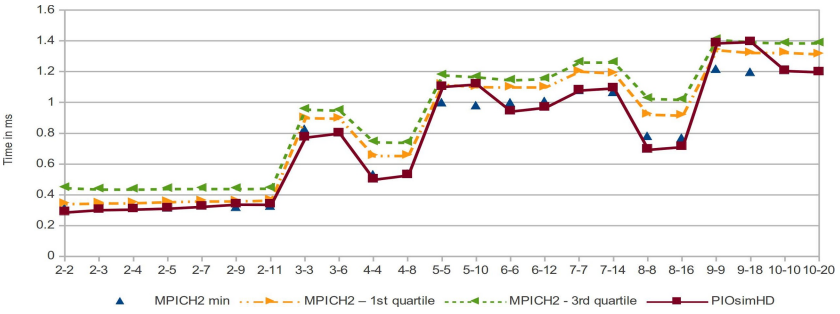
One experiment was conducted that changed the packet size of the store-and-forward network. The simulated network relays packets of 100 KiB size; a lower packet size of 10 KiB can be chosen, which improves concurrency of the components and the theoretic performance.



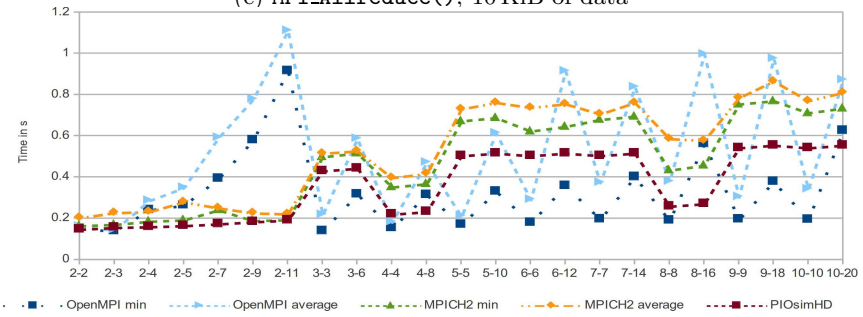
(a) `MPI_Allgather()`, 10 KiB of data



(b) `MPI_Allgather()`, 10 MiB of data

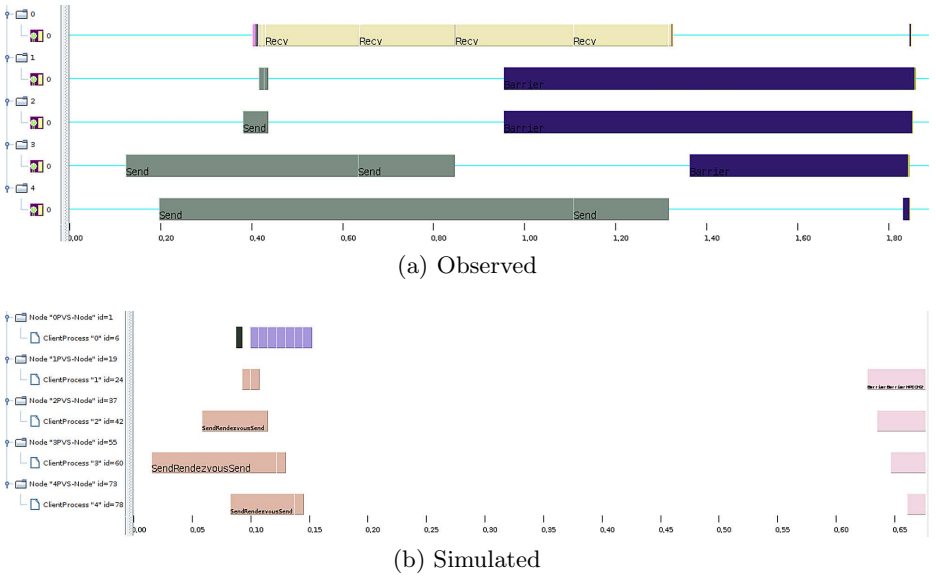


(c) `MPI_Allreduce()`, 10 KiB of data



(d) `MPI_Allreduce()`, 10 MiB of data

Fig. 3. Simulation of inter-node collective communication for a variety of configurations (# of nodes, # of processes)



(a) Observed

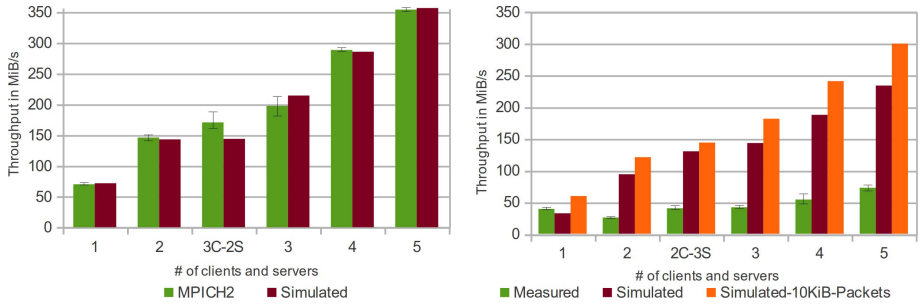
(b) Simulated

Fig. 4. Final phase of a Jacobi PDE solver – traces of observation and simulation

During the validation of the Jacobi PDE, an inefficiency in the PVFS module of MPI-IO could be identified and resolved. The PDE outputs the matrix diagonal for later inspection as a sequential 64 KiB data block. Internally, a memory datatype is used to address the matrix diagonal in one write call. However, the write takes about 70 ms while the simulator estimates 2 ms[4]. Using HDTrace, a detailed analysis of client and server activities has been made revealing that PVFS split the 64 KiB block into 512 bytes requests. The reason is the handling of non-contiguous datatypes by ROMIO. Since ROMIO does not use an additional buffer to store data, every non-contiguous region in memory is normally accessed with an individual operation. With ListIO, PVFS supports encapsulating to 64 non-contiguous operations in one request. Since the matrix diagonals are 8 byte, 512 byte requests are created. For the application, the problem could be fixed by setting the undocumented hint *romio_pvfs2_listio_write* which enables handling of memory and file datatypes in ROMIO. By setting the hint, the average time for a single write is reduced to 3.4 ms which is close to the estimation.

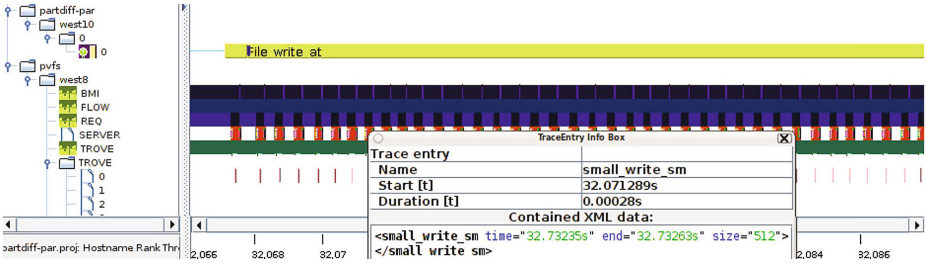
A screenshot of the obtained traces for one iteration of a write phase are shown in Figure 6. In the default operation, server and trove timeline show many small operations. With the applied hint, one large write operation can be seen in the timeline (the additional small write operation left updates the header of the file in both cases).

[4] The actual time depends on the current activity on the accessed servers.

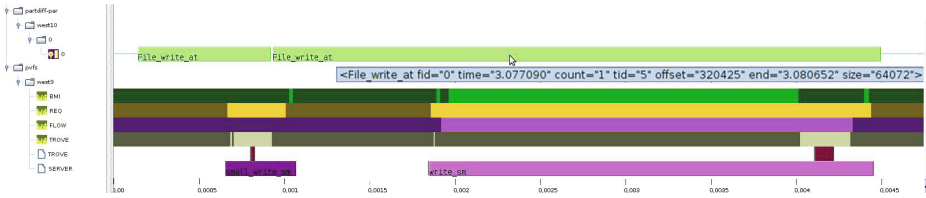


(a) 100 MiB records (b) 100 KiB records

Fig. 5. Performance of independent contiguous I/O with a variable number of clients and servers Data is stored on tmpfs



(a) Default operation; details of one request are shown



(b) With supplied hint

Fig. 6. Screenshot of the PDE's data exchange for one client and one server

Difficulties to Identify Causes of Performance Degradation. The intention of performance analysis is not only to localize but to resolve the causes of potential performance degradation. However, as it turned out the identification of triggered issues is non-trivial. We invested several month trying to localize the general issues in our network stack and to identify and fix the performance issue with PVFS. The latter issue could only be analyzed in detail thanks to the detailed tracing mechanisms of HDTrace. The fix involved communication with the developers, but also detailed code inspection of PVFS and MPI-IO.

Unfortunately, debugging of the network issue showed little success. As it turned out, operations sometimes take much longer than expected (10 times the average time, an overhead of about 0.2 s), and there was the network limitation

of 67 MiB/s. To identify the reason for the performance degradation, several regulating knobs have been evaluated on our production system: TCP-tuning, alternative MPI implementations (MPICH2 and Open MPI), different Linux kernel and also CentOS as an alternative distribution. Also, existing network tools and benchmarks have been used to validate the observed performance.

The insight of all this effort is: Performance could only be improved a little by testing many alternative sets of TCP-options. With newer kernel versions, the throughput improved to 71 MiB/s and finally with kernel 3.5 to 117 MiB/s. Interestingly, by using CentOS, the variance of network packets stabilizes and throughput is good. As these issues disappeared by using newer kernels and another distribution, a detailed analysis of involved libraries and kernel is required.

5 Summary and Conclusions

In this paper, we describe the benefit of using simulation for validating performance of MPI-IO implementations. To estimate performance, the simulator executes the activity of a parallel program on a virtual cluster with similar characteristics as the experimental system. In many cases, a very good match to observations is achieved, which validate that system and implementation behavior is consistent. However, an excerpt of experiments is given in which performance gaps become visible. For example, observed performance of `MPI_Sendrecv()` and `MPI_Allreduce()` fall behind the expectations which indicate a demand for further investigation. Automatic performance assessment could be an integral part in a test-suite – a simulator complements existing benchmarks by creating run-time estimates. Shipped with MPI implementations, these tests would spot unexpected performance inefficiencies directly. While we tried to identify the causes of the network performance degradation in kernel, libraries and system hardware, we did not succeed so far. Nevertheless, without systematic testing we would be unaware of these inefficiencies, showing the necessity of automatic tools and the involvement of developers.

Suboptimal behavior during parallel I/O has been investigated but it is much more complex than collective I/O. For example, timing effects may overload individual servers. Finally, a performance problem in an application could be identified and with the help of advanced tracing of client and server behavior, the reasons could be identified and fixed by applying an MPI hint. Overall, the virtual laboratory of HDTrace allows us to identify inefficiencies and to study behavior of communication and file system, to conduct research on new algorithms, and to evaluate future systems. In the future, we will try to build the envisioned system for automatic validation of MPI-IO behavior.

Acknowledgements. I want to thank the PVFS development team to help resolving the performance degradation of writing the non-contiguous matrix diagonal.

References

1. Thakur, R., Rabenseifner, R., Gropp, W.: Optimization of Collective Communication Operations in MPICH. International Journal of High Performance Computing Applications 19(1), 49–66 (2005)
2. Faraj, A., Yuan, X., Lowenthal, D.: STAR-MPI: Self Tuned Adaptive Routines for MPI Collective Operations. In: Proceedings of the 20th Annual International Conference on Supercomputing, ICS, pp. 199–208. ACM, New York (2006)
3. Kielmann, T., Hofman, R.F.H., Bal, H.E., Plaat, A., Bhoedjang, R.A.F.: MagPIe: MPI's Collective Communication Operations for Clustered Wide Area Systems. In: Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP, pp. 131–140. ACM, New York (1999)
4. Miller, S., Kendall, R.: Implementing Optimized MPI Collective Communication Routines on the IBM BlueGene/L Supercomputer. Technical report, Iowa State University (2005)
5. Gabriel, E., Huang, S.: Runtime Optimization of Application Level Communication Patterns. In: International Parallel & Distributed Processing Symposium, IPDPS, pp. 1–8. IEEE (2007)
6. Thakur, R., Gropp, W., Lusk, E.: Optimizing Noncontiguous Accesses in MPI-IO. Parallel Computing 28, 83–105 (2002)
7. Singh, D.E., Isaila, F., Pichel, J.C., Carretero, J.: A Collective I/O Implementation Based on Inspector–Executor Paradigm. The Journal of Supercomputing 47(1), 53–75 (2009)
8. Worringen, J.: Self-adaptive hints for collective I/O. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) PVM/MPI 2006. LNCS, vol. 4192, pp. 202–211. Springer, Heidelberg (2006)
9. Buntinas, D., Mercier, G., Gropp, W.: Design and evaluation of Nemesis, a scalable, low-latency, message-passing communication subsystem. In: Sixth IEEE International Symposium on Cluster Computing and the Grid, CCGRID 2006, vol. 1, p. 10 (2006)
10. Graham, R., Shipman, G., Barrett, B., Castain, R., Bosilca, G., Lumsdaine, A.: Open MPI: A high-performance, heterogeneous MPI. In: 2006 IEEE International Conference on Cluster Computing, pp. 1–9 (2006)
11. Kunkel, J., Ludwig, T.: Performance Evaluation of the PVFS2 Architecture. In: PDP 2007: Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, Euromicro, pp. 509–516 (2007)
12. Rodrigues, A.F., Murphy, R.C., Kogge, P., Underwood, K.D.: The Structural Simulation Toolkit: Exploring Novel Architectures. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, SC. ACM, New York (2006)
13. Hoefler, T., Schneider, T., Lumsdaine, A.: LogGOPSim: Simulating Large-Scale Applications in the LogGOPS Model. In: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, HPDC, pp. 597–604. ACM, New York (2010)
14. Girona, S., Labarta, J., Badía, R.M.: Validation of Dimemas Communication Model for MPI Collective Operations. In: Dongarra, J., Kacsuk, P., Podhorszki, N. (eds.) PVM/MPI 2000. LNCS, vol. 1908, pp. 39–46. Springer, Heidelberg (2000)
15. Hermanns, M.A., Geimer, M., Wolf, F., Wylie, B.J.N.: Verifying Causality between Distant Performance Phenomena in Large-Scale MPI Applications. In: Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, pp. 78–84 (2009)

16. Tu, B., Fan, J., Zhan, J., Zhao, X.: Accurate Analytical Models for Message Passing on Multi-core Clusters. In: Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, pp. 133–139 (2009)
17. Cope, J., Liu, N., Lang, S., Carns, P., Carothers, C., Ross, R.: CODES: Enabling Co-design of Multilayer Exascale Storage Architectures. In: Proceedings of the Workshop on Emerging Supercomputing Technologies 2011 (2011)
18. Kunkel, J.: Simulating Parallel Programs on Application and System Level. Computer Science – Research and Development (online first) (May 2012)
19. Kuhn, M., Kunkel, J., Ludwig, T.: Simulation-Aided Performance Evaluation of Server-Side Input/Output Optimizations. In: 20th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, pp. 562–566 (2012)
20. Mordvinova, O., Runz, D., Kunkel, J., Ludwig, T.: I/O Performance Evaluation with Parabench – Programmable I/O Benchmark. Procedia Computer Science, 2119–2128 (2010)

Software Design Space Exploration for Exascale Combustion Co-design*

Cy Chan, Didem Unat, Michael Lijewski, Weiqun Zhang, John Bell, and John Shalf

Lawrence Berkeley National Laboratory

Abstract. The design of hardware for next-generation exascale computing systems will require a deep understanding of how software optimizations impact hardware design trade-offs. In order to characterize how co-tuning hardware *and* software parameters affects the performance of combustion simulation codes, we created ExaSAT, a compiler-driven static analysis and performance modeling framework. Our framework can evaluate hundreds of hardware/software configurations in seconds, providing an essential speed advantage over simulators and dynamic analysis techniques during the co-design process. Our analytic performance model shows that advanced code transformations, such as cache blocking and loop fusion, can have a significant impact on choices for cache and memory architecture. Our modeling helped us identify tuned configurations that achieve a 90% reduction in memory traffic, which could significantly improve performance and reduce energy consumption. These techniques will also be useful for the development of advanced programming models and runtimes, which must reason about these optimizations to deliver better performance and energy efficiency.

1 Introduction

One of the challenges facing the scientific computing community is to ensure applications will perform well on future exascale machines years in advance of their arrival. Meeting the extreme power and performance challenges of HPC system design over the next decade requires a tightly coupled hardware/software co-design process that optimizes both the application *and* the hardware to meet target performance, power, and cost requirements [1]. Tuning software or hardware in isolation is insufficient to reach the optimal balance of these design goals. To this end, we require a capability to rapidly estimate the performance of scientific applications in various potential hardware and software configurations.

We present the *ExaSAT* (Exascale Static Analysis Tool) framework, which enables us to rapidly explore the effects of code optimizations on the performance of a target application in the context of varying hardware parameters.

* This manuscript has been authored by an author at Lawrence Berkeley National Laboratory under Contract No. DE-AC02-05CH11231 with the U.S. Department of Energy. The U.S. Government retains, and the publisher, by accepting the article for publication, acknowledges, that the U.S. Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for U.S. Government purposes.

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 196–212, 2013.

Previous work includes cycle-accurate hardware simulators such as RAMP Gold [2] and discrete event simulators such as SST [3], which produce more accurate performance predictions than are feasible with static analysis but are more computationally expensive. Dynamic binary instrumentation tools such as Pin [4] can also be used to analyze the performance of a code by capturing events during code execution, but are subject to the quirks of the x86 ISA and compiler. In contrast, our framework provides a quantitative measure of application requirements through static code analysis, allowing us to characterize the co-design parameter space much more quickly than would be possible with simulators or dynamic analysis alone. Aspen [5] is a recent notation based language for analytical modeling where the programmer inserts a description of the application's performance behavior into the code. ExaSAT automatically generates a performance model directly from the source code without requiring programmer intervention, allowing us to analyze larger codes more easily.

We applied our framework to two combustion *proxy applications* (CNS and SMC) that were developed by the DOE Exascale Combustion Codesign Center (ExaCT) [6] to provide a representative set of core computational kernels required for combustion simulation. The majority of stencil computations at the heart of these codes are memory bandwidth bound on current architectures [7,8] and are predicted to become even more so on future architectures as computational throughput is expected to increase faster than memory bandwidth [9,10]. Furthermore, data movement is expected to become an increasingly important contributor to power consumption for exascale machines [11,12].

Because memory traffic is so critical, our analysis focuses on the effects of software optimizations that are intended to reduce data movement between the CPU and memory, rather than reducing the number of floating point operations. We examine optimal *cache blocking* (or *tiling*) and *loop fusion* code transformations and their effect on hardware design trade-offs as they relate to application performance for our combustion proxy applications. The software design space is parameterized to expose many of the potential realizations of the application and constituent kernels so that the best implementation can be selected. Applying our framework, we observe up to a 45% and 90% reduction in memory traffic when we apply optimal tiling and aggressive loop fusion, respectively.

Hardware complexity has increased to the point that current compilers are no longer able to automatically produce the code optimizations needed to achieve optimal performance on every target architecture. This paper demonstrates the impact of advanced code transformations that are beyond the capability of current compilers to produce and provides guidance for the development of new programming models and runtimes that will support these transformations. We discuss the following contributions in this work:

- We designed and implemented a fast, flexible static analysis and performance modeling framework and XML-based intermediate representation that can be used to estimate the performance of stencil computations and help explore trade-offs for co-design.

- We utilized our framework to profile the characteristics and estimate the performance of two combustion code variants: CNS and SMC, under a variety of hardware and software configurations.
- We used this model to illustrate the impact of cache blocking and loop fusion optimizations on hardware trade-offs, in particular how they affect cache size and memory bandwidth requirements.
- Our framework provides the deep code analysis and modeling necessary for future programming models and runtimes to reason about choices and make adaptations without a costly combinatorial search.
- Our analysis serves as a key vehicle for communicating with our industry partners for co-designing an exascale machine.

2 CNS and SMC Combustion Codes

To represent the characteristics of a range of combustion applications, we studied two proxy applications developed by the ExaCT project. The CNS code is a simple proxy that integrates the compressible Navier-Stokes equations assuming constant transport properties. It is intended to capture the computational characteristics of the dynamical core of a combustion simulation. The SMC code is a more advanced proxy for the direct numerical simulation combustion code S3D [13], adding detailed models for chemical species diffusion and kinetics. SMC contains the key elements of both the dynamical core and the chemical kinetics components of S3D; however, it uses a simpler temporal integration algorithm that does not include automatic error control. Both codes are based on the high-accuracy solution of a system of PDEs of the form:

$$\frac{\partial U}{\partial t} + \nabla \cdot \mathcal{F}(\mathcal{U}) = \nabla \cdot \mathcal{D}(\mathcal{U}) + \mathcal{S} \quad .$$

Here U is a vector of unknowns, representing density, energy and three components of momentum with an additional density for each chemical species, for a total of $5 + N_s$ unknowns where N_s is the number of species in the problem (1 for CNS). The terms $\mathcal{F}$, $\mathcal{D}$, and S correspond to hyperbolic transport, nonlinear diffusive processes and chemical source terms, respectively. The dynamical core uses 8th-order stencil operations to approximate spatial derivatives, converting the system into a large collection of ordinary differential equations that are integrated using a third-order, low-storage, TVD Runge-Kutta scheme [14,15]. The chemical source term is a computationally intensive single-point physics routine that uses a large number of computationally expensive transcendental function evaluations. Further details on our approach will be presented in a forthcoming paper [16].

Figure 1 illustrates the most data-intensive stencil access pattern used in the CNS and SMC codes. This stencil reads values four grid elements deep in both directions of each of three dimensions; however, there are many other stencil patterns in the code that read only a subset of the points shown. Our tool separately analyzes each stencil access for every array in every loop to estimate working sets and data movement.

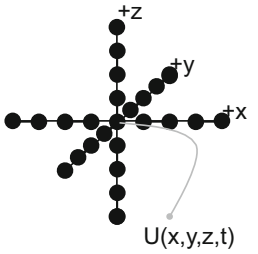


Fig. 1. 3D stencil access pattern in the SMC and CNS codes

Table 1 shows some computational properties of the codes studied. In addition to the ones shown, we also analyzed versions of the SMC code with 21, 71, and 107 chemical species, where the number of species varies with fuel type, from hydrogen to methane to biofuels. The CNS code (which simulates dynamics only) and the SMC dynamics codes have very different computational characteristics compared with the SMC chemistry codes. In our analysis, we utilize the *bytes per flop (B/F) ratio*, which represents the required number of bytes to be transferred between the processor and off-chip memory divided by the required number of floating point operations needed for a particular code. The division and transcendental operations are weighted since they cost more than adds and multiplies (see Section 4.2 for details).

An algorithm with a low B/F ratio will likely be computationally bound, while one with a high B/F ratio will likely be memory bandwidth bound. The CNS and SMC dynamics codes have a relatively high B/F ratio especially in a cache constrained environment (up to 2.22). They exhibit a high degree of data reuse both within and across loops, resulting in a lot of potential to reduce memory traffic using the optimizations discussed in this work. In contrast, the chemistry code is dominated by expensive floating point divisions and trancendental functions with a relatively light memory access requirement, resulting in a much lower B/F ratio. For the 53 species SMC code, the difference is roughly two orders of magnitude (0.01 vs. 1.48, cache-constrained). We expect the dynamics code to be bandwidth bound on most current and future architectures while the chemistry code will remain compute bound.

Although this paper focuses mainly on memory traffic optimizations that improve the performance of the dynamics codes, there are computational optimizations such as vectorization or pipelining that could help improve the throughput of the chemistry codes. Furthermore, the disparity in arithmetic intensity between the dynamics and chemistry codes suggests that co-scheduling could have each code utilize different parts of the processor simultaneously. Support for such optimizations within a programming model and runtime will be explored in future work.

Table 1. CNS and SMC code characteristics (for 1, 9, and 53 chemical species). RK = Runge-Kutta step. †Weighted flops. ‡Cache available to group of threads cooperating on a working set.

		CNS	SMC Dynamics		SMC Chemistry	
Num. 3D spatial loops		14	27		1	
Number of species		1	9	53	9	53
Flops/point per RK	Adds	821	7005	30677	619	7923
	Muls	797	7871	34860	815	9432
	Divs	6	66	374	39	540
	Trans	1	1	1	51	710
Arrays/RK	Resident	40	157	685	20	108
	Reads	92	557	2405	20	108
	Writes	40	253	1045	9	53
Bytes/Flop†	Unlimited $	1.32	0.82	0.74	0.03	0.01
	512 kB $‡	2.22	1.25	1.48	0.03	0.01

3 Software Design Space

3.1 Cache Blocking

The first optimization, cache blocking, focuses on reducing capacity misses (see [17] for more on 3C's cache model) as a core sweeps through the problem's iteration space. Tiling the iteration sweep reduces the size of the working set required to enable temporal reuse of data. If the working set is reduced to within the size of available on-chip memory, capacity misses can be reduced or eliminated, thus decreasing the necessary memory traffic between the CPU and DRAM.

The relationship between working set, cache size, and memory traffic can sometimes cause unexpected performance effects. For example, many programmers may parallelize a triply nested loop by associating an OpenMP `parallel for` pragma with the outermost loop (see Figure 2). This strategy yields the coarsest grain parallelism, which minimizes the overhead of spawning and syncing the resulting threads. However, parallelizing the middle loop instead reduces the working set of each thread by a factor of four (the number of threads). If the reduced working set now fits into the cache, then there may be a significant performance benefit. There is a trade-off in this scenario between cache size, memory bandwidth, and the costs of spawning and syncing threads.

Another trade-off for tiling is redundant *ghost zone* traffic that must be pulled in for each block. The ghost zone consists of neighboring cells outside of the tile that must be read due to the shape of the stencil access pattern (see Figure 1). The left diagram in Figure 3 shows a single, unblocked tile with ghost zones on the outside the tile. As the tile size is decreased (center and right diagrams), the ghost zones overlap with neighboring blocks (indicated with deeper shading).

Previous work has shown the benefits of cache blocking stencil codes [8,7]. In this paper, we are interested more in illustrating the co-design trade-offs that are exposed by software optimizations such as blocking, rather than the

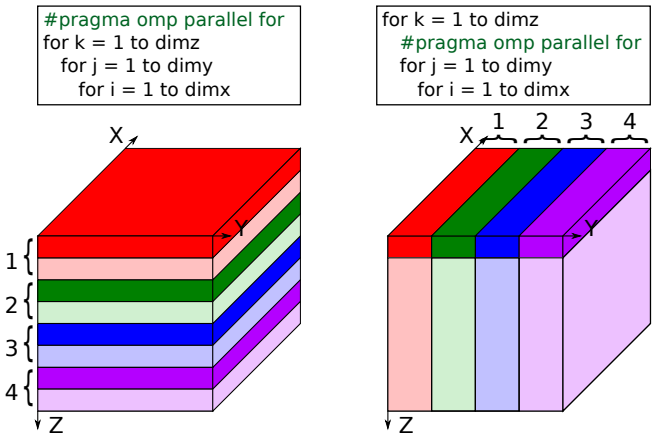


Fig. 2. Working sets that result from using OpenMP `parallel for` with the outermost vs. middle loop with four threads. Each color/number indicates the subgrid updated by a thread. Bold regions indicate the working set tile.

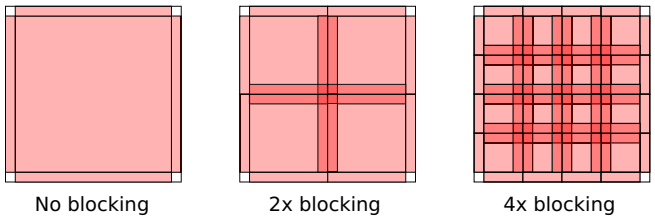


Fig. 3. 2D representation of the cache blocking optimization in the X and Y dimensions. Overlapping ghost zones are indicated by the deeper shading in the diagram.

raw performance benefits enabled. To this end, we developed a cache model to estimate the level of data reuse given different configurations. For any particular cache configuration, a blocking strategy may be chosen that balances the penalty of capacity misses against the overhead of redundant ghost-cell traffic. This optimization exposes a trade-off in hardware between cache size and memory bandwidth explored further in Section 5.

3.2 Loop Fusion

The second optimization, loop fusion, focuses on eliminating the need to stream arrays in and out of memory. While some compilers already implement fusion, they tend to do so to enhance instruction level parallelism and to help hide latency. In contrast, we apply loop fusion for the purpose of decreasing memory traffic by reducing the number of times arrays are transferred to or from memory [18].

Figure 4 shows an example of a loop fusion optimization. In Scenario 1, array A must be streamed from memory twice compared to just once for Scenario 2.

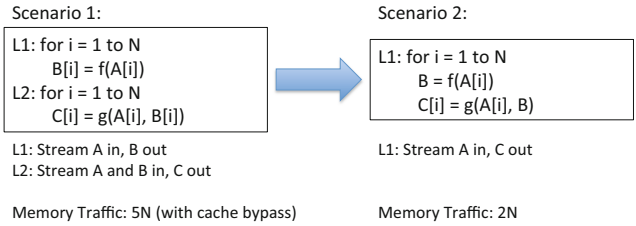


Fig. 4. Example of loop fusion code optimization

Also in Scenario 1, array B must be streamed to memory and back, while in the fused case the array can be replaced with a temporary variable (so long as B is not needed afterwards). Assuming cache bypassed writes, this optimization reduces memory traffic from $5N$ to $2N$, where N is the size of each array.

The trade-off for fusing loops is that the register and cache working sets grow, potentially causing a reduction in performance if the working sets no longer fit within on-chip memory. Loop fusion exposes a trade-off in the hardware involving the balance of memory bandwidth with registers and cache size. Our framework allows us to explore the impact of this transformation on memory traffic in the context of varying on-chip memory capacities. We will explore a couple strategies for applying loop fusion and their effects in Section 5.

4 Methodology

4.1 Framework and Toolchain

We developed a stencil-specific static analysis and performance modeling tool to help estimate the performance of target codes on various potential hardware platforms. Figure 5 illustrates our framework, which consists of roughly two stages of analysis. The first stage, which is built on top of the ROSE compiler [19], takes Fortran code as an input and extracts key characteristics about the computation and data access patterns and stores them in an XML intermediate representation (XML-IR). Data in the XML-IR include (but are not limited to) the following:

- Loop nest structure, bounds, and strides
- Floating point operations
- Scalar accesses (number of reads and writes)
- Array accesses (number of reads and writes for each index)

The second stage (written in Python) combines the XML-IR with user-provided problem parameters (e.g. box size, number of chemical species), machine parameters (e.g. computational throughput, cache size, memory bandwidth), and software optimizations (e.g. loop transformations) to produce estimates of key performance metrics such as working set sizes, DRAM traffic, B/F ratio, and execution time. The resulting performance model can be executed within a script

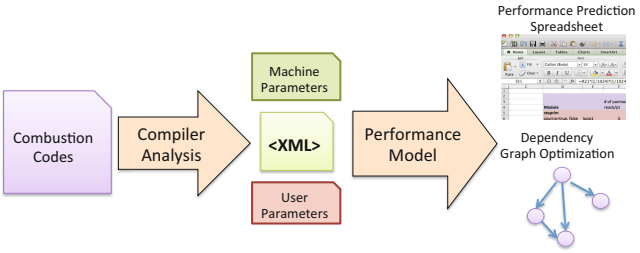


Fig. 5. ExaSAT Tool Chain

to rapidly explore parameter configurations, and it can additionally produce spreadsheets, dependency graphs, and tables with additional details such as array residency and access frequency, memory footprints, and state variable statistics. When applied to the SMC proxy application, our framework can evaluate roughly 900 hardware/software configurations per minute on a laptop. More details will be provided in future work [20].

4.2 Hardware and Performance Model

We utilize a simple hardware model (shown in Figure 6) that abstracts the machine as a collection of parallel hardware cores with some parameterized on-chip memory. Our hardware model exposes the following architectural parameters, which were identified through discussion with industry participants in the DOE Fast Forward program:

- Aggregate computational throughput
- Aggregate memory bandwidth
- Cache or scratchpad size
- Cache line and word sizes
- Cost of special functions (e.g. divisions or transcendentals)

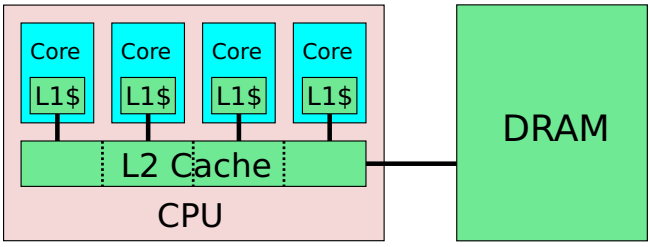


Fig. 6. Hardware model featuring the CPU with two levels of on-chip cache and separate DRAM connected by a bus

The CPU model is agnostic to the number of cores, instead taking the aggregate computational throughput as a model parameter. The memory model

similarly utilizes a parameter specifying the aggregate bandwidth between the CPU and the DRAM (i.e. the STREAM bandwidth) over the memory bus. Figure 6 shows an example cache configuration with private L1 caches and a partitioned, shared L2 cache. While there are many other possible on-chip memory configurations, we are mainly interested in the resulting bandwidth filtering capability, which is primarily determined by the total amount of (non-inclusive) on-chip memory per thread or group of cooperating threads. The performance model focuses on aggregate performance of the machine rather than simulating individual components and their interactions, It captures the costs of the computational workload and data movement and the performance implications of data reuse (or lack thereof).

The performance of an application is estimated in the following way: let α be the aggregate computational throughput of the machine and β be the aggregate memory bandwidth. Also, let C be the application's total computational work and D be the total necessary data movement between the CPU's on-chip memory and DRAM. Then the estimated running time is $T = \max(T_c, T_d)$, where $T_c = \frac{C}{\alpha}$ is the CPU time and $T_d = \frac{D}{\beta}$ is the DRAM time. Our modeling framework is not intended to provide exact performance predictions, but rather sets a performance upper-bound in the spirit of the Roofline model [21].

Since some floating point operations such as divides and transcendentals can take significantly longer to execute than adds and multiplies, our performance model can weight these special operations according to their relative costs. For this paper, we weighted the costs of these operations according to their non-SIMD throughput on the Intel Sandy Bridge architecture [22,23]. The resulting *weighted flop* count determines the estimated CPU time of the computation. Figures 7(a) and 7(b) show the significance of weighting the floating point operations by their cost. In the chemistry module of the SMC proxy application, the CPU time is dominated by transcendentals, even though the transcendental operation count is a small percentage of the total flops.

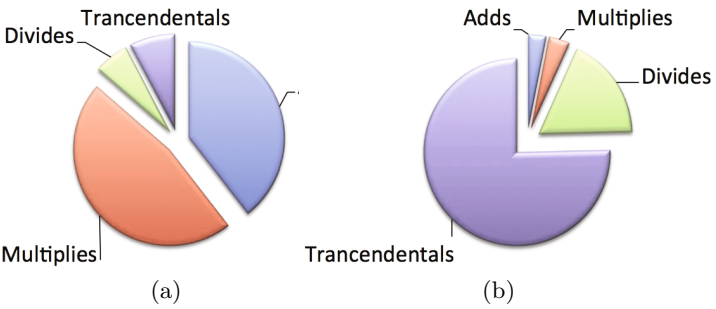


Fig. 7. (a) Floating point instruction mix and (b) CPU Time (T_c) for the chemistry part of the SMC code for 53 chemical species, assuming divides and trancendentals take a relative factor of 39x and 125x longer than the adds and multiplies, respectively

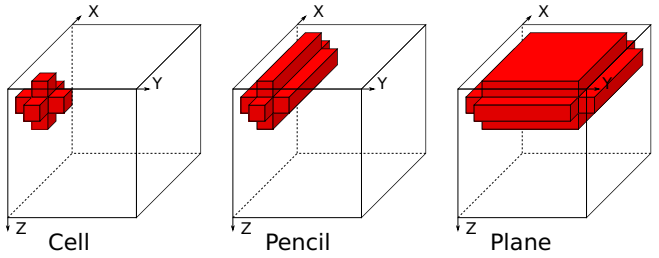


Fig. 8. Working set sizes to enable different levels of reuse for an example 7-point 3D stencil. The block is swept in a triply nested loop with the X dimension swept in the innermost loop and Z in the outermost.

Memory and Cache Model. In order to determine D, the total data movement, our cache model captures the data reuse pattern that occurs with stenciled array accesses. The on-chip memory is modeled as an ideal, fully-associative cache with a least-recently used (LRU) replacement policy. Our model assumes reuse of data will occur if the associated working set is small enough to fit in on-chip memory. Actual cache behavior is likely to under-perform in comparison due to conflict misses and imperfect replacement, though our model should capture the first-order behavior of a finite-cache memory system. Threads on a chip may cooperate to gain the benefit of a larger aggregate memory space in which to store a shared working set, possibly enabling larger block sizes and reduced memory traffic; however, the costs of sharing data between the caches on the chip are not included in our model. The amount of memory traffic required to execute a particular stencil loop is determined by the amount of the on-chip memory available per group of threads collaborating on a working set. Our model uses the specified cache size to 1) determine what temporal reuse of data will occur as threads sweep through the grid and 2) estimate the resulting cache miss traffic.

Figure 8 shows the working set sizes needed to enable reuse between cells, pencils, and planes. If no on-chip memory is available, every array access in the kernel requires data to be transferred from DRAM. If the cell working set (left) fits in cache, then those values will remain in cache for reuse on the next cell iteration. Similarly, if the pencil working set (middle) fits into cache, there will be reuse between pencil sweeps, and so forth. Based on the shape of the stencil access pattern, our model computes 1) the sizes of the working sets and 2) the resulting memory traffic for each of the reuse cases. This information is then combined with hardware and software parameters to determine the estimated memory traffic and DRAM time required for each array in every loop in the code.

If on-chip data movement is a concern, a conservative estimate can be made by limiting the size of modeled on-chip memory to the size of the private L1 cache; however, the resulting memory traffic estimates produced by our model will be the total traffic between the L1 cache and the next level of on-chip memory rather than the traffic between the CPU and DRAM. Our methodology could

potentially be extended in future work to do a multi-level analysis that computes bandwidth filtering and performance modeling at every level of cache.

Model Validation. Figure 9 shows the effect of tiling with three simple 7-point stencil benchmark kernels on a single node of the NERSC "Hopper" Cray XE6 using 384^3 data grids. Table 2 shows properties of the benchmarked machine. We measured the execution time for 24 concurrent threads (1 thread per core) with no software prefetch or cache bypass used. To first-order approximation, the measured execution times correlate well with our model's predictions with respect to optimal execution times and block sizes. The model departs from measurement for smaller blocks as the hardware prefetchers are no longer able to hide load latencies for short stanza accesses. We also observe the effect of the machine's randomized cache replacement policy, which smooths the sharp transition in the model at the point where the working set grows larger than the cache and capacity misses begin to occur. Since static analysis does not resolve system behavior to the same degree of precision as an event simulator in the interest of speed and flexibility, our results are necessarily more comparative than absolute in nature. That said, we believe valuable lessons can be learned from examining the trade-offs exposed by our analysis framework in the co-design parameter space.

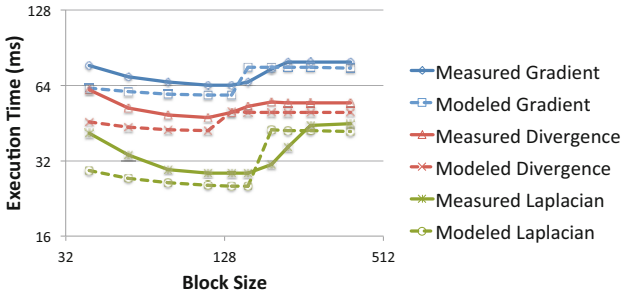


Fig. 9. Measured and modeled execution times for various blocking sizes for three benchmark finite difference kernels

5 Results

Since the optimizations studied do not change the amount of computation (flops) required, the B/F ratio can be used as a proxy metric for memory traffic (and thus execution time) in memory bandwidth bound codes. In this context, the B/F ratio is an indicator of relative code performance independent of a particular machine's specifications, and is useful when making comparisons between code requirements and machine capabilities during the design process.

Table 2. Properties of a NERSC Cray XE6 compute node [24]. All numbers given are *per node* except cache, which are given *per core*. †Data cache.

One NERSC "Hopper" Node	
CPUs	Opteron 6172
Sockets / Cores	2 / 24
Peak Compute	201.6 Gflop/s
Priv. L1/L2 $	64 kB† / 512 kB
Shared L3 $	1 MB / core
Mem. Interface	DDR3-1333
Mem. Channels	8
Peak Mem. BW	72 GiB/s
STREAM BW	∼51 GiB/s
Peak B/F Ratio	0.38

5.1 Cache Blocking

Figure 10 shows the B/F ratio for the CNS proxy application for different block sizes and on-chip cache sizes. In many cases there is insufficient cache to enable the best reuse case outlined in Section 4.2. Blocking the iteration space reduces the sizes of the working sets needed to enable reuse, but incurs the overhead of pulling in additional ghost zones for the smaller blocks. This overhead is illustrated by the unlimited cache case, where the B/F ratio increases as the block size decreases. For a fixed block size, as the amount of cache is reduced, more capacity misses occur, increasing the B/F ratio. For a fixed cache size, we observe the minimum B/F ratio typically occurs at the largest block size whose working sets still fit within cache. The multiple inflection points are due to the code's various loops having different working set sizes and reuse behaviors.

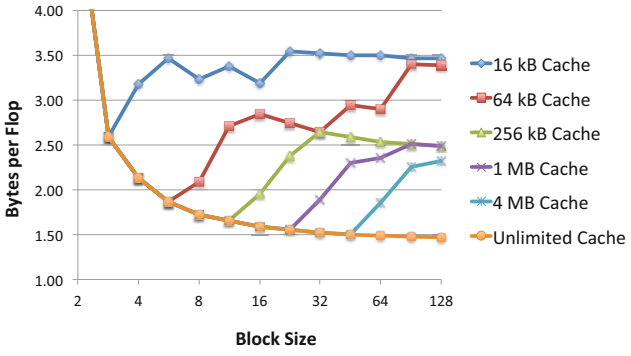


Fig. 10. Byte to flop ratio for various block and cache sizes

Figure 11 shows that as chemical species are added to the simulation, the memory traffic required per Runge-Kutta step increases across all block sizes.

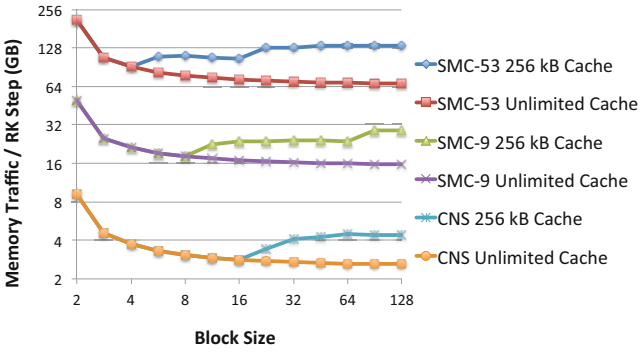


Fig. 11. Memory traffic per Runge-Kutta step as block size, cache size, and number of chemical species are varied

Since the working set size increases with number of species, the optimal block size for a fixed amount of cache decreases. In cases with a large number of chemical species, augmenting the cache resources on the chip would reduce the blocking overhead and ease memory bandwidth requirements.

5.2 Loop Fusion

We examined two variants of the loop fusion optimization: simple and aggressive. We use the term *stencil dependency* to refer to a data dependency between loops where data written by the first loop is read by the second in a stencil (non-pointwise) access pattern. In the simple fusion case, only loops with no stencil dependencies are fused, while in the aggressive fusion case, all loops in the solver are fused. Simple fusion can be applied without major changes to the loop bodies, but aggressive fusion requires the introduction of temporary buffers and a staggered update strategy to replace arrays with stencil dependencies. While our framework is able to model the effects of cache blocking without any manual code modification, the loop fusion transformations studied here were implemented manually using the dependency graphs generated by our tool for guidance. Analysis of the resulting fused code was then handled by our framework.

Figure 12 shows the dependency graphs generated by our framework (simplified for clarity) corresponding to the different fusion cases for the CNS code. The ovals correspond to loops in the solver, while rectangles represent data arrays. The arrows show which arrays are read and written by each loop (dashed arrows represent stencil dependencies).

Figure 13 shows the bandwidth filtering that results for the CNS code using various cache sizes and loop fusion strategies. For each point in the graph, the cache blocking strategy was independently chosen to minimize the resulting memory traffic in our model. The stair-step pattern observed with the simple fusion scenario is a result of the transition between cell, pencil, and plane reuse

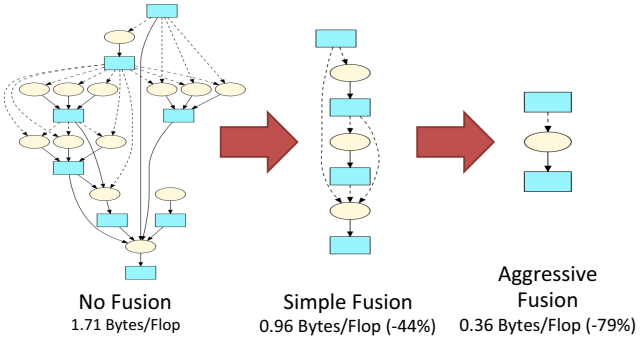


Fig. 12. Dependency graph showing loop fusion optimizations

cases as explained in Section 4.2. As expected, increasing the cache size introduces the opportunity to substantially reduce memory bandwidth requirements.

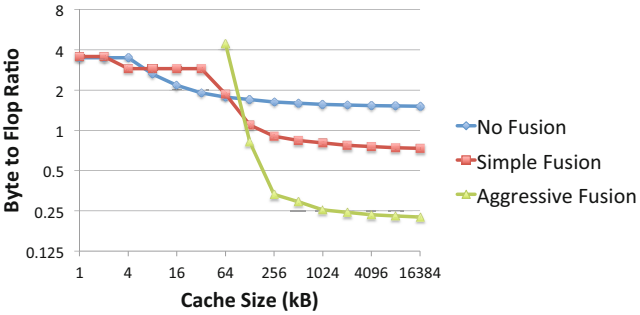


Fig. 13. Bandwidth filtering diagram for the CNS code for different loop fusion strategies

For small caches, there is typically only modest benefit from using loop fusion because the larger fused working sets require smaller block sizes to fit into cache. However, once the cache is large enough to fit the fused working sets, the benefits can be dramatic. For the largest cache, a 6.7x reduction in B/F ratio can be attained for the CNS code using the aggressive fusion strategy, but even with only 128 kB of cache the traffic is reduced by 2.1x compared to the unfused strategy.

5.3 Analysis

Given the size of the on-chip memory in our hardware configuration, we can choose the best overall optimization strategy to minimize memory traffic. As a result of applying the best combination of blocking and fusion strategies, the realizable bandwidth filtering curve is the minimum across the curves shown in Figure 13.

In some cases, making the trade-off of dedicating extra die area for cache can lead to a substantial benefit in power and performance from reduced memory

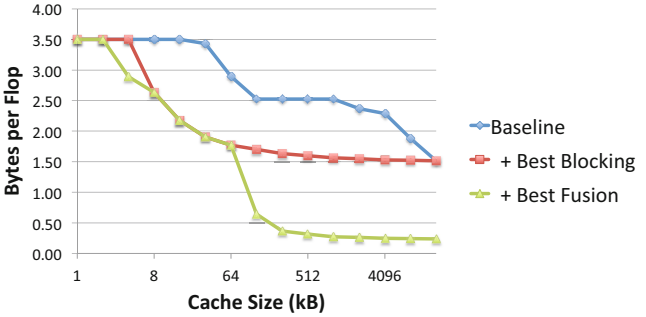


Fig. 14. Impact of software optimizations on the trade-off between cache size and memory bandwidth for the CNS code

traffic and lower bandwidth requirements. The effect on this trade-off due to software optimization is illustrated in Figure 14 for the CNS code. Tiling provides up to a 45% improvement versus the baseline unoptimized code at cache capacities larger than 8 kB. Loop fusion has the potential to filter bandwidth by as much as 90% compared to baseline, but it requires a cache larger than 64 kB. Similarly, Figure 15 shows the impact of the code optimizations on the 53 species SMC dynamics code. The stair-step pattern resulting from the transition between reuse cases is more prominent here due to the larger working sets. The SMC code has a lower baseline B/F ratio compared with CNS due to the arithmetic complexity of the code, but the minimum cache size for blocking improvements is higher than CNS due to the increased amount of data handled. Furthermore, several loops in the SMC code can be fused without large working set penalties, improving performance even with small cache sizes. Tiling provides up to a 39% improvement versus the baseline unoptimized code, while fusion can reduce traffic by up to 60% versus baseline.

The lowered bandwidth requirements due to these optimizations could have a significant beneficial impact on energy efficiency of future systems. However,

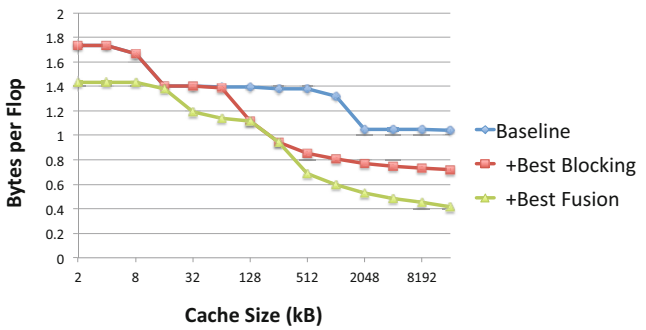


Fig. 15. Impact of software optimizations on the trade-off between cache size and memory bandwidth for the 53 species SMC dynamics code

the larger cache sizes required could be difficult to implement without moving towards a scratchpad memory implementation, such as the one used in the STI Cell processor. Our results show that it is essential to consider both software optimizations and hardware design parameters simultaneously. These observations would not have been apparent from benchmarking on fixed hardware alone.

6 Conclusions and Future Work

We developed a compiler-based framework that is able to automatically construct performance models directly from source code and applied it to explore trade-offs in the hardware design space using the CNS and SMC combustion proxy applications. Using this approach, we demonstrate tuned hardware/software configurations that achieve up to 45% and 90% reductions in compulsory memory traffic with the application of optimal data tiling and aggressive loop fusion, respectively. We believe this kind of deep code analysis and performance modeling demonstrates the importance for future advanced runtimes to make dynamic adaptations in the context of changing computing environments without a costly combinatorial search. Our analysis serves as a key vehicle for communicating with our vendor partners for co-designing an exascale machine.

We wish to generalize our approach and make it practical to apply these techniques to larger, more complex codes. Because the optimizations studied here require significant code transformations, current compilers are unable to perform them automatically. We are using the lessons learned here to guide the development of programming models and frameworks that will enable the automation of our code transformation and performance analysis techniques. For example, we are exploring the use of functional semantics and annotations to help reason about data flows and on-chip memory footprints. In summary, our work demonstrates the utility of a co-design approach, which explores the design space of software optimizations with parameterized hardware and offers deeper insight into the future of application and machine design.

Acknowledgements. The authors would like to thank George Michelogiannakis and Sam Williams for their insightful comments during the preparation of this paper. All authors from Lawrence Berkeley National Laboratory were supported by the Office of Advanced Scientific Computing Research in the Department of Energy Office of Science under contract number DE-AC02-05CH11231. This work is part of the DOE Center for Exascale Simulation of Combustion in Turbulence (ExaCT) and the DOE Co-Design for Exascale (CoDEx) projects.

References

1. Mohiyuddin, M., et al.: A design methodology for domain-optimized power-efficient supercomputing. In: SC 2009, pp. 12:1–12:12. ACM, New York (2009)
2. Tan, Z., et al.: RAMP Gold: An FPGA-based architecture simulator for multiprocessors. In: 2010 47th ACM/IEEE Design Automation Conference (DAC), DAC 2010, pp. 463–468 (June 2010)

3. Janssen, C.L., et al.: A simulator for large-scale parallel computer architectures. International Journal of Distributed Systems and Technologies 1(2), 57–73 (2010)
4. Luk, C.-K., et al.: Pin: building customized program analysis tools with dynamic instrumentation. In: PLDI 2005, pp. 190–200. ACM, New York (2005)
5. Spafford, K.L., Vetter, J.S.: Aspen: a domain specific language for performance modeling. In: SC 2012, pp. 84:1–84:11. IEEE Computer Society Press, Los Alamitos (2012)
6. ExaCT: Center for exascale simulation of combustion in turbulence. Website (2013), http://exactcodesign.org
7. Datta, K., et al.: Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: SC 2008, pp. 4:1–4:12. IEEE Press, Piscataway (2008)
8. Rivera, G., Tseng, C.-W.: Tiling optimizations for 3d scientific computations. In: Supercomputing 2000. IEEE Computer Society, Washington, DC (2000)
9. Kogge, P., et al.: Exascale computing study: Technology challenges in achieving exascale systems (2008)
10. Shalf, J., Dosanjh, S., Morrison, J.: Exascale computing technology challenges. In: Palma, J.M.L.M., Daydé, M., Marques, O., Lopes, J.C. (eds.) VECPAR 2010. LNCS, vol. 6449, pp. 1–25. Springer, Heidelberg (2011)
11. Miller, D.A.B.: Rationale and challenges for optical interconnects to electronic chips. In: Proc. IEEE, pp. 728–749 (2000)
12. Borkar, S.: Design challenges of technology scaling. IEEE Micro 19(4), 23–29 (1999)
13. Chen, J.H., et al.: Terascale Direct Numerical Simulations of Turbulent Combustion Using S3D. Comput. Sci. Disc. 2(015001) (2009)
14. Gottleib, S., Shu, C.: Total variation diminishing Runge-Kutta schemes. Mathematics of Computation 67(221), 73–85 (1998)
15. Qiu, J., Shu, C.: Runge-Kutta discontinuous Galerkin method using WENO limiters. SIAM J. Sci. Comp. 26(3), 907–929 (2005)
16. Zhang, W., et al.: Multirate higher-order discretization approaches for the multicomponent, reaction compressible Navier-Stokes equations (in preparation)
17. Hill, M.D., Smith, A.J.: Evaluating associativity in cpu caches. IEEE Trans. Comput. 38(12), 1612–1630 (1989)
18. Ding, C., Kennedy, K.: Improving effective bandwidth through compiler enhancement of global cache reuse. J. Parallel Distrib. Comput. 64(1), 108–134 (2004)
19. Quinlan, D.J., Miller, B., Philip, B., Schordan, M.: Treating a user-defined parallel library as a domain-specific language. In: IPDPS 2002, p. 324. IEEE Computer Society (2002)
20. Unat, D., Chan, C., et al.: Exasat: A static analysis and performance modeling tool for exascale co-design (in preparation)
21. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Commununications of the ACM 52(4), 65–76 (2009)
22. Williams, S.: Intel Sandy Bridge SVML benchmark results (2012)
23. Vladimirov, A.: Arithmetics on Intel's Sandy Bridge and Westmere CPUs: not all FLOPS are created equal. Colfax International (2012)
24. NERSC: Cray XE6 (Hopper). Website (2013), http://www.nersc.gov/users/computational-systems/hopper

Beyond the CPU: Hardware Performance Counter Monitoring on Blue Gene/Q

Heike McCraw[1], Dan Terpstra[1], Jack Dongarra[1],
Kris Davis[2], and Roy Musselman[2]

[1] Innovative Computing Laboratory (ICL), University of Tennessee, Knoxville
1122 Volunteer Blvd, Knoxville TN, 37996
{mccraw,terpstra,dongarra}@icl.utk.edu
[2] Blue Gene System Performance
Dept. KOKA, Bldg. 30-2, IBM Rochester, MN 55901
{krisd,mussel}@us.ibm.com

Abstract. The Blue Gene/Q (BG/Q) system is the third generation in the IBM Blue Gene line of massively parallel, energy efficient supercomputers that increases not only in size but also in complexity compared to its Blue Gene predecessors. Consequently, gaining insight into the intricate ways in which software and hardware are interacting requires richer and more capable performance analysis methods in order to be able to improve efficiency and scalability of applications that utilize this advanced system.

The BG/Q predecessor, Blue Gene/P, suffered from incompletely implemented hardware performance monitoring tools. To address these limitations, an industry/academic collaboration was established early in BG/Q's development cycle to insure the delivery of effective performance tools at the machine's introduction. An extensive effort has been made to extend the Performance API (PAPI) to support hardware performance monitoring for the BG/Q platform. This paper provides detailed information about five recently added PAPI components that allow hardware performance counter monitoring of the 5D-Torus network, the I/O system and the Compute Node Kernel in addition to the processing cores on BG/Q.

Furthermore, we explore the impact of node mappings on the performance of a parallel 3D-FFT kernel and use the new PAPI network component to collect hardware performance counter data on the 5D-Torus network. As a result, the network counters detected a large amount of redundant inter-node communications, which we were able to completely eliminate with the use of a customized node mapping.

1 Introduction

With the increasing scale and complexity of large computing systems the effort of performance optimization and the responsibility of performance analysis tool developers grows more and more. To be of value to the High Performance Computing (HPC) community, performance analysis tools have to be customized

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 213–225, 2013.

as quickly as possible in order to support new processor generations as well as changes in system designs.

The Blue Gene/Q (BG/Q) system is the third generation in the IBM Blue Gene line of massively parallel, energy efficient supercomputers. BG/Q is capable of scaling to over a million processor cores while making the trade-off of lower power consumption over raw processor speed [4]. BG/Q increases not only in size but also in complexity compared to its Blue Gene predecessors. Consequently, gaining insight into the intricate ways in which software and hardware are interacting requires richer and more capable performance analysis methods in order to be able to improve efficiency and scalability of applications that utilize this advanced system.

Performance analysis tools for parallel applications running on large scale computing systems typically rely on hardware performance counters to gather performance relevant data from the system. The Performance API (PAPI) [3] has provided consistent platform and operating system independent access to CPU hardware performance counters for more than a decade. In order to provide the very same consistency for BG/Q to the HPC community - and thanks to a close collaboration with IBMs Performance Analysis team - an extensive effort has been made to extend PAPI to support hardware performance monitoring for the BG/Q platform. This customization of PAPI to support BG/Q also includes a growing number of PAPI components to provide valuable performance data that not only originates from the processing cores but also from compute nodes and the system as a whole. More precisely, the additional components allow hardware performance counter monitoring of the 5-dimensional (5D) Torus network, the I/O system and the Compute Node Kernel in addition to the CPU component.

This paper provides detailed information about the expansion of PAPI to support hardware performance monitoring for the BG/Q platform. It offers insight into supported monitoring features. Furthermore, it will discuss performance counter data of a parallel 3-dimensional Fast Fourier Transform (3D-FFT) computation. We explore the impact of a variety of node mappings on the performance of a 3D-FFT kernel and use the recently introduced PAPI network component for BG/Q to collect hardware performance counter data on the 5D-Torus network.

This paper is organized as follows. The next section provides a brief overview of the BG/Q hardware architecture with focus on the features that are particularly relevant for this project. Section 3 goes into detail on how PAPI has been expanded with five components to support hardware performance counter monitoring on the BG/Q platform. Our case study is discussed in Section 4 which includes a short description of the implementation of the parallel 3D-FFT algorithm with a two-dimensional data decomposition as well as results of the experimental study. We conclude and summarize our work in Section 5.

2 Overview of the Blue Gene/Q Architecture

2.1 Hardware Architecture

The BG/Q processor is an 18-core CPU of which 16 cores are used to perform mathematical calculations. The 17th core is used for node control tasks such as offloading I/O operations which "talk" to Linux running on the I/O node. (Note, the I/O nodes are separate from the compute nodes; so Linux is not actually running on the 17th core.) The 18th core is a spare core which is used when there are corrupt cores on the chip. The corrupt core is swapped and software transparent. In the remainder of this paper we focus on the 17 usable cores only, since there are really only 17 logical units available on the CPU.

The processor uses PowerPC A2 cores, operating at a moderate clock frequency of 1.6 GHz and consuming a modest 55 watts at peak [6]. The Blue Gene line has always been known for throughput and energy efficiency, a trend which continues with the A2 architecture. Despite the low power consumption, the chip delivers a very respectable 204 Gflops [6]. This is due to a combination of features like the high core count, support for up to four threads per core, and a quad floating-point unit. Compared to its Blue Gene predecessors, BG/Q represents a big change in performance, thanks to a large rise in both core count and clock frequency. The BG/Q chip delivers 15 times as many peak FLOPS as its BG/P counterpart and 36 times as many as the original BG/L design (see Table 1 for comparison).

Table 1. Brief summary of the three Blue Gene versions

Version	Core Architecture	Instruction Set	Clock Speed	Core Count	Interconnect	Peak Performance
BG/L	PowerPC 440	32-bit	700 MHz	2	3D-Torus	5.6 Gigaflops
BG/P	PowerPC 450	32-bit	850 MHz	4	3D-Torus	13.6 Gigaflops
BG/Q	PowerPC A2	64-bit	1600 MHz	17	5D-Torus	204.8 Gigaflops

This PowerPC A2 core has a 64-bit instruction set compared to the 32-bit chips used in the prior BG/L and BG/P supercomputers. The A2 architecture has a 16 KB private L1 data cache and another 16 KB private L1 instruction cache per core, as well as 32 MB of embedded dynamic random access memory (eDRAM) acting as an L2 cache, and 8 GB (or 16 GB) of main memory [9]. The L2 cache as well as the main memory are shared between the cores on the chip.

Every BG/Q processor has two DDR3 memory controllers, each interfacing with eight slices of the L2 cache to handle their cache misses (one controller for each half of the 16 compute cores on the chip) [1,10]. This is an important feature that will be described in more detail in the discussion of the PAPI `L2Unit` component in Section 3.2.

BG/Q peer-to-peer communication between compute nodes is performed over a 5D-Torus network (note that BG/L and P feature a 3D-Torus). Each node has 11 links and each link can simultaneously transmit and receive data at 2 GB/s for a total bandwidth of 44 GB/s. While 10 links connect the compute

nodes, the 11th link provides connection to the I/O nodes. The I/O architecture is significantly different from previous BG generations since it is separated from the compute nodes and moved to independent I/O racks.

3 PAPI BG/Q Components

The general availability of PAPI for BG/Q is due to a cooperative effort of the PAPI team and the IBM performance team. This joint effort started with careful planning long before the BG/Q release, with the goal to design PAPI for Q as well as to design the Blue Gene Performance Monitoring API (BGPM) according to what is needed by PAPI and other HPC performance analysis tools, like e.g. HPCToolkit [2] that heavily use PAPI under the covers.

In general, hardware performance event monitoring for BG/Q requires user code instrumentation with either the native BGPM API or a tool like PAPI which relies on BGPM. The following five sections talk about the five different components that have been implemented in PAPI to allow users to monitor hardware performance counters on the BG/Q architecture through the standard Performance API interface.

3.1 Processor Unit Component

The PAPI `PUnit` component is handled as component 0 in PAPI - which is the default CPU component. Each of the 17 usable A2 CPU cores has a local Universal Performance Counting (UPC) module. Each of these modules provides 24 counters (14-bit) to sample A2 events, L1 cache related events, floating point operations, etc. Each local UPC module is broken down into five internal sub-modules: functional unit (FU), execution unit (XU), integer unit (IU), load/store unit (LSU) and memory management unit (MMU). These five internal sub-modules are easily identifiable through the event names. Table 2 shows an example selection of native `PUnit` events provided by the PAPI utility `papi_native_avail`.

In addition to native events, a user can select predefined events (Presets) for the `PUnit` component on BG/Q. Out of 108 possible predefined events, there are currently 43 events available of which 15 are derived events made up of more than one native event.

Overflow: Only the local UPC module, L2 and I/O UPC hardware support performance monitor interrupts when a programmed counter overflows [1]. For that reason, PAPI offers overflow support for only the `PUnit`, `L2Unit`, and `IOUnit` components.

Fast versus Slow Overflow: `Punit` counters freeze on overflow until the overflow handling is complete. However, the `L2Unit` and `IOUnit` counters do not freeze on overflow. The L2 and I/O counts will be stopped when the interrupt is handled. The signal handler restarts L2 and I/O counting when done [1].

`PUnit` counters can detect a counter overflow and raise an interrupt within approx. 4 cycles of the overflowing event. However, according to the BGPM

Table 2. Small selection of PUnit events available on BG/Q

PUnit Event	Description
PEVT_AXU_INSTR_COMMIT	A valid AXU (non-load/store) instruction is in EX6, past the last flush point. - AXU uCode sub-operations are also counted by PEVT_XU_COMMIT instead.
PEVT_IU_IL1_MISS	A thread is waiting for a reload from the L2. - Not when CI=1. - Not when thread held off for a reload that another thread is waiting for. - Still counts even if flush has occurred.
PEVT_IU_IL1_MISS_CYC	Number of cycles a thread is waiting for a reload from the L2. - Not when CI=1. - Not when thread held off for a reload that another thread is waiting for. - Still counts even if flush has occurred.
PEVT_IU_IL1_RELOADS_DROPPED	Number of times a reload from the L2 is dropped, per thread - Not when CI=1 - Does not count when not loading cache due to a back invalidate to that address
PEVT_XU_BR_COMMIT_CORE	Number of Branches committed
PEVT_LSU_COMMIT_LD_MISSES	Number of completed load commands that missed the L1 Data Cache. - Microcoded instructions may be counted more than once. - Does not count dcbt[st][ls][ep]. - Include larx. - Does not includes cache-inhibited loads
PEVT_MMU_TLB_HIT_DIRECT_IERAT	TLB hit direct entry (instruction, ind=0 entry hit for fetch)
PEVT_MMU_TLB_MISS_DIRECT_IERAT	TLB miss direct entry (instruction, ind=0 entry missed for fetch)
...	...

documentation it takes up to approx. 800 cycles before the readable counter value is updated. This latency does not affect the overflow detection, and so we refer to a `PUnit` overflow as a "Fast Overflow".

The `IOUnit` and `L2Unit` take up to 800 processor cycles to accumulate an event and detect an overflow. Hence, we refer to this as a "Slow Overflow", and the program counters may alter up to 800 cycles or more after the event. This delay is due to the distributed nature of the performance counters. The counters are spread throughout the chip in multiple performance units. The hardware design consolidates the counters into one memory space continually, however it takes 800 cycles to visit all of the distributed units, hence the delay. The `IO` and `L2Units` are not thread specific, so there is no basis to stop counting for a single thread on overflow. However, the `PUnit` counters can be threaded, and the hardware has the ability to arm the distributed counts and freeze on overflow.

Multiplexing: PAPI supports multiplexing for the BG/Q platform. BGPM does not directly implement multiplexing of event sets. However, it does indirectly support multiplexing by supporting a multiplexed event set type. A multiplexed event set type will maintain sets of events which can be counted simultaneously, while pushing conflicting events to other internal sets [1].

3.2 L2 Unit Component

The shared L2 cache on the BG/Q system is split into 16 separate slices. Each of the 16 slices has a L2 UPC module that provides 16 counters with fixed events that can be gathered separately or aggregated into 16 counters (depending on the events chosen). Those 16 counters are node-wide, and cannot be isolated to a single core or thread. As mentioned earlier, every BG/Q processor has two DDR3 memory controllers, each interfacing with eight slices of the L2 cache to

handle their cache misses (one controller for each half of the 16 compute cores on the chip) [1,10]. The counting hardware can either keep the counts from each slice separate, or combine the counts from each slice into single values (which is the default). The combined counts are significantly important if a user wants to sample on overflows. Actually, the separate slice counts are not particularly interesting except for perhaps investigating cache imbalances because consecutive memory lines are mapped to separate slices. The node-wide "combined" or "sliced" operation is selected by creating an event set from the "combined" (default), or "sliced" group of events. Hence a user cannot assign events from both groups. Currently, there are 32 `L2Unit` events (16 events for the "combined" and "sliced" case, respectively) available on the BG/Q architecture.

Overflow: If `L2Unit` event overflow is desired, the overflow signal is "slow" (see the end of Section 3.1 for details that describe the difference between fast and slow overflow). As mentioned before, PAPI supports overflow for `PUnit` events as well as `L2Unit` and `IOUnit` events.

3.3 I/O Unit Component

The Message, PCIe, and DevBus modules - which are collectively referred to as I/O modules - together provide 43 counters. These counters are node-wide and cannot be isolated to any particular core or thread [1]. Note, the PCIe module is only enabled on the I/O nodes but disabled on the compute nodes. The counters for this specific I/O sub-module exist, however, there is currently no BGPM support for the I/O nodes. Currently, there are 44 `IOUnit` events available on the BG/Q architecture. The two I/O sub-modules - Message, and DevBus - are transparently identifiable from the `IOUnit` event names.

Overflow: If `IOUnit` event overflow is desired, the overflow signal is "slow" (see the end of Section 3.1 for details that describe the difference between fast and slow overflow).

3.4 Network Unit Component

The 5D-Torus network provides a local UPC network module with 66 counters - each of the 11 links has six 64-bit-counters. As of right now, a PAPI user cannot select which network link to attach to. Currently, all 11 network links are attached and this is hard-coded in the PAPI `NWUnit` component. We are considering options for supporting the other enumerations for network links as well. We can easily change to attaching the ten torus links only and leave the I/O link out. As for measuring the performance of an application's communication, both of the two configurations will work without limitations because the I/O links are not used for sending packets to another compute node. However, if users want to evaluate the I/O performance of an application, then they can do this via the current network component as well. This would not be the case when we use the torus links only. Currently, there are 31 `NWUnit` events available on the BG/Q architecture.

3.5 CNK Unit Component

By default a custom lightweight operating system called Compute Node Kernel (CNK) is loaded on the compute nodes while I/O nodes run Linux OS [4]. The CNK OS is the only kernel that runs on all the 16 compute cores. In general, on Linux kernels the "/proc" file system is the usual access method for kernel counts. Since CNK does not have a "/proc" filesystem, PAPI uses BGPM's "virtual unit" that has software counters collected by the kernel. The kernel counter values are read via a system call that requests the data from the lightweight compute node kernel. Also, there is a read operation to get the raw value since the system has been booted. Currently, there are 29 `CNKUnit` events available on the BG/Q architecture. Table 3 provides a small selection of `CNKUnit` events. The CNK functionality is heavily used by tools that support sample-based profiling like e.g. HPCToolkit [2]. Hence, with the `CNKUnit` Component, this is much easier handled on BG/Q than it was on BG/P.

Table 3. Small selection of CNKUnit events, available on the BG/Q architecture

`CNKUnit` Event	Description
`PEVT_CNKNODE_MUINT`	Number of Message Unit non-fatal interrupts
`PEVT_CNKNODE_NDINT`	Number of Network Device non-fatal interrupts
`PEVT_CNKHWT_SYSCALL`	System Calls
`PEVT_CNKHWT_FIT`	Fixed Interval Timer Interrupts
`PEVT_CNKHWT_WATCHDOG`	Watchdog Timer Interrupts
`PEVT_CNKHWT_PERFMON`	Performance Monitor interrupts
`PEVT_CNKHWT_PROGRAM`	Program Interrupts
`PEVT_CNKHWT_FPU`	FPU Unavailable Interrupts
...	...

4 Case Study: Parallel 3D-FFT on BG/Q

As a case study, we implemented a parallel 3D-FFT kernel and want to explore how well the communication performs on the BG/Q network. The Fast Fourier Transforms (FFT) of multidimensional data are of particular importance in a number of different scientific applications but they are often among the most computationally expensive components. Parallel multidimensional FFTs are communication intensive, which is why they often prevent the application from scaling to a very large number of processors. A fundamental challenge of such numerical algorithms is a design and implementation that efficiently uses thousands of nodes. One important characteristics of BG/Q is the organization of the compute nodes in a 5D-Torus network. We will explore that in order to maintain application performance and scaling, the correct mapping of MPI tasks onto the torus network is a critical factor.

4.1 Definition of the Fourier Transformation

We start the discussion with the definition and the conventions used for the Fourier Transformation (FT) in this paper. Consider $A_{x,y,z}$ as a three-dimensional array of $L \times M \times N$ complex numbers with:

$$\begin{aligned} A_{x,y,z} \in \mathbb{C}\ x \in \mathbb{Z}\ \forall x,\ 0 \leq x < L \\ y \in \mathbb{Z}\ \forall y,\ 0 \leq y < M \\ z \in \mathbb{Z}\ \forall z,\ 0 \leq z < N \end{aligned}$$

The Fourier transformed array $\tilde{A}_{u,v,w}$ is computed using the following formula:

$$\tilde{A}_{u,v,w} := \underbrace{\sum_{x=0}^{L-1} \underbrace{\sum_{y=0}^{M-1} \underbrace{\sum_{z=0}^{N-1} A_{x,y,z} \exp(-2\pi \mathrm{i} \frac{wz}{N})}_{\text{1st 1D FT along } z}}_{\text{2nd 1D FT along } y} \exp(-2\pi \mathrm{i} \frac{vy}{M})}_{\text{3rd 1D FT along } x} \exp(-2\pi \mathrm{i} \frac{ux}{L}) \quad (1)$$

As shown by the under-braces, this computation can be performed in three single stages. This is crucial for understanding the parallelization in the next subsection. The first stage is the one-dimensional FT along the z dimension for all (x, y) pairs. The second stage is a FT along the y dimension for all (x, w) pairs, and the final stage is along the x dimension for all (v, w) pairs.

4.2 Parallelization

Many previous parallel 3D-FFT implementations have used a one-dimensional virtual processor grid - i.e. only one dimension is distributed among the processors and the remaining dimensions are kept locally. This has the advantage that one all-to-all communication is sufficient. However, for problem sizes of about one hundred points or more per dimension, this approach cannot offer scalability to several hundred or thousand processors as required for modern HPC architectures. For this reason the developers of the IBMs Blue Matter application have been promoting the use of a two-dimensional virtual processor grid for FFTs in three dimensions [5]. This requires two all-to-all type communications, as shown in Figure 1, which illustrates the parallelization of the 3D-FFT using a two-dimensional decomposition of the data array A of size $L \times M \times N$. The compute tasks have been organized in a two-dimensional virtual processor grid with P_c columns and P_r rows using the MPI Cartesian grid topology construct. Each individual physical processor holds an $L/P_r \times M/P_c \times N$ sized section of A in its local memory. The entire 3D-FFT is performed in five steps as follows:

1. Each processor performs $L/P_r \times M/P_c$ one-dimensional FFTs of size N

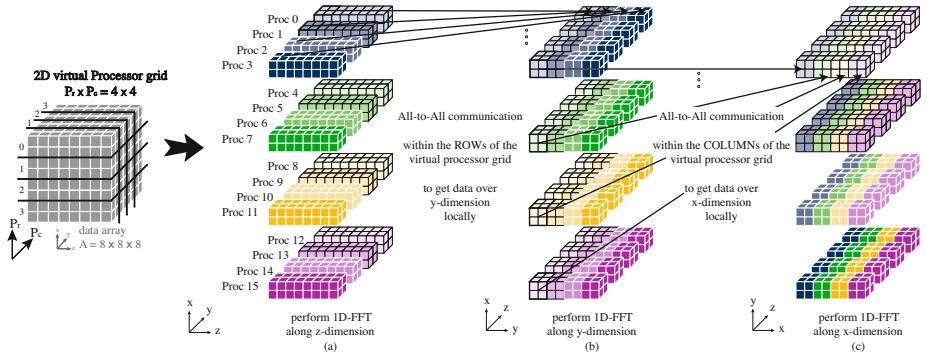


Fig. 1. Computational steps of the 3D-FFT implementation using 2D-decomposition

2. An all-to-all communication is performed within each of the rows - marked in the four main colors - of the virtual processor grid to redistribute the data. At the end of the step, each processor holds an $L/P_r \times M \times N/P_c$ sized section of A. These are P_r independent all-to-all communications.
3. Each processor performs $L/P_r \times N/P_c$ one-dimensional FFTs of size M.
4. A second set of P_c independent all-to-all communications is performed, this time within the columns of the virtual processor grid. At the end of this step, each processor holds a $L \times M/P_c \times N/P_r$ size section of A.
5. Each processor performs $M/P_c \times N/P_r$ one-dimensional FFTs of size L

For more information on the parallelization, the reader is referred to [5,8].

4.3 Communication Network Topology

As mentioned before, the network topology for BG/Q is a 5D-Torus. Every node is connected to its ten neighbor nodes through bidirectional links in the $\pm A$, $\pm B$, $\pm C$, $\pm D$, and $\pm E$ directions. This appears to be a significant change compared to BG/Q predecessors, both of which feature a 3D-Torus. Here every node is connected to its six neighbor nodes through bidirectional links in the $\pm A$, $\pm B$, and $\pm C$ directions. To maintain application performance, an efficient mapping of MPI tasks onto the torus network is a critical factor.

The default mapping is to place MPI ranks on the BG/Q system in $ABCDET$ order where the rightmost letter increments first, and where $< A, B, C, D, E >$ are the five torus coordinates and $< T >$ ranges from 0 to $N - 1$, with N being the number of ranks per node [7]. If the job uses the default mapping and specifies one process per node, the following assignment results:
MPI rank 0 is assigned to coordinates $< 0, 0, 0, 0, 0, 0 >$
MPI rank 1 is assigned to coordinates $< 0, 0, 0, 0, 1, 0 >$
MPI rank 2 is assigned to coordinates $< 0, 0, 0, 1, 0, 0 >$
The mapping continues like this, first incrementing the E coordinate, then the D coordinate, and so on, until all the processes are mapped. The user can choose

a different mapping by specifying a different permutation of $ABCDET$ or by creating a customized map file.

4.4 Network Performance Counter Analysis

We ran our 3D-FFT kernel on a 512 node partition, utilizing half a rack on the BG/Q system at Argonne National Laboratory, using all 16 compute cores per node for each run. Table 4 summarizes the number of nodes that are available in each of the five dimensions. Also the torus connectivity is shown for each dimension, while 1 indicates torus connectivity for a particular dimension, 0 indicates none.

Table 4. Torus connectivity and number of nodes in each of the five dimensions for a 512 node partition

	A	B	C	D	E	T
nodes	4	4	4	4	2	16
torus	1	1	1	1	1	-

For the 512 node partition, we have a total of 8,192 MPI tasks, and for the virtual two-dimensional process grid, we chose 16×512, meaning that each subgroup has 16 MPI tasks and we have 512 of those subgroups. Since we want to know how well the communication performs on the 5D-Torus network, we use the new PAPI network component to sample various network related events. The number of packets sent from each node is shown in Figure 2 for a problem size of 512^3. This includes packets that originate as well as pass through the current node. It is important to note, these are the numbers for only the all-to-all communication within each subgroup (only first all-to-all), not including the second all-to-all communication between the subgroups.

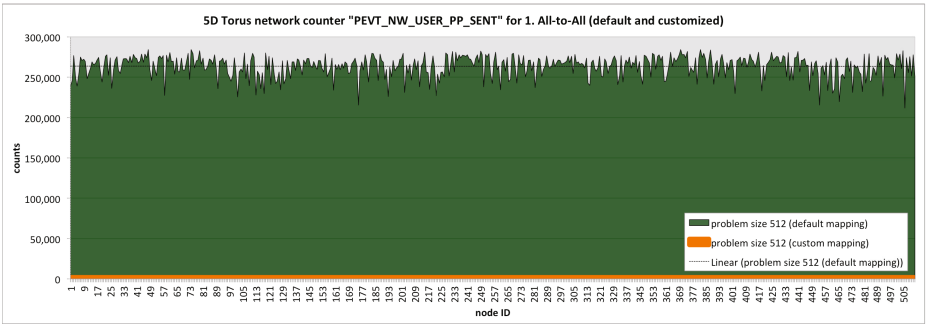


Fig. 2. Network counter data collected with PAPI. This event counts the number of packets originating and passing through the current node for the first all-to-all communication.

The PAPI network counter data greatly helped to evaluate the communication of the parallel 3D-FFT kernel as it clearly shows an unnecessary large number of packets that cross a node if the default MPI task mapping is used. The collected network data for a medium problem size of 512^3 counts approx. 270,000 packets which originate from and pass through each single node. Note, the count variation from node to node is due to the routing algorithm which may pass more packets through some nodes but not others based on how congested alternative routes are in the network. On the whole, we consider this number of packets for such a fairly small problem size extremely high, which is also the cause of the discovered network congestions. Without the network counter data, a user may merely pursue with speculations about various reasons of the poor performance. However, the data allows a much more concentrated analysis that assists with taking more settled instead of speculative actions.

In order to resolve this type of network congestion, we examined a variety of customized MPI task mappings, which heavily depend on the chosen 2D processor grid of the parallel 3D-FFT implementation. For each experiment, the network counter data distinctly indicated either a stationary or improved congestion of the network. The analysis shows that the reason for the high numbers is the placement of MPI tasks onto the network using the default mapping which results in a lot of inter-node communications. It appears that even when using a total of five dimensions for a torus network, the default mapping can still result in severe performance degradations due to congestions. This stresses all the more how critical the correct mapping of MPI tasks onto the torus network is, even when we utilize a five-dimensional torus. The default mapping places each task of a subgroup on a different node, as can be seen from Table 5(a) that summarizes the default MPI task mapping on the 5D-Torus for one communicator.

Table 5. MPI task mappings on the 5D-Torus for the 512 node partition run, using a 2D virtual processor grid 16×512. For simplicity, each table presents the mapping of only one out of a total of 512 communicators.

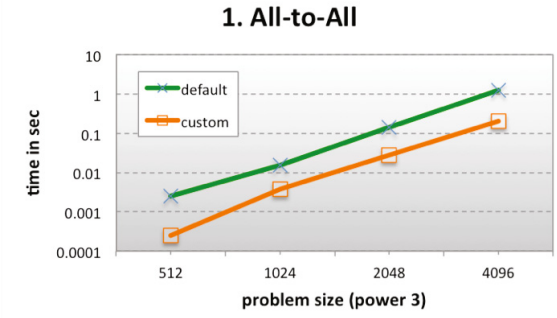
(a) Default MPI-task mapping

rank	A	B	C	D	E	T
0	0	0	0	0	0	0
512	0	1	0	0	0	0
1,024	0	2	0	0	0	0
1,536	0	3	0	0	0	0
2,048	1	0	0	0	0	0
2,560	1	1	0	0	0	0
3,072	1	2	0	0	0	0
3,584	1	3	0	0	0	0
4,096	2	0	0	0	0	0
4,608	2	1	0	0	0	0
5,120	2	2	0	0	0	0
5,632	2	3	0	0	0	0
6,144	3	0	0	0	0	0
6,656	3	1	0	0	0	0
7,168	3	2	0	0	0	0
7,680	3	3	0	0	0	0

(b) Customized MPI-task mapping

rank	A	B	C	D	E	T
0	0	0	0	0	0	0
512	0	0	0	0	0	1
1,024	0	0	0	0	0	2
1,536	0	0	0	0	0	3
2,048	0	0	0	0	0	4
2,560	0	0	0	0	0	5
3,072	0	0	0	0	0	6
3,584	0	0	0	0	0	7
4,096	0	0	0	0	0	8
4,608	0	0	0	0	0	9
5,120	0	0	0	0	0	10
5,632	0	0	0	0	0	11
6,144	0	0	0	0	0	12
6,656	0	0	0	0	0	13
7,168	0	0	0	0	0	14
7,680	0	0	0	0	0	15

The above mentioned network counter data analysis for various customized mappings promotes a mapping that places all the tasks from one subgroup onto the same node which significantly reduced the amount of communication. Table 5(b) presents the optimum customized MPI task mapping on the 5D-Torus for the same communicator as was used in Table 5(a). Since each subgroup has 16 MPI tasks, and since we have 16 compute cores per node, we can place one entire subgroup on each node. By doing so, all the high numbers reported for the network counter were reduced to zeroes, resulting in no inter-node communication at all. The results presented in Figure 3 show that the customized mapping gives us a performance improvement of up to a factor of approx. 10 (depending on the problem size) for the first all-to-all. Note, there was no degradation in performance for the second all-to-all with the customized mapping. For the entire 3D-FFT kernel - which consists of three 1D-FFT computations and two all-to-all communications - we see an improvement ranging from 10 to 18% for various mid-size problems.



Comm	10.2x	4x	5.2x	6x
3D FFT	18%	16%	15%	10%

Fig. 3. Performance comparison for the first all-to-all communication using default and customized mapping. The table at the bottom presents the performance improvement of the customized mapping for each problem size and for the communication as well as the entire 3D-FFT kernel respectively.

5 Conclusion

Performance analysis tools for parallel applications running on large scale computing systems typically rely on hardware performance counters to gather performance relevant data from the system. In order to allow the HPC community to collect hardware performance counter data on IBM's latest Blue Gene system BG/Q, PAPI has been extended with five new components.

The PAPI customization for BG/Q accesses the BGPM interface under the covers, allowing users and third-party programs to monitor and sample hardware

performance counters in a traditional way using the default PAPI interface. The recently added PAPI components allow hardware performance counter monitoring not only for the processing units but also for the 5D-Torus network, the I/O system, and the Compute Node Kernel.

As a case study for using hardware performance monitoring beyond the CPU we implemented a parallel 3D-FFT kernel and instrumented it with PAPI for communication evaluation on the BG/Q system at Argonne National Laboratory. The collected network counter data considerably helped evaluating the communication for the 5D-torus partition as well as made us look deeper into where tasks are located by default on the 5D network, and how to improve the task location based on the algorithm's features. With the default mapping of MPI tasks onto the torus network, the network counters detected a large amount of redundant inter-node communications. By employing a custom mapping, we were able to eliminate the unnecessary communication and achieve more than a ten-fold bettering for the all-to-all communication which consequently leads to up to 18% performance improvement for the entire 3D-FFT kernel on 8,192 cores.

Acknowledgments. This material is based upon work supported by the U.S. Department of Energy Office of Science under contract DE-FC02-06ER25761. Access to the early access BG/Q system at Argonne National Laboratory was provided through the ALCF Early Science Program.

References

1. BGPM Documentation (2012)
2. Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., Tallent, N.R.: HPCToolkit: Tools for performance analysis of optimized parallel programs. Concurrency and Computation: Practice and Experience 22(6), 685–701 (2010)
3. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A portable programming interface for performance evaluation on modern processors. International Journal of High Performance Computing Applications 14(3), 189–204 (2000)
4. Budnik, T., Knudson, B., Megerian, M., Miller, S., Mundy, M., Stockdell, W.: Blue Gene/Q Resource Management Architecture (2010)
5. Eleftheriou, M., Moreira, J.E., Fitch, B.G., Germain, R.S.: A Volumetric FFT for BlueGene/L. In: Pinkston, T.M., Prasanna, V.K. (eds.) HiPC 2003. LNCS (LNAI), vol. 2913, pp. 194–203. Springer, Heidelberg (2003)
6. Feldman, M.: IBM Specs Out Blue Gene/Q Chip (2011), `http://www.hpcwire.com/hpcwire/2011-08-22/ibm_specs_out_blue_gene_q_chip.html`
7. Gilge, M.: IBM system Blue Gene solution: Blue Gene/Q application development. IBM Redbook Draft SG24-7948-00 (2012)
8. Jagode, H.: Fourier Transforms for the BlueGene/L Communication Network. Master's thesis, EPCC, The University of Edinburgh (2006), `http://www.epcc.ed.ac.uk/msc/dissertations/2005-2006/`
9. Morgan, T.P.: IBM Blue Gene/Q details (2011), `http://www.multicoreinfo.com/2011/02/bluegeneq`
10. Morgan, T.P.: IBM's Blue Gene/Q super chip grows 18th core (2011), `http://www.theregister.co.uk/2011/08/22/ibm_bluegene_q_chip`

Maximizing Application Performance in a Multi-core, NUMA-Aware Compute Cluster by Multi-level Tuning

Gilad Shainer[1], Pak Lui[1], Martin Hilgeman[2], Jeffrey Layton[2], Cydney Stevens[2], Walker Stemple[2], Scot Schultz[3], Guy Ludden[3], Joshua Mora[3], and Georg Kresse[4]

[1] Mellanox Technologies, California, United States
[2] Dell Inc., Texas, United States
[3] Advanced Micro Devices, California, United States
[4] University of Vienna, Vienna, Austria

Abstract. Achieving good application performance on a modern compute cluster of multi-core, multi-socket, NUMA-aware systems can be challenging. In this paper, we use VASP, a popular ab-initio quantum-mechanical MD simulation software, to investigate the various levels of the software, hardware, and network tuning that boosts performance on a Dell PowerEdge R815 HPC cluster with AMD "Interlagos" and "Abu-Dhabi" processors. We implement code changes with the free software stack that supports FMA and AVX CPU instructions on the Bulldozer/Piledriver architecture. We analyze the MPI communications by profiling, compare the scalability performance of different interconnects, and discuss various MPI tuning parameters show effects of the advanced features that are crucial to the scalability performance of InfiniBand, including MXM and SRQ, which optimize the network resources for MPI communications. We investigate the importance of the MPI process placement, and introduce a process allocation tool that facilitates the affinity grouping on a multicore architecture.

Keywords: Performance, Multi-Level Tuning, VASP, AMD Bulldozer, InfiniBand, MPI.

1 Introduction

High-performance computing (HPC) simulations are typically carried out on high-performance computing clusters, as they require an effective compute resource that can handle complex and parallel simulations. HPC clusters are scalable performance compute solutions based on industry standard hardware connected by a private system high speed network. The main benefits of clusters are affordability, flexibility, availability, high-performance, and scalability. A cluster uses the aggregated power of compute server nodes to form a high-performance solution for parallel applications. When more compute power is needed, it can be achieved simply by adding more server nodes to the cluster.

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 226–238, 2013.

Fig. 1. An Example of HPC System Architecture with System Nodes of NUMA Architecture - Demonstrating that the Difference between Intra-Node Latency and Inter-Node Latency can Affect Application Performance

The architecture of HPC clusters (that is, multi-core, multi-processor based HPC servers with high-speed interconnects) has a great influence on the overall application performance and productivity. The cluster interconnect is very critical for delivering efficiency and scalability for the applications, as it needs to handle the networking requirements of each CPU core without imposing additional networking overhead.

In a multi-core, multi-socket HPC server-based cluster, the driving factors of performance and scalability for HPC simulations have shifted from the frequency and cache size per core to the memory and interconnect latency and throughput.

In a clustered environment, multiple servers are connected via the cluster interconnect. This architecture typically imposes higher latency for communication between compute cores located on different nodes or servers. A high-latency interconnect can dramatically reduce the efficiency of the compute cluster and will make the solution ineffective for HPC simulations. This concern drove the notion that gathering as many compute elements onto a single board (SMP) is a better solution than using the cluster interconnect for critical data transfers between the application's processes.

With the increased availability of low-latency interconnect solutions such as InfiniBand [1], the penalty of inter-node latency compared to the intra-node latency has decreased, making the architecture the leading solution for HPC applications. The InfiniBand Architecture (IBA) [1] is an industry-standard fabric designed to provide high bandwidth/low-latency computing, scalability to ten-thousand nodes and multiple CPU cores per server platform, and efficient utilization of compute processing resources. InfiniBand adapters and switches deliver 56Gb/s bandwidth today and are expected to deliver 100Gb/s by 2014. This high-performance bandwidth is matched with ultra-low application latency of nearly 1 μsec to enable efficient scale-out of compute systems.

For intra-node and inter-node communications, PCI-Express (PCIe) and AMD HyperTransport (HT) have become the technologies that are being used to connect between CPUs, as well as between CPU and memory. Fig. 1 illustrates the overall system architecture of NUMA nodes interconnected over a network to form a cluster. The latency between the CPU cores within the NUMA node is the intra-node latency, and the latency across the nodes is the inter-node latency.

Even with the low-latency interconnect solutions that eliminate most of the intra-node and inter-node latencies to make parallel computing effective, there are still steps required to make the multi-core, NUMA-aware compute cluster run efficiently.

This paper will describe how these methods of multi-level tuning are conducted in order to maximize the application performance.

2 Application Example – VASP

We have chosen to use the VASP (Vienna Ab-initio Simulation Package) [6-9] to demonstrate the performance benefits of various optimization methods. VASP performs ab-initio quantum-mechanical molecular dynamics (MD) using pseudopotentials and a plane wave basis set. The VASP code is written in FORTRAN 90 with MPI support and is designed to be an efficient massively parallel computing code that takes advantage of advanced high-performance computing systems. The approach used in VASP is based on techniques that employ a finite-temperature local-density approximation and an exact evaluation of the instantaneous electronic ground state at each MD-step using efficient matrix diagonalization schemes and an efficient Pulay mixing.

The analysis described in this paper was conducted as part of the HPC Advisory Council [2] research activities, using the HPC Advisory Council HPC Compute Center. The cluster configuration was a Dell™ PowerEdge™ R815 11-node cluster. Each R815 system has 4-socket 16-core AMD Opteron 6276 "Interlagos" processors @ 2.30 GHz CPUs and 128GB memory (DDR3, 1333 MHz) per node. Mellanox® ConnectX®-3 QDR InfiniBand adapters and QDR InfiniBand switches were used as the cluster networking. The operating system was RHEL 6 Update 2 with OFED 1.5.3 InfiniBand SW stack. MPI tested were Intel MPI 4 Update 3, MVAPICH2 1.8.1, and Open MPI 1.6.3 with dell_affinity 0.88. VASP version 5.2.7 used the Pure Hydrogen MD simulation case with ten iconic steps, 60 electronic steps, and 264 bands.

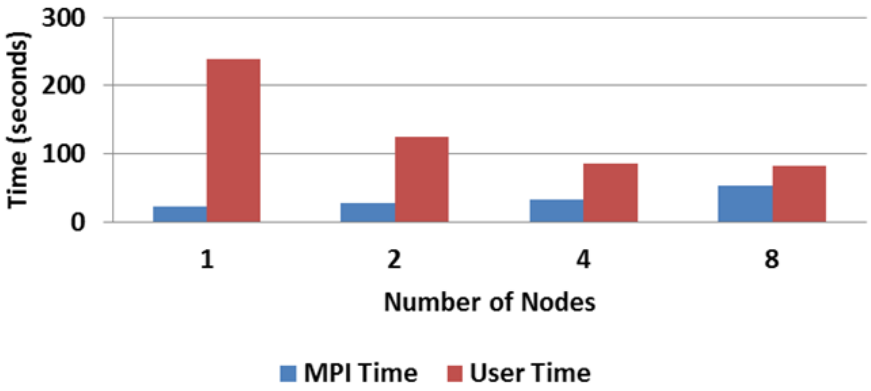


Fig. 2. VASP Profiling Comparing MPI Time versus User Time (MPI Time Versus Compute Time), from One Server Node to 8 Server Nodes

3 Selecting the Best Method for Tuning

Before tuning for the best application performance, one should understand how time is spent during the application runtime. One way to learn more about how the runtime is spent is to use performance analyzing tools or profiling tools. By tackling the biggest time consumers determined by a profiling tool on the application, it is then easy to identify the proper direction to tune. It yields the most benefit by tuning those components for which the application spends the most time, resulting in a large runtime reduction for the application.

Fig. 2 describes the overall compute time and the MPI time of the Pure Hydrogen benchmark. As more nodes are used, the overall compute time is reduced, since more cores are used for the parallel computations. It also means that by optimizing for the CPU performance at a small node count, it yields a greater time difference. On the other hand, the MPI time (that is, communication time) stays nearly constant for the higher node, higher CPU core counts. This can be explained by understanding how VASP works, and also by the MPI profiling, which will be explained later.

4 Tuning by Selecting the Right CPU Cores

To understand the AMD Bulldozer module concept for making maximal use of available resources, we must first analyze the CPU architecture of the AMD Interlagos CPU.

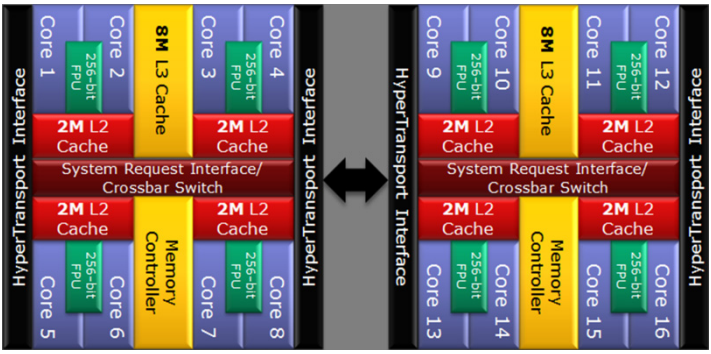


Fig. 3. AMD Opteron™ 6200 Series Processor ("Interlagos")

As depicted in Fig. 3, the Interlagos processor is composed of two die. Each die is composed of eight "Bulldozer" modules. The two Bulldozer cores inside an Interlagos processor contain 16 CPU cores. Two of the neighboring cores share a 256-bit Floating Point Unit (FPU) within a module. The module divisions are transparent to shared hardware, operating systems, or applications.

When a lightly threaded workload sends half the "Bulldozer" modules into C6 sleep state, but also requests max performance, AMD Turbo Core technology can increase clock speeds by up to 1 GHz+ across half the cores.

By correctly selecting the placement of the MPI process to use, the application is able to take advantage of this performance boost from the Turbo Core technology.

By running only one core active inside a Bulldozer "core-pair" module unit, the core can run at a higher speed. The performance boost by using 32 processes per node from having one active core in a module yields about 42% better performance than when all 64 processes per node are used in a 256-process job. Likewise, by strategically placing one active core in a module, the yield is about 30% faster than using the 32 cores located on the first 2 CPU sockets.

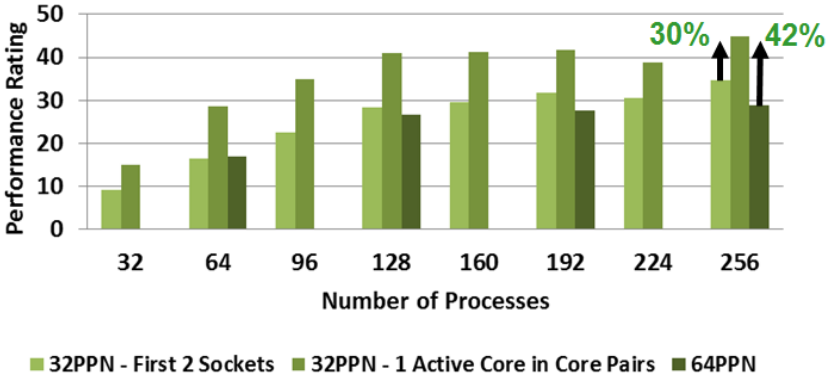


Fig. 4. Example for Processes per Node (PPN) Placement

5 Tuning by MPI Process Placement Based on Locality

The process placement program is written in C99 with approximately 2,000 lines of code. It works on all major distributions, such as RHEL 5, RHEL 6, SLES11 SP2, and other open variants. It supports all major MPI libraries, such as MVAPICH, MVAPICH2, Open MPI, Platform MPI/HP MPI, and Intel MPI. The executable has been tested with over 5,000 core runs. It supports hybrid MPI/OpenMP runs as well.

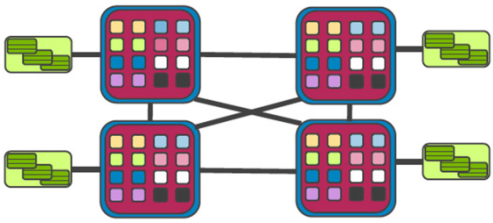


Fig. 5. Naïve Placement of MPI Processes to CPU Cores

Fig. 5 shows an example of the naïve placement of MPI processes to available CPU cores. By placing MPI processes on the CPU cores in this fashion, the communication between the processes causes MPI internode communication to occur between the inter-node bus (over the AMD HyperTransport bus), which results in a bottleneck in MPI communications and eventually causes a degradation in runtime performance.

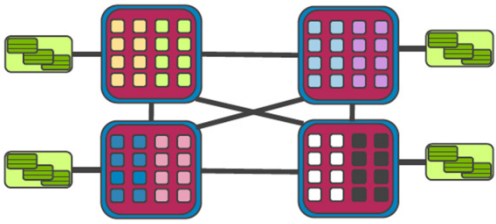


Fig. 6. Optimal Placement Using dell_affinity.exe

Fig. 6 shows the improvement made by utilizing the "dell_affinity.exe" tool to place the processes that are in close proximity by ordering the adjacent MPI processes closely together on a NUMA node, thereby reducing the time it takes to communicate between the MPI processes and resulting in higher runtime performance.

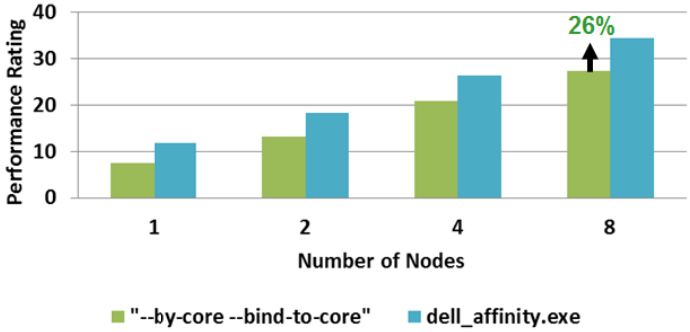


Fig. 7. Performance and Importance of Optimal Process Placement

Fig. 7 demonstrates the runtime productivity gain by using the "dell_affinity.exe" tool to enable processor binding versus the "--by-core --bind-to-core" flags available in Open MPI. The default process selection in Open MPI is the "--by-core" algorithm, which puts sequential MPI processes on adjacent CPU cores. The "--bind-to-core" specifies to Open MPI to enable processor binding.

The problem is that the default processor binding selection in Open MPI is retrieved through the Portable Hardware Locality (hwloc). In some instances, the hwloc utility may not correctly identify the CPU cores when the BIOS does not enumerate the processors in the sequential or adjacent order. The "dell_affinity.exe" tool is developed and tested on the Dell PowerEdge series of server platforms, and works around the issue by providing itself as a runtime wrapper program to identify and bind

to the correct CPU cores, thereby allowing the most optimal process placement to CPU cores.

6 Tuning of Software Stack

The third element in tuning the AMD Opteron architecture is to tune the compilation of the software stack and to link with the software library that takes advantage of the AMD Opteron architecture.

The main target for VASP simulations are clusters based on the x86 architecture, preferably with a fast interconnect. For these systems, the commercially available Intel compilers and Intel Math Kernel library are an obvious choice for obtaining good performance by using SIMD instructions on modern processors. By default, VASP contains build instructions for these systems. For AMD based systems, this is not an obvious choice. For the first time, support has been implemented for the free software stack (of Open64 compilers, ACML, and MVAPICH2) that can deliver comparable or better application performance for the AMD Bulldozer architecture with the support of the FMA instructions and AVX CPU extensions available on the Open64 compiler. While VASP is written in FORTRAN 90 and uses modern features like module procedures and interface blocks, some changes are required to these interface blocks, and lower optimization levels are required in order for some routines to be compiled with the Open64 compiler. The code changes for the aforementioned interface blocks of certain modules to support Open64 are then ported back to the original VASP code base. VASP users can access and acquire the code changes from the developers at the University of Vienna upon request and from the HPC Advisory Council [10]. The free software stack consists of Open64 Compiler 4.5.2, ACML 5.2.0, and MVAPICH2 1.8.1. The commercial software stack consists of Intel Compiler 13.0, MKL 11.0, and Intel MPI 4 Update 3.

Some of the VASP code is modified so that it can be compiled by the Open64 compilers. The advantage of using Open64 compilers is that they contain CPU optimization extensions suitable for the Bulldozer and Piledriver architectures available in the AMD Opteron 6200 “Interlagos” and Opteron 6300 “Abu Dhabi” series. The AVX and FMA instruction sets and the compiler flags “-march=bdver1 -mavx -mfma”, were used to enable such processor extension on the Bulldozer architecture. Likewise, for the AMD Opteron 6300 “Abu Dhabi” series, it supports using AVX and FMA instruction sets with the compiler flags “-march=bdver2 -mavx -mfma” to enable such processor extensions for the Piledriver architecture.

Fig. 8 shows the Pure Hydrogen dataset being used to compare between the commercial and open source stacks. The VASP performance illustrates that the open source MVAPICH2 MPI, Open64 compiler, and ACML libraries are able to achieve comparable performance in most cases. However, it is assumed that improved performance can be expected with additional compiler optimizations, some of which were tuned to enable the open64 port to be compiled. Besides the optimization, additional tuning can be handled by using the latest development tools (such as Open64 and ACML) that support AMD “Interlagos” and “Abu Dhabi” CPU architecture.

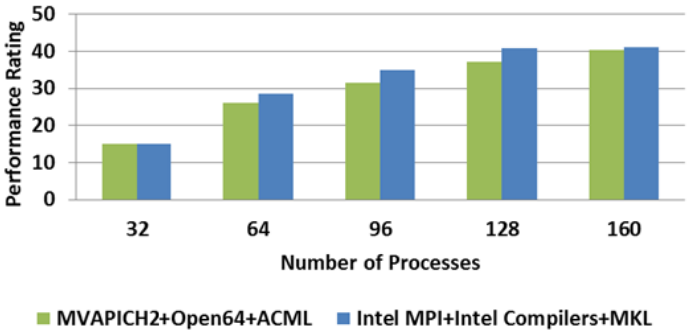


Fig. 8. Performance of Free Software Stack (MVAPICH2 MPI, Open64 Compilers, and ACML) versus the Intel Software Stack Using the Pure Hydrogen Problem

7 Profiling the MPI Traffic in VASP

To tune the network communication effectively to achieve the best runtime performance, we must first analyze the type of MPI traffic that occurs on the network fabric during the runtime of VASP.

To do so, we must profile the VASP usage of MPI collective communications using an MPI profiler that allows us to visualize the time it takes for the MPI calls to complete, and the volume of MPI calls made during runtime. This information makes the types of MPI communication that occur transparent, and might explain why there is a need for some of the network optimization we are about to introduce.

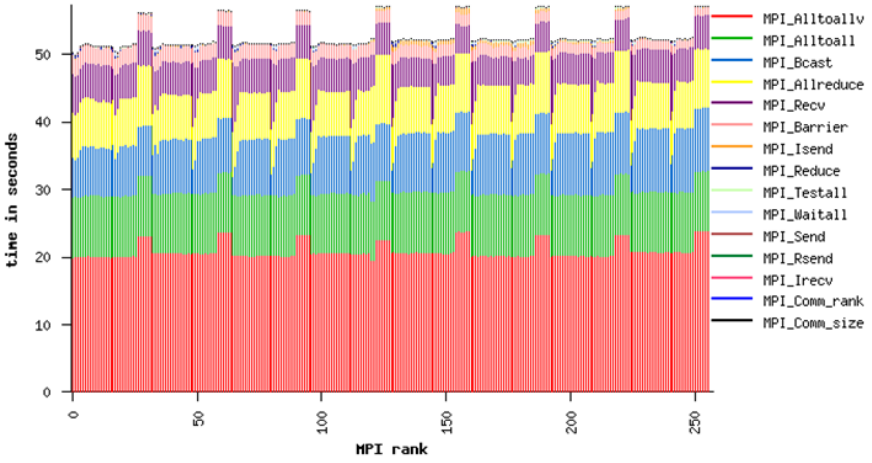


Fig. 9. Provides the Timing Information per MPI Call at 256 Processes

Fig. 9 demonstrates the large usage of the MPI collective communications in VASP. MPI_Alltoallv is the most used collective operation, responsible for 40% of the MPI time. This, of course, was expected, as VASP scatters and gathers the data

among the available cores using the MPI_Alltoallv collective communication. After MPI_Alltoallv, MPI_Alltoall (17%) and MPI_Reduce (15%) are the most time-consuming MPI calls. By using a low latency networking technology like InfiniBand, it would reduce the latency for this kind of collective communications. Fig. 9 also shows that uneven communication occurs for MPI messages, as more MPI collective communication (MPI_Alltoallv) takes place on certain MPI ranks than on others.

8 Scalability Performance Comparisons

We now compare the scalability performance of various high speed network interconnects, and we show the effects of the advanced features of the InfiniBand technology (such as RDMA) that would allow kernel bypass access of remote memory.

The performance measurements of various network interconnects are performed on systems with different network interconnect hardware to ensure the tests are completed in a consistent manner.

The performance comparison shows that QDR InfiniBand delivers the best performance for VASP among the various network interconnects being tested. QDR InfiniBand outperforms 40GbE by up to 186% on 8 nodes with 32 processes per node. Compared with 10GbE, QDR InfiniBand delivers over five times better performance on 8 nodes. Likewise, when comparing against 1GbE, QDR InfiniBand provides nearly nine times better performance.

The crucial observation here is that both latency and bandwidth are important for VASP. With the Ethernet network interconnects, there is a workload scalability limitation that the Ethernet network is unable to scale beyond two systems. The performance results can be explained by the large number of MPI processes per node responsible for sending and receiving messages per node. This creates a large amount of data that must be sent across the network. By inspecting the scalability across multiple nodes, we see that VASP requires a network interconnect such as QDR InfiniBand, which can provide both low latency and high bandwidth suitable for VASP.

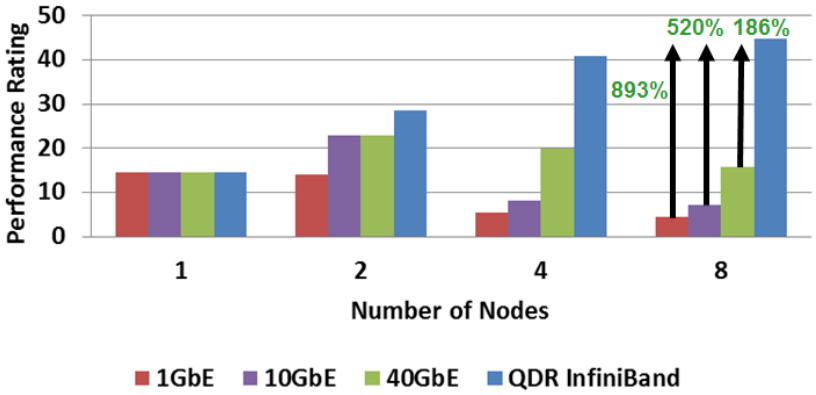


Fig. 10. Performance Differences of Various Network Interconnects

9 Tuning of MPI Communications Using SRQ

When operating in large clusters, there is a need to reduce the memory footprint and to keep it constant regardless of the number of processes. InfiniBand defines the concept of Shared Receive Queue (SRQ), such that receive resources can be shared among multiple endpoints.[4]

A scalability effect is seen when running VASP on 256 MPI processes with Open MPI. We observe an improvement of 24% by enabling SRQ using these MCA parameters in Open MPI (`--mca btl_openib_receive_queues S,9216,256,128,32:S,65536,256,128,32`)

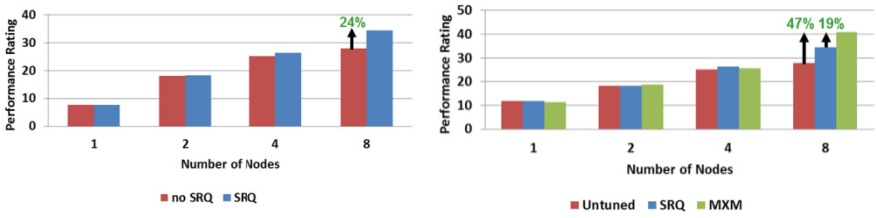


Fig. 11. Performance and Scalability Improvement Due to Shared Receive Queue (SRQ) (left) and Performance and Scalability Improvement Due to Reliable Messaging of MXM (right). Both Optimized for Mellanox HCA

10 Tuning of MPI Communications Using MXM

Mellanox has introduced a reliable messaging acceleration optimized for Mellanox HCA called MXM. It utilizes a hybrid transport mechanism that allows efficient memory registration, as well as receive-side tag matching when transporting messages that would allow MXM to deliver better performance for VASP at high CPU core counts.

MXM provides the extension for the network transport, memory management, matching operations, and more. In particular, for the support of the InfiniBand

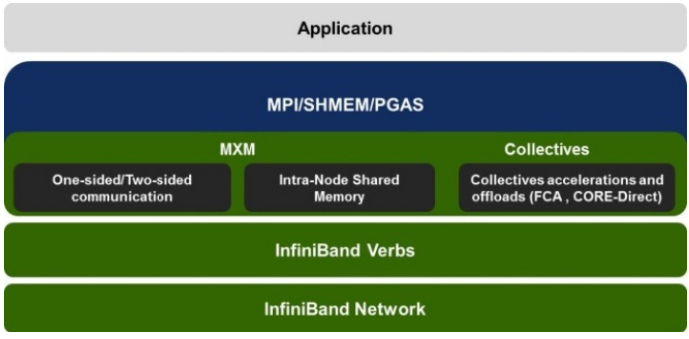


Fig. 12. Software Layer Architecture of the co-design implementation

network, MXM extends the management of the networking I/O channels into the communication libraries and offers the ability to optimize the implementation of the memory allocations and usage. MXM takes ownership of both the implementation of one-sided and two-sided communications.

The MXM interface includes the function calls for send/receive operations. The API provides the complete set of communication calls related to communication algorithms, and includes the interconnect management and usage of the available offloading and accelerations.

Fig 12. shows that MXM is implemented as part of the co-design architecture for exascale systems. The design of MXM was previously described and validated by using micro-benchmarks that demonstrate great reduction in latency in both MPI and SHMEM [12]. In this paper, we deploy MXM in Open MPI on VASP, which translates the latency reduction into application performance gain.

By comparing between un-tuned Open MPI, SRQ-enabled Open MPI, and MXM-enabled Open MPI, we have observed that the MXM run at 8 nodes delivers up to 47% higher job productivity than the un-tuned Open MPI run at 8 nodes. MXM also drives about 19% higher job productivity than the SRQ-enabled Open MPI at 8 nodes. Processor binding with "dell_affinity.exe" is used when comparing all three cases. The comparison is shown in Fig. 11.

The flags enabled for the MXM-enabled Open MPI run are as follows: "`--mca mtl mxm --mca btl_openib_free_list_num 8192 --mca btl_openib_free_list_inc 1024 --mca mpi_preconnect_mpi 1 --mca btl_openib_flags 9`"

11 Summary and Future Plans

The steps in achieving good application performance on a modern compute cluster of multi-core, multi-socket, NUMA-aware systems can be performed by first using profiling tools to systematically analyze the time used for each component involved. Then, begin by tuning at each level within the software, hardware, and networking that enables the best interaction between the software applications and the system hardware by extending the interconnect capabilities into the software communication algorithms.

By determining what takes the most time in runtime, whether in compute or network communications, we can then decide to tune the area that would be most profitable. The first approach is to tune by profiling the amount of time an application runs. Then, understand and exploit the CPU architecture that yields the best application performance, such as CPU core selection, MPI process placement based on locality, and tuning of the software stack to use different compiler and math libraries.

We have investigated and shown the importance of the MPI process placement. By correctly placing the MPI processes on the core in Bulldozer core pairs, it allows the active core to run at a higher frequency, which translates into 42% better performance than running with both cores in a Bulldozer core pair package, and 30% better

performance than running the same number of cores in all the CPU cores on the first two sockets of a 4-socket system.

We have introduced a tool to facilitate the allocation of MPI processes into affinity grouping to take advantage of multicore CPU architecture that is available in the AMD "Interlagos" processors. The tool ensures proper process allocation support in all flavors of MPI implementations on the Dell systems, which shows a 26% better performance than using a process placement that has not taken into account the actual process number enumeration that takes place in the BIOS.

We have analyzed and implemented code changes to the code of VASP, which has been developed and used predominately on the commercial software stack to support the free software stack. The change in software stack delivers a comparable application performance with the AMD Bulldozer architecture with the support of the FMA instructions and AVX CPU extensions in the software stack. We expect additional performance gain when enabling additional compiler optimizations that were turned off, and when tuning using the latest development tools (such as Open64 and ACML) that support AMD "Interlagos" and "Abu-Dhabi" architecture.

We analyzed the MPI communications by profiling, and we discovered that heavy MPI collective communication occurs during the runtime of VASP. We have also determined that a low latency and high bandwidth network communication is required to make VASP scalable, which explains why InfiniBand would benefit for VASP over other network interconnects.

We compared the scalability performance of various high speed network interconnects, and we were able to see that the effect of deploying a low latency network interconnect such as the QDR InfiniBand improves application performance by over 186% against 40GbE, by over five times compared to 10GbE, and by over 8 times versus 1GbE on 8 nodes (256 processes). Ethernet does not scale and becomes inefficient to run beyond two nodes.

We have discussed various MPI tuning parameters that are crucial to the scalability performance of InfiniBand interconnect, including features such as Mellanox Messaging (MXM) and Shared Reduced Queue (SRQ), which optimize the network resource for MPI communications. SRQ can provide improvement of 24% over the un-tuned run for VASP running at 8 nodes. In addition, by deploying MXM, the reliable messaging acceleration optimized for Mellanox Host Channel Adapter (HCA), we are able to show a 47% performance improvement compared to a baseline run with Open MPI at 8 nodes.

We plan to extend the testing to include the Abu Dhabi testing and compiler optimization for the new machine architecture, so as to take advantage of the new processor extensions, and to publish our results in future papers.

Acknowledgements. Part of this work was supported by the resources of the HPC Advisory Council HPC Cluster Center [2]. The authors would also like to acknowledge the VASP license provided by Professor Kresse of the University of Vienna to make this research possible. [6-9]

References

1. InfiniBand Trade Association, http://www.infinibandta.org/
2. HPC Advisory Council HPC Center, http://www.hpcadvisorycouncil.com/cluster_center.php
3. The TOP500 list, http://www.top500.org
4. Shipman, G.M., Woodall, T.S., Graham, R.L., Maccabe, A.B., Bridges, P.G.: InfiniBand Scalability in Open MPI. In: IEEE Parallel and Distributed Processing Symposium (IPDPS), Rhodes Island, Greece (May 2006)
5. Bailey, D.H., Lucas, R.F., Williams, S.W.: Performance Tuning of Scientific Applications (2011) ISBN 978-1-4398-1569-4
6. Kresse, G., Hafner, J.: Ab initio molecular dynamics for liquid metals. Phys. Rev. B 47, 558 (1993)
7. Kresse, G., Hafner, J.: Ab initio molecular-dynamics simulation of the liquid-metal-amorphous-semiconductor transition in germanium. Phys. Rev. B 49, 14251 (1994)
8. Kresse, G., Furthmüller, J.: Efficiency of ab-initio total energy calculations for metals and semiconductors using a plane-wave basis set. Comput. Mat. Sci. 6, 15 (1996)
9. Kresse, G., Furthmüller, J.: Efficient iterative schemes for ab initio total-energy calculations using a plane-wave basis set. Phys. Rev. B 54, 11169 (1996)
10. Code changes that supports Open64 Compiler on VASP, http://www.hpcadvisorycouncil.com/pdf/open64.diff, http://www.hpcadvisorycouncil.com/pdf/open64.diff
11. Shainer, G., Lui, P., Liu, T., Wilde, T., Layton, J.: The Impact of Inter-Node Latency versus Intra-Node Latency on HPC Applications. In: Parallel and Distributed Computing and Systems. ACTA Press (2011)
12. Shainer, G., Wilde, T., Lui, P., Liu, T., Kagan, M., Dubman, M., Shahar, Y., Graham, R., Shamis, P., Poole, S.: The Co-design Architecture for Exascale Systems, A Novel Approach for Scalable Designs. In: ISC 2012. Springer (2012) ISSN 1865-2034

Offload Compiler Runtime for the Intel® Xeon Phi™ Coprocessor*

Chris J. Newburn, Rajiv Deodhar, Serguei Dmitriev, Ravi Murty,
Ravi Narayanaswamy, John Wiegert, Francisco Chinchilla, and Russell McGuire

Intel Corporation

Abstract. The Intel® Xeon Phi™ coprocessor platform enables offload of computation from a host processor to a coprocessor that is a fully-functional Intel® Architecture CPU. This paper presents the C/C++ and Fortran compiler offload runtime for that coprocessor. The paper addresses why offload to a coprocessor is useful, how it is specified, and what the conditions for the profitability of offload are. It also serves as a guide to potential third-party developers of offload runtimes, such as a gcc-based offload compiler, ports of existing commercial offloading compilers to Intel® Xeon Phi™ coprocessor such as CAPS®, and third-party offload library vendors that Intel is working with, such as NAG® and MAGMA®. It describes the software architecture and design of the offload compiler runtime. It enumerates the key performance features for this heterogeneous computing stack, related to initialization, data movement and invocation. Finally, it evaluates the performance impact of those features for a set of directed micro-benchmarks and larger workloads.

Keywords: multicore, heterogeneous, coprocessor, offload, compiler, runtime, acceleration.

1 Introducing Offload for a Fully-Capable Compilation Target

This paper describes the runtime infrastructure and performance features of the offload runtime for the Intel® Xeon Phi™ coprocessor. This section covers the architecture, offload execution model, suitability criteria for offload, and contributions.

There has been a long-standing interest in using many power-efficient and more memory- and I/O-bandwidth-capable GPUs and CPUs to efficiently accelerate computation [20,5,26]. Taking advantage of such architectures has often depended on largely rewriting user codes. Intel has taken a different approach: make the coprocessor compilation target very similar to the host, incrementally extend existing programming models for parallelism to take advantage of that coprocessor, and leverage the host when single-thread performance is important, e.g. for I/O. This paper describes one way of doing that: through offload of computation from a host CPU to a fully-capable Intel® Architecture processor that enables higher levels of thread and SIMD parallelism and bandwidth for data-parallel workloads than the Intel® Xeon® processor line. It also offers greater power efficiency: it won Green 500 at SC12 [6]. The first product in that family is

* For more complete information about compiler optimizations, see Intel's Optimization Notice at `http://software.intel.com/en-us/articles/optimization-notice`

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 239–254, 2013.

codenamed Knights Corner [10]. It is a coprocessor in a PCIe [2] card form factor. One or more cards may connect to host chips that may be part of a larger cluster, as shown in Figure 1.

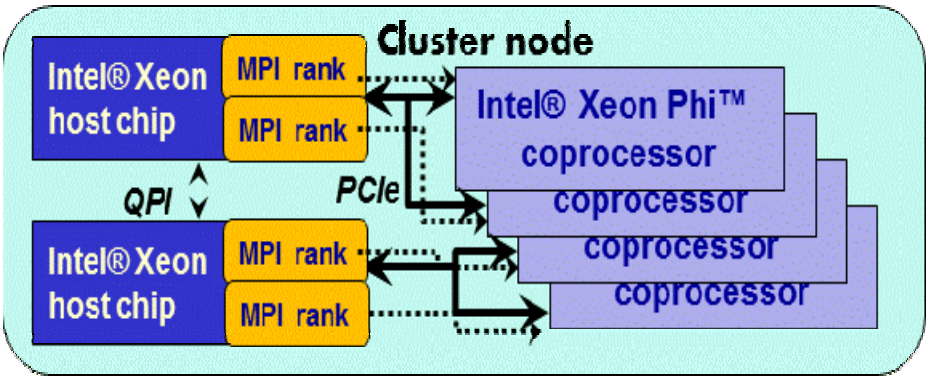


Fig. 1. Intel Xeon Phi™ coprocessor system architecture. MPI ranks may execute on subsets of host cores, and computation may be offloaded to the coprocessors from those ranks, or ranks may be distributed across the host and coprocessor (not shown).

1.1 Execution Models

This platform supports several execution models, as shown in Figure 2. Applications may use the different execution models in different phases, or use a mix of execution models concurrently. And in the symmetric message passing interfaces [7,8, 13] programming model, ranks may run on a heterogeneous set of nodes, concurrently mixing execution on hosts and coprocessors. In hybrid [25] execution, MPI ranks on the host or coprocessor are threaded with OpenMP [22].

- When applications have more than a nominal serial fraction, they are best executed in **host** mode, on a general-purpose CPU that has a higher clock speed and a more aggressive micro-architecture, e.g., an Intel® Xeon® processor (left).
- When applications are highly parallel, with many threads and either highly-SIMD-vectorized code [24] or that demand very high bandwidth to the memory system, **native** execution on an Intel Xeon Phi™ coprocessor may offer higher performance and greater power efficiency (middle or right).
- The highly-parallel phases of the application may be **offloaded** from the Intel® Xeon® host processor to the Intel®Xeon Phi™ coprocessor (middle). In this mode, shown in Figure 1, input data and code are sent to the coprocessor from the host, and output data is sent back to the host when offloaded computation completes. Execution may be **concurrent** on host and coprocessor.

Users may choose to use the offload model instead of native execution because that's higher performing, or for other reasons, such as minimizing complexity or fitting into memory. For example, it may be easier to make an Intel® Math Kernel Library [12] call that transparently results in offload to a coprocessor than to heterogeneously mix MPI ranks on hosts and coprocessors. If the working set [4] fits in the memory on the coprocessor (up to 8GB initially), but the entirety of the memory footprint does

not, it may be necessary to explicitly manage memory locality by partitioning work into computational subtasks that are offloaded to the coprocessor.

1.2 Suitability for Offload

Offloading incurs overhead costs for initialization, marshaling and transferring data, and invocation. These costs vary in their significance. Thus offload may or may not provide a speedup, even if native execution would result in a speedup, as shown in Section 5. There are two important metrics for evaluating the profitability of offload: host-side throughput and response time (latency). Host-side throughput improves if there is a reduction in host-side work that exceeds the host-side overhead of offloading it. Response time improves if the reduction in host-side work exceeds the sum of host-side overhead and the time spent waiting for the coprocessor. The latency formulation accounts for overheads on both the host and coprocessor. Response time and throughput speedups will be identical in the case of single-threaded workloads doing synchronous offload. Offload profitability depends on a mix of the following factors:

- Application characteristics: Applications must have a high ratio of computation to communication (sending code and input data, invocation, and returning output data) for the offloaded portion. Offload communication overheads can be hidden if code is structured to overlap them with computation, or if data is reused across offloads.
- Offload runtime: Offload profitability depends on the efficiency of the host- and coprocessor-side runtimes in marshaling and moving data and invoking code .
- Performance on the *coprocessor and system*: It must benefit from the highly-parallel hardware (SIMD width, threads, and high bandwidth) of the Intel® Xeon Phi™ coprocessor. For example, a serial application that is I/O intensive is unlikely to be a good candidate for that coprocessor.

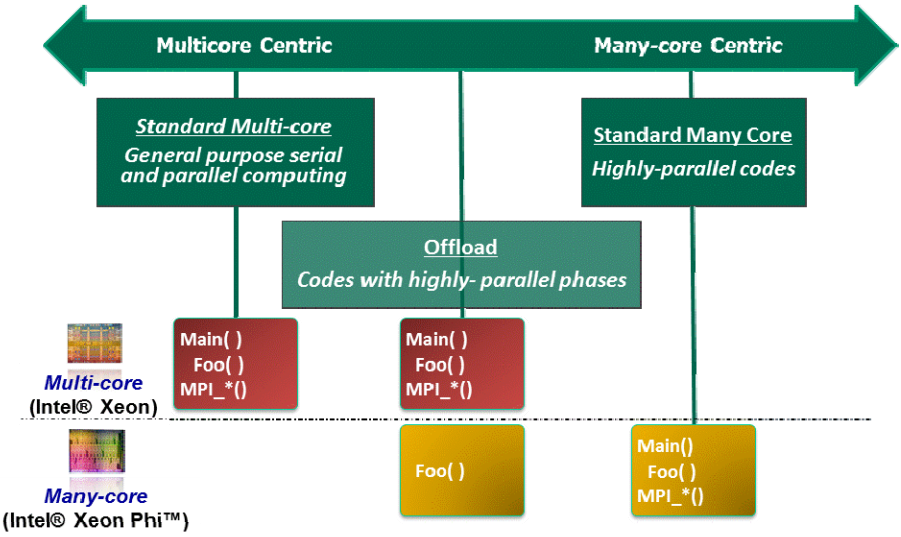


Fig. 2. Execution models: host-only, offload from host to coprocessor and native on coprocessor only

The paper shows examples of workloads with high and with low computation to communication ratios, as well as good and poor native speedups. It focuses on compiler runtime implementation and performance issues related to reducing offload overheads on the host and coprocessor. The performance is evaluated on a few workloads and several directed tests that are in the form of micro- benchmarks.

1.3 Contributions and Scope

The contributions of the paper are as follows. This is the first known description of a production offload compiler and offload runtime software infrastructure for a widely-deployed commercial coprocessor that is as capable as the host CPU. This is the first published description of the Intel® Xeon Phi™ coprocessor offload runtime software architecture. It enumerates issues that matter most to offload compiler runtime performance, describes the solutions to those performance issues, and evaluates their impact on both micro-benchmarks and customer workloads.

There are several other issues of potential interest which are out of scope for this paper. One is a detailed description of the offload programming models described here, and an assessment of their ease of use, performance and productivity relative to other programming models such as CUDA® [20] or OpenACC® [21] or hand-coded assembly. Another is a comparison of our OpenMP 4.0 TR1 [23] implementation with the `#pragma offload` approach. Another is techniques for evaluating and improving the suitability of parallelized code for Intel® Many-Integrated Core, which is documented online [11]. This paper does not delve into the details of the non-compiler runtime components and their performance. The paper's focus on the offload runtime leaves aside micro-architecture-specific issues related to code generation, like vectorization and prefetching. Finally, issues related to sharing the virtual address space between the host and coprocessor [29] are not covered in this paper.

2 Software Architecture

The Intel® Xeon Phi™ coprocessor software architecture is shown in Figure 3. There are essentially four layers in the software stack: offload tool runtimes, user-level offload libraries, a low-level communication layer that's split between user-level libraries and kernel drivers, and the operating system. There is a host-side and coprocessor-side component for each. Everything below the offload runtimes is part of the Intel® Manycore Platform Software Stack (MPSS).

This paper focuses on the compiler offload runtime of the Intel® Composer XE 2013 compiler, but there are several other tool offload runtimes listed in Section 3. Intel provides a user-level offload library, called the Intel® Coprocessor Offload Infrastructure (COI) [15]. This library provides services to create coprocessor-side processes, create FIFO pipelines between the host and coprocessor, move code and data, invoke code (functions) on the coprocessor, manage memory buffers that span the host and coprocessor, enumerate available resources, etc. Offload runtime implementations aren't strictly required to use COI, but doing so can relieve developers of significant implementation complexity and tuning effort, and it provides portability

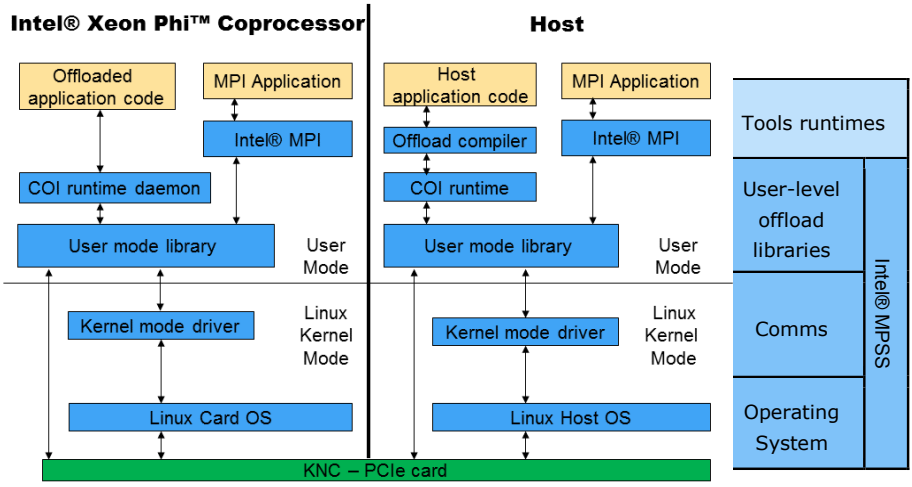


Fig. 3. Intel® Xeon Phi™ coprocessor software architecture

to other host OSes such as Microsoft® Windows™. This paper discusses COI's APIs, but the paper covers implementation and performance optimizations only for the compiler runtime, not other lower-level runtimes like COI and SCIF [15, 28].

Finally, the host and coprocessor have separate operating systems. The snapshot of supported host OSes at the time of evaluation includes RHEL [27] 6.0 kernel 2.6.32-71 and 6.1 kernel 2.6.32-131 and SLES 11 SP1 kernel 2.6.32.12-0.7 [31]. The coprocessor OS is a stock Linux kernel 2.6.34.11 with few architecture-specific modifications. Using a standard OS eases restrictions on what code is offloaded, enables building third-party software and eliminates the need for many proprietary services.

3 Programming Models

A brief background on programming models is provided here, for context. This is not the focus of this paper. Offload is accomplished by runtimes which implement:

- **language pragmas**, e.g. `#pragma omp target` data device(1) map() used by various compilers, e.g. from Intel [9] and CAPS [3], and #pragma offload, an Intel-specific extension (see Figure 4)
- **language keywords**, e.g. the _Cilk_shared keyword used by the Intel compiler [9,29] or language constructs used by CUDA [20] or OpenCL [16] such as Intel® SDK for OpenCL Applications XE for the coprocessor [14]
- **library calls**, e.g. Intel® Math Kernel Library (MKL) [12], MAGMA [17,1] or NAG [19] calls that divide work across the host and coprocessor

There are also third-party preprocessors that use cost models to evaluate when and how to make use of those tools [26].

This paper focuses on a preliminary version of a proposed but not yet ratified `#pragma omp` offload extension to the OpenMP 4.0 standard [23] implemented by the Intel compiler. An extended version of this paper [18] also covers an Intel-proprietary

set of directives, based on `#pragma offload`. Single-threaded code that uses various offload constructs is illustrated in Figure 4. This code shows how the host synchronously invokes work on the coprocessors, as though DGEMM were invoked on the host with the Intel® Math Kernel Library. See Section 4 for details.

In general, the programmer has to explicitly specify what to do using the offload pragmas. But in some cases, there is a default, for ease of programming. In the `#pragma omp` case, data transfer of variables to and from the coprocessor for offloaded code is explicitly named with `map(to:…)`, `map(from:…)` and `map(tofrom:…)` clauses, but the `map(tofrom:…)` clause is implicitly used for all variables that are not explicitly named but are visible in the scope of a construct during compilation. The coprocessor offload target is specified like `device(1)`.

The offload compiler runtime doesn't manage parallelism. It's the programmer's responsibility to enable and expose vectorization and the compiler's responsibility to extract it, to take advantage of SIMD parallelism. It's the programmer's responsibility to take advantage of thread parallelism. There are two approaches to using hardware threads: using MPI to parallelize across ranks, or using a threading runtime like pthreads, OpenMP [22] or Threading Building Blocks [32] to create and parallelize across a team of threads. These threading runtimes execute natively on the coprocessor. The offload runtime's only interaction with threading runtimes is to set up environment variables they need, e.g. for CPU affinities. By default, the offload runtime sets the CPU affinity mask to allow threading runtimes to execute across all coprocessor threads except those that are used by system software.

4 Offload Compiler Runtime Design

Offload compiler runtime performance matters when the act of offloading is on the critical path. Offload can sometimes be taken off of the critical path with extra coding effort. But in the general case, offload compiler runtime performance can be critical to the profitability of offload, as shown by the data below. This section outlines the key compiler runtime (not COI) performance features that impact performance.

4.1 Overview

The left of Figure 4shows the host-side code for an offload example, and the right of the figure shows the corresponding sequence of compiler offload runtime actions. When the coprocessor is booted, before execution of the application ever begins, a COI daemon is created (not shown), which handles process creation. Compiler offload runtime initialization discovers the coprocessor and establishes a connection to the COI daemon, which creates a coprocessor-side process that corresponds to the host-side process and that has appropriate environmental settings. At appropriate times, code that is embedded in the host-side binary is extracted, moved and invoked, and memory management is begun.

When data transfers or invocations are specified by the programmer, they trigger specific actions that are executed by the compiler offload runtime and its supporting libraries. Data must be allocated, marshaled, efficiently transferred, unmarshaled and deallocated as necessary. The addresses of coprocessor-side functions must be looked

up and they must have their stack frames set up and be invoked with appropriate parameters, once data transfer dependencies are satisfied.

The COI daemon and the host-side COI infrastructure robustly handle planned and unexpected process termination and error handling on the host and coprocessor.

In the following sections, we focus on each of the performance-critical cases.

4.2 Initialization

The OFFLOAD_INIT environment variable controls whether initialization happens at program startup (on_start), on first use for each device (on_offload, the default), or for all devices at the first offload to any device (on_offload_all).

The first step of initialization is to evaluate the conditions for offload. The compiler offload runtime checks for the presence of the COI software stack and the availability of Intel® Xeon Phi™ coprocessors in the system: it gets the count of "engines" (coprocessors) that are currently available. If all offload conditions are satisfied, the compiler offload runtime creates a process on the coprocessor with COIProcessCreateFromFile that corresponds to the host process; creates, marshals and moves code and data; and the startup image that is part of the compiler offload runtime that is loaded from disk performs initialization there.

The coprocessor side of the application consists of only those parts of code and data which are marked by the user for offloading. These parts include all routines defined with the target attribute, regions of code following offload pragmas, and variables with static storage which are defined with the target attribute. All other code and data variables are filtered out. The offload compiler statically generates the coprocessor code and data segments, and embeds it in the host-side binary executable. These get loaded onto the coprocessor at process creation time using COIProcessLoadLibraryFromMemory. Additional coprocessor-side code may be available in dependent shared libraries, and these dependencies are detected, resolved, and copied to the coprocessor by COI.

The compiler offload runtime needs the coprocessor-side address for all variables marked with the `target mirror` attribute, in order to know where to transfer data. The address is transferred from the coprocessor to the host during initialization and it is used throughout the execution for transferring data between CPU and coprocessor. This early transfer of the address for heap and static data avoids the need to DMA from the host into a temporary buffer on the coprocessor, and to rely on coprocessor-side code to copy the data to the final destination.

4.3 Invocation

If the offload initialization conditions are not met, there is no invocation of the coprocessor. If the user has mandated the use of offload, e.g. with the `device(1)` clause, then the runtime produces an error. In the absence of such a mandate, a host version of the offloaded region is executed on the host instead of using offload.

Each host thread that has offload pragmas has a corresponding coprocessor-side thread, linked by a data object called a COIPipeline. This object facilitates ordered invocation with a FIFO command queue. That thread and pipeline span multiple transactions between the host and coprocessor, enabling offloaded code to preserve thread-local variable values, OpenMP teams, etc. from one offload to another.

Host-side source code	**Corresponding offload runtime actions** **Only highlighted actions are on coprocessor**

double A[SIZE], B[SIZE], C[SIZE];
…
void f() {
  int transa, transb, N;
  double alpha, beta;
  double *A,* B,,*C;
 	
  A = malloc(..);
  B=malloc(..)
  C= malloc(…)
 	
  …
 	
// Define a data region and allocate
// memory for A, B and C on the
// coprocessor
#pragma omp target data device(0)\
  map (alloc : A[0:length], \
  B[0:length], C[0:length]) {
A = …; B = …
 	
// Xfer the data to the coprocessor
#pragma omp target update \
  device(0) to (A[0:length], \
  B[0:length]
// Perform the computation
#pragma omp target device(0) \
  map (to : transa, transb, N, alpha, \
  beta)
 	
{dgemm(&transa, &transb, &N, \
  &N, &N, &alpha, A, &N, B, &N, \
  &beta, C, &N);
}
 	
// Transfer the result from the
// coprocessor
#pragma omp target update \
  device(0) from (C[0:length])
 	
// End of data region
}
	// Coprocessor initialization, on demand by default
	if ((count = COIEngineGetCount()) <= 0)
	  // fatal error if target unavailable
	eng = COIEngineGetHandle(0)
	proc=COIProcessCreateFromFile(eng, startup_image…)
	COIProcessLoadLibraryFromMemory(proc,codegen_image)
	thunk = COIProcessGetFunctionHandles(…)
	pipe = COIPipelineCreate()
	/
	
	/ transfer variable mapping info – not shown
	// Buffer allocation, same for B, C
	bufAcpu = COIBufferCreateFromMemory(A,bufAsize)
	bufAmic = COIBufferCreate(bufAsize)
	COIBufferAddReference(bufAmic)
	
	/ Separate data transfer since >30KB; same for B
	COIBufferCopy(bufAmic, bufAcpu, bufAsize
	  0, 0, NULL) // no input dependencies
	
	//Start dgemm on the coprocessor
	// pack misc_data with name of function calling dgemm
	memcpy(misc_data, nameOfFunctionCallingDgemm)
	
	COIPipelineRunFunction(misc_data,
	  thunk, // offload function in runtime
	  0, 0, // no buffers
	  0, 0, // no dependecies
	  &runEvent)
	
	// Transfer results from coprocessor to CPU
	COIBufferCopy(bufCcpu, bufCMic, bufCsize,
	  &runEvent, 1, // input dependencies
	  &outEventC) // completion event
	dgemm(…), same for B, C
	COIBufferReleaseReference(bufAmic)
	// Destroy buffer for A, same for B, C
	COIBufferDestroy(bufAcpu)
	COIBufferDestroy(bufAmic)

Fig. 4. Code example illustrating the use of OpenMP offload extensions in discussion for OpenMP 4.0 RC2

Each offloaded region following an offload pragma is outlined by the compiler into a separate routine having a unique name. The routine's name and address compose a unique entry which is added to a compiler-generated function lookup table.

Function invocation uses indirection, through a thread-safe, coprocessor-side thunk. An invocation helper function is used to invoke coprocessor code that is shared across pipelines. COIProcessGetFunctionHandles is used by the host to get a handle for the thunk. The host puts a string with the coprocessor-side function name in a "misc_data" buffer, and uses COIPipelineRunFunction for function invocation. Its function arguments include a handle for that thunk, a pointer to misc_data, input buffers, and dependence objects. The resolution of those dependencies, e.g. movement of data down to the co-processor, as checked by COIPipelineRunFunction, gates the invocation of the thunk. The thunk extracts the coprocessor function name string from misc_data, looks it up in a function address table, and invokes the outlined function. The compiler-generated outlined function is responsible for demarshaling the data passed during invocation and executing the offloaded region on the coprocessor.

4.4 Memory Management

In the example in Figure 4, a buffer object is created on the host by COIBufferCreateFromMemory. This function creates a corresponding buffer on the coprocessor, and the transfer is accomplished with COIBufferCopy. Having distinct host and coprocessor buffers, rather than sharing across the PCIe aperture, protects from subsequent modification of the host buffer that could lead to data races.

Coprocessor-side data may be global/static, on the heap, or on the stack. The handling is different for each case. For global/static data, the addresses are already fixed at code generation time, and a mapping table is created upon initialization that the compiler offload runtime uses to set up direct DMAs between host and coprocessor with the correct physical addresses. For heap data, a host-coprocessor address mapping is established and communicated at runtime. In both cases, extra copies through temporary buffers are not necessarily required, and data is persisted as needed.

Each offload invocation creates and destroys its own coprocessor-side stack frame. Coprocessor-side interrupts that occur between invocations use, and hence clobber, the user stack. Because of this, data from the stack frame of one offload function may not persist until the next such invocation. The host-side data that falls within the invoker's scope gets moved and copied onto the coprocessor-side stack upon each invocation and copied back after completion. The programming interface can hide that, thereby emulating persistence, but keeping large variables on the stack isn't recommended for performance reasons. Therefore, use of stack variables for offload is less efficient than using heap or global variables.

Heap data introduces some special issues: allocation and alignment. `#pragma omp` allocates and frees data on coprocessor on the outermost `#pragma omp` region. Data inside nested constructs are implicitly tracked and are ignored if they are also specified in the `map` clause. To synchronize variables between host and coprocessor inside a `#pragma omp data` construct use the `#pragma omp update` construct. COI uses a reference counting scheme to manage data persistence, which explains the use of COIBufferAddReference and COIBufferRelease-Reference.

Handling of multiple coprocessors is generally outside the scope of this paper, but it should be noted that the offload runtime supports more than one coprocessor using the target clause. It is entirely the user's responsibility to synchronize data across different coprocessors, unlike the transparent abstraction offered by MYO [29].

Fortran arrays use dope vectors to describe the number and size of dimensions. This extra 72 or more bytes of metadata must be included in data transfers. At present, with `#pragma omp` directives, only contiguous data transfers are supported, although the base languages permit writing array-slice expression with arbitrary strides.

Table 1. Platform configuration parameters

Host	SNB-EP (2 sockets) 2.6 GHz, Intel® Xeon® E5-2670, Crown Pass Platform
Coprocessor	Pre-production Intel® Xeon Phi™ coprocessor, 61 4-thread cores, 1.09GHz, 5.5GTransfers/s, 8GB (all apps fit)
Host OS	RHEL 6.2, kernel 2.6.32-220.el6.x86_64
Compiler	Composer XE Beta
MPSS	2.1.3653-8, kernel 2.6.34.11

5 Performance Evaluation

When there are multiple architectural families available, like Intel® Xeon® processors and Intel® Xeon Phi™ coprocessors, developers must make a choice as to which is most suitable. The Intel® Xeon Phi™ product family is an extension of the Intel® Xeon® processor family, with different design objectives. A careful consideration of how application characteristics map onto each platform helps direct users to perform a theoretical or empirical analysis of the app, to optimize accordingly.

Performance of offload runtime features is evaluated in two ways: at the directed test level and at the workload level. Directed tests measure the impact of a particular feature in isolation; workloads measure the impact of that feature in the context of a workload. Evaluation at the workload level focuses on overall overheads and profitability. We first evaluate overall profitability, and then deep-dive into the impact of specific features. The evaluations below were based on the `#pragma offload` form of directives, on the platforms described in Table 1. The performance of the `#pragma omp` form is materially identical.

5.1 Offload Profitability

Table 2 provides a summary characterization of a few workloads, and the numerical analysis below is based on that data only, vs. being a general characterization of customer code. The workloads were selected to span a variety of application domains. They came from customers, and are meaningfully representative to them. Some workload names are occluded in deference to customers. The first row of data shows the overall speedup attained using the coprocessor with the offload runtime, relative to host-only execution. Workloads in this paper use OpenMP, but not MPI.

Table 2. Offload performance and overheads for workloads

Workload	3DFD PDE Stencil	Convolution Resampling	Hogbom-Clean	QR	Iterative closest point, DP	Adaptive Sparse Grid	Black Scholes Compute SP
Domain	Seismic	Astronomy	Astronomy	Physics	Manufac-turing	Physics	Financial
Speedup (x): offload vs. host only	2.03	1.56	2.31	1.40	1.54	1.32	6.92
Compute % of total execution	97.7	44.3	72.9	97.3	94.5	95.3	99.4
Host offload overhead (x), with (top) & without (bottom) init	0.21	1.77	0.44	0.03	0.02	0.06	0.00
	0.18	1.60	0.25	0.01	0.01	0.05	0.00
Computation/communication ratio	5.42	0.62	3.99	68.6	90.1	19.7	1640

The percentage of total execution time that is offloaded computation varies from 44% to 99%. The overhead for moving data and invocation varies from being negligible to being a 1.77x multiple of the computation time. Coprocessor-side overheads were measured but are not shown, since they are negligible. The ratio of computation to communication ranges from 0.62x to 1640x. It was generally over 4x, except for the convolution case, whose scaling across threads and SIMD elements on the coprocessor allows the computation speedup to outweigh the overheads.

Black Scholes, which has the highest computation to communication ratio, the highest offload fraction, and the lowest overheads, is the big performance winner at 6.92x speedup of a dual-socket Sandy Bridge. Recall that the offload speedup depends on several factors, so it is sometimes but not always highest when the computation to communication ratio is high. To illustrate that point, there was temporary compiler regression in which the coprocessor execution time shot up, and the speedup for Adaptive Sparse Grid fell to parity between host only and offload.

The impact of communication and offload overheads can be partially hidden with the use of asynchronous invocation. Using the asynchronous form of `#pragma offload` (see [18]) benefitted SHOC [30] Triad by 1.65x (3.16/1.91), as shown by comparing the two columns at the right of Table 3. That speedup didn't come from reducing the offload overhead, which was only a 1.11x (8.51/7.67) improvement; it was from overlapping computation with communication, only some of which (0.96/0.72=1.33x) can be captured by the available runtime stats.

Table 3. Offload performance and overheads and native performance for SHOC

Workload	FFT-DP	FFT-SP	GEMM-DP	GEMM-SP	MD-DP	MD-SP	Reduction-DP	Reduction-SP	S3D-DP	S3D-SP	SCAN	Sort	SPMV-DP	SPMV-SP	Triad Sync	Triad Async
Data set	N= 16777216	N= 33554432	2048 x 2048	4096 x 4096	73728_atoms	73728_atoms	8388608_items	16777216_items	262144_gridPoints	262144_gridPoints	8388608 items	4007383_elements _62451_rows 4007383_elements _62451_rows	4007383_elements _62451_rows	2684354_elements	4096KB	4096KB
Speedup with offload (top) and native only	0.30	0.45	3.30	3.25	0.81	0.44	0.20	0.20	0.95	1.03	0.25	0.21	0.07	0.07	1.91	3.16
	3.33	5.13	4.47	3.88	1.94	1.20	4.14	4.25	1.40	1.37	4.29	0.71	0.80	0.62	NA	NA
% execution time in offload	3.58	3.62	33.0	59.1	19.3	0.26	37.3	19.0	7.53	6.92	68.0	17.4	8.22	6.91	3.4	5.8
Host offload overhead, with & without init	13.9	14.8	1.08	0.34	2.42	17.1	0.92	2.42	9.75	9.45	0.19	3.66	6.09	6.99	8.51	7.67
	1.20	1.09	0.13	0.07	0.52	0.54	0.41	0.48	4.96	3.09	0.02	2.47	1.60	1.54	1.39	1.04
Computation: communication	0.83	0.92	7.74	14.0	1.92	1.85	2.46	2.10	0.20	0.32	56.9	0.40	0.63	0.65	0.72	0.96

Offload runtime and application performance tends to improve with tuning. For example, allocation and initialization time, which depend on the amount of memory to be initialized, was reduced from several seconds down to between 0.25 and 2.5s for the workloads in this paper. As shown in the SHOC [30] data, allocation and initialization costs are still significant for small kernels. Further efforts to hide those costs is expected to bring them down.

5.2 Runtime Performance Feature Impact

As mentioned above, the performance of the offload runtime in initialization, data transfer, and invocation can determine whether offload is profitable. This section analyzes the most important offload compiler runtime performance features: transfer avoidance, copy avoidance and page size.

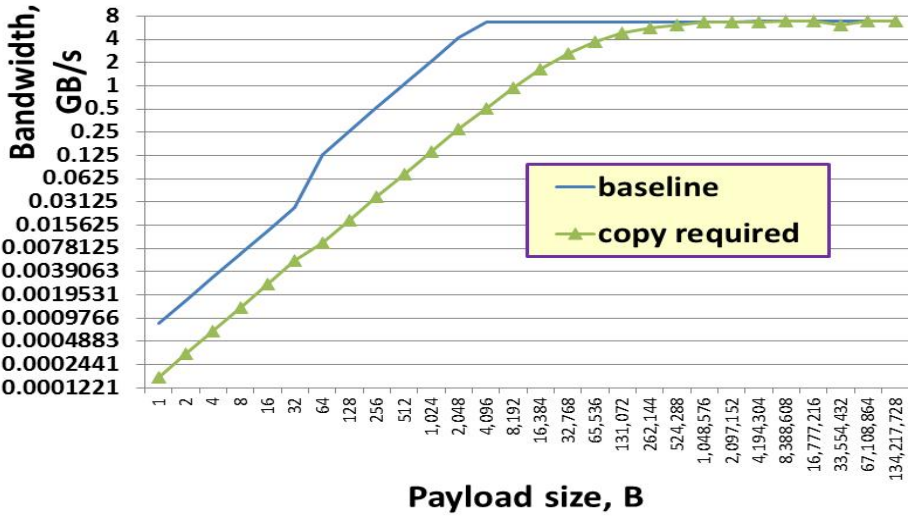


Fig. 5. DMA performance, as a function of payload size, for host to coprocessor transfers. For the baseline, the destination memory location is known. The other curve shows the cost of an extra copy.

When data can be persisted, there may be no need to transfer data at all. Consider the case of a temporary variable that is only written to on the coprocessor side, is never used on the host side, but that is needed by several offloaded functions. If that variable is static, it never needs to be copied back and forth across the PCIe, but if it lives on the stack of an enclosing function, it must be copied in and updates must be copied back to the host in each function. Notice, in Tables 2 and 3, that the ratio of computation to communication can be low enough that eliminating communication can significantly impact performance.

The cost of an extra copy is shown in Figure 5, where the median speedup from copy avoidance is 2.6x and the mean ratio is 3.9x. The baseline curve is for transfers whose destination address is already known to the host, that use 2MB pages and that otherwise meet the optimal conditions.

When large amounts of data must be transferred between the host and coprocessor, the hardware DMA (direct memory access) engine is used. We describe three cases where a direct transfer without extra copies to the final destination may be impossible.

The first case is mutual misalignment between host and coprocessor offsets of addresses within a cache line. For example, there is mutual misalignment when the modulus of differences in global/static addresses isn't 0, or when a host address is an odd multiple of 32, and there is a `#pragma offload` with an `align` clause value of 64. This is important because mutual alignment within a 64B cache line enables a direct DMA into the destination, whereas mutual misalignment within a 64-byte cache line on the initial production member of the Intel® Xeon Phi™ product family isn't supported by the DMA engine. Without mutual alignment, an extra buffer copy is necessary, incurring the costs quantified below. Static data can have mutual misalignment within a cache line. By default, the alignment within a cache line of the coprocessor-side heap data is set to match that of the host-side data.

In the second case, data that is resident on the stack has to be copied from a pinned DMA buffer onto the coprocessor's stack. In the third case, when the destination physical address is not known to the host prior to the DMA, there is a DMA to a temporary buffer, followed by a copy to the address which is already known to the coprocessor code. The compiler offload runtime avoids that copy by sending the coprocessor variable memory locations at initialization.

The overall median improvement of 2MB vs. 4KB pages for several customer work-loads is 1.051x and the average is 3.20x. Almost all of those workloads benefitted from large pages, whose use is enabled by the offload runtime environment variable. The average gain for using large pages in DMA transfers is only 1.064x. See [18] for more detail.

6 Summary and Conclusions

We demonstrated the potential for making offload to the Intel® Xeon Phi™ coprocessor profitable. We showed sample workloads with speedups over a dual-socket Intel® Xeon® E5-2670, codenamed Sandy Bridge, in the 1.3x to 6.9x range as customer-relevant examples that span different application domains. But offload is not always profitable, as the SHOC data showed. The paper explores what can enhance or inhibit offload speedup. First, where response time is of concern, there must be a speedup from native execution on only the coprocessor, relative to execution on the host. This is based on the performance of the code on coprocessor and system, such as whether it's highly threaded, and either uses SIMD well or its bandwidth demand exceeds what's available on the host. Second, the ratio of computation to communication must generally be high. In the workloads evaluated, that ratio ranged from 4x to 1640x for all but one case. This depends on characteristics of the application, and how it is structured for offload. For example, a simple restructuring of SHOC Triad to use of asynchronous offload boosted performance. Third, the offload runtime overheads must be small relative to the computation. This depends on the offload runtime implementation.

Runtime offload overheads ranged from negligible to a factor of 1.77x of the actual computation time, on sample workloads for which offload is profitable. These overheads are important to performance; an optimized implementation of the offload compiler runtime is a key aspect of platform performance for offloaded apps. This paper highlights the key facets of the offload compiler runtime implementation which impact performance, describes those performance features, and evaluates their performance. The performance impact of the offload compiler runtime performance features range from a few percent for page size, up to a mean across payload sizes of 3.9x for incurring an extra copy in cases of stack residency, DMA mutual misalignment, or an unknown DMA destination address.

This paper provides the first description of the Intel® Xeon Phi™ coprocessor software architecture, and serves as a guide both to end users wanting to evaluate whether offload is likely to be profitable, and to third-party implementers of heterogeneous runtimes for Intel® Xeon Phi™ coprocessors and potentially other targets seeking guidance in design principles and in what's most important to optimize.

The authors would like to acknowledge many contributors to this paper, who include implementers of the tools and system software described herein, and the many

applications engineers who tuned and measured performance on customer codes. We thank the customers who shared their workload examples, and reviewers who helped improve the presentation of the material.

References

1. Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., Tomov, S.: Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA Projects SciDAC 2009: Scientific Discovery through Advanced Computing, San Diego, California. Journal of Physics: Conference Series, vol. 180, p. 012037. IOP Publishing (2009)
2. Budruk, R., Anderson, D., Shanley, T.: PCI Express System Architecture, 1st edn., 1120 pages (2003) ISBN 978-0-321-15630-3
3. CAPS, http://www.caps-entreprise.com/technology/hmpp
4. Denning, P.J., Schwartz, S.C.: Properties of the Working-Set model. Communications of the ACM 15, 191–198 (1972)
5. Donaldson, A.F., Dolinsky, U., Richards, A., Russell, G.: Automatic offloading of C++ for the Cell BE Processor: A case study using offload. In: Proceedings of the 2010 International Conference on Complex, Intelligent and Software Intensive Systems, pp. 901–906 (2010)
6. Green 500: The Green500 List (November 2012), http://www.green500.org
7. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message Passing Interface, 2nd edn. MIT Press, Cambridge (1999)
8. Gropp, W., Lusk, E., Thakur, R.: Using MPI-2: Advanced Features of the Message-Passing Interface. MIT Press, Cambridge (1999)
9. Intel® C/C++ compiler, http://www.intel.com/Software/Products
10. Intel® Many Integrated Core, http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html
11. Intel® Many Integrated Core SW development pages, http://software.intel.com/mic-developer
12. Intel® Math Kernel Library, http://www.intel.com/Software/Products
13. Intel® Message Passing Interface, http://software.intel.com/en-us/intel-mpi-library/
14. Intel® OpenCL for Intel® Xeon Phi™ Coprocessor, http://software.intel.com/en-us/vcsource/tools/opencl-sdk-xe
15. Jeffers, J., Reinders, J.: Intel® Xeon Phi™ Coprocessor High Performance Programming. Morgan Kaufmann (2013)
16. Khronos, http://www.khronos.org/opencl/
17. MAGMA, http://icl.cs.utk.edu/magma/
18. Newburn, C., Deodhar, R., Dmitriev, S., Murty, R., Narayanaswamy, R., Wiegert, J., Chin-chilla, F., McGuire, R.: Offlad Runtime for the Intel® Xeon Phi™ Coprocessor, http://software.intel.com/en-us/articles/offload-runtime-for-the-intelr-xeon-phitm-coprocessor
19. Numerical Algorithms Group, Ltd., http://www.nag.com/

20. NVIDIA CUDA reference manual, version 5.0 (October 2012), http://docs.nvidia.com/cuda/pdf/CUDA_Toolkit_Reference_Manual.pdf
21. OpenACC, http://www.openacc-standard.org/
22. OpenMP (March 2013), http://www.openmp.org/mp-documents/OpenMP_4.0_RC2.pdf
23. OpenMP (November 2012), http://www.openmp.org/mp-documents/TR1_167.pdf
24. Patterson, D., Hennessey, J.: Computer Organization and Design: the Hard-ware/Software Interface, 2nd edn., p. 751. Morgan Kaufmann Publishers, Inc., San Fran (1998)
25. Rabenseifner, R., Hager, G., Jost, G., Keller, R.: Hybrid MPI and openMP parallel programming. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) PVM/MPI 2006. LNCS, vol. 4192, p. 11. Springer, Heidelberg (2006)
26. Ravi, N., Yang, Y., Bao, T., Chakradhar, S.: Apricot: An optimizing compiler and productivity tool for x86-compatible many-core coprocessors. In: Proc. of the 26th ACM International Conference on Supercomputing, pp. 47–58. ACM, New York (2012)
27. Redhat, http://www.redhat.com/products/enterprise-linux/
28. Reinders, J., http://parallelbook.com/blogs/james
29. Saha, B., Zhou, X., Chen, H., Gao, Y., Yan, S., Rajagopalan, M., Fang, J., Zhang, P., Ronen, R., Mendelson, A.: Programming model for a heterogeneous x86 platform. SIGPLAN Not. 44(6), 431–440 (2009)
30. SHOC 1.1.1 manual, http://ft.ornl.gov/doku/_media/shoc/shoc-manual-1.1.1.pdf
31. SUSE, https://www.suse.com/promo/sle11.html
32. Threading Building Blocks, http://threadingbuildingblocks.org

Fork-Join and Data-Driven Execution Models on Multi-core Architectures: Case Study of the FMM

Abdelhalim Amer[1], Naoya Maruyama[2], Miquel Pericàs[1], Kenjiro Taura[3], Rio Yokota[4], and Satoshi Matsuoka[1]

[1] Tokyo Institute of Technology, Tokyo, Japan
[2] RIKEN, Kobe, Japan
[3] The University of Tokyo, Tokyo, Japan
[4] KAUST, Saudi Arabia

Abstract. Extracting maximum performance of multi-core architectures is a difficult task primarily due to bandwidth limitations of the memory subsystem and its complex hierarchy. In this work, we study the implications of fork-join and data-driven execution models on this type of architecture at the level of task parallelism. For this purpose, we use a highly optimized fork-join based implementation of the FMM and extend it to a data-driven implementation using a distributed task scheduling approach. This study exposes some limitations of the conventional fork-join implementation in terms of synchronization overheads. We find that these are not negligible and their elimination by the data-driven method, with a careful data locality strategy, was beneficial. Experimental evaluation of both methods on state-of-the-art multi-socket multi-core architectures showed up to 22% speed-ups of the data-driven approach compared to the original method. We demonstrate that a data-driven execution of FMM not only improves performance by avoiding global synchronization overheads but also reduces the memory-bandwidth pressure caused by memory-intensive computations.

1 Introduction

Hardware manufacturers now focus on multi-core and many-core technologies as a way to increase performance and make use of the continually increasing number of transistors. The multi-core road-map provides a slowly increasing number of processing units with each new generation, while maintaining high single thread performance thanks to their sophisticated control logic, out-of-order execution, and their complex memory hierarchy. However, this architectural design leads to unprecedented programming difficulties to extract their potential due mainly to its memory subsystem. Non Uniform Memory Access (NUMA), memory bandwidth limitations, and complex memory hierarchies are key properties that hinder programmer productivity.

In order to exploit parallel architectures, different execution models can be adopted. In a fork-join model, independent tasks run concurrently while task

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 255–266, 2013.

dependencies are ensured by global synchronization barriers. Data-driven (also called data-flow) models have been proposed in order to avoid global synchronizations and improve resources exploitation. However, proponents of the fork-join model argue that data-flow models have worse memory behavior. As a result, fork-join and data-driven methods have their trade-offs, and the achievable performance will depend both on the algorithm and the target architecture.

In the present work, we study these execution models on state-of-the-art multi-core architectures by using the *Fast Multipole Method* (FMM). We choose the FMM, given its wide usage in many scientific domains such as astrophysics[1], electrodynamics[2], and fluid dynamics[3]. Furthermore, FMMs are composed of heterogeneous computations following a complex execution flow. Moreover, we rely on one of the fastest FMM codes and its highly tuned OpenMP fork-join parallel implementation [4][5]. To implement a data-driven execution, there are many runtime schedulers or programming models which have the ability to express task dependencies and perform an asynchronous execution. StarPU[6], OmpSs[7], and Quark[8] are examples of such tools. However, we choose to implement our data-driven FMM using the lightweight thread library MassiveThreads [9]. The low overhead, flexibility, and load-balancing mechanism of this library let us implement an efficient fine-grain data-driven FMM which uses a distributed scheduling approach.

We summarize our contributions and findings as follows:

- We implement a novel thread-based data-driven FMM by using a source centric approach and efficiently managing the task dependencies using a distributed scheduling approach. We further reduce data movements by exploiting the tree nature of the FMM data-structures and using sub-tree based working sets per thread.
- We perform an in-depth analysis of the original implementation which reveals that at large scale the often neglected stages at smaller scale consume more time than the usually compute intensive ones.
- Our evaluation on state-of-the art x86 multi-core architectures, showed that the data locality issues of the data-driven execution are not significant, and when eliminating the synchronization overheads, this method achieved up to 22% speed-ups over the original implementation.
- We also found that the data-driven approach can reduce localized high memory bandwidth stress by spreading the memory traffic along the execution. Moreover, we prove that the memory bound nature of one of the kernels is not only due to its low arithmetic intensity but also bound by remote memory transfers and a non-unit-stride memory access pattern.

The rest of the paper is organized as follows: Section 2 introduces the Kernel Independent FMM and its different computational stages. Then, we discuss our data-driven implementation in Section 3. We describe the configuration of our tests and the details of the target multi-core machines in Section 4. In Section 5, we evaluate and analyze both execution models on the target machines. In Section 6, we discuss related work and we conclude in Section 7.

2 The Fast Multipole Method

Nbody problems can be encountered in many disciplines such as mathematical physics, machine learning, approximation theory, etc. The problem is how to efficiently evaluate pairwise interactions between N bodies. It can be formally described as follows:

$$f(x_i) = \sum_{j=1}^{N} K(x_i, y_j)s(y_j), i = [1..N] \quad (1)$$

where $f(x_i)$ is the potential at the target x_i resulting from the sources y_j, s the source density, and K the interaction kernel. A direct computation results in a $O(N^2)$ complexity which makes it very expensive for large problem sizes. First attempts towards a faster method brings the complexity to $O(NLogN)$ like the Barnes-Hut method[10]. The FMM was proposed as an even faster solution that uses a rapidly convergent method achieving a $O(N)$ complexity [11].

Most of FMMs rely on analytic expansions to evaluate pairwise interactions. Analytical expansions are problem dependent, not always available, and difficult to build. In this work we use the Kernel-Independent FMM (KIFMM) developed by Ying et al. which relies only on kernel evaluations, thus enabling FMMs to a wider range of engineering and scientific problems [12][13]. In KIFMM, the domain is represented by an octree of cells, where interaction lists are built for each cell following Greengard notation [14], namely: U-list, V-list, W-list, and X-list. The KIFMM implements the force evaluation through the following large stages: U-list, Upward, V-list, X-list, W-list, and Downward. These stages are synchronized by global barriers, are embarrassingly parallel, and traverse the tree cells independently (for the list computations) or level-by-level (Upward and Downward). We distinguish two independent flows of computation: the near field *direct evaluation* represented by the U-list computation, and the *far-field approximation* starting from the Upward stage, computing V-list, W-list, and X-list stages and finishing by the Downward stage. We note that the W-list and X-list computations are negligible for a uniform distribution of bodies.

3 Data-Driven Implementation

In this section we discuss the implementation of our data-driven solution. That is, the flow of execution goes from the sources to the targets where the far-field and direct evaluation computations are merged into a single flow by starting the Upward and the direct evaluation at the same time.

3.1 From Target Centric to Source Centric

In KIFMM, the data structures are built from a target centric point of view, thus these data structures need to be rebuilt from a source centric view to enable a data-driven execution. Although in theory if a cell A interacts with cell B, B

will interact with A whether symmetrically (U and V lists) or dually (W and X lists), in practice it depends on how a cell's neighbors are determined. Indeed, a cell's interaction lists are only built around a neighborhood, and in KIFMM this neighborhood does not ensure bidirectional interactions between two cells. In order to maintain a correct behavior of the algorithm in a data-driven execution, we compute these lists from a source point of view.

3.2 Thread-Based Data-Driven Implementation

The dependencies in the data-driven execution can be seen as a producer-consumer synchronization problem as shown in the simplified FMM far-field computation task dependency graph in Figure 1.(a). In our implementation, each task is aware of the tasks that depend on it and may trigger their execution upon termination. Moreover, the task dependencies are satisfied using a combination of *recursive calls* and *atomic counters*. For instance, an atomic counter is used at the `Down` task dependency in Figure 1.(a) which is updated by other `Down` tasks or `V` tasks. In the following we give an example on how a V-list task is executed for a source cell (`src`) after it was called by an `Up` task:

```
void* V (src){
  for(trg in Vlist(src))              //Compute the contribution of src
  {                                     to all the target cells that
     compute_V(trg,src);                depend on it
     trg.down_counter++;              //Atomic incrementation of
                                        the synchronization counter
     /* Test if all dependencies are satisfied */
     if(trg.down_counter = nb_input_depend(trg))
           create_task(Down, trg);    //Start Down computation.
     }
  }
}
```

This pseudo-code shows the V-list and Downward computation tasks (`V` and `Down` resp.) and the target's synchronization counter (`trg.down_counter`) between them. In the Massivethreads library, tasks are embedded in lightweight threads scheduled to be executed by *workers*. Each *worker* is an OS-thread and has a private queue of ready tasks which is managed by a LIFO (Last In First Out) scheduler and a FIFO (First In First Out) work stealing policy between *workers* is adopted. As a result, the `Down` task will be executed first and the `V` task goes at the front of the worker's ready queue. We note that the creation of tasks is incremental and done at the worker level, while the first created tasks, which are situated at the back of the ready queue, may be stolen by other workers ensuring good load-balancing. Since each *worker* is scheduling the tasks to be executed independently from the others and uses a private task queue, this method results in a *distributed scheduling* scheme avoiding a *centralized scheduler* that will constitute a potential bottleneck. We note that, being oblivious of which stage in KIFMM, contributions from many cells may be reduced at a target cell. While this is naturally serialized in the original target approach, in our

source approach, we serialize these updates by using the *locks* provided by the lightweight thread library.

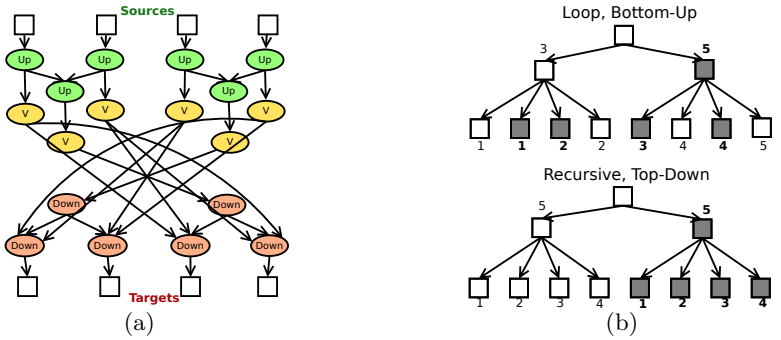


Fig. 1. (a) Simplified FMM far-field computation task dependencies. (b) Simple example of the Upward tasks executed by two *workers*: white for the first *worker* and gray for a second *worker*. The numbering shows a possible task execution order.

3.3 Effects of the Data-Driven FMM on Data Locality

The dependency between a `V` and a `Down` task, as described in Section 3.2, corresponds to a *read* after *write* hazard and results in temporal data reuse. Such data reuse can be observed along the paths going from sources to targets. However assessing the spatial data reuse is more subtle since it depends on how the tasks are scheduled. First, one may implement the task graph of Figure 1.(a) by traversing the leaf boxes and spawning the Upward and V-list tasks. This method requires *synchronization counters* where each Upward task atomically increments its parent's *counter* upon termination, and triggers the Upward task of its parent if it is the last child. In addition, *workers* will likely access non-contiguous cells in the tree. Indeed, the *serial* code to create all the leaf tasks is also considered as a task which will be preempted and put in the *worker*'s ready queue. A second *worker* will steal that task and create the second leaf task and so on. As a result, the *workers* will access randomly the data as shown by the upper part of Figure 1.(b).

To overcome this issue, we use a top-down recursive algorithm to spawn the tasks as shown in the lower part of Figure 1.(b). In this approach, the work stealing happens in the upper levels of the tree and results in a sub-tree working-set per *worker*, thus, ensuring a better spatial and temporal locality and avoiding additional synchronization variables.

4 Test-Bed Configuration

We choose to follow the same input problems as in [4]. That is, we simulate the evaluation of a single step with 4 million bodies following two distributions: a unit

cube uniform and an elliptical non-uniform distribution. As for the interaction kernel we use the Laplace kernel. For each target machine, we manually tune the maximum number of bodies per cell parameter. We only consider double-precision computation because of its higher pressure on the memory subsystem and the document space limitations. As for the target architectures, we select representatives of NUMA multi-core architectures, with a 2 socket Intel Sandy-Bridge-EP, a 4 socket Intel Nehalem-EX, and a 4 socket 8 NUMA-nodes AMD Magny-Cours with their detailed specifications given in Table 1.

Table 1. Target machine specifications. We report the memory bandwidth as the maximum value achieved by the Stream benchmark [15].

	Sandy-Bridge-EP	Nehalem-EX	Magny-Cours
Processor	Xeon E5-2620	Xeon X7550	Opteron 6172
CPU Frequency (Ghz)	2.0	2.0	2.1
# Sockets	2	4	4
# NUMA-Nodes	2	4	8
#Cores/NUMA-Nodes	6	8	6
L3 Cache size (MB)	15	18	6-1
Memory BW (MB/s)	52590.4	68827.3	74720.4
Compiler	GCC 4.4.6	ICC 11.1	GCC 4.4.5

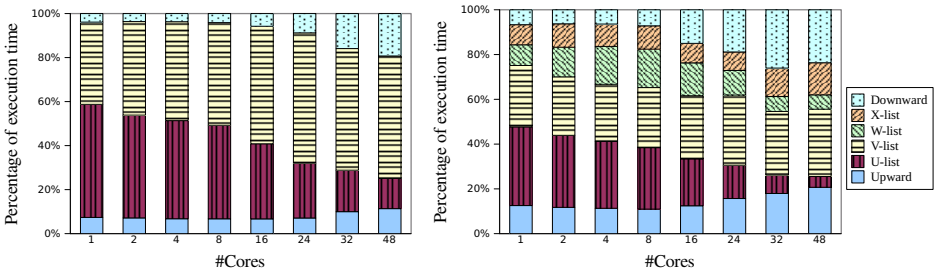
5 Performance Evaluation

In this section we will conduct a performance analysis of the original KIFMM design approach as described in [4] and [5]. However we do not consider the intermediate and advanced tuning techniques introduced in [5], as these techniques can also be adapted for a data-driven execution. Our methodology is guided by the high-level knowledge of the application and also relying on hardware performance monitoring tools. For the latter purpose, we use the Vampir tool-set [16][17] combined with native hardware counters accessible through the PAPI library [18]. In addition, we use the VTune tool to report memory-bandwidth measurements on the Sandy-Bridge-EP machine [19].

5.1 FMM Stages at Large Scale

It is well known in the FMM literature that the U-list and V-list computations dominate the serial execution time. However, after parallelization, not all of the stages scale in the same way, and the dominant stages at larger scale may differ. To verify our assumptions, we run strong scaling simulations using the original implementation on the Magny-Cours machine and we report the percentage of execution time taken by each stage as shown in Figure 2. These results were reported using high resolution timers without tracing the execution in order to

avoid unnecessary overheads. We observe that although the U-list stage takes the longest time when running sequentially, at full concurrency it takes the smallest amount of time while the opposite is observed for the other stages. In the following section, we shed light on the reasons behind this disparity in parallel efficiency of the stages while performing a deep comparative analysis of both FMM implementations on the target machines.



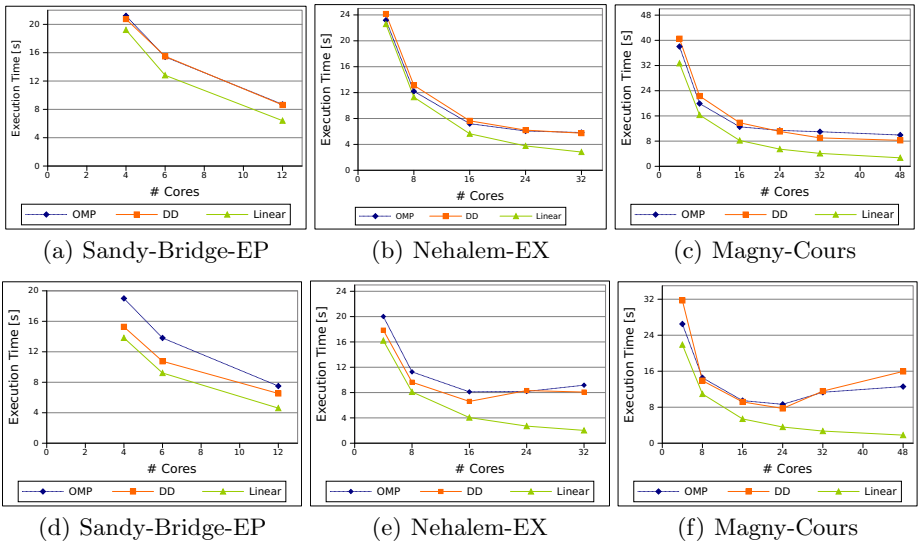
(a) Uniform distribution (b) Elliptical distribution

Fig. 2. Percentage of execution time for each stage on a the Magny-Cours machine for uniform and elliptical distributions

5.2 Comparative Analysis

To decrease the negative effects of NUMA in the data-driven execution, we use the `numactl` command to interleave the memory allocation on the NUMA-nodes where there exists a MassiveThreads *worker*. For both implementations, the OS-threads are scattered across the sockets and bound to the cores to optimize the memory bandwidth. Figure 3 shows the strong scaling of each implementation on each machine using both distributions and indicates an overall better scaling of the data-driven execution. However we observe that both implementations exhibit very limited speed-ups after using more than half the cores. In particular, the Magny-Cours machine has the worst scaling likely due to a smaller last level cache. Also, for the elliptical distribution, there is a 22%, 18%, and 10% speed-up of the data-driven execution over fork-join when using half of the cores on the Sandy-Bridge-EP, Nehalem-EX, and Magny-Cours machine, respectively. To better understand this scaling disparity, a deeper analysis of the latter case is performed as it showed the greatest gap between the two methods.

We record statistics for each stage and also for the total force evaluation as shown in Table 2. The computation times do not include scheduling and synchronization overheads thus, the differences between the methods are only due to data movements. We used native counters rather than PAPI preset counters which were not enough to gather the information of interest. Native counters are machine dependent, thus we follow the guidelines of the hardware manufacturers to derive our metrics for the Magny-Cours [20] and the Sandy-Bridge-EP [21]



(a) Sandy-Bridge-EP (b) Nehalem-EX (c) Magny-Cours

(d) Sandy-Bridge-EP (e) Nehalem-EX (f) Magny-Cours

Fig. 3. Strong scaling of the OpenMP fork-join (OMP) and the data-driven (DD) implementations for uniform (a,b,c) and elliptical (d,e,f) distributions. To better appreciate the scaling results we added a linear scaling plot.

machines. However, to the best of our knowledge, similar guidelines are not available for the Nehalem-EX machine, thus we do not report its memory-bandwidth measurements.

We can observe that the data-driven computation time is close to that of the original implementation, indicating that the synchronization overheads eliminated by our method did not detrimentally affect the data locality. In this experiment we observed improvements of 14%, 17.3%, and 14% resp. for the data-driven which deviate slightly from the above mentioned speed-ups due likely to tracing overheads and operating system noise.

We notice that our method ensures a better locality for the Upward computations, which hides the slower V-list execution time. In order to verify the locality benefit of using sub-tree based working-sets, a similar approach was implemented for the Upward stage using the fork-join model by manually partitioning the tree among the threads. The results, as reported in the last row of Table 2, show that the computation runs faster at the cost of a very large synchronization overhead. We also observe that most of the synchronization overheads stem from the X-list and W-list computations. This is not surprising since these computations exhibit the highest variation in the work per cell and results in high load-imbalance. An attempt to fix this by means of a `dynamic` or a `guided` OpenMP scheduler resulted in worsening the data locality and increasing the OpenMP scheduling overhead leading to a longer execution time. This validates our strategy which achieves a better trade-off between locality and synchronization overhead.

Table 2. Computation time (without scheduling and synchronization overheads), OpenMP synchronization overhead, average bandwidth consumption, and relative computation time of the data-driven execution per machine for the elliptical distribution running on half the cores. Note that important information is highlighted. Abbreviations: DD (Data-Driven), SB (Sandy-Bridge-EP), NH (Nehalem-EX), MC (Magny-Cours), and N/A (Not Available).

	Comput. Time(s)			Sync. Overhead(%)			Bandwidth(GB/s)			DD Relative Time(%)		
	SB	NH	MC	SB	NH	MC	SB	NH	MC	SB	NH	MC
U-list	27	30.5	38.5	7	14.8	14	0.1	N/A	0.4	-2.96	-2.30	-11.69
Upward	9.56	15.7	43.2	1.2	0.2	7.36	1.2	N/A	0.68	**-4.60**	**17.20**	**44.44**
V-list	13.42	24.7	36.7	1.8	8	1.34	**6**	N/A	**6.8**	**-8.05**	**-10.12**	**-32.15**
W-list	7.3	8.05	10.67	**56.7**	**62**	**61**	0.1	N/A	0.2	0.00	-4.60	-7.78
X-list	7	7.96	15	**29.4**	**25.5**	**24**	0.1	N/A	0.38	-2.86	-3.27	23.00
Downward	5.9	14.9	51.9	2.1	0.2	1.9	1.8	N/A	0.4	0.00	-0.54	1.54
Total OpenMP	70.18	101.8	195.9	**15.6**	**18.8**	**15**	1.3	N/A	1.8	**-3.59**	**-1.19**	**3.22**
Data-Driven	72.7	103	190	0	0	0	**2.7**	N/A	**4.4**			
Upward static	**9.32**	**13.4**	**18.58**	**55**	**24.5**	**45.3**						

For a uniform distribution, we observed less synchronization overheads, a worse data locality, and more bandwidth consumption, due to a larger V-list computation, which reduces the effectiveness of the data-driven execution.

5.3 Analysis of the Memory Bandwidth Consumption

According to the memory bandwidth measurements of Table 2, most of the memory traffic comes from the V-list stage which is known to be memory bound. The memory behavior of this computation can be explained as follows: V-list target-source interactions can be seen as a sparse matrix pattern with high spatial locality and temporal reuse regions at the diagonal [5]. These regions are limited (roughly half of the total sources for a uniform distribution) while the rest of the sources are streamed in a non-unit-stride fashion. In addition to the source cells, V-list uses translation vectors, which are also accessed in a non-unit-stride pattern, and further increases the working-set size. We conclude that the V-list bandwidth is consumed by streaming a large working-set following mostly a non unit-stride memory access pattern. Furthermore, reading the sources and translation vectors in a NUMA-aware fashion is not guaranteed.

The data-driven execution of FMM resulted in a homogeneous memory bandwidth consumption rather than concentrated only in the V-list computation. Thus, on a machine with a low memory bandwidth, this execution model will help reduce localized high memory traffic and the performance may improve as long as the overall data locality is not severely hindered. However, for our target machines this was not observed when comparing the V-list bandwidth in Table 2 with the Stream bandwidth in Table 1 for each machine. The limited scaling of V-list can be explained by the Roofline model [22]. We draw the Roofline plot for the the Sandy-Bridge-EP machine along with the performance achieved by U

and V-list computations at full concurrency in Figure 4. V-list has a low arithmetic intensity, as opposed to the compute bound U-list, a mixture of unit-stride and non-unit-stride memory accesses, and also local and remote DRAM accesses. Hence, V-list is partially affected by each memory ceiling in the Roofline plot which explains the limited achievable bandwidth and the performance.

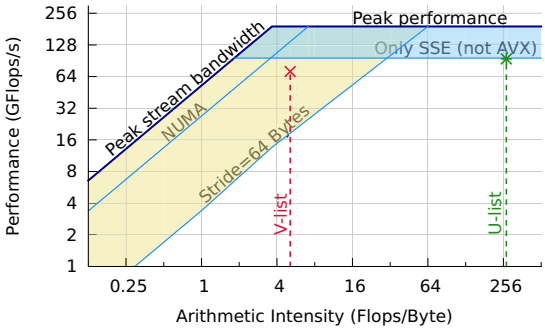


Fig. 4. Roofline of the Sandy-Bridge-EP machine. The NUMA memory ceiling was obtained using the Stream benchmark with only remote accesses, which was then augmented with 64 bytes strided accesses to plot the stride ceiling. We use SSE vector instructions and do not exploit AVX instructions which halves the computational power. The arithmetic intensity of the computations were derived from machine counters.

6 Related Work

Data-driven execution of FMM is not novel. Yokota et al. [23] proposed a data-driven execution of FMM in order to overcome load-balancing issues. Agullo et al. proposed to pipeline the FMM computations on heterogeneous architectures over a runtime [24]. Pericàs et al. implemented a data-driven execution of ExaFMM, a fast open-source FMM [25]. Although these works used a data-driven approach to implement the FMM, their objectives were to load-balance the work among the computational units. Our work also achieves this goal, proposes a novel distributed scheduling scheme, and further presents an in-depth comparison with a fork-join execution model. Using also the MassiveThreads library, Taura et al. described parallel recursions as an alternative to parallel loops for implementing ExaFMM [26]. Our methodology can be applied to this work in order to evaluate the implications of using recursions to implement task parallelism. Towards understanding the performance of multi-core machines, some authors used stencil-computation and sparse matrix-vector multiplication kernels and optimized them for state-of-the-art multi-core architectures[27] [28]. These works use small kernels as benchmarks while going deeper in architectural details in order to get insight into performance trade-offs. In our work, we use FMM, a sizable algorithm which uses multiple kernels, as a benchmark to get insight into the performance of different execution models.

7 Conclusion and Future Work

In this work, we study multi-core architectures' performance given a fork-join and a data-driven execution models. We implemented a data-driven execution of the FMM using a distributed scheduling approach and observed improvements of up to 22% in execution time as compared to the fork-join approach. We concluded that for an algorithm such as a FMM, a data-driven execution is more suitable on our target machines as trading-off the inferior data locality by removing the synchronization overheads was beneficial. The benefit of the data-driven execution grows at scale reaching the best speed-ups with half of the cores, after which both methods are limited by the scalability of the memory intensive kernel. This kernel is not limited by the memory bandwidth in our experiments, but it is rather a combination of low arithmetic intensity and a sparse NUMA pattern that reduces the achievable bandwidth. This work can be extended to analyze the effects of task-coarsening and adapting the advanced optimizations applied to the original OpenMP implementation [5]. Our preliminary attempts in task-coarsening, by aggregating the work of leaf siblings in the tree, resulted in an average 5% speed-up, which encourages pursuing this direction.

References

1. Dehnen, W.: A hierarchical o(n) force calculation algorithm. Journal of Computational Physics 179(1), 27–42 (2002)
2. Chaillat, S., Bonnet, M., Semblat, J.F.: A multi-level fast multipole bem for 3-d elastodynamics in the frequency domain. Computer Methods in Applied Mechanics and Engineering 197, 4233–4249 (2008)
3. Yokota, R., Narumi, T., Barba, L.A., Yasuoka, K.: Petascale turbulence simulation using a highly parallel fast multipole method (2011)
4. Chandramowlishwaran, A., Williams, S., Oliker, L., Lashuk, I., Biros, G., Vuduc, R.: Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures. In: 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS), pp. 1–12 (April 2010)
5. Chandramowlishwaran, A., Madduri, K., Vuduc, R.: Diagnosis, tuning, and redesign for multicore performance: A case study of the fast multipole method. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2010, pp. 1–12. IEEE Computer Society, Washington, DC (2010)
6. Augonnet, C., Thibault, S., Namyst, R.: StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines. Rapport de Recherche RR-7240, INRIA (March 2010)
7. Duran, A., Ayguade, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: Ompss: a proposal for programming heterogeneous multi-core architectures. Parallel Processing Letters, 173–193 (2011)
8. YarKhan, A., Kurzak, J., Dongarra, J.: Quark users' guide: Queueing and runtime for kernels. Technical report, University of Tennessee Innovative Computing Laboratory (April 2011)
9. `http://code.google.com/p/massivethreads/`

10. Barnes, J., Hut, P.: A hierarchical O(N log N) force-calculation algorithm. Nature 324(6096), 446–449 (1986)
11. Greengard, L.F.: The rapid evaluation of potential fields in particle systems. PhD thesis, New Haven, CT, USA, AAI8727216 (1987)
12. Ying, G.L., Biros, Zorin, D., Langston, H.: A new parallel kernel-independent fast multipole method. In: Supercomputing, 2003 ACM/IEEE Conference, p. 14 (November 2003)
13. Ying, L., Biros, G., Zorin, D.: A kernel-independent adaptive fast multipole algorithm in two and three dimensions (2003)
14. Greengard, L.: The Rapid Evaluation of Potential Fields in Particle Systems, vol. 52. MIT Press (1988)
15. McCalpin, J.D.: Memory bandwidth and machine balance in current high performance computers. IEEE Computer Society Technical Committee on Computer Architecture TCCA Newsletter, 19–25 (1995)
16. Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M.S., Nagel, W.E.: The Vampir Performance Analysis Tool-Set. In: Resch, M., Keller, R., Himmler, V., Krammer, B., Schulz, A. (eds.) Tools for High Performance Computing, pp. 139–155. Springer, Heidelberg (2008)
17. Brunst, H., Knüpfer, A.: Vampir. In: Encyclopedia of Parallel Computing. Springer (2011)
18. `http://icl.cs.utk.edu/PAPI/`
19. `http://software.intel.com/en-us/intel-vtune-amplifier-xe`
20. Drongowski, P.J.: Basic performance measurements for amd athlontm 64, amd opterontm and amd phenomtm processors (September 25, 2008)
21. Intel xeon processor e5-2600 product family uncore performance monitoring guide (March 2012)
22. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Commun. ACM 52, 65–76 (2009)
23. Bergstrom, L.: Measuring numa effects with the stream benchmark (2011)
24. Agullo, E., Bramas, B., Coulaud, O., Darve, E., Messner, M., Takahashi, T.: Pipelining the fast multipole method over a runtime system (2012)
25. Pericas, M., Amer, A., Fukuda, K., Maruyama, N., Yokota, R., Matsuoka, S.: Towards a dataflow fmm using the ompss programming model. In: 136th IPSJ Conference on High Performance Computing
26. Taura, K., Yokota, R., Maruyama, N.: A task parallelism meets fast multipole methods. In: Proceedings of SCALA 2012 (November 2012)
27. Datta, K., Kamil, S., Williams, S., Oliker, L., Shalf, J., Yelick, K.: Optimization and performance modeling of stencil computations on modern microprocessors (2009)
28. Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., Demmel, J.: Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, SC 2007, 38:1–38:12. ACM, New York (2007)

VLI – A Library for High Precision Integer and Polynomial Arithmetic

Timothée Ewart[1], Andreas Hehn[2], and Matthias Troyer[2]

[1] Université de Genève, Switzerland
timothee.ewart@unige.ch
[2] Eidgenössische Technische Hochschule Zürich, Switzerland

Abstract. We present a high-performance C++ library for high but fixed precision (128 to 512 bit) integer arithmetic and symbolic polynomial computations. While the large integer and polynomial computation parts of the library can be used independently optimized kernels for symbolic polynomials with large integer coefficients are provided. The kernels were manually optimized in assembly language for the x86-64 and power64 architectures. Our main target application is high-temperature series expansions which requires inner products of large vectors of polynomials with large integer coefficients. For this purpose we implemented a tunable hybrid CPU/GPU inner product function using OpenMP and NVIDIA CUDA with inline PTX assembly. This way we make optimal use of today's and upcoming hybrid supercomputers and attain 49% of the peak performance of the current NVIDIA Kepler GPU. Compared to a pure CPU solution using the GNU Multiple Precision Arithmetic Library (GMP) we gain a speedup of 13x for a pure CPU inner product and 38x using a GPU accelerator.

1 Introduction

High precision integers and symbolic polynomials play an important role in many fields of science. Most notably the basis of modern cryptographic systems like the RSA algorithm [1] rely on high precision integer arithmetic. Here we present a library which we developed with one particular application from statistical and quantum physics in mind. Specifically we target high-temperature series expansions [2], which require symbolic polynomials in multiple variables with arbitrary precise coefficients. These coefficients can be represented as integers whose maximal value can be determined before the actual calculation and do not exceed 512 bits. The computational hot spot of the method are inner products of large vectors of such symbolic polynomials. Since this method is computationally expensive and offers multiple levels of high parallelism, we require optimized kernels for these operations to make optimal use of today's supercomputers and not waste valuable resources.

Many multiple precision libraries, like the popular GNU Multiple Precision Arithmetic Library (GMP) [3] or the Number Theory Library (NTL) [4], provide an arbitrary precise integer class, which is dynamic in size and optimized to cover

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 267–278, 2013.

a wide range of integer sizes up to several thousand bits. Since the size of the integers in our application ranges only from 128 bit to 512 bit and the maximal size is known beforehand, the dynamic size poses a significant overhead and much more optimized implementations are possible. The Number Theory Library (NTL) also offers a polynomial class with large integer coefficients. However, it does not support multivariate polynomials.

While the existing libraries may be optimized for common CPU architectures, current trends in supercomputing require applications to exploit massively multi-threaded GPU accelerators to benefit from their enormous computational capacity. Previous studies successfully explored polynomial multiplications on GPUs using FFT methods [5,6,7]. Here we are interested in the inner product of vectors of polynomials, which offers an additional level of parallelism. In order to meet the special requirements of the GPU on the memory layout the different components (vector, polynomial, high precision integer) need to be closely connected in order to interleave them for efficient parallel kernels. As this close connection is almost impossible if one of the components is provided by an external library, we decided to develop our own library to achieve best interoperability and not be restricted by external design decisions. While maintaining this high interoperability, the high precision integer part and the symbolic polynomial part of the library are designed in such a way that they can also be used independently. In the following sections we will show key features and implementation decisions of these components and benchmark the library against a GMP solution.

2 Large Integers

The library provides a template class `integer<n>` to represent signed integers with a fixed size of $128 \leq n \leq 512$ bits. We implemented all standard integer operations starting from basic arithmetic operations, comparison operations to bit operations, like bit-shift or bit-wise logical operations. This way objects of the class can be used like the regular C++ type `int` in most cases. In addition to these standard integer operations we also provide a multiply-add function and an extended multiplication which doubles the number of bits when two integers of the same size get multiplied.

The data of the integer is stored in an array of 64 bit unsigned integer segments, where we use a standard two-complement representation of the number. Due to the fixed size of the integer, we are able to allocate the memory for the data on the stack. This way we save expensive heap allocations without implementing a manual memory pool management. For the basic arithmetic operations like addition and multiplication we use standard schoolbook algorithms, since the integers are too small for Toom [8] or FFT based approaches like the Schönhage-Strassen algorithm [9]. An exception is the 256 bit multiplication on the GPU where we use the Karatsuba algorithm [10] as we will explain later. We implemented the schoolbook algorithms in optimized assembly code to make optimal use of the hardware features that are not accessible from C++, like the carry addition `adcq` or the 64 bit multiplication `mulq` which calculates the low

and high 64 bit part of the product on the x86-64 architecture. Low-level programming methods for high-precision integer arithmetic and assembly tips can be found in the famous book "The Art of Assembly Language" [11]. Since writing assembly code can result in very long error-prone code, we generate many parts of the assembly code with the C++ preprocessor using the `BOOST_PP` library.

3 Polynomials

The second part of the library is a template class

`polynomial<CoeffType, Structure<N>, Var0, Var1, Var2, Var3>`

for symbolic polynomials in 1 to 4 variables

$$p(x, y, z, w) = \sum_{i,j,k,l} c_{ijkl} \cdot x^i y^j z^k w^l \tag{1}$$

having either a "dense" $(i, j, k, l \leq N)$ or a "triangular" $(i+j+k+l \leq N)$ structure. The truncation order N is fixed at compile time. The coefficients c_{ijkl} of the polynomial may have any C++ type supporting basic arithmetic operations. The polynomial itself supports additions, subtractions and multiplications with other polynomials or monomials. We also provide a special multiplication function returning a polynomial with truncation order $2N$, such that no terms are dropped. It is also possible to mix and match polynomials having different sets of variables. The coefficients will be automatically mapped to the corresponding symbols during compile time. All operations call free hook functions with default implementations for all coefficient types. These functions may be overloaded by the user to provide optimized implementations for specific coefficient types. The user may for example add a specialized implementation for `float` using Streaming SIMD Extension intrinsics without touching the polynomial class itself. In addition to the general default implementations of the operations the library provides optimized routines for large integer coefficients `integer<n>`.

4 Optimized Inner Product

Since the hot spots of our main target application are inner products of large vectors with symbolic polynomials as components, where the coefficients of the polynomials are large integers, we implemented an optimized inner product function for this special case. All polynomials of the vector are assumed to have the same structure, the same truncation order N and the same `integer<n>` type as coefficients. The inner product will result in a polynomial of twice the order of the original polynomials with coefficients of twice the width (`integer<2n>`) of the original coefficients. To employ today's supercomputers as efficiently as possible the inner product is a hybrid CPU/GPU implementation, where the inner product is split into a part calculated on the CPU and a part calculated on the GPU. The ratio of these two parts can be set when building the library. It is also possible to deactivate the GPU part completely and compile a pure CPU version.

4.1 CPU Implementation

On the CPU we perform the inner product by element-wise polynomial-polynomial multiplications using the implementation of the polynomial. This operation is easily parallelized using OpenMP by splitting and distributing equal-sized chunks of the vectors among the available threads. Since all polynomials are of the same structure the work is well balanced between the threads.

The multiplication of two polynomials itself seems well suited for a SIMD implementation processing multiple coefficients in parallel. However, this is not advisable for large integers on current x86-64 architectures, even if the operations on the large integers are entirely independent. Using the Streaming SIMD Extensions (SSE) all kernels need to be based on 32 bit integer operations instead of 64 bit operations which are available for the sequential implementation. This reduction of segment size increases the number of required multiply instructions for a large integer multiplication by a factor of 4. The current SSE instruction set only supports an integer SIMD multiplication for two pairs of 32 bit numbers, each yielding a 64 bit result. Thus a SIMD implementation using SSE will be two times slower than our serial version using 64 bit integers. Note that we also neglected the carry bit propagation of the required additions. While the sequential version benefits from the hardware support for carry bit propagation, an SSE implementation needs to handle the carry manually.

The upcoming AVX2 instruction set will feature integer SIMD operations with twice the vector width of SSE. However, it will neither support 64 bit integer multiplications nor carry bit propagation. Therefore it will be at most as fast as our sequential version.

4.2 GPU Implementation

While on the CPU a SIMD approach is not promising, the highly parallel structure of the problem is well suited for GPU accelerators. In a nutshell GPUs offer a hybrid between SIMD and massively multithreaded execution, where the GPU schedules thousands of threads in so called warps. A warp is a group of usually 32 threads which are executed in a lockstep manner. We implemented the inner product in NVIDIA CUDA to exploit this powerful architecture. Our target hardware is the NVIDIA Tesla K20X. We perform element-wise polynomial multiplications where each thread calculates one coefficient

$$c_{IJKL} = \sum_{i,j,k,l} a_{ijkl} \cdot b_{I-i,J-j,K-k,L-l} \tag{2}$$

of the product polynomials, where a_{ijkl}, $b_{i'j'k'l'}$ are the coefficients of the polynomials to be multiplied. Once all coefficients are calculated we perform a reduction over the result vector to obtain the final result of the inner product. This way we avoid race conditions and minimize the number of synchronization points. However, this method leads to a load inbalance since the number of terms in the sum (2) depends on the orders I, J, K, L of the resulting coefficient. To overcome this problem we set up an execution plan before the actual calculation.

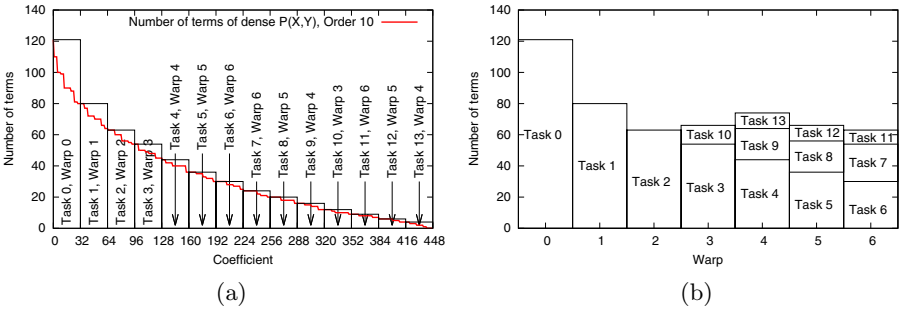
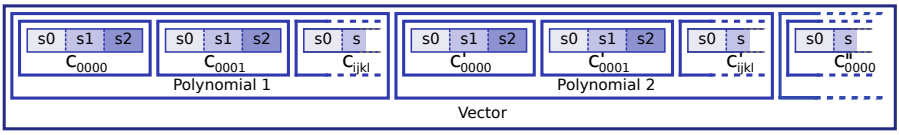


Fig. 1. (a) Number of contributions to each coefficient of the resulting polynomial (red line) sorted and split into tasks. (b) Load balancing of the tasks over the warps.

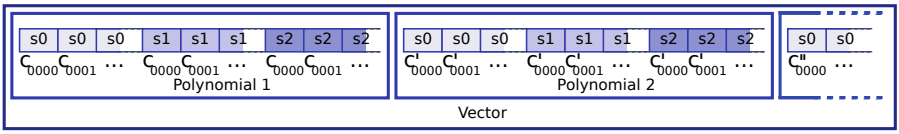
Therefore we determine the number of terms of each coefficient of the product polynomial, sort them according to this number and assemble groups of 32 coefficients. Starting with the most expensive group the groups get scheduled on the available warps balancing the work among them (Fig. 1). Note that by sorting the coefficients we also minimize the work-load difference within a warp, such that the threads within the warp idle as little as possible until the other threads finished their lockstep calculation.

Once the `inner_product` function is called the vectors to be multiplied are copied asynchronously from the host to the device. We store the data in the texture memory to take advantage of the texture cache which is optimized for 2D access. The intermediate results, i.e. the vector of the product polynomials, is written to the global memory. At this point we change the memory layout from the usual Array (vector) of Array (polynomial) of Structures (large integer) to an Array of Structures of Arrays where the data segments of the large integer coefficients are interleaved within the polynomial (Fig. 2), such that the least significant segments of all coefficients are contiguous in memory followed by the next more significant segments. This memory layout allows for an efficient coalesced access during the reduction at the end of the calculation. Once the reduction is completed the result is transferred from the device memory back to the host using the regular memory layout.

Like for the SSE approach on the CPU our kernels on the GPU rely on 32 bit unsigned integer arithmetic. However, on the GPU we were able to benefit from hard-wired addition with carry (`addc.cc`) and multiply-add with carry (`madc.cc`) operations using NVIDIA parallel thread execution (PTX) inline assembly. Contrary to the CPU where we used only schoolbook algorithms, we employ the Karatsuba algorithm [10] for the 256 bit integer multiplication on the GPU. Even though the method does not require less operations than the schoolbook algorithm it is advantageous on the GPU, because it uses less multiplications and more additions on the 32 bit segments, which have a 5 times higher throughput on the NVIDIA Kepler [12].



(a) Memory layout - CPU



(b) Memory layout - GPU

Fig. 2. Memory layout of a vector of polynomials with large integer coefficients. Each coefficient c_{ijkl} consists of 64 bit segments `s`. On the CPU the coefficients are stored in nested arrays. Each coefficient is stored contiguously. For the intermediate results on the GPU, the coefficients are stored in an interleaved way grouping the segments according to their significance. This assures coalesced memory access. Note that on the GPU we will have segments of only 32 bit width, which doubles the number of segments compared to the 64 bit CPU version.

5 Benchmarks

5.1 Simple Integer Operations

We benchmarked the large integer part of the library on an Intel Sandy Bridge node and a Power7 node. We compared against the commonly used GNU Multiple Precision Arithmetic Library (GMP) version 5.1.1. The system configuration and compilation information is given in table 1. The libraries were tested and validated by the GMP benchmark. Comparing the achieved GMPbench scores to reference values on the GMP web page [3] for slightly different systems shows good agreement. The benchmarks and the VLI library were compiled with `-O2 -m64`. Comparing the performance of simple addition and multiplication operations (Tab. 2) our implementation for fixed size integers is between 10% and

Table 1. Benchmark system configuration

	Sandy Bridge
CPU	Intel Xeon E5-2670 (16 physical cores, 2.6 GHz)
Compiler	GCC 4.7.1
GMP compile flags	`-O2 -fomit-frame-pointer -m64 -march=corei7`
GMPbench 0.2 score	41580 (multiply), 1245 (full score/Ghz)
	Power7
CPU	Power7 720 Express (8 physical cores, 3.0 GHz)
Compiler	GCC 4.7.1
GMP compile flags	`-O3 -m64 -mtune=power7`
GMPbench 0.2 score	26040 (multiply), 670 (full score/Ghz)

320% faster for the addition and up to 150% faster for the multiplication. We obtain larger speed-ups for the addition of small high precision integers, since the overhead to manage the dynamic integer size in GMP is approximately independent of the integer size. Since our integer library is stack based we do not need expensive calls to `malloc` like GMP, which will give an additional speedup for the inner product, where multiple allocations are necessary.

Table 2. Performance of the GNU Multiple Precision Arithmetic Library and the VLI library for addition (+) and multiplication (×) in 10^6 large integer operations/s. Mean and standard deviation from 100 runs.

	Power7			**SandyBridge**		
op.	Performance [10^6 op/s]		Speed-Up	Performance [10^6 op/s]		Speed-Up
	GMP	VLI		GMP	VLI	
128 bit +	58.2 ±1.3	107 ±1.4	×**1.8**	96.9 ±0.1	405 ±0.3	×**4.2**
192 bit +	61.1 ±1.2	79.4 ±2.9	×1.3	94.1 ±0.0	268 ±0.2	×2.8
256 bit +	59.2 ±1.4	66.7 ±1.7	×1.1	94.1 ±0.0	143 ±0.1	×1.5
320 bit +	53.5 ±1.1	59.7 ±1.0	×1.0	86.7 ±0.1	97.0 ±0.1	×1.1
384 bit +	51.7 ±0.9	64.6 ±1.6	×1.2	84.5 ±0.0	122 ±0.1	×1.4
448 bit +	54.7 ±1.2	56.7 ±1.1	×1.0	86.5 ±0.1	114 ±0.1	×1.3
512 bit +	52.5 ±1.0	50.0 ±3.3	×1.0	82.2 ±0.0	106 ±0.1	×1.3
128 bit ×	31.2 ±0.4	103 ±2.7	×**3.3**	81.9 ±0.1	126 ±0.1	×1.5
192 bit ×	18.6 ±2.4	61.4 ±1.0	×3.3	37.1 ±0.1	94.1 ±0.1	×**2.5**
256 bit ×	16.0 ±2.2	34.9 ±0.5	×2.2	30.1 ±0.1	54.6 ±0.0	×1.8

5.2 Optimized Inner Product

The benchmarks for the inner product were again performed on the Intel Sandy Bridge node for the pure CPU version of the inner product. The GPU benchmarks were performed on Todi, a Cray XK7 with NVIDIA Tesla K20X GPUs and AMD Opteron CPUs, at the Swiss Center for Scientific Computing (CSCS). The test cases are inner products of two vectors of dimension 4096 with "dense" and "triangular" polynomials of order 1 to 14 with 128 and 256 bit integer coefficients. Since the inner product will double the order of the polynomial and the width of the large integer coefficients, the test cases will result in polynomials of order 2 to 28 with 256 and 512 bit integer coefficients, respectively.

CPU Implementation. Figures 3a and 3b show the performance behavior of the inner product for polynomials with 128 bit integer coefficients for different truncation orders. Let us first focus on the pure CPU based inner products, which are represented by dashed lines. The naive implementation using the GNU Multiple Precision Arithmetic Library (GMP) and OpenMP performs rather poorly reaching $0.1 \cdot 10^9$ large integer operations per second for any polynomial type. As suggested before a major reason for the poor performance is the dynamic

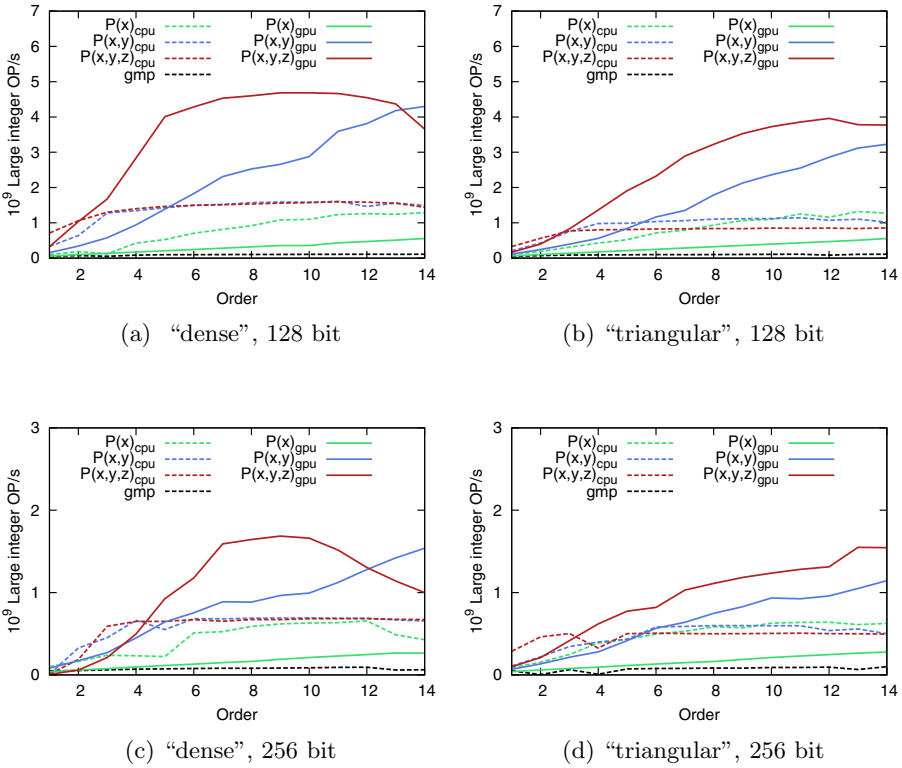
Table 3. VTune Performance profile of the inner product using GMP on Intel Sandy Bridge for polynomials of order 10 in three variables.

Function	Time [%]
`__gmp_default_reallocate`	**34**
`__gmp_default_allocate`	**14**
`vli::detail::inner_product_cpu`	10
`__gmpz_add`	10
`__gmpz_mul`	9
`__gmpn_mul_2`	7
`__gmpz_realloc`	**3**
`__gmpz_init`	3
`__gmpn_add_n`	2
`__gmpn_sub_n`	2

memory management of GMP. A VTune profile analysis (Tab. 3) reveals approximately 50% of the execution time is spent in memory allocation.

Using our optimized CPU based inner product, we reach a limit of $1.6 \cdot 10^9$ large integer OP/s for "dense" polynomials in two and three variables for almost any order. The performance of the optimized inner product for univariate polynomials remains below this limit and grows slowly with the order of the polynomial, because the amount of work per thread is too low and the computation is dominated by spawning the OpenMP threads. The performance behavior of the "triangular" polynomials is similar (Fig. 3b), however the upper limit depends on the number of variables of the polynomial. We achieve $1.1 \cdot 10^9$ large integer OP/s for polynomials in two symbolic variables, and only $0.9 \cdot 10^9$ large integer OP/s for three symbolic variables. This difference is due to the more complex index calculations required to map the "triangular" structure to the linear memory layout. In conclusion we managed to gain a speed-up of 13x for "dense" polynomials and up to 10x for "triangular" polynomials using our CPU based optimized inner product and large integer functions.

GPU Implementation. Using the optimized inner product only on GPUs yields an even higher performance (Fig. 3, solid lines). Note that the performance estimate includes the time needed for the data transfer between the host and the device. For very low orders as well as for univariate polynomials, the GPU is outperformed by the CPU implementation, since these calculations do not offer enough parallelism to saturate the available number of threads on the GPU. The performance, however, increases with increasing order until it reaches a plateau at $4.6 \cdot 10^9$ large integer OP/s for a 7th order "dense" polynomial in three variables (Fig. 3a). This increase is mainly due to the growing number of coefficients which allow us to employ more threads and eases the load balancing as discussed in the previous section. Beyond 13th order the performance decreases again, since the texture memory cache is saturated and the number of cache misses increases. For polynomials in two variables this threshold is not



(a) "dense", 128 bit

(b) "triangular", 128 bit

(c) "dense", 256 bit

(d) "triangular", 256 bit

Fig. 3. Comparison of the inner product with and without GPU accelerator. Inner product of "dense" polynomials (a and c) and "triangular" polynomials (b and d) up to 3 variables, with 128 bit (a and b) and 256 bit (c and d) coefficients. Size of the vector 4096. GMP gives similar results independent of the polynomial structure.

reached within our benchmark. The performance of the inner product for "triangular" polynomials increases less with increasing order in general, as the number of coefficients grows slower and the calculation of the memory location is more complex than for "dense" polynomials. Compared to the naive GMP solution we achieve speed-ups of up to 38x and 33x for "dense" and "triangular" polynomials, respectively. This corresponds to an additional speed-up of 3x with respect to our pure CPU inner product on the Sandy Bridge node.

The performance profile of the inner product for polynomials with 256 bit integer coefficients (Fig. 3c and 3d) is very similar to the 128 bit version. Since the 256 bit integer multiplication requires approximately four times the number of operations of the 128 bit integer multiplication, one would expect at most 1/4 of the large integer operation performance of the 128 bit operation. However, the number of large integer operations per second is only reduced by a factor of

approximately 2/5. The reason for this non-linear behavior is twofold. First, the load of the data is only twice as expensive for the 256 bit operation, since during the operation all data is kept in registers and each number needs to be loaded only once per large integer multiplication. This is true for the CPU and the GPU. The second reason is architecture dependent. The CPU can employ instruction level parallelism techniques more effectively, since the 256 bit multiplication contains more independent operations than the corresponding 128 bit version. On the GPU we profit from the Karatsuba algorithm for 256 bit multiplications trading slow 32 bit integer multiplications for additions, which have a much higher throughput on the GPU. Thus the additional operations can be executed in a more efficient way.

5.3 Efficiency of the Optimized Inner Product

To estimate the efficiency of our GPU kernel we computed the number of 32 bit integer operations per second (IOP/s) for the inner product with 128 bit integer coefficients. We only considered the instructions of the polynomial-polynomial multiplication and neglect the reduction at the end of the inner product. Since to our knowledge there exist neither benchmarks nor official peak values for the integer performance of the NVIDIA Kepler GK110, we calculate a theoretical limit based on the instruction throughput of the 32 bit integer multiply-add (`madc`) instruction [12] which is used almost exclusively in our multiplication kernel. The theoretical maximum of the NVIDIA Tesla K20X is given by

$$\begin{aligned} n_{\mathrm{SMX}} \cdot f \cdot \mu \cdot r &= 14\,\mathrm{SMX} \cdot 732 \cdot 10^6\,\frac{\mathrm{cycle}}{\mathrm{s}} \cdot 32\,\frac{\mathrm{instr}}{\mathrm{SMX}\cdot\mathrm{cycle}} \cdot 2\,\frac{\mathrm{IOP}}{\mathrm{instr}} \\ &= 655\,\mathrm{GIOP/s}\,, \end{aligned} \tag{3}$$

where n_{SMX} is the number of streaming multiprocessors (SMX), f denotes the clock frequency, μ is the instruction throughput of the `madc` instruction and r is the number of integer operations per instruction. Since GPUs are optimized for single precision floating point arithmetic, the theoretical integer performance of the NVIDIA Tesla K20X is just 655 GIOP/s, which is far less than the floating point performance of 3.95 TFLOP/s. The maximal performance of our implementation of $4.6 \cdot 10^9$ large integer OP/s in figure 3 corresponds to $4.6 \cdot 10^9\,\mathrm{OP/s} \cdot 32\,\mathrm{instr/OP} \cdot 2\,\mathrm{IOP/instr} = 294\,\mathrm{GIOP/s}$ (32 bit). This number includes the time needed to transfer the data between the CPU memory and the GPU memory. In an independent measurement we determine the pure kernel performance to be 319 GIOP/s. We therefore reach 49% of the theoretical peak performance of the NVIDIA Tesla K20X.

6 Summary and Outlook

We presented a C++ library for efficient fixed high precision integers up to 512 bits and polynomials in 1 to 4 symbolic variables with an optimized inner

product function. The purely CPU based high precision integer part of the library outperforms the GNU Multiple Precision Arithmetic Library (GMP) by a factor up to 4.2x for the integer addition and 2.5x for the integer multiplication. The optimized hybrid CPU-GPU inner product function for vectors of polynomials with high precision integer coefficients in pure CPU mode showed an excellent speed-up factor of up to 13x compared to a naive solution using GMP and OpenMP. Using a single NVIDIA Tesla K20X we were able to push this speed-up factor to 38x. While the library was originally intended as a special purpose library for high-temperature series expansions, its modular design offers a high flexibility and renders the library also interesting to other applications in science and engineering.

We would like to extend the large integer class to allow for sizes up to 2048 bits, which will make it more attractive to cryptography applications. The polynomial part of the library should allow for more complex structures, like individual truncation orders for each symbolic variable, and could be extended to support lazy evaluation. Until now the hybrid inner product function requires the library user to specify the ratio that is used to split the vectors to be multiplied between GPU and CPU when building the library. In future versions we would like to provide a tool to generate a look-up table based on a small benchmark to select the appropriate ratio automatically. Beside the optimized kernels for the x86-64, power64 and Kepler architectures we would also like to explore the new Intel Xeon Phi SIMD architecture, which supports carry propagation natively and might be a better candidate for integer arithmetic than current GPUs.

The library is released under the Boost Software License and available at `http://www.comp-phys.org/vli/`.

Acknowledgments. We would like to thank Maxim Milakov (NVIDIA), Peter Messmer (NVIDIA), William Sawyer (CSCS), Gilles Fourestey (CSCS), Lukas Gamper (ETH Zürich) and Thierry Giamarchi (Université de Genève) for their helpful advices, insights and resources. The project has been supported by the Swiss Platform for High-Performance and High-Productivity Computing (HP2C).

References

1. Rivest, R.L., Shamir, A., Adleman, L.: A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Commun. ACM 21(2), 120–126 (1978)
2. Oitmaa, J., Hamer, C., Zheng, W.: Series Expansion Methods for Strongly Interacting Lattice Models. Cambridge University Press (2006)
3. Granlund, T., The GMP development team: GNU MP: The GNU Multiple Precision Arithmetic Library. 5.1.1 edn. (2012), `http://gmplib.org/`
4. Shoup, V., The NTL development team: NTL: A Library for Doing Number Theory, `http://www.shoup.net/ntl/`
5. Emeliyanenko, P.: Efficient Multiplication of Polynomials on Graphics Hardware. In: Dou, Y., Gruber, R., Joller, J.M. (eds.) APPT 2009. LNCS, vol. 5737, pp. 134–149. Springer, Heidelberg (2009)

6. Govindaraju, N.K., Lloyd, B., Dotsenko, Y., Smith, B., Manferdelli, J.: High Performance Discrete Fourier Transforms on Graphics Processors. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, SC 2008, pp. 2:1–2:12. IEEE Press, Piscataway (2008)
7. Maza, M.M., Pan, W.: Fast Polynomial Multiplication on a GPU. Journal of Physics: Conference Series 256(1), 012009 (2010)
8. Toom, A.J.: The Complexity of a Scheme of Functional Elements Simulating the Multiplication of Integers. Soviet Mathematics 3, 714–716 (1963)
9. Schönhage, A., Strassen, V.: Schnelle Multiplikation grosser Zahlen. Computing 7, 281–292 (1971)
10. Karatsuba, A., Ofman, Y.: Multiplication of Multidigit Numbers on Automata. Soviet Physics - Doklady 7, 595–596 (1963)
11. Hyde, R.: The Art of Assembly Language. No Starch Press (2003)
12. NVIDIA Corporation: CUDA C Programming Guide v5.0 (2012)

Performance-Portable Finite Element Assembly Using PyOP2 and FEniCS

Graham R. Markall[1], Florian Rathgeber[1], Lawrence Mitchell[2], Nicolas Loriant[1], Carlo Bertolli[3], David A. Ham[1], and Paul H.J. Kelly[1]

[1] Imperial College London, UK
[2] EPCC, University of Edinburgh, UK
[3] IBM T.J. Watson Research Center, NY, USA

Abstract. We describe a toolchain that provides a fully automated compilation pathway from a finite element domain-specific language to low-level code for multicore and GPGPU platforms. We demonstrate that the generated code exceeds the performance of the best available alternatives, without requiring manual tuning or modification of the generated code. The toolchain can easily be integrated with existing finite element solvers, providing a means to add performance portable methods without having to rebuild an entire complex implementation from scratch.

1 Introduction

FEniCS [1] is a toolchain that is widely used by scientists and engineers for developing finite element models. The models are specified in the Unified Form Language (UFL), a domain-specific language for writing finite element variational forms. UFL forms are translated into C++ code conforming to the Unified Form-Assembly Code (UFC) specification by the FEniCS Form Compiler (FFC), then just-in-time (JIT) compiled and linked back into the DOLFIN finite element library, where they are executed. This mechanism provides rapid prototyping and development of finite element solvers that execute with high performance on CPU architectures, but does not presently provide performance-portable code for accelerators and heterogeneous architectures. Given the current prevalence of multi- and many-core architectures, it is desirable to find an alternative intermediate representation that provides good performance on these architectures. We propose PyOP2 as a suitable representation for this purpose.

PyOP2 [2] is a framework for performance-portable parallel computations on unstructured meshes. PyOP2 code consists of data declarations to identify the entities of a mesh, and kernels, which define a portion of code that is executed for every mesh entity. Finite element computations consist of executing the same assembly function for every element of a mesh and naturally fit into the PyOP2 model.

We have extended the FEniCS Form Compiler (FFC) to be able to generate PyOP2 kernels instead of UFC code. The software stack that performs the translation from UFL forms to target-specific code is shown in Figure 1. The UFL

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 279–289, 2013.

library performs preprocessing of the variational forms which are then translated into PyOP2 kernels. These local assembly kernels are further translated into a target-specific implementation and are scheduled accordingly by the PyOP2 runtime when parallel loops are executed. PyOP2 organizes the data structures such that the required data is passed to each invocation of the assembly kernel.

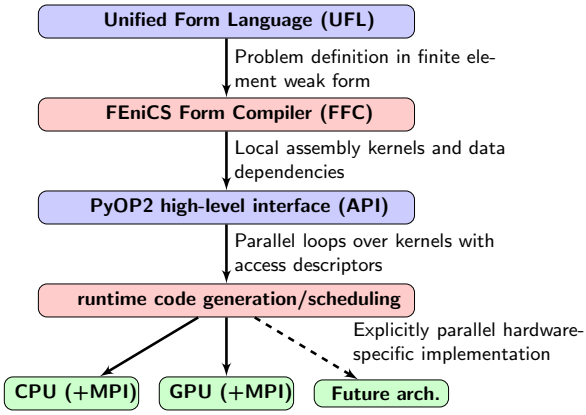


Fig. 1. Overview of the UFL/PyOP2 tool chain

To demonstrate the performance-portability achieved by the toolchain, a single-source implementation of a finite element solver for an advection-diffusion problem has been benchmarked on CPU and GPU targets and compared against a DOLFIN implementation of the same problem. The generated implementation has been integrated into Fluidity [3], a computational fluid dynamics package written in Fortran 90, which demonstrates how the toolchain can be integrated into existing complex finite element codes.

The work in this paper builds on our preliminary experimental work presented in [2], which showed the feasibility of generating code for multiple platforms from the UFL source, by demonstrating the C and CUDA backends with structured meshes. The specific contributions of this paper are:

- We present experimental results using the toolchain's CUDA, OpenMP and MPI backends using unstructured meshes, which provide a much more representative benchmark than in our previous work.
- We demonstrate that the performance provided by the toolchain exceeds the best available alternatives by benchmarking an advection-diffusion problem using our toolchain, the FEniCS toolchain, and Fluidity's built-in advection-diffusion solver.

We begin by briefly introducing the computations performed in implementations of the finite element method in Section 2, then describe in detail the abstraction

layers in Section 3. We give an exposition of the experiments performed using this framework in Section 4. We discuss related work in Section 5 and conclude in Section 6.

2 The Finite Element Method

A partial differential equation has the general form:

$$L(u) = q \tag{1}$$

In order to solve this equation with the finite element method, the *weak form* of the equation is derived by multiplying both sides by a *test function* v and replacing the exact solution u with the numerical solution u^δ, then integrating over the entire domain [4]. This gives:

$$\int_\Omega vL(u^\delta)\,\mathrm{d}X = \int_\Omega vq\,\mathrm{d}X \tag{2}$$

The discretized solution u^δ is represented as a combination of a finite number of basis functions that span the computational domain. The choice of basis functions affects the accuracy and stability of the computed solution, and the optimal choice varies depending on the equation being solved. It is common for finite element solvers to only implement a small set of different types of basis functions.

The computational domain is divided up into the elements that form the mesh. For each element, integrals are evaluated to produce a *local* matrix and vector. Terms from the local matrix and vector are added to form a sparse *global* matrix and vector that are indexed by the global degrees of freedom (i, j and k in the diagram). After all elements have been assembled, the global matrix and vector form a linear system of equations to be solved. An overview of the flow of data in this process is given in Figure 2.

In practice, the system of linear equations is usually solved using an iterative method, such as the Conjugate Gradient or GMRES method. Most finite element programs rely on an external library such as PETSc [5] for this purpose.

3 Multilayered Abstractions for PDEs

3.1 The Unified Form Language

UFL is an embedded Domain-Specific Language (eDSL) implemented as a Python library, which allows the weak forms of PDEs to be expressed in near-mathematical notation. The library performs preprocessing and (if necessary) automatic differentiation of the forms in order to convert them into tensor contractions on the basis functions and coefficients. A tensor contraction expression has a simple correspondence to a loop nest that iterates over the indices in the expression, and can be algorithmically generated. The automatic generation of

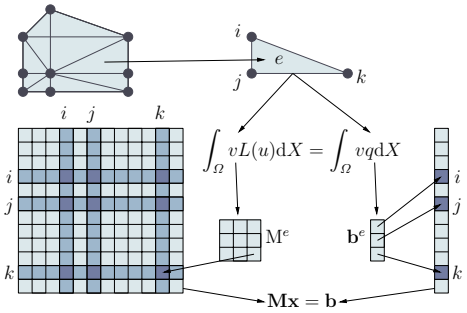


Fig. 2. Computations in the finite element solution of a partial differential equation (PDE). A local matrix and vector are produced for every mesh element, and their terms are added into the global matrix and vector. The global matrix and vector form a system of linear equations.

low-level code facilitates rapid development and allows generation of variants that provide hardware-aware performance. Examples of the different representations include the quadrature- and tensor-representations available in FFC [6] and the symbolic factorisation used in Excafé [7].

In order to exemplify UFL, we will consider the Helmholtz equation (3) and its weak form (4):

$$\nabla^2 u + \lambda u = -f \tag{3}$$

$$\int_\Omega \nabla v \cdot \nabla u - \lambda v u \, \mathrm{d}X = \int_\Omega v f \, \mathrm{d}X \tag{4}$$

The UFL code for expressing this weak form is given in Figure 3. In the example, the basis functions are chosen as order 1 Lagrange functions, but a wide range of basis functions are available in UFL. As can be seen from the example, the use of UFL code does not prescribe any data structures or algorithms used to evaluate the forms - it provides the user with the means to declare the forms to be evaluated. This enables code transformations to be made that optimize the code for a specific architecture - for example, data layouts that attempt to maximize coalescing of memory accesses on GPUs can be used, and the evaluation kernel can be generated with code that stages unstructured data into shared memory in order to maximize performance.

A translation of the left-hand side of the weak form of the Helmholtz equation (`a` in Figure 3) into a local assembly kernel is shown in Figure 4. This kernel is executed for every element in the mesh to produce the local matrices which are returned in `A`, which is a fully-parallel loop.

```
P = FiniteElement("Lagrange", "triangle", 1)
v = TestFunction(P)
u = TrialFunction(P)
f = Coefficient(P)

a = ( dot(grad(v),grad(u)) - lambda*v*u )*dx
L = v*f*dx
```

Fig. 3. The weak form of the Helmholtz equation written in UFL, discretized with order 1 Lagrange basis functions on a triangular mesh

```
void helmholtz(double A[3][3],
               double x[3][2]) {
  for (uint j = 0; j < 3; j++)
    for (uint k = 0; k < 3; k++)
      for (uint ip = 0; ip < 3; ip++)
        for (uint d = 0; d < 2; d++)
          A[j][k] +=
           (dCG1[d][ip][j]*dCG1[d][ip][k]
           - lambda*CG1[ip][j]*CG1[ip][k])
           * W[ip]*detJ;
}
```

Fig. 4. The local element kernel `helmholtz` for assembling a Helmholtz matrix, omitting the declaration of finite element linear basis functions `CG1` and their derivatives `dCG1`, quadrature weights `W` and the computation of the Jacobian determinant `detJ` from the local coordinates `x` for brevity

3.2 The PyOP2 Framework

PyOP2 is an implementation of the OP2 paradigm for parallel unstructured mesh applications [8]. Unstructured meshes are a popular and efficient choice for computational science and engineering applications on complex geometries. Unlike structured meshes, the mesh topology is defined through explicit connectivity information between entities of the mesh. Finite element methods are commonly used with unstructured meshes to resolve complex geometries with a high level of accuracy.

Data Model. PyOP2 allows the definition of mesh data structures using primitives provided as part of the API, where the user is freed - in fact prevented - from having to define the low-level representation of this data. Mesh entities are represented as *sets* and connectivity relationships between the sets, called *mappings*. For the unstructured two-dimensional triangular mesh in Figure 5, vertex and cell entities are declared in PyOP2 as:

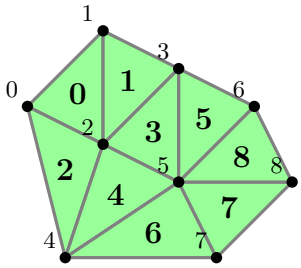


Fig. 5. An unstructured mesh consisting of nine vertices and nine cells

```
ncells = 9
nvertices = 9
cells = op2.Set(ncells)
vertices = op2.Set(nvertices)
```

The connectivity between `cells` and `vertices` is given by a mapping associating three incident vertices with any cell:

```
map_data = [ [0,1,2], [1,3,2], ...]
cell_vertex = op2.Map(cells, vertices, 3, map_data)
```

Mesh data in OP2 is associated with a *set*. The coordinate field defined on the set of `vertices` uses two floats to represent the coordinates of each vertex:

```
coord_data = [ [0.,0.], [.5,.5], [.5,-.25], ... ]
coords = op2.Dat(vertices, 2, coord_data, float)
```

Execution Model. Computations are specified as *kernels* that are executed in parallel for all elements of a given set, which we refer to as the *iteration set*. The order in which set elements are visited is determined by the PyOP2 runtime and cannot be specified by the user. This allows the runtime system freedom in the scheduling of the execution.

A kernel defines computations for a single element of the iteration set. The following example kernel launched over the set of `cells` computes the coordinates of the midpoint `m` of each cell given the coordinates `x` of its incident vertices:

```
midpoint = op2.Kernel("""
void midpoint(double m[2], double *x[2]) {
  m[0] = (x[0][0] + x[1][0] + x[2][0]) / 3.0;
  m[1] = (x[0][1] + x[1][1] + x[2][1]) / 3.0;
}
""", "midpoint")
```

Note that the kernel code is a multi-line string with the kernel definition in C. Support for kernel declarations in a subset of Python is ongoing work.

In a *parallel loop* the kernel is called for each element of the iteration set with the appropriate input data passed in. Data defined on the iteration set is

accessed directly, whereas data defined on another set is accessed indirectly via a mapping.

Our `midpoint` computation kernel requires two arguments: the midpoint of the cell, and the vector of vertex coordinates. The midpoint dataset can be defined as two floats per cell:

```
midpoints = op2.Dat(cells, 2, [0.]*2*ncells, float)
```

The following parallel loop executes the `midpoint` kernel over the iteration set `cells` with `midpoints` as a directly accessed argument via the *identity map* and `coords` indirectly accessed via the `cell_vertex` mapping. *Access descriptors* specify the arguments as write-only and read-only respectively.

```
op2.par_loop( midpoint, cells,
                midpoints(op2.IdentityMap, op2.WRITE),
                coords(cell_vertex, op2.READ) )
```

Implementation. PyOP2 provides an API for the implementation of target-specific backends. At present, backends for sequential C code, C with OpenMP, and CUDA code are fully supported. An OpenCL backend is a work-in-progress at the time of writing. PyOP2 also supports MPI with the sequential backend, and overlaps communication and computation where possible. MPI with other backends is also in progress.

The runtime system is responsible for the efficient scheduling of kernels and marshalling/staging data for each iteration of the set. Prior to kernel invocation, a *plan* for its execution is generated. A plan consists of a set of partitions and a renumbering of the maps used to access datasets so that they can be staged into shared memory. The partitions are appropriately sized so that they can fit within a certain level of the cache hierarchy - for example, the shared memory in CUDA. During execution, the data is staged into shared memory before the user's kernel is invoked for all the set elements in the partition. The partitions are colored so that adjacent partitions have different colors in order to avoid race conditions. Coloring is also done within a partition, in order to prevent interference between the threads executing on a partition.

In common with most finite element toolchains, PyOP2 does not perform linear algebra operations natively but makes use of a linear algebra library. An API for interfaces to linear algebra libraries is provided. At present, PETSc [5] and CUSP [9] are supported.

4 Experiments

In order to demonstrate the performance portability of the UFL-FFC-PyOP2 toolchain, we benchmark an implementation of an advection-diffusion solver. The entire code that is required to solve the advection-diffusion equation is given in Figure 6. This test problem models the movement of a tracer species through a simulation velocity field over time, accounting for both how the tracer is carried by the flow (advection) and how it becomes dispersed over time (diffusion).

The specified variational forms implement a finite element spatial discretisation, with an Euler method for advancing the advection term in time and a Theta scheme for the diffusion term.

The toolchain-generated code is tested with the sequential, CUDA, MPI and OpenMP backends. We also benchmark DOLFIN, as it is known to generate performant CPU codes and can run in parallel using MPI, and Fluidity's built-in advection-diffusion solver, which also supports MPI parallelism. In order to demonstrate integration of the toolchain into existing finite element codes with little overhead and interoperation with the existing data structures, our toolchain-generated code is run from within Fluidity. It runs in an embedded Python interpreter via a thin interface layer that wraps Fluidity's native data structures in NumPy arrays which are suitable for initialising PyOP2 data structures.

Experiments were performed on a machine with 2 6-core Intel Xeon X5650 (Westmere-EP, 2.66GHz with 12MB of L3 cache) with 16GB of RAM and a Tesla M2050 GPU. Software used included RHEL6.3, the CUDA toolkit version 5.0.35 for the CUDA code and the Intel compiler version 11.1.073 for the CPU code. PETSc 3.3 was used for linear algebra with Fluidity, DOLFIN, and PyOP2 on CPU, and CUSP 0.3.1 was used with the CUDA backend. All meshes are fully unstructured and are numbered using a Hilbert space-filling curve numbering to improve locality.

Table 1 shows the execution times for each implementation and mesh. The speedup relative to the sequential Fluidity implementation is given in Figure 7. It can be seen that the PyOP2 generated code compares favourably with the DOLFIN and Fluidity implementations running on 12 cores. The MPI and CUDA implementations exceed the performance of DOLFIN and Fluidity for the largest mesh sizes. Note that the simulation speed is constrained by the available bandwidth rather than computational throughput of the target machines.

```
t=state.scalar_fields["Tracer"]
u=state.vector_fields["Velocity"]

p=TrialFunction(t)
q=TestFunction(t)

M=p*q*dx
D=M-0.5*d

adv_rhs = (q*t+dt*dot(grad(q),u)*t)*dx
d=-dt*diffusivity*dot(grad(q),grad(p))*dx
diff_rhs=action(M+0.5*d,t)

solve(M == adv_rhs, t)
solve(D == diff_rhs, t)
```

Fig. 6. Advection-diffusion UFL implementation. The advection and diffusion terms are split, with advection solved first.

Table 1. Execution times of each version of the solver and each mesh size. Mesh sizes are specified as the number of elements. Times given in seconds.

Mesh	Fluidity	PyOP2 Seq	Dolfin Seq	Dolfin MPI	Fluidity MPI	PyOP2 CUDA	PyOP2 MPI	PyOP2 OpenMP
201712	121.0	46.0	50.2	7.6	16.1	8.2	11.8	15.1
252013	147.8	55.0	59.2	9.0	19.4	8.8	14.3	20.2
302618	181.0	63.3	73.9	12.0	23.0	16.	16.1	22.1
353054	202.3	75.5	79.6	13.8	26.4	18.2	16.4	26.9
402863	231.4	85.3	98.3	17.5	30.8	20.4	21.6	27.6
454182	263.4	95.3	144.9	26.7	36.2	27.6	22.7	36.6
503852	291.5	108.5	150.1	37.3	39.3	28.8	30.1	45.1
553962	321.2	118.5	128.6	30.2	45.0	32.1	32.2	46.4
604454	362.4	128.6	238.9	59.6	50.5	36.7	35.5	48.9
655224	394.2	148.3	276.0	70.2	55.6	38.8	44.2	63.4
705120	413.1	151.4	172.2	40.6	62.2	32.8	42.7	56.5
755413	440.6	163.9	199.5	52.1	67.3	36.9	45.6	68.1
806110	469.3	177.3	239.9	67.2	71.0	43.5	53.1	75.0

5 Related Work

The OP2 C/Fortran implementation [8] shares many design decisions with PyOP2. The main difference between these implementations is that OP2 uses static code translation to generate the kernels whereas PyOP2 uses just-in-time code generation. OP2 is being used to port the Rolls-Royce HYDRA CFD code, a Fortran 77 application, such that it can exploit clusters of GPUs and multicore CPUs.

The DOLFIN Problem Solving Environment (PSE) [10,1] provides an environment for the rapid development of finite element solvers from within Python. Portions of the toolchain used by DOLFIN are used in our work, in particular UFL and FFC. However, the branch of FFC used in this work supports the generation of PyOP2 code, as well as code that conforms to the UFC specification required by DOLFIN. Parallel execution using MPI and OpenMP is supported by DOLFIN, but GPU architectures are not presently supported for finite element assembly.

Nektar++ [11] is a C++ library for implementing finite element methods that allows one of several different assembly algorithms to be chosen from. The key insight of experimental work done with Nektar++ is that the optimal choice of algorithm, even on the same machine, varies depending on the problem parameters [12]. Nektar++ currently targets only CPU architectures. However, it may be expected that the optimal implementation for a given set of parameters will vary with the target architecture.

Liszt [13] is a DSL for unstructured mesh applications that is embedded in Scala. From the programmer's point of view, Liszt differs from PyOP2 in making a distinction between different classes of mesh entities. Cells, edges, vertices etc. have distinct types, whereas in PyOP2 all mesh entities are represented as

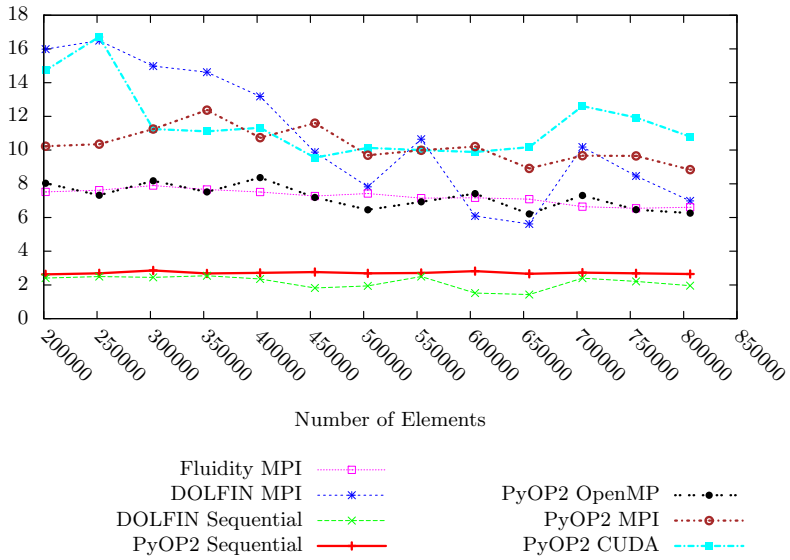


Fig. 7. Speedup of each implementation relative to the Fluidity sequential implementation, which is chosen as a baseline because it is the slowest. All executions (apart from DOLFIN and PyOP2 Sequential) running on 12 cores.

sets. Liszt has been used to implement solvers for the Euler, Navier-Stokes and Shallow Water equations, and a finite element solver for the Laplace equation.

6 Conclusions and Further Work

The toolchain we have presented allows the generation of performance-portable finite element solvers from a single UFL source. Since the input language is the same as that used by the DOLFIN solver from the FEniCS toolchain, a wide range of finite element models can be run with high performance on the CPU and GPU architectures supported by PyOP2.

The power of the code generation approach is that it allows us to explore alternative code generation schemes. We plan to look at hybrid CPU/GPU execution, improved communication overlap, intra-kernel vectorization and warp-wide parallelization, and variants of local vs. global assembly. The key in each case is to adapt the implementation to the application context and the hardware capabilities.

Acknowledgments. The authors acknowledge contributions to the OP2 project from Mike Giles, Gihan Mudalige and Istvan Reguly at the University of Oxford, Leigh Lapworth, Yoon Ho and David Radford at Rolls-Royce, Adam Betts at Imperial College London, Jamil Appa and Pierre Moinier at BAE Systems, Tom Bradley at NVIDIA and Nick Hills at the University of Surrey.

This research is partly funded by EPSRC (EP/I00677X/1, EP/I006079/1), APOS (EU FP7 Project ID 277481) and NERC (DTG NE/G523512/1, NE/I021098/1).

References

1. Logg, A., Mardal, K.-A., Wells, G.N.: Automated Solution of Differential Equations by the Finite Element Method. Springer (2012)
2. Rathgeber, F., Markall, G.R., Mitchell, L., Loriant, N., Ham, D.A., Bertolli, C., Kelly, P.H.J.: PyOP2: A High-Level Framework for Performance-Portable Simulations on Unstructured Meshes. In: WOLFHPC 2012: Workshop on Languages for High-Performance Computing at SC 2012 (November 2012)
3. Applied Modelling and Computation Group Department of Earth Science and Engineering, South Kensington Campus, Imperial College London, London, SW7 2AZ, UK: Fluidity Manual. Version 4.0-release edn. (November 2010)
4. Karniadakis, G.E., Sherwin, S.J.: Spectral/hp Element Methods for Computational Fluid Dynamics, 2nd edn. Oxford University Press (2005)
5. Balay, S., Brown, J., Buschelman, K., Eijkhout, V., Gropp, W.D., Kaushik, D., Knepley, M.G., McInnes, L.C., Smith, B.F., Zhang, H.: PETSc users manual. Technical Report ANL-95/11 - Revision 3.3, Argonne National Laboratory (2012)
6. Kirby, R.C., Logg, A.: A compiler for variational forms. ACM Transactions on Mathematical Software 32(3), 417–444 (2006)
7. Russell, F.: An Active-Library Based Investigation into the Performance Optimisation of Linear Algebra and the Finite Element Method. PhD thesis, Imperial College London (June 2011)
8. Mudalige, G.R., Giles, M.B., Spencer, B., Bertolli, C., Reguly, I.Z.: Designing OP2 for GPU Architectures. Journal of Parallel and Distributed Computing (2012) (in press)
9. Bell, N., Garland, M.: Cusp: Generic parallel algorithms for sparse matrix and graph computations, Version 0.3.0 (2012)
10. Logg, A., Wells, G.N.: Dolfin: Automated finite element computing. ACM Trans. Math. Softw. 37(2), 20:1–20:28 (2010)
11. Vos, P.E.J., Eskilsson, C., Bolis, A., Chun, S., Kirby, R.M., Sherwin, S.J.: A generic framework for time-stepping partial differential equations (pdes): general linear methods, object-oriented implementation and application to fluid problems. Int. J. Comput. Fluid Dyn. 25(3), 107–125 (2011)
12. Cantwell, C.D., Sherwin, S.J., Kirby, R.M., Kelly, P.H.J.: From h to p Efficiently: Selecting the Optimal Spectral/hp Discretisation in Three Dimensions. Mathematical Modelling of Natural Phenomena 6(3), 84–96 (2011)
13. DeVito, Z., Joubert, N., Palacios, F., Oakley, S., Medina, M., Barrientos, M., Elsen, E., Ham, F., Aiken, A., Duraisamy, K., Darve, E., Alonso, J., Hanrahan, P.: Liszt: a domain specific language for building portable mesh-based pde solvers. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2011, pp. 9:1–9:12. ACM, New York (2011)

Container-Based Job Management for Fair Resource Sharing*

Jue Hong[1,6], Pavan Balaji[2], Gaojin Wen[3], Bibo Tu[4], Junming Yan[5], Chengzhong Xu[6], and Shengzhong Feng[6]

[1] Oracle Corporation
[2] Mathematics and Computer Science Division, Argonne National Laboratory
[3] National Laboratory of Pattern Recognition, Institute of Automation, CAS
[4] Institute of Information Engineering, CAS
[5] Tencent, Inc.
[6] Shenzhen Institutes of Advanced Technology, CAS

Abstract. Achieving fair resource sharing is rapidly becoming an essential requirement in cluster computing systems. Although many fair scheduling algorithms have been proposed in recent decades, controlling resource sharing among jobs on servers remains a challenging problem that, if not handled well, may result in chaotic resource contention and service-level agreement violation of jobs. To address this problem, we propose a resource container–based job management approach for fair resource sharing. In our approach, we first design and implement a general container-based job management module, providing lightweight and fine-grained resource allocation and isolation for job execution. With this module, we propose a resource-aware management scheme to enable fair resource sharing in job scheduling and dispatching. We conduct experiments by implementing the proposed module and applying the scheme on `TCluster`, a self-developed cluster computing system of a worldwide top Internet corporation. Results show that our approach performs well in guaranteeing fair resource sharing with negligible overhead.

1 Introduction

Unlike past decades when batch job submission prevailed, today various types of jobs are being deployed simultaneously by using cluster computing systems. Each job usually has a specific service-level agreement (SLA), which can be mapped to resource requirements such as CPU, memory, or I/O bandwidth. How to fairly partition and share such resources among running jobs in a cluster, is key to guaranteeing SLAs.

Many job-scheduling algorithms based on fair strategies have been proposed in recent decades [9, 15, 19, 25, 26, 28]; they determine which job should be scheduled to run according to the job's resource requirements and the available quota of job owners. Controlling resource sharing among running jobs on servers remains a challenge, however; and if not handled well, chaotic resource

* This work was partially supported by NSFC 61202417, 60903116 and 61003063.

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 290–301, 2013.

contention and SLA violation of jobs may result. Two popular approaches for resource sharing are process-level sharing and virtual machine (VM)-level sharing. In a process-level approach, it is difficult to track and control the resources used by programs with multiple processes or with a newly spawned process while running. Moreover, fine-granularity isolation of important resources such as the CPU and network bandwidth cannot be guaranteed [10]. The VM-level approach [16] relies on various virtualization technologies such as XEN [11], VMware [8], and KVM [4]. It typically runs programs inside a VM configured with the required resources. Although the VM-level approach provides good resource-isolation, it usually incurs high overhead for controlling, setting up, and running programs [25], especially in the case of small short-lived programs.

In this paper we propose to use a resource container to build a job management approach for fair resource sharing on cluster computing systems. Resource containers are based on OS-level virtualization that has been popularized in recent decades; examples are LRP [10], VServer [25], OpenVZ [7], and Linux Container (LXC) [5]. Resource containers partition the resources of a single operating system into isolated groups and can give programs the illusion of running on a separate machine. By running instructions native to the core CPU without any special interpretation mechanisms, resource containers introduce little or no overhead. Moreover, modern container technology such as LXC can provide fine-grained resource partitioning, for example, assigning half a CPU to a program. The features of low cost and fine-grained partitioning make resource containers particularly suitable for job execution in cluster computing systems, in which each server hosts a homogeneous operating system. Furthermore, recent container technologies such as LXC provide good support for resource management of multiprocess programs.

The contributions of our paper are twofold:

- We propose a general container-based job management module (CJMM) as the kernel of our fair resource sharing approach, providing resource isolation and a sharing mechanism for running jobs on one server.
- Based on CJMM, we present a resource-aware management scheme, including resource-aware job scheduling and dispatching, that enables fair resource sharing on cluster computers.

We note that our resource-aware management scheme provides only a framework for implementing resource fairness using the proposed module. The job scheduling and dispatching algorithms can be chosen as needed and thus are not the focus of this paper. As the underlying container technology, we use LXC, which is a recent implementation of OS-level virtualization and has been included in the mainstream Linux kernel. Several other container technologies (e.g., OpenVZ and VServer) are also popular and have performance comparable to that of LXC. However, they all require customized Linux kernels. Moreover, issues of security, stability, and maintenance make them less competitive, compared to LXC, in production environments. The proposed module and scheme are implemented on `TCluster`, a self-developed cluster computing system of a worldwide top Internet corporation. Experiment results show that our approach performs well in

enabling fair resource sharing among running jobs. Also, the proposed scheme helps improve the resource utilization of the whole cluster.

The rest of this paper is organized as follows. We introduce related work in Section 2. The detailed design and implementation of the job management module are discussed in Section 3. We present the resource-aware management scheme in Section 4. In Section 5 we present our experimental results. We conclude in Section 6 with a brief look at future work.

2 Related Work

This section briefly discusses related work in two areas: fair scheduling algorithms and resource containers.

2.1 Fair Scheduling Algorithms

Fair job scheduling is an important research problem for cluster computing systems. One approach to fair scheduling is lottery and stride scheduling [26], proposed by Waldspurger, using both random and deterministic manner to allocate resource for jobs. In [25], Soltesz et al. proposed using a hierarchical token bucket to assign quotas among jobs in order to achieve fairness. Recently, with the popularization of Hadoop, some simple yet effective fair scheduling algorithms based on a round-robin approach have been proposed [9, 28]. Isard et al. proposed a fair scheduling named Quincy [19] for the Microsoft Dryad cluster by modeling the scheduling as a network flow problem. For multiple resources cases, Ghodsi et al. proposed a dominant resource fairness (DRF) scheduling algorithm [15]. However, all these algorithms focus only on how to determine a fair order of jobs running in a cluster. They do not provide a mechanism for controlling resource sharing of running jobs on servers.

2.2 Resource Containers

The concept of resource containers was first proposed by Gaurav Banga et al. [10]. Similar concepts on non-Linux system are Solaris Zone and FreeBSD Jail. Early resource containers on Linux systems are OpenVZ, VServer, and FreeVPS. Most of these technologies require customized Linux kernels, however, and thus are unacceptable in many product scenarios, especially for large corporations. On the other hand, LXC has been combined into the mainstream Linux kernel; and, different from traditional machine-level virtualization, LXC requires neither instruction-level emulation nor just-in-time compilation.

The building block of LXC's resource sharing is the `cgroup` framework and various subsystems [2, 22]. The `cgroup` framework allows LXC to track, control, and audit the resources used by process groups. Subsystems in current mainstream Linux kernels, such as cpu, cpuset, cpuacct, memory, and net_cls, enable LXC to support sharing and accounting on such resources. For resource isolation,

LXC employs the kernel namespace [6] and the `pivot_root` system call. Additionally, the LXC toolkit provides a `liblxc` library and a series of userspace tools for container management. Given the advantages and the popularity of LXC, we use it as the underlying container technology in this paper.

3 Container-Based Job Management

In this section we describe the design and implementation of the container-based job management module (CJMM).

3.1 Design of Job Management Module

The architecture of a typical cluster computing system can be separated into two parts: a central manager that responsible for global control, and an execution engine that is responsible for running and managing jobs on cluster servers. The CJMM is plugged into the execution engine, taking over the job execution, resource provisioning, and isolation. More specifically, the execution engine passes the job information to our job management module. The module then creates a container, assigns resources, and starts the job inside it. A handler of the running job (e.g., the PID) will be returned to the execution engine for monitoring and controlling. Each job runs in its own container, unaware of jobs in other containers.

The CJMM consists of two components: `JobManager` exposes a simple, high-level, container-based job management interface to the execution engine; and `Container` represents the data structure and operations of a real container.

JobManager. `JobManager` starts jobs and manages their containers. Job's data (e.g., executing function and arguments) is passed in via a `JobPtr` object when starting. The PID of the job is stored and returned for monitoring and control. While the configuration of a container (CPU shares, memory limits, etc.), is stored in a `ContainerConf` object.

In addition, `JobManager` assigns and accounts for the resource usage on the server. When a job is being started, `JobManager` checks whether the required resource can be allocated according to the available amount. If allocatable, a container is created and the required resources are deducted from the available amount; they will be returned when the job finishes. Otherwise, the request is declined. The resource requirement is stored in a `ResInfo` entity, an example of which is "resinfo.cpus = 0.5; resinfo.memory = 3GB...".

Container. The `Container` represents the data structure and the operations of a real container. For the sake of extensibility, `Container` is designed as a virtual class. Two key operations of `Container` are task execution and resource usage information retrieval. The execution command of a job will be passed to the real container's executing API via `RunTask` method. Real-time resource usage information can be obtained via `GetUsage`. We have designed a class `LXCContainer` inherited from `Container` using the LXC technology.

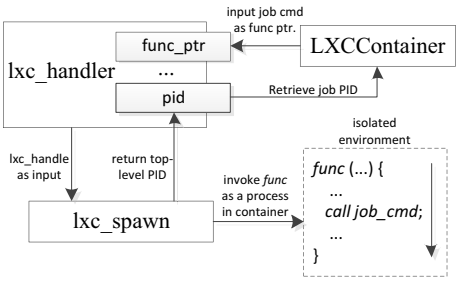


Fig. 1. Modified job-startup mechanism in LXCContainer

3.2 Implementation Issues

We have implemented a prototype of the job management module using C++ based on the LXC 0.7.2. In this version, the low-level control of programs outside their containers is obscured. And the LXC toolkit has no direct method to obtain the real-time resource usage. Next we describe our solutions to these two issues, including the job-startup mechanism and resource usage information retrieval for a container.

Job-Startup Mechanism. LXC's application execution mode allows starting a program in a container through the `lxc-execute` command. However, because of the hierarchical PID namespace, the PID of a running program in a container is usually different from that outside. Neither the LXC userspace tools nor the open API of `liblxc` provides a direct way to get the outer-layer PID. Moreover, in LXC's application mode, a process called `lxc-init` will be started with PID 1, acting as the parent of all other processes in the container. As a result, the ending signal and exit code of a job cannot be captured by the execution engine, which is undesirable for the JobManager.

To handle these problems, we hacked the program-starting mechanism of LXC's application mode in the `LXCContainer` as in Figure 1. We directly use the `lxc_handler` structure in LXC's source code as well as the open API of `liblxc`. We first replace `lxc-init` with a function executing the job's commands directly (e.g., `func`), which makes the job itself the root process of a container. Then we store a pointer of the function in the `lxc_handler` and invoke the `lxc_spawn` method in `liblxc`'s open API, which will set up a container and start the job finally. The job's PID in the top-level namespace is stored in the `lxc_handler` and passed to the execution engine.

Usage Information Retrieval. We use the statistical functions of subsystems with the `cgroup` framework to obtain real-time usage information of containers.

For CPU usage, we employ the subsystem cpuacct, which accumulates the CPU time of all tasks in a group in cpuacct.usage. Given a timing period, we can calculate the real-time CPU usage of a container by dividing its CPU time by the elapsed time. For simplicity, we let the timing period be the time between

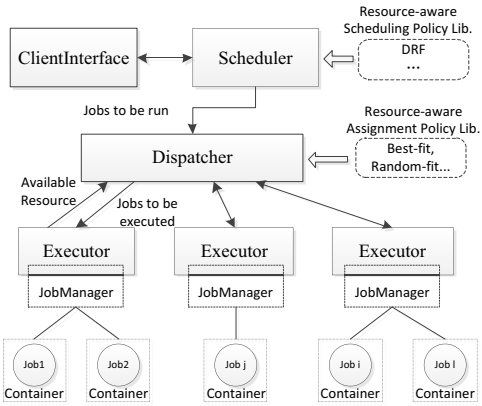


Fig. 2. Architecture of resource-aware TCluster

the most recent two calls of the `GetUsage` method. For memory usage, we employ the memory subsystem, and access the memory.usage_in_bytes to obtain memory usage of all tasks in a group.

4 Resource-Aware Management Scheme

Based on the CJMM, we propose a management scheme and show its application on the `TCluster` system.

4.1 TCluster

`TCluster` is a typical cluster computing system for job processing and cluster resource management (Figure 2), which consists of four main modules: a *ClientInterface* for submitting jobs, a *scheduler* for scheduling jobs, a *dispatcher* for assigning jobs to servers, and an *executor* for running jobs on each server. Using a process-level isolation and job-number based scheduling, the original `TCluster` is not resource-aware and is unable to conduct SLA-guaranteed job processing. To remedy this situation, we apply our resource-aware management scheme on `TCluster`.

4.2 Resource-Aware Scheme on TCluster

Our proposed scheme consists of a resource-aware scheduler, a resource-aware dispatcher, and a container-based execution engine, which can be applied on any cluster computing system.

Resource-Aware Scheduler. The resource-aware scheduler employs the required resources of each job as a key metric while scheduling. We require users

to declare the resource requirements for their submitted jobs and then employ the DRF scheduling algorithm in `TCluster`'s scheduler, which works well in multiresource scenarios. This renders `TCluster`'s scheduler able to generate fair scheduling results in terms of multiple resources (e.g., both CPU and memory).

Resource-Aware Dispatcher. The resource-aware dispatcher tries to find a good match between jobs' required resources and the available resources on servers in assignment. We use a simple matching policy as blow among various candidates (e.g., [12–14, 17, 18, 21, 23, 24, 27]).

Let $\overrightarrow{R_{req}} = \{x_1, x_2, \ldots, x_i\}$ be the required resources for a job, where x_i is the amount of the ith resource. Also let $\overrightarrow{R_k} = \{y_1, y_2, \ldots, y_i\}$ be the available resources on server k. Both x_i and y_i are normalized values. We then find the server m with the minimum Euclidean distance (Affinity Number) of $\overrightarrow{R_{req}}$ and $\overrightarrow{R_k}$ for the given job. This simple policy works well in scenarios where short-lived jobs dominate and the scale of cluster is large (e.g., over 5,000 servers), reducing both computation complexity (e.g. the backfilling with online bin-packing [20]) and resource fragments .

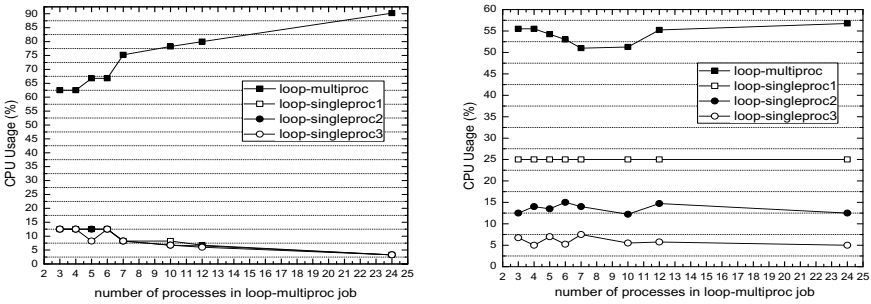
Container-Based Execution Engine. We modified the executor of `TCluster` by merging the `JobManager` class, taking over the job execution, and providing resource guarantees and isolation. The executor reports the available resources to the dispatcher in each heartbeat to facilitate the resource-aware assignment.

Additionally, resource requirements of each job is expressed using XML and passed via the ClientInterface. An example is like "cpu_num=0.5; memory=0.5GB".

5 Performance Evaluation

In this section we evaluate our approach via experiments. The OS used on each server is SUSE Linux Enterprise 11-sp1 with kernel version of 2.6.32.29-x86_64. The version of the LXC toolkit used is 0.7.2.

To evaluate resource sharing, isolation, and utilization performance, we set up a cluster consisting of six servers on the same rack and connected with 1G Ethernet, and deploy `TCluster` on them. Each server is equipped with four Intel 3 GHz Xeon CPUs and 2 GB memory. One server hosts the ClientInterface, the scheduler, and the dispatcher, and the other five servers run the container-based executor. Here we focus on the CPU and memory resources. To see the CPU, we employ two CPU-intensive programs: "loop-singleproc" and "loop-multiproc". Both programs repeatedly execute some increment instruction (e.g.,"i++") with single and multiple process respectively. And we increase the number of processes in "loop-multiproc" gradually (i.e., 3, 4, 5, 6, 7, 10, 12, and 24). For each version of `TCluster`, one job of "loop-multiproc" and three jobs of "loop-singleproc" are submitted simultaneously. In the modified `TCluster`, the CPU resource ratio set to the "loop-multiproc" job, and the other three "loop-singleproc" jobs is 8:4:2:1.



(a) In original TCluster. (b) In modified TCluster.

Fig. 3. CPU usage of each job in TCluster (ratio 8:4:2:1)

For memory testing, we use a memory-intensive program that continuously allocates and touches memory.

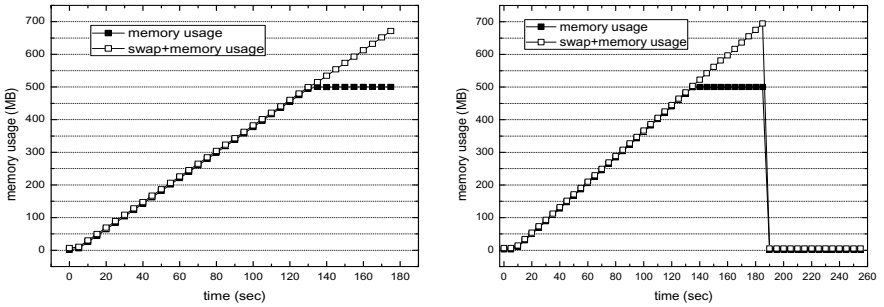
In the overhead evaluation, we use an experimental IBM x3550 server. The server is equipped with a quad-core Xeon E5504 2 GHz CPU and 15 GB memory. We use GeekBench [3] and UnixBench [1] to evaluate the performance of CPU, memory, disk I/O, and system operations.

5.1 Resource Sharing and Isolation

Figure 3(a) shows that the "loop-multiproc" job consumes a significant portion of CPU, indicating that the original `TCluster` cannot ensure sharing and isolation of CPU resource among jobs. While in the modified container-based `TCluster`, the CPU times of the four jobs are approximately in accordance with the preset ratio of 8:4:2:1 (Figure 3(b)), which means the container-based approach helps guarantee fair sharing of the CPU.

Since no memory control mechanism was in original `TCluster`, we only evaluate the modified `TCluster` with the memory-intensive benchmark. We first set the memory limit of the job to 500 MB with unlimited swap space, and then limit the total space of swap and memory to 700 MB. In both cases the used physical memory never exceed 500 MB (Figure 4(a) and Figure 4(b)). In former the job can still obtain the memory since the swap space is unlimited, while in latter the amount of used swap and physical memory drops to zero quickly once the limit is reached. The reason is that in the memory subsystem, a default out-of-memory (OOM) behavior is to kill the programs.

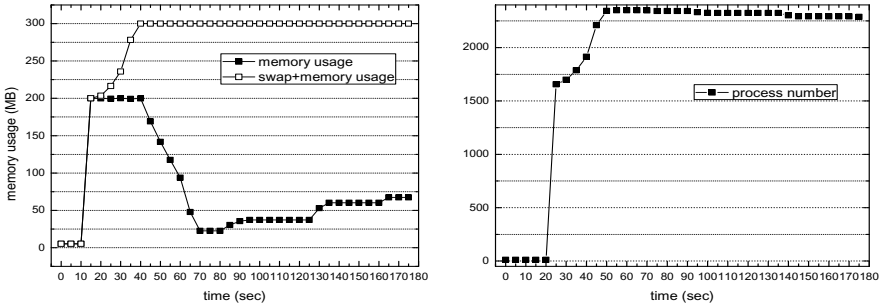
Next we show with our approach, the famous "forking attack" can be alleviated. We implement a program called "bomb," which keeps the forking process using a nonpaused and infinite loop. We submit "bomb" to both versions of `TCluster`, and establish SSH sessions to the server on which the "bomb" job is assigned to see the isolation effect. On the original `TCluster`, when the "bomb" begins running, the SSH session quickly becomes unresponsive. On the modified `TCluster` with memory limit of 200 MB and a total limit for swap and memory



(a) Limited memory with unlimited swap.

(b) Limited memory and limited swap.

Fig. 4. Memory usage of the experimental job in modified TCluster



(a) Memory usage of the bomb job.

(b) Process number in system.

Fig. 5. Memory usage of bomb job and process number in system

of 300 MB, both memory usage and process number are well controlled (Figure 5(a) and Figure 5(b)), and the SSH session is completely responsive. The reason is that the default OOM killer of the memory subsystem keeps killing the processes forked by "bomb" and releases system resource.

5.2 Resource Utilization

Here we compare the resource utilization under the FF, RF in original `TCluster` and affinity number-based best-fit (ANBF) in our resource-aware scheme, described in Section 4.2.

The total available resources (CPU (cores) and memory (MB)) of the experimental cluster are shown in Table 1. For each server, we reserve 0.2 CPU for the executor process itself. The workload we use consists of a series of production jobs with different resource requirements (Table 2).

Table 1. Available resources of cluster

Server ID	CPU(s)	Memory
1	3.8	1847
2	3.8	1851
3	3.8	1858
4	3.8	1859
5	3.8	1855

Table 2. Information of experimental jobs

Job ID	CPU(s)	Memory
1	1.2	1000
2	2.5	900
3	2.5	1200
4	3.0	800
5	3.0	1500
6	0.7	300
7	1.0	600
8	1.7	800
9	1.1	900
10	0.5	800

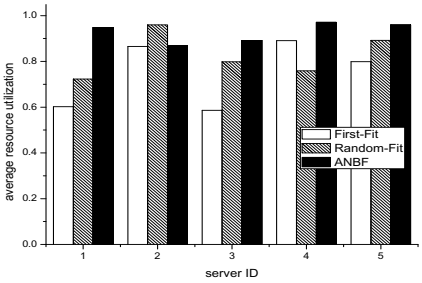


Fig. 6. Average resource utilization of each server with different policy

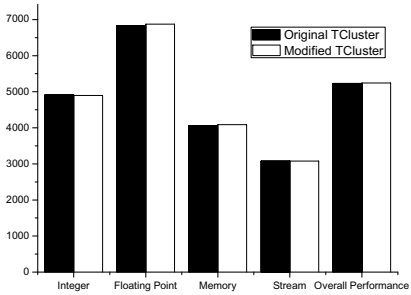


Fig. 7. CPU and memory overhead (Higher score is better)

We submit all ten jobs to the modified `TCluster` with FF, RF, and ANBF, and the available resources of each server is shown in Table 3. And the status of submitted jobs in `TCluster` with each assignment policy is: FF {running 8, pending Jobs (8,9)}, RF {running 9, pending Job (9)}, ANBF {running 10, no pending}. This result indicates that with our resource-aware scheme, `TCluster` can produce fewer resource fragments and thus improve resource utilization. We also calculate the average resource utilization of each server with $avg\_util = 0.5 * cpu\_util + 0.5 * mem\_util$, and the results are shown in Figure 6. We can see that in most cases, our approach with the ANBF policy can produce higher resource utilization on cluster servers (over 85%).

5.3 Overhead

First we analyze the overhead of CPU and memory. From Figure 7 we see no noticeable disparity between the two versions of `TCluster`. Hence our LXC-based approach has negligible overhead for CPU or memory-intensive programs. Then we observe the disk I/O overhead by comparing the data from UnixBench. From Table 4 (higher score is better) we can see that our approach causes at most 1.78% degradation compared with that of the original `TCluster`. Finally we observe the overhead on microsystem through UnixBench as well. The observed items and scores are shown in Table 5. The modified `TCluster` incurs minor overheads in almost all items except the process creation and the pipe-based context switching.

Table 3. Free resources on each server

Server ID	First-Fit		Random-Fit		ANBF	
	CPU(s)	Mem(MB)	CPU(s)	Mem(MB)	CPU(s)	Mem(MB)
1	1.9	547	1.6	247	0.3	47
2	0.3	351	0.2	51	0.9	51
3	1.8	658	0.8	358	0.3	258
4	0.3	259	1.1	359	0.1	59
5	0.8	355	0.3	255	0.2	55

Table 4. Disk I/O overhead

	Original TCluster	Modified TCluster	Gain
File Copy 1024 bufsize 2000 maxblocks	766476.3	761287.11	-0.68%
File Copy 256 bufsize 500 maxblocks	226551.2	222517.2	-1.78%
File Copy 4096 bufsize 8000 maxblocks	1609916.2	1606169	-0.23%

Table 5. System operation overhead

	Original TCluster	Modified TCluster	Gain
Pipe Throughput	1642770.8	1646751.1	0.24%
System Call Overhead	2771390	2771562.9	0.01%
Process Creation	14374.9	13785	-4.10%
Pipe-based Context Switching	259403.2	226531.7	-12.67%
Shell Scripts (1 concurrent)	6270.2	6345.1	1.19%
Shell Scripts (8 concurrent)	2153.7	2171	0.80%

6 Conclusion and Future Work

Although many fair scheduling algorithms have been proposed for cluster computing systems, few suitable mechanisms exist that control resource sharing among jobs on servers. To address this problem, we introduced a container-based job management approach. We first designed and implemented the CJMM to control the resource sharing and isolation for jobs on servers. Based on CJMM, we then proposed a resource-aware management scheme to enable fair resource sharing. We experimented with our approach on `TCluster`, a self-developed cluster computing system for a worldwide top Internet corporation. Results show that our approach performs well in providing fair resource sharing at a very low overhead, as well as a higher resource utilization of above 85%. An adaptive and automatic reconfiguration mechanisms and strategies will be our future work.

References

1. byte-unixbench, `http://code.google.com/p/byte-unixbench/`
2. Cgroup, `http://www.kernel.org/doc/documentation/cgroups/cgroups.txt`
3. Geekbench, `http://www.primatelabs.ca/geekbench/`
4. KVM, `http://www.linux-kvm.org/`
5. Linux container, `http://lxc.sourceforge.net/`
6. Linux kernel namespace, `http://lxc.sourceforge.net/index.php/about/kernel-namespaces/`
7. OpenVZ, `http://download.openvz.org/doc/openvz-intro.pdf`
8. Vmware, `http://www.vmware.com/`

9. Hadoop fair scheduler (2010), http://hadoop.apache.org/mapreduce/docs/r0.21.0/fair_scheduler.html
10. Banga, G., Druschel, P., Mogul, J.C.: Resource containers: A new facility for resource management in server systems. In: OSDI, pp. 45–58 (1999)
11. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: SOSP, pp. 164–177 (2003)
12. Chang, F., Ren, J., Viswanathan, R.: Optimal resource allocation in clouds. In: ICCC (2010)
13. Diaz, C.O., Guzek, M., Pecero, J.E., Danoy, G., Bouvry, P., Khan, S.U.: Energyaware fast scheduling heuristics in heterogeneous computing systems. In: HPCS (2011)
14. Gambosi, G., Postiglione, A., Talamo, M.: Algorithms for the relaxed online bin-packing model. SIAM Journal on Computing 30(5), 1532–1551 (2000)
15. Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., Stoica, I.: Dominant resource fairness: Fair allocation of multiple resource types. In: NSDI (2011)
16. Grit, L., Irwin, D., Marupadi, V., Shivam, P., Yumerefendi, A., Chase, J., Albrecht, J.: Harnessing virtual machine resource control for job management. In: VTDC (2007)
17. He, Y., Elnikety, S., Sun, H.: Tians scheduling: Using partial processing in beste ort applications. In: ICDCS (2011)
18. Huang, Q., Huang, T.: An optimistic job scheduling strategy based on QoS for cloud computing. In: ICISS (2010)
19. Isard, M., Prabhakaran, V., Currey, J., Wieder, U., Talwar, K., Goldberg, A.: Quincy: fair scheduling for distributed computing clusters. In: SOSP, pp. 261–276 (2009)
20. Lee, C.C., Lee, D.T.: A simple on-line bin-packing algorithm. J. ACM 32, 562–572 (1985)
21. Liu, H., Abraham, A., Hassanien, A.E.: Scheduling jobs on computational grids using a fuzzy particle swarm optimization algorithm. In: FGCS (2009)
22. Menage, P.B.: Adding generic process containers to the linux kernel. In: OLS (2007)
23. Pinel, F., Pecero, J.E., Bouvry, P., Khan, S.U.: A two-phase heuristic for the scheduling of independent tasks on computational grids. In: HPCS (2011)
24. Sanders, P., Sivadasan, N., Skutella, M.: Online scheduling with bounded migration. Journal of Mathematics of Operations Research 34(2) (2009)
25. Soltesz, S., Pötzl, H., Fiuczynski, M.E., Bavier, A., Peterson, L.: Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In: EuroSys, pp. 275–288 (2007)
26. Waldspurger, C.A.: Lottery and stride scheduling: Flexible proportional-share resource management (1995)
27. Wang, M., Meng, X., Zhang, L.: Consolidating virtual machines with dynamic bandwidth demand in data centers. In: INFOCOM (2011)
28. Zaharia, M., Borthakur, D., Sarma, J.S., Elmeleegy, K., Shenker, S., Stoica, I.: Job scheduling for multi-user mapreduce clusters. Technical Report UCB/EECS-2009-55 (2009), http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-55.html

One Size Does Not Fit All: Clustering Supercomputer Failures Using a Multiple Time Window Approach

Catello Di Martino

Coordinated Science Laboratory
University of Illinois at Urbana Champaign
1307 W Main St., 61801, Urbana, IL, USA
dimart@illinois.edu

Abstract. This paper proposes a heuristic to improve the analysis of supercomputers error logs. The heuristic is able to estimate the error on the measurement induced by the clustering process of error events and consequently drive the analysis. The goal is to reduce errors induced by the clustering and be able to estimate how much they affect the measurements. The heuristic is validated against 40 synthetic datasets, for different systems ranging from 16k to 256k nodes under different failure assumptions. We show that i) to accurately analyze the complex failure behavior of large computing systems, multiple time windows need to be adopted at the granularity of node subsystems, e.g. memory and I/O, and ii) for large systems, the classical single time window analysis can overestimate the MTBF by more than 150%, while the proposed heuristic can decrease the measurement error of one order of magnitude.

1 Introduction

As we walk our way towards the exascale era we prepare to face new challenges imposed by the need of the simultaneous use and control of hundreds of thousands or even millions of processing, storage, and networking elements. As the number of components becomes bigger and software more complex, the frequency and the propagation of failures poses a serious threat for applications to make progress, requiring new fault tolerant and failure avoidance solutions at scale. In this perspective, operational data (e.g., system logs) are going to play a key role to derive novel failure containment [1] and prediction [2, 3] techniques, since they allow to study failure happening during the operational phase. Therefore, the success of next generation of computing systems will deeply rely on how well we are able to understand and analyze current operational data.

A fundamental step for sifting system logs is to accurately group events related to the same cause[1],i.e., the same fault/error. In facts, failure events happening close in time may correspond to different manifestation of the same fault/error on the system, i.e., as the effects of a single fault propagate through a system,

[1] In this paper grouping, clustering and coalescence are used interchangeably.

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 302–316, 2013.

different error detectors are triggered resulting in multiple logged events. Clearly, the accurate grouping of events is crucial to accurately evaluate even simple metrics such as the MTBF [4, 5].

A widely adopted heuristic [2, 6–14] consists in selecting a fixed time window and to group the events that manifest close in time (within the time window) into a unique "tuple" [15, 7]. However, the identification of a proper coalescence time window becomes tricky when analyzing logs of several thousands of nodes due to the overlapping of independent events and disparities in the failure dynamics. For instance independent events happening close in time might be grouped together by mistake and accounted as a single failure, or multiple events due to the same cause erroneusly grouped as independent. As a consequence, with the number of heterogeneous computing cores (e.g., hybrid nodes equipped with CPU and GPGPUs) approaching to the million, the analysis performed with a single time window approach will result inaccurate, i.e., *one size does not fit all*.

This paper proposes a heuristic to improve the analysis of supercomputers error logs. The heuristic is based on the use of multiple time windows selected at the granularity of single node subsystems. The goal of the heuristic is to to reduce, if not eliminate, errors induced by imperfections of the event clustering process on the final measurements such as the MTBF and other metrics evaluated by analyzing error logs.

The heuristic, referred as MTW (multiple time windows) operates in three main steps at four different level of granularity, respectively, i.e., subsystem, node, nodes involved in the same job, and system. It uses information from both failure logs and workload logs. In each step the coalescence process is tuned (i.e., coalescence time windows are selected) in order to minimize the error on the final measuremnets, and, at the end of the analysis, the herurisic is capable of providing an estimate of overall error on the measuremnts accumulated during the analysis.

The MTW heuristic is compared with the standard single time windows approach and extensively validated over a variety of 40 synthetic datasets generated using a model-based log generator tool [4][2]. Datasets are generated with respect to different settings and assumptions and they are made publicly available[3]. The systems considered in this study range from 16384 to 262144 nodes, totaling 6GB of failure data for a total of 1,966,080 nodes, collected during a simulated period of 6 months and under different failure assumptions on failure rates, failure and recovery distributions and on spatial failure propagation patterns. To the best of our knowledge this is the first work i) proposing a time-based heuristic tailored able to take into account the measurement error to drive the analysis, and ii) performing a validation campaign using such a large dataset.

The key findings of this study are: (i) the estimation of coalescence mistakes makes it possible to express the error caused by the time coalescence process as

[2] The log generator in [4] has been extended to be able to generate workload logs along with failure logs. Details on the extension are not described here due to space limitation.

[3] `www.catellodimartino.it`

two linear functions of the selected time window; (ii) a more accurate coalescence can be achieved by analyzing the data starting at the granularity of single node subsystems, and can reduce the error on the measurement of one order of magnitude, i.e., from 70% to 7% in the average; and (iii) with reference of the worst case (higher failure rate and spatial failure propagations) simulated for 262144 nodes, the MTBF is overestimated of more than 150% if employing the commonly adopted single time window, while the proposed MTW heuristic can limit the error to the 22%.

The rest of the paper is organized as it follows. Section 2 presents the state of the art and discuss about the limitation of current time based coalescence approaches. Section 3 presents a preliminary set of experiments tailored to provide some basis to the formulation of the MTW heuristic, discussed in section 4. Section 5 presents the case studies. Finally section 6 concludes the paper discussing future improvements.

2 Related Work

The analysis of supercomputer logs usually accounts three consecutive steps: i) data filtering [10, 6, 16, 8], concerning the removal of log events not related to failures, ii) data coalescence, concerning the grouping of redundant or equivalent failure events, and iii) data analysis, concerning the evaluation of measurements. Clearly, the quality of the final analysis is tied to the goodness of the former phases [4, 7, 8, 17].

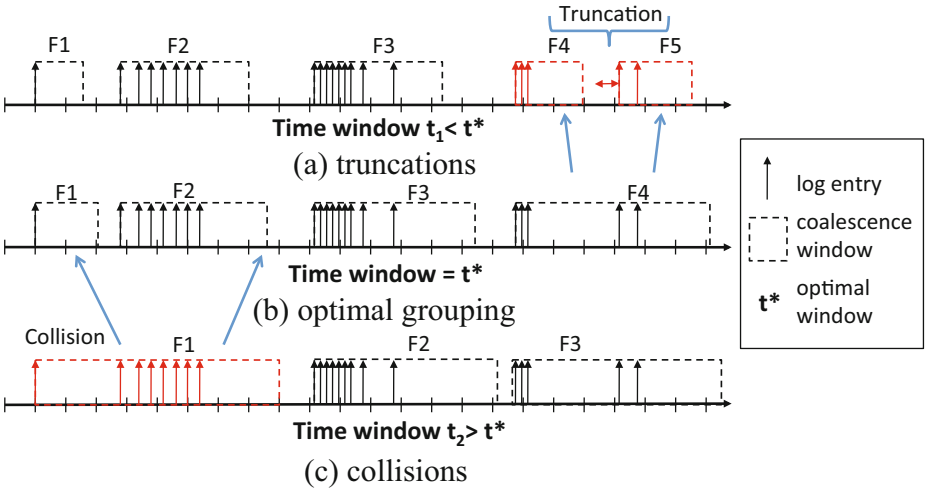


Fig. 1. Example of wrong grouping: (a) truncations and (c) collisions with respect to optimal grouping (b)

The Coalescence process aims to group events manifesting presumably due to a unique failure. This process is sensitive to bad grouping referred as truncations and collisions. A truncation occurs when the time between two or more events caused by a single failure is greater than the clustering time (i.e., the used time window), thus causing the events to be split into multiple tuples (Figure 1.(a)). A collision occurs when two independent failures occur close enough in time such that their events overlap and they are erroneously combined into a single tuple (Figure 1.(c)).

An early work in [7] proposes a heuristic for the selection of a single coalescence time window. The work shows that the number of tuples (i.e., a collection of events that happens close in time) of a given log is a monotonically decreasing function of the time window, with a characteristic "L" shaped curve. This work was proposed for past Tandem systems and assumes that all events due to a same cause manifest close in time. Therefore, the point corresponding to the "knee" of the curve is suggested as a suitable time window able to group all the related events with negligible rate of collisions and truncation, i.e., all the events due to the same fault are clustered together for time windows greater than the time indicated by the knee. However, recent work [4, 17] demonstrate that the rate of collisions is expected to be sensibly larger in modern supercomputers due to the interleaving of uncorrelated events and spatial failure propagations. Despite this results, the knee rule and the single time window technique still constitute the baseline of may work in the field on large-scale systems. The common trend is to use given values for time windows, such as 5min [6–10], 20min [18, 19, 5], 60min [11] and without conducting any tuning (such as, the knee rule) or validation.

New emerging technique are based on content-based coalescence and on the extraction and processing of signals from logs. Content-based coalescence techniques are emerging recently [20–22, 17] and are based on the grouping of events by looking at the specific contents of log messages. For instance [17] and [22] apply the lift data mining operator to find frequent event patterns starting from log content, hence isolating accidental patterns.

Another direction of analyzing logs is given in [3] where authors investigate the use of singnal processing techiques for analyzing error logs in real time and to predict future occurrence of failures. Despite the promising results of all these new techniques, time (and spatial) coalescence techniques are still the most used in the field [6, 23, 7–10, 18, 11, 2, 24], due to the simplicity of the underlying algorithms. Spatial coalescence of data is often used jointly with temporal coalescence to describe failure propagations among machines in the case of large-scale systems [8, 6].

3 Coalescence and Measurement Error Relationship

The effectiveness of time based coalescence has been proved to depend on several factors [4, 17, 8]. In order to develop a better coalescence heuristic, in this section we discuss a set of preliminary tests conceived to identify how truncations and collisions are tied to the error on MTBF estimation obtained after the analysis.

To this aim, we use a model based tool presented in [4] to generate several synthetic supercomputer logs, i.e., artificial logs containing failure dynamics for which the ground truth is know. The ground truth represents the *actual failure process* of the system being analyzed (the objective reality), which may not correspond to the *presumed failure process* reconstructed by coalescing the logs. Synthetic logs are similar to the format of real system logs: each entry contains a time stamp, the system component/node that wrote the entry, and the error message. The synthetic logs are generated along with so-colled oracle logs storing the ground truth on what generated in the syntethic logs. They contains detailed information about every single failure affecting the system, including its start time, end time, type, and the set of system resources involved (e.g., nodes and subsystems). The assessment is performed by running real coalescence algorithm (e.g., the MTW and the fixed time window technique) on the synthetic dataset and comparing the results obtained from the analysis of synthetic logs (e.g, estimated MTBF) against the real dynamics present in the oracle log (e.g., the real simulated MTBF). An example of synthetic and related oracle logs is shown in tables 1 and 2. The generation of both synthetic and oracle logs is parameterized with respect to a number of aspects such as, the number of nodes in the system, the workload, the failure type and propagation patterns, allowing sensitivity studies and to evaluate how such aspects impact on the accuracy of the results achieved when using a specific data analysis technique.

Table 1. Example of synthetic logs

Time	NodeID	Subsys	Message
11/22/11 19:06:41	191	IO	IO Error no. 1
11/22/11 19:06:41	212	IO	IO Error no. 1
11/22/11 19:06:41	212	IO	IO Error no. 2
11/22/11 19:06:41	191	IO	IO Error no. 2
11/22/11 19:06:41	195	IO	IO Error no. 1
11/22/11 19:06:41	195	IO	IO Error no. 2
11/22/11 19:06:41	212	IO	IO Error no. 3
11/22/11 19:06:43	195	SW	SW Error no. 1
11/22/11 19:06:43	192	IO	IO Error no. 1
11/22/11 19:06:46	192	IO	IO Error no. 2
11/22/11 19:06:46	191	IO	IO Error no. 3
11/22/11 19:06:49	195	SW	SW Error no. 2
11/22/11 19:07:01	195	SW	SW Error no. 3
11/22/11 19:07:02	161	NET	NET Error no. 1
11/22/11 19:07:03	195	IO	IO Error no. 3
11/22/11 19:07:09	195	IO	IO Error no. 4
11/22/11 19:07:09	161	NET	NET Error no. 2
11/22/11 19:07:09	195	SW	SW Error no. 4
11/22/11 19:07:47	161	NET	NET Error no. 3
...			
12/17/11 23:22:39	297	PROC	CPU Error no. 1
12/17/11 23:22:42	297	MEM	MEM Error no. 1
12/17/11 23:22:43	297	MEM	MEM Error no. 2
12/17/11 23:22:49	297	MEM	MEM Error no. 3

For this preliminary experiments we used the public synthetic datasets[4], consisting in 24 synthetic log files obtained by simulating the behavior of an hypothetic supercomputer composed by 1024 to 32768 nodes under different failure

[4] The dataset is available on `www.catellodimartino.it`

Table 2. Example of Oracle log

Failure Start Time	NodeID	Type of Failure	Events	Duration	Signature
11/22/11 19:06:39	191	IO_PROPAGATION(192,212,195)+SW(195)	112	124.332	75829788
11/22/11 19:07:02	161	NET	9	154.8	35924727
12/17/11 23:22:39	297	PROC+MEMORY	20	143.85	71162829

assumptions on the failure rate and processed workload. The dataset consists in log files mimicking the content of Linux syslog, reporting timestamp, Id of the node that generate an event, facility responsible for the event (e.g., kernel, network cards) and a text message. The datasets encompass two case studies, both ranging between 1024 and 32768 nodes: 1) logs generated according to simplistic failure assumptions (i.e., exponential and no failure propagations), and 2) logs generated according to more realistic failure assumptions (e.g., lognormal and weibull distributions, spatial failure propagations) to evaluate how system-representative aspects impact on the quality of the coalescence. More details on the dataset can be found in [4].

In this preliminary study we opted to analyze the following response variables: (i) $MTBF_{err}$: the error on the MTBF estimate, defined as $MTBF_{err} = (MTBF_{est} - MTBF_{real})/MTBF_{real}$, where $MTBF_{est}$ is the MTBF estimated from the coalesced log as the average distance between failures in the coalesced data; (ii) $trunc_{\%}$: the percentage of truncations with respect to F, defined as $trunc_{\%} = 100 \cdot number\_of\_truncations/F$; (iv) $coll_{\%}$: the percentage of truncations with respect to F, defined as $coll_{\%} = 100 \cdot num\_collisions/F$. F: the real number of failures; $MTBF_{real}$: the real MTBF of the simulated failure.

Figure 2 shows a subset of the results concerning the sensitivity analysis conducted for the *sliding* algorithm (fixed time window is slided) introduced in section 2 with respect to the time window W. This is a mandatory step when adopting the classical time based coalescence with the knee rule to drive the selection of the time window W. The upper part of the figure reports the tuple count (the "L" shaped curve used by the knee heuristic to select the time window), whereas the lower part reports $MTBF_{err}$, $coll_{\%}$ and $trunc_{\%}$ as a function of W. First, it can be noted that *there is a value W^* for the time window for which the sum of $coll_{\%}$ and $trunc_{\%}$ is minimum and $MTBF_{err}$ tends to 0* (that is, the point where the $coll_{\%}$ curve crosses the $trunc_{\%}$ curve). Interestingly, this point corresponds to the last knee of the tuple count curve, and the same observation repeats for all the performed experiments, showcasing the effectiveness of the knee rule in selecting good time windows for the considered case study.However, it is worth noting that in this case, the knee rule "as it is" will chose as time window the point just after the first knee (a time window shorter that W^*), therefore causing several truncation errors in the coalesced data.

Figure 2.(b) shows the same phenomenon in the case of more complex logs, generated with several different failure inter-arrival distributions, and considering failure propagations and a increased failure rate (ten times). Recall that in this case study 1 only exponential failure dynamics with big inter-arrival are

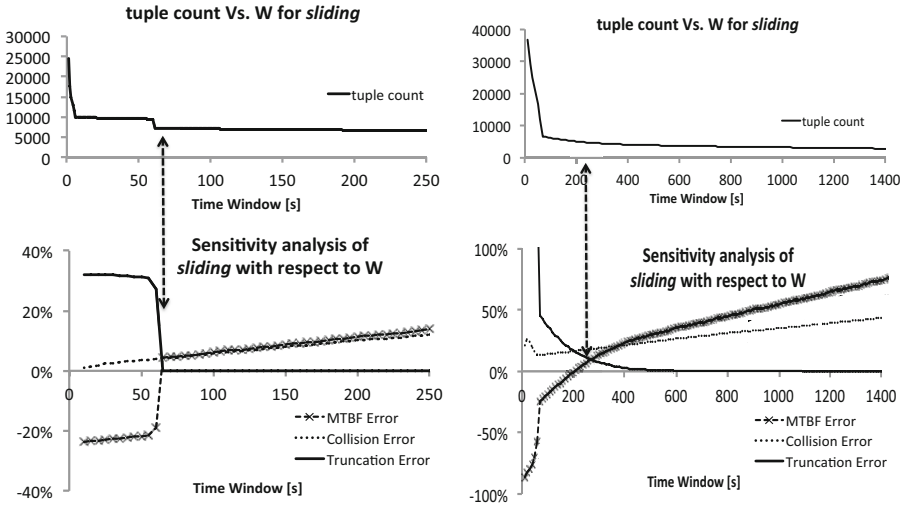


Fig. 2. Sensitivity analysis for the *sliding* algorithm: (a) case study 32768 nodes, failure rate 1E-9 fps, simple(hardware) failures, NO propagations, (b) case study 32768 nodes, failure rate 5 E-9, Complete(ALL) failures, Complete propagations

present in the dataset and that failure propagations have not been simulated. In case 2 we notice from the tuple count curve (Figure 2.(a)) that only one knee is visible, not corresponding to the optimal window, despite several simulated failure dynamics: this is due to the presence of several events accidentally overlapping (e.g., concurrent failures on different nodes), which cause a smoother knee in the tuple count curve, making it difficult to clearly identify the right time window, differently from the former case.

Figures 2.(a) and (b) also show an interesting phenomenon related to the $MTBF_{err}$ which interestingly shows to be correlated with the collisions and truncations. In particular, if W^* is the optimal window (identified by the vertical arrow in Figures 2.(a) and (b)), we can note that:

$$MTBF_{err} = \begin{cases} -m_{trunc_{\%}} \cdot (W - W^*) + trunc_{\%|W=W^*} & \text{for } W \leq W^*, (1) \\ m_{coll\%} \cdot (1 + \sigma) \cdot (W - W^*) + coll_{\%|W=W^*} & \text{for } W \geq W^*. (2) \end{cases}$$

where $m_{trunc_{\%}}$ and $m_{coll\%}$ are the angular coefficient of the linear approximation of $trunc_{\%}$ and $coll_{\%}$ around W^*, respectively[5], and sigma is an real number ≥ 0.

[5] The sharp knee on the $trunc_{\%}$ in figure 2.(a) can be approximated with (i) a discontinuity in W^* so that $\lim_{W \to W^{*+}} trunc_{\%} = coll_{\%|W=W^*}$, and with a linear behavior in $(0, W^*)$.

```
/* STEP 1 */
1. For each subsystem S of the node j
2.    Find Optimal Time window W_Sj
3.    For each W=1s, 20s,... 3600s and W≠W_Sj*
4.       Evaluate trunc%and coll%
5.    End for
6. End for
/* STEP 2 */
10. For each node j in the failure data
11.    For each W in [min_j, W_SJ, max_jWS_J]
12.       Evaluate CI_trunc% and Ci_coll%
13.       Use CI_trunc% and CI_coll% to estimate MTBF_err(W)
13.       Find the W_NJ: MTBF_err(W_NJ) is min
14.    end for
15.    Coalesce node data for using W_NJ as time window
16    Node J Failure Data ←Coalesce(W_NJ) of all S_j in J Failure Data
17.    Node J MTBF_err ← MTBF_err(W_NJ)
18. End for
/* STEP 3 */
19. For each Job T_i
20.    Fetch the failure data using job start and stop time and
       the list of nodes in T_i
21.    If any node is present
22.       Perform spatial coalescence
23.       T_i Failure Data ← Spatial Coalescence all nodes in T_i
24.       For each Failure F in Ti Failure Data
24.          Evaluate CI_trunc%, CI_coll%, MTBF_err
25.       End For
26.       Evaluate CI_trunc%, CI_coll%, MTBF_errfor job J_i
27.    End If
28. End For
29. For Each failure F_i in the whole dataset
30.    System Failure Data ← spatial coalescence all F_i
31.    Evaluate CI_trunc%, CI_coll%, MTBF_errfor System failure data
32. End For
```

Fig. 3. The Multiple Time Window Algorithm

4 The Proposed Solution

The observations provided in section 3 allow us to conclude that:

- *the knee rules is useful but its application can be tricky for large systems due the presence of complex failure propagations and overlapping which should be isolated and analyzed separately;*
- *estimating the probability of truncations and collisions would allow to estimate the expected error on measurements*, which is useful to *allows to optimally tune W to the point such that $trunc_{\%} = coll_{\%}$, i.e., the point where $MTBF_{err} = 0$*, and to correctly interpret and weight the results.

Starting from these observations we propose the multiple time heuristic (MTW) which pseudo code is showed in showed in Figure 3. The algorithm operates in 3 steps.

Step1. Analysis of failure data from single node subsystems and assessment of truncation and collisions probabilities.

For each subsystem S_j of the node j, a suitable time window W_{Sj} is determined using the knee rule. Using the identified time window W_{Sj} the truncations and collisions in the data if coalescing the data of S_j with time windows $W \neq W_{Sj}$ are evaluated as (i) $trunc_{Sj}\%$: the percentage of truncations with respect to $F_{W_{Sj}}$, defined as $trunc_{\%} = 100 \cdot num\_of\_truncations/F_{W_{Sj}}$; (ii) $coll_{\%}$: the percentage of truncations with respect to $F_{W_{Sj}}$, defined as $coll_{\%} = 100 \cdot num\_collisions/F_{W_{Sj}}$. $F_{W_{Sj}}$ is the number of failures for S_j when coalescing with W_{Sj}.

Step2. Analysis of failure data corresponding to a node and selection of the node coalescence time window

The coalescence time windows identified at the end of Step1 may still not be good enough for coalescing the failure behavior of a single node. For instance, a

node may require two time windows, due to its subsystems failing close in time, but independently and with different dynamics.

Therefore, for each node j in the failure data, calling N_{Sj} the number of tuples obtained with the time window W_{Sj} for the subsystem S and $N_j = \sum_j^{N_{Sj}}$, we calculate the following *corruption indices* CI_{trunc} and CI_{coll} (line 12 of Figure 4):

$$CI_{trunc} = \sum_j \frac{N_{Sj}}{N_j} \left[trunc_{\%Sj}\right] \text{for} W = \min_j W_{Sj}, ... \max_j W_{Sj} \tag{3}$$

$$CI_{coll} = \sum_j \frac{N_{Sj}}{N_j} \left[coll_{\%Sj}\right] \text{for} W = \min_j W_{Sj}, ... \max_j W_{Sj} \tag{4}$$

The *corruption indices* in equation (3) and (4) evaluates a sum of the percentages of collisions and truncations weighted by the the total number of tuples of each subsystem over the total number of tuples in the node. The objective is to take into account the contribution brought by each subsystem to the overall truncations and collisions in the node failure data for time windows $\neq W_{Sj}$. The results are then used in equation (2) to identify (i) the time window causing the smallest error in the node dataset and (ii) to estimate the residual error on the measurements.

Let us consider an example. Let us assume that in node j two different subsystems S_1 and S_2 fail almost concurrently. Due to the different nature of S_1 and S_2 they manifest different failure dynamics requiring different time windows W_{S1} and W_{S2} identified with the knee rule. When coalescing all the data of the node, we must chose a time window W_{Nj} to coalesce the failure data of the two failed subsystems. In order to reduce the number of truncations and collisions, we compute the collisions and truncations that we would have in $S1$ and $S2$ when using the time window ranging from W_{S1} and W_{S2}. We then weight the so computed collisions and truncations percentages by the total number of tuples each subsystem is responsible for, divided by the total number of tuples, to take into account the real contribution brought to the failure data of node j by S_1 and S_2 failure data, respectively. The result in CI will encompass the total number of collisions and truncations due to the disparities in the subsystem failure dynamics, when coalescing the node j with a single time window.

Step3. spatial coalescence. The objective of this step is to perform the spatial coalescence considering first the nodes executing the same job and then all the nodes in the system. For this step, workload logs are required to gather information of the IDs of the nodes executing the same job. The objective is to analyze spatial propagations of failures and to estimate the error in the analysis due to the accumulation of the error of present in the data coalesced for each node. This step shares the same principles of the former and start analyzing the data first at the level of the job and then at the level of the system. The objective, as for the first step, is to reduce the granularity of the system to decrease the possibility of accidental overlapping in the data.

For each job, the heuristic extracts subset of the failure data containing only the participating nodes. Then for each subset of failure data, it evaluates corruption indices (equations (3) and (4)) and the estimation of measurement error as weighted sum of the indices and error estimation computed at the end of step 2. At the end of the analysis the two computed indices are used to weight the results and understand the confidence of the measurements. To this aim, confidence intervals are computed concerning the estimation of measurement error and delivered together with the final coalesced dataset.

5 Validation

The proposed solution is validated against several logs generated using the tool in [4]. In particular, the proposed solution required to extend the log generator in [4] in order to make it able to generate also workload log as needed by the approach. The assumed format for the workload log is compliant with the standard workload log format[6].

In order to assess what are the parameters that influence the effectiveness of the proposed solution, synthetic logs are generated in this study following a rigorous experimental plan chosen according to a full factorial design of experiments. The factors of the study describe the system while the response variables identify how well the coalescence is able to reconstruct the ground truth. The goal is to test the hypothesis that the response variables (e.g., error on the measurements when using a specific coalescence technique) depends on the levels of the explanatory factors. We choose to factorize the dataset with respect to i) size of the system (number of nodes), ii) type of failures, iii) presence of spatial failure propagations, iv) failure rates. The synthetic experimental plan is reported in table 3. Factors levels are chosen to represent the size of the system, assumed to range from 16384 up to 262144 nodes and limit situations, such as presence or absence of spatial propagations, high or low failure rate, in order to reduce the number of experiments to run, while still be able to generate enough heterogeneous data point to test the hypothesis. A total of 40 logs, totaling 6GB of filtered failure data for 1,966,080 nodes are generated during a simulated period of 6 months per dataset. In the following the effectiveness of the proposed solution is assessed and compared with the standard time based coalescence adopting the knee rule and single time window (see section 2). The comparison is carried out by i) comparing the error caused on the MTBF measurement by the two techniqyes, ii) by analyzing the percentage of truncations, collisions, and iii) by assessing the error on the MTBF estimation provided by the MTW. In the following experiment we refer to the metrics used in section 3 for the preliminary experiments.

Figure 4 shows the error caused by the the MTW and the knee coalescence heuristics. Recall $MTBF_{err} = (MTBF_{est} - MTBF_{real})/MTBF_{real}$ and is computed as the mean of the time intervals between consecutive failures in the coalesced data. All the plots are reported against increasing size of the systems.

[6] `http://www.cs.huji.ac.il/labs/parallel/workload/swf.html`

Table 3. Synthetic plan of experiments

Factors	Levels				
System Size	16384	32768	65536	131072	262144
Failure Rate	FR		5*FR		
Failure Type	Simple (2 Exponential dynamics)		Complex (several dynamics including Weibull, Lognormal and Exponential failure distributions)		
Spatial Propagation	NO		YES		

Figure 4.(a) shows the effectiveness of the two heuristics for low exponential failure distributions with low failure rate and no spatial propagations. In this case both the techniques perform equally and the error is bound to 3% and 5% for MTW and the knee heuristic, respectively. This is due to the low failure rate simulated and to the absence of spatial failure propagations. i.e., despite the high number of simulated nodes, the probability of having concurrent independent failures (i.e., accidental collisions) during the observed period is negligible, as shown in in Figure 5.(a), reporting the collisions, truncations caused by the two techniques. The small error still present in the MTBF is in both the heuristic is mainly due to the disparity in the simulated failure dynamics and time to recovers, and in the truncations caused still caused by the knee rule (see section 3), which in part affects also the MTW.

Figure 4.(b) shows the experiments performed with an increased failure rate (5 times). In this case, the probability of having accidental collisions is higher, as shown in Figure 5.(b). The higher percentage of collisions causes longer tuples since several consecutive failures are merged and accounted as one longer failure.

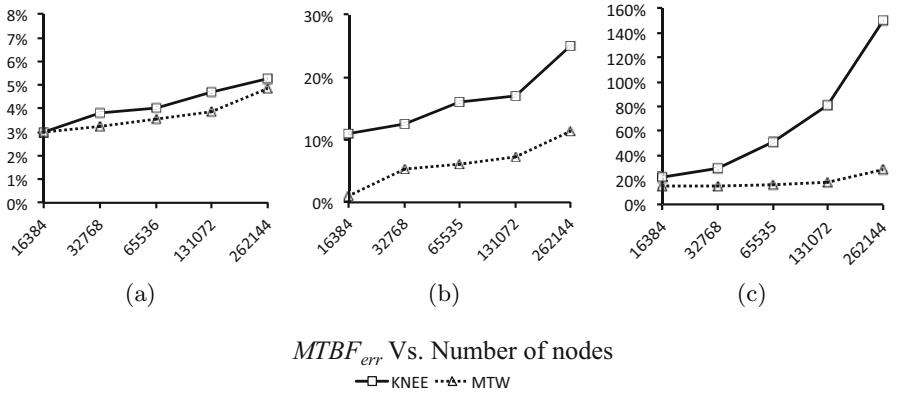


Fig. 4. MTBF error committed by the KNEE and MTW heuristics for systems ranging from 16384 to 262144 nodes in the case of: exponential failure dynamics with low and failure rates, with no spatial failure propagations (a)(b), and complex failure dynamics (mixed exponential, lognormal and weibull) with spatial failure propagations (c). Simulated MTTR are about 60s and 500s depending on the type of failure.

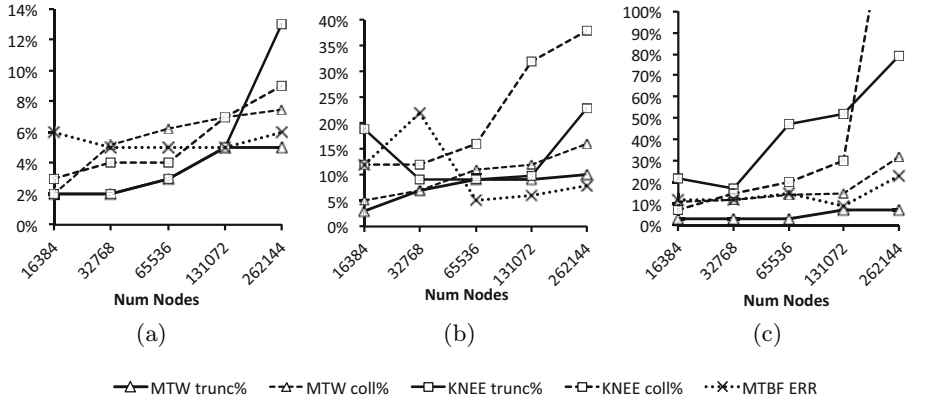


Fig. 5. $trunc_{\%}$, $coll_{\%}$ and $MTBF_{err}$ caused by the MTW and knee heuristics for systems ranging from 16384 to 262144 nodes in the case of: exponential failure dynamics with low (a)(b) and high (c) failure rates, with no spatial propagations (a), and with spatial propagations (b)(c). $MTBF_{err}$ refers to the value computed in step 3 of the MTW (line 31 Figure 3).

This in turn causes an overestimation of the MTBF in both the techniques when increasing the size of the system. The overestimation is a dangerous mistake, since it causes the system to be presumed as more reliable than it really is, underestimating the risk of potential failures. In this case, the MTW heuristic shows less sensitivity to the increased failure rate. In this case, the $MTBF_{err}$ is kept less than 10.8% for 262144 nodes for the MTW while the knee heuristic is not able to go lower than 28%. This is due to the analysis performed at the granularity of the single node subsystems using the estimation of $MTBF_{err}$ as feedback to drive the selection of the time windows. In particular, this last feature allow to solve the higher amount of truncations caused by of the knee (see section 3), as showed in Figure 4(a)-(c). In addition, it is worth noting that the prediction of the MTBF error delivered by MTW (Figure 4.(b)) shows to be valuable in weighting the results. In this case, using equation 2, the MTW is able to estimate to be committing an error of 8% on the results while actually overestimating the MTBF by 10.8%. In addition, the results delivered by the MTW for the $MTBF_{err}$ can be further analyzed by interpreting the $CI_{coll_{\%}}$ and $CI_{trunc_{\%}}$ indices. For instance, indices are valuable in identifying situations in which a low error is due to aliasing of truncations and collisions (high values for the indices means).

Figures 4.(c) and 5.(c) reports the MTBF, collision truncation errors evaluated when generating logs taking into account complex failure dynamics, failure propagations between nodes and higher failure rates. In this case logs show to be more complex and difficult to analyze. More collisions and truncations are caused than in the former simpler scenario, impacting negatively on MTBF for both the knee technique and MTW. For instance, the classic time coalescence approach shows to not be able to scale with the system and the complexity of

failures, causing an error on the MTBF estimate as high as 150% (Figure 5.(c)). The MTW seems to be less vulnerable to the simulated scenario, however causing as much as 30% of truncations and collisions when analyzing the data from 262144 nodes. In this case, the error on the MTBF (Figure 4.(c)) is bigger than in the former cases (22%)and interestingly, one order of magnitude lower than for the knee rule. Also in this case, the $MTBF_{err}$ estimated by MTW (about 27%) is helpful in weighting the results, i.e., warning that the provided measurements could be affected by an error as high as 27%.

6 Conclusions

This paper presented a heuristic for the temporal analysis of event logs of supercomputers. The approach is to be able to drive the analysis taking into account the error committed on the final measurements. The approach has been validated against 40 different synthetic logs generated for a variety of scenarios ranging from smaller systems of 16384 nodes to large supercomputers of 262144 nodes Experiments have been conducted for different extreme-cases failure assumptions and results have been compared with those obtained by using the standard single time window coalescence heuristic.

MTW showed to be promising in reducing the MTBF error by i) reducing the impact of collisions and truncations on the data analyzing the logs starting from the single node subsystems, and ii) using an estimation of the MTBF error on the measurement to drive the analysis. In difficult situations such as for the largest system analyzed in this study, the approach showed to be more still sensitive to collisions in the data, however reducing the error on the MTBF of one order of magnitude if compared with standard time based coalescence. In particular, results allows to conclude that when dealing with sporadic failures for which the MTBF is >> than MTTR both the two tested heuristics perform fairly the same. With reference to the used datasets, for large system and high failure rates, the standard time coalescence fails in delivering confident results causing an error reaching the 150%, while the MTW is able to avoid most of the accidental collisions. Even in the case of higher committed error, the indices computed by the MTW are valuable to weight the final results, therefore adding an extra level of introspection to the measurements. To the best of our knowledge, this is the first work proposing an approach to analyze supercomputer logs able to estimate the error caused on the final measurement, such as for the MTBF. Future work will consider augmenting the proposed MTW heuristic with data mining and machine learning techniques [17, 3] that showed to be effective in further reducing the impact of accidental collisions.

References

1. Guermouche, A., Ropars, T., Snir, M., Cappello, F.: Hydee: Failure containment without event logging for large scale send-deterministic mpi applications. In: 2012 IEEE 26th International on Parallel Distributed Processing Symposium (IPDPS), pp. 1216–1227 (May 2012)

2. Fu, S., Xu, C.: Exploring event correlation for failure prediction in coalitions of clusters. In: SC 2007: Proc. of the 2007 ACM/IEEE Conference on Supercomputing, pp. 1–12. ACM (2007)
3. Gainaru, A., Cappello, F., Snir, M., Kramer, W.: Fault prediction under the microscope: a closer look into hpc systems. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, pp. 77:1–77:11. IEEE Computer Society Press, Los Alamitos (2012)
4. Di Martino, C., Cinque, M., Cotroneo, D.: Assessing time coalescence techniques for the analysis of supercomputer logs. In: IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012), pp. 1–12 (2012)
5. Buckley, M.F., Siewiorek, D.P.: A comparative analysis of event tupling schemes. In: FTCS 1996: Proc. of the The Twenty-Sixth Annual Int. Symp. on Fault-Tolerant Computing (FTCS 1996), p. 294. IEEE Computer Society, Washington, DC (1996)
6. Liang, Y., Zhang, Y., Sivasubramaniam, A., Jette, M., Sahoo, R.: Bluegene/l failure analysis and prediction models. In: Int. Conference on Dependable Systems and Networks, DSN 2006, pp. 425–434 (2006)
7. Hansen, J., Siewiorek, D.: Models for time coalescence in event logs. In: Twenty-Second Int. Symp. on Fault-Tolerant Computing, FTCS-22, Digest of Papers, pp. 221–227 (July 1992)
8. Oliner, A., Stearley, J.: What supercomputers say: A study of five system logs. In: 37th Annual IEEE/IFIP Int. Conference on Dependable Systems and Networks, DSN 2007, pp. 575–584 (June 2007)
9. Sahoo, R.K., Sivasubramaniam, A., Squillante, M.S., Zhang, Y.: Failure data analysis of a large-scale heterogeneous server environment. In: DSN 2004: Proc. of the 2004 Int. Conference on Dependable Systems and Networks, p. 772. IEEE Computer Society, Washington, DC (2004)
10. Liang, Y., Sivasubramaniam, A., Moreira, J.: Filtering failure logs for a bluegene/l prototype. In: DSN 2005: Proc. of the 2005 Int. Conference on Dependable Systems and Networks, pp. 476–485. IEEE Computer Society, Washington, DC (2005)
11. Simache, C., Kaâniche, M., Saidane, A.: Event log based dependability analysis of windows nt and 2k systems. In: PRDC 2002: Proc. of the 2002 Pacific Rim Int. Symp. on Dependable Computing, p. 311. IEEE Computer Society, Washington, DC (2002)
12. Casola, V., Cuomo, A., Rak, M., Villano, U.: Security and performance trade-off in perfcloud. In: Guarracino, M.R., Vivien, F., Träff, J.L., Cannatoro, M., Danelutto, M., Hast, A., Perla, F., Knüpfer, A., Di Martino, B., Alexander, M. (eds.) Euro-Par-Workshop 2010. LNCS, vol. 6586, pp. 633–640. Springer, Heidelberg (2011)
13. Palmieri, F., Pardi, S., Veronesi, P.: A fault avoidance strategy improving the reliability of the EGI production grid infrastructure. In: Lu, C., Masuzawa, T., Mosbah, M. (eds.) OPODIS 2010. LNCS, vol. 6490, pp. 159–172. Springer, Heidelberg (2010)
14. Barone, G.B., Boccia, V., Bottalico, D., Carracciuolo, L., Doria, A., Laccetti, G.: Modelling the behaviour of an adaptive scheduling controller. In: 2012 Sixth International Conference on Complex, Intelligent and Software Intensive Systems (CISIS), pp. 438–442 (2012)
15. Tsao, M.M., Siewiorek, D.P.: Trend analysis on system error files. In: Thirteenth Annual International Symposium on Fault-Tolerant Computing, pp. 116–119 (July 1983)

16. Di Martino, C., Cotroneo, D., Kalbarczyk, Z., Iyer, R.K.: A framework for assessing the dependability of supercomputers via automated log analysis. In: DSN 2008: Sup. Volume of Proc. of the Int. Conference on Dependable Systems and Networks, Anchorage, AK, pp. 383–384 (2008)
17. Pecchia, A., Cotroneo, D., Kalbarczyk, Z., Iyer, R.K.: Improving log-based field failure data analysis of multi-node computing systems. In: 2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN), pp. 97–108 (June 2011)
18. Thakur, A., Iyer, R.K.: Analyze-now-an environment for collection and analysis of failures in a network of workstations. IEEE Transactions on Reliability 45(4), 561–570 (1996)
19. Kalyanakrishnam, M., Kalbarczyk, Z., Iyer, R.: Failure data analysis of a lan of windows nt based computers. In: Proc. of the 18th IEEE Symp. on Reliable Distributed Systems, pp. 178–187 (1999)
20. Stearley, J., Oliner, A.J.: Bad words: Finding faults in spirit's syslogs. In: 8th IEEE International Symposium on Cluster Computing and the Grid, CCGRID 2008, pp. 765–770 (May 2008)
21. Vaarandi, R.: Mining event logs with slct and loghound. In: IEEE Network Operations and Management Symposium, NOMS 2008, pp. 1071–1074 (April 2008)
22. Zheng, Z., Lan, Z., Park, B., Geist, A.: System log pre-processing to improve failure prediction. In: IEEE/IFIP International Conference on Dependable Systems Networks, DSN 2009, June 29-July 2, pp. 572–577 (2009)
23. Lal, R., Choi, G.: Error and failure analysis of a unix server. In: Proc. Third IEEE Int High-Assurance Systems Engineering Symp., pp. 232–239 (November 1998)
24. Cinque, M., Cotroneo, D., Kalbarczyk, Z., Iyer, R.K.: How do mobile phones fail? a failure data analysis of symbian os smart phones. In: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, pp. 585–594. IEEE Computer Society, Washington, DC (2007)

Tracking the Performance Evolution of Blue Gene Systems

Darren J. Kerbyson[1], Kevin J. Barker[1], Diego S. Gallo[2,*], Dong Chen[3], Jose R. Brunheroto[3], Kyung Dong Ryu[3], George L. Chiu[3], and Adolfy Hoisie[1]

[1] Pacific Northwest National Laboratory
Richland, WA 99352
{darren.kerbyson,kevin.barker,adolfy.hoise}@pnnl.gov
[2] IBM Research Brazil
São Paulo, SP 04007-900
dsgallo@br.ibm.com
[3] IBM T.J. Watson Research Center
Yorktown Heights, NY 10598
{chendong,brunhe,kryu,gchiu}@us.ibm.com

Abstract. IBM's Blue Gene supercomputer has evolved through three generations from the original Blue Gene/L to P to Q. A higher level of integration has enabled greater single-core performance, and a larger concurrency per compute node. Although these changes have brought with them a higher overall system peak-performance, no study has examined in detail the evolution of performance across system generations. In this work we make two significant contributions – that of providing a comparative performance analysis across Blue Gene generations using a consistent set of tests, and also in providing a validated performance model of the NEK-Bone proxy application. The combination of empirical analysis and the predictive performance model enable us to not only directly compare measured performance but also allow for a comparison of system configurations that cannot currently be measured. We provide insights into how the changing characteristics of Blue Gene have impacted on the application performance, as well as what future systems may be able to achieve.

Keywords: High Performance Computing, Performance Evaluation, Performance Modeling, Massively Parallel Processing.

1 Introduction

Since its introduction in 2004 IBM Blue Gene® has seen some of the largest systems deployed for High Performance Computing applications. Its approach has always been to provide leading overall system performance through extreme levels of parallelism but with modest single-core performance. Blue Gene has also achieved leading levels of energy efficiency placing each generation at the top of the Green500 [1].

[*] Work done while at IBM T. J. Watson Research Center.

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 317–329, 2013.

In 1999 IBM was in the process of developing two massively parallel computing technologies known at that time as Blue Gene [2] and Blue Light [3]. They had similar 3-D system topologies but had very different processor core designs and physical packaging. Blue Gene soon became known as Blue Gene/C and later as Cyclops64 [4] and subsequent systems have been produced [5]. Blue Light became known as Blue Gene/L (BG/L) – the first in the series of systems that we are concerned with here.

A direct precursor to BG/L was the QCDOC system that was designed and implemented with a 6-D nearest neighbor network for QCD calculations [6,7]. This system was deployed at Columbia University and Edinburgh University. A 128-node BG/L prototype that used early silicon was also produced in 2003 [8].

To date there have been three generations of Blue Gene systems which have appeared at various scales throughout the world. Each successive generation has seen substantial gains in performance capabilities from increases in both levels of concurrency and in single-core performance. BG/L was the first [9], available in 2004, that used a dual-core processing node with a 3-D torus network for data transfer. The quad-core based BG/P was introduced in 2007 [10] and retained the 3-D system topology. The latest Blue Gene, BG/Q, was introduced in 2011 [11], having 16 application cores per node with a 5-D system topology [12].

In this work we provide a unique perspective on the evolution of Blue Gene through a performance analysis that examines sub-system characteristics, including memory and communication performance, as well as application level performance. The approach that we take combines both empirical analysis as well as predictive performance models that enable system configurations that cannot be measured to be analyzed and compared. We chose the NEK-Bone application as a vehicle to compare application performance. This, a proxy application, has been recently developed within the DOE CESAR (Center for Exascale Simulation of Advanced Reactors) co-design center [13] and represents a significant part of the NEK5000 CFD spectral element application [14]. At the core of NEK-Bone is a Conjugate Gradient (CG) solver that is a key component in many large-scale scientific applications.

The main contributions of this work are:

- a side-by-side performance comparison of all three generations of Blue Gene using a common set of tests,
- the development and validation of a performance model of NEK-Bone that can be used to accurately predict performance at scales unavailable for measurement, and
- an analysis of the relative performance of the Blue Genes on NEK-Bone using a combination of empirical analysis and performance modeling.

The model for NEK-Bone not only validates the observed performance across the existing three generations of Blue Gene but can also be used gain insights into the design of possible future systems.

The rest of the paper is organized as follows. In Section 2 we provide an overview of the three generations of Blue Gene as well as a comparison of their architectural characteristics. An empirical analysis of each system's memory, communication, collectives, and O/S noise characteristics is given in Section 3. An overview of NEK-Bone along with its performance model is described in Section 4. A comparison of the application performance on the three Blue Genes is provided in Section 5. Conclusions from this work are discussed in Section 6.

2 Blue Gene Systems

The IBM Blue Gene line of massively parallel supercomputers has been designed from the outset to be highly power efficient, scalable, reliable, with highly dense packaging. An overview of each of the three generations is given below and a summary of their architectural characteristics is listed in Table 1. Each of the systems has similar packaging: a single compute-card is comprised of 32 compute-nodes, 16 compute-cards compose a mid-plane, and two mid-planes compose a single-rack.

Blue Gene/L: The first generation, BG/L, used a compute-node ASIC that incorporated two embedded PowerPC 440 cores, 4 MB EDRAM as an on-chip cache and multiple networks. Each core is capable of two double-precision fused multiply-adds (four flops per cycle), and at 700 MHz each compute-node has a peak performance of 5.6 Gflops. A compute-node has a modest main memory of either 512 MB or 1 GB. The integrated 3-D (XYZ) torus network for inter-node data transfers has six communication links, each running at a peak of 175+175 MB/s (send+receive). An independent collective network has 3 ports per compute-node, with each running at 350+350 MB/s. The largest installed BG/L system at LLNL consists of 104 racks (106,496 nodes), with a peak performance of 598 Tflops.

Blue Gene/P: The second generation, BG/P, saw an increase in the number of cores in each compute-node to four and used the PowerPC 450 with a 850MHz clock rate that still issued two fused multiply-adds per cycle. The node peak-performance increased to 13.6 Gflops as did the on-chip cache to 8 MB. Each node has two memory controllers and can support either 2 or 4 GB of main memory. BG/P retained BG/L's 3-D network topology and separate collective network but both had higher communication speeds. The peak link speed of the 3-D torus network increased to 425+425 MB/s and the collective network link speed increased to 850+850 MB/s. The largest installed system at Juelich has 72 racks with a peak performance of 1.003 Pflops.

Blue Gene/Q: The third and latest generation, BG/Q, has further increased the core count per compute-node to 18. Though each core is identical in its functionality, only 17 are used to increase manufacturing yield. 16 cores are available to applications with the 17th dedicated to running a light weight OS. Each of the PowerPC A2 cores can support four threads using 4-way Simultaneous Multi-Threading (SMT) and each can issue at most one instruction, integer or floating-point, per cycle. Each core contains a quad floating-point processing unit (QPU) that can perform up to four double-precision fused multiply-adds per cycle, resulting in a peak-performance of 204.8 Gflops per node. The on-chip L2 cache increased to 32 MB, and is shared by all cores. Each node has 16 GB of main memory. BG/Q's system network has a 5-D (ABCDE) toroidal topology and an increased peak link speed of 2+2 GB/s for an aggregate of 20+20 GB/s bandwidth from a node. The fifth (E) dimension is always two wide, and contained within a compute-card to minimize wiring. In addition, the function of the two earlier generations collective network has been integrated into the torus network. A BG/Q rack is water-cooled unlike BG/L and BG/P that are air-cooled. The largest system at LLNL has 96 racks and a peak of 20 Pflops.

Table 1. Summary of Blue Gene Architectural Characteristcs

System	Blue Gene/L	Blue Gene/P	Blue Gene/Q
Year of Introduction	2004	2007	2011
Largest system (Racks)	104 (LLNL)	72 (Juelich)	96 (LLNL)
Largest system (nodes)	106,496	73,728	98,304
Largest system (Peak Pf/s)	0.6	1	20
Processor core			
type	PowerPC 440	PowerPC 450	PowerPC A2
Clock speed (GHz)	0.7	0.85	1.6
Threads	1	1	4
Flops/cycle	4	4	8
Peak performance (Gf/s)	2.8	3.4	12.8
L1 cache I+D	32KB+32KB	32KB+32KB	16KB+16KB
L2 shared cache	4MB	8MB	32MB
Node			
Cores	2	4	16
Peak performance (Gf/s)	5.6	13.6	204.8
Memory (GB)	0.5 - 1	2 - 4	16
Memory channels per node	1	2	2
Memory speed	DDR-350	DDR2-425	DDR3-1333
Peak memory bandwidth (GB/s)	5.6	13.6	42.6
System Network			
Topology	3D torus	3D torus	5D torus
Peak link speed (GB/s)	0.175 + 0.175	0.425 + 0.425	2.00 + 2.00
Aggregate bandwidth (GB/s)	1.03 +1.03	2.55 + 2.55	20.0 + 20.0
Topology of largest system	104×32×32	72×32×32	16×16×16×12×2
Collective Network			
Type	Independent	Independent	Integrated in torus
Collective latency (72 racks)	6.0 µs	5.0 µs	6.0 µs
Collective bandwidth (per-port)	0.325 + 0.325	0.850 + 0.850	2.0 + 2.0

3 Performance Evolution of Sub-systems

Our approach, prior to the analysis of application performance on any system, is to use microbenchmarks to examine separately important system characteristics that have a direct impact on performance. Here we examine the: memory bandwidth, OS noise, point-to-point and collective communication, and network congestion.

Three Blue Gene systems available at IBM T.J. Watson Research Center were used to obtain all empirical data. Each system contained either 16K or 32K cores. The BG/L system consisted of 16 racks, the BG/P had four, and the BG/Q consisted of a single rack. Their configurations, compiler and software driver/MPI versions are listed in Table 2. Note that the process allocation ordering on BG/L and BG/P are in terms of their X, Y and Z dimensions and on BG/Q in terms of its 5-D network labeled as A, B, C, D and E. In all cases processes are packed within a node.

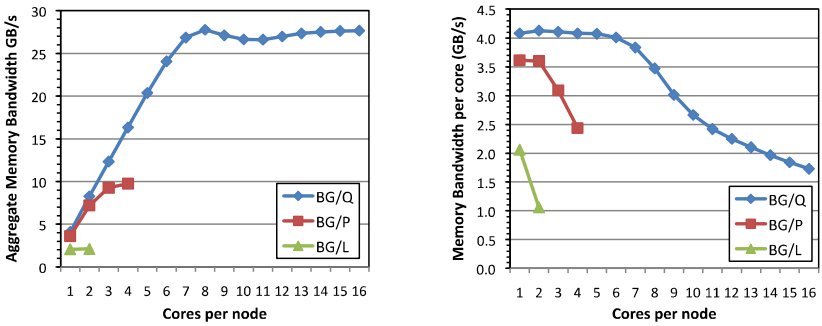
3.1 Memory Performance

The memory bandwidth of each node was examined using the MPI version of the STREAM benchmark [16]. The achieved aggregate memory bandwidth is shown in

Table 2. Configurations of the test systems

	BG/L	BG/P	BG/Q
Racks	16	4	1
Nodes	16K	4K	1K
Cores	32K	16K	16K
Memory / Node (GB)	1	4	16
Network Topolgy	32×32×16	16×16×16	8×4×4×4×2
Min. node allocation	512	32	32
Allocation ordering	*TXYZ*	*TXYZ*	*ABCDET*
Driver/MPI verions	V1R3_300	V1R4M2_200	V1R1M1
Compiler versions	XLC 9.0 XLF 11.1	XLC 9.0 XLF 11.1	XLC 12.1 XLF 14.1

Fig. 1(a) when varying the number of cores used per node. The maximum bandwidth observed for a single node is 2.1 GB/s, 9.7 GB/s and 27.8 GB/s for a BG/L, BG/P and BG/Q node respectively. It can be seen that the aggregate bandwidth generally increases with the core-count but achieves a maximum when using fewer cores than the maximum available. The same data is presented in Fig. 1(b) but in terms of memory bandwidth per-core. The per-core bandwidth on BG/Q shows a consistent performance up to 6 cores before it degrades to 43% of its peak. Interestingly, the per-core bandwidth on BG/P exceeds that of BG/Q when using over 11 cores per node.



(a) aggregate bandwidth (b) per-core bandwidth

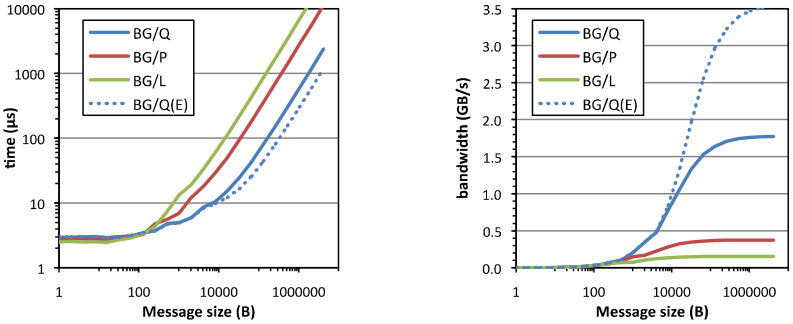
Fig. 1. Memory Bandwidth (STREAM)

3.2 Operating System Noise

The impact of the OS on each core was examined using P-SNAP [15]. This uses a fixed-work-quantum approach in which a single computation with known expected run-time (e.g. 1ms) is repeatedly measured. The computation is measured for several million iterations and the actual time taken to complete each is recorded. The observations were found to be highly consistent across all nodes within each system with an average slow-down of 0.035%, 0.06%, and 0.01% on BG/L, BG/P and BG/Q respectively. All systems exhibit levels of noise that are much lower than on other commodity-based supercomputers. In particular the virtually zero level of noise on BG/Q is achieved through the dedication of its 17th core to offload OS activities.

3.3 Point-to-Point Communication Performance

The MPI point-to-point performance is shown in Fig. 2 between a pair of processors on adjacent nodes. The message latency in Fig. 2(a) and bandwidth is shown in Fig. 2(b). The small message latency is comparable across the three systems at 2.5µs, 2.7µs, 2.4µs and the large message bandwidth is 0.154GB/s, 0.374GB/s, 1.77GB/s on BG/L, BG/P and BG/Q respectively. Note that nodes that are adjacent in the E dimension on BG/Q (node 0 and 1 in this case) can use both ±E communication channels and hence achieve double bandwidth compared to communication with other nodes.



(a) message latency (b) message bandwidth

Fig. 2. Unidirectional MPI point-to-point performance

The MPI point-to-point performance was also measured from core 0 in the first allocated node to core 0 in each of the other nodes in each system. This test exposes the topology of each system's network and results in the interesting curves in Fig 3. On BG/L the latency varies from a minimum of 2.48µs to 5.45µs in a saw-tooth waveform at a high frequency of width 32 (the number of nodes in a row on the test system), and low frequency of 1024 (the number of nodes in an XY plane). Similarly on BG/P the waveform reflects the 16 node row size, and 256 XY plane size with a latency that varies between 2.68µs and 3.83µs. The 5-D topology of BG/Q results in a more consistent latency of between 2.35µs and 2.68µs. The per-hop latency is approximately 100ns, 60ns and 30ns on BG/L, BG/P and BG/Q respectively.

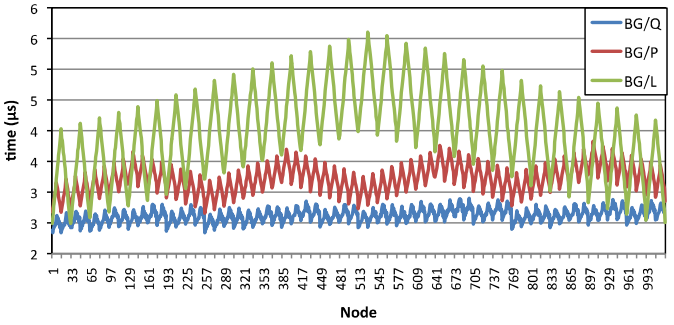
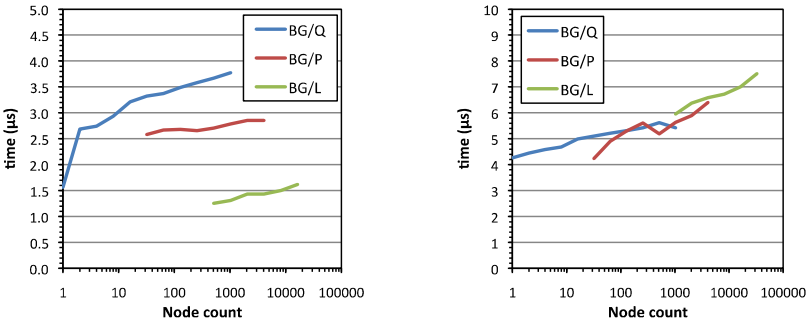


Fig. 3. 0-byte latency from node 0 to all others (first 1024 nodes shown)

3.4 Collective Performance

MPI_barrier and MPI_allreduce (1-word) performance is shown in Fig. 4 using all cores in a node. Note that the minimum node count for BG/L and BG/P is that of the minimum allocation size. A software collective is used when using less than the allocated number of nodes on these systems rather than the hardware collective network and hence results in much larger times. Barrier is slightly faster than allreduce on all systems. When using 1,024 nodes the time for allreduce is similar across the three systems at 6.4µs, 5.6µs, 5.4µs on BG/L, BG/P and BG/Q respectively.



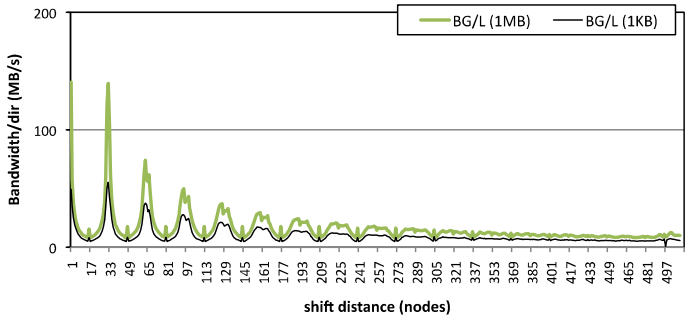
(a) Barrier (b) Allreduce

Fig. 4. Performance of collectives on the three systems

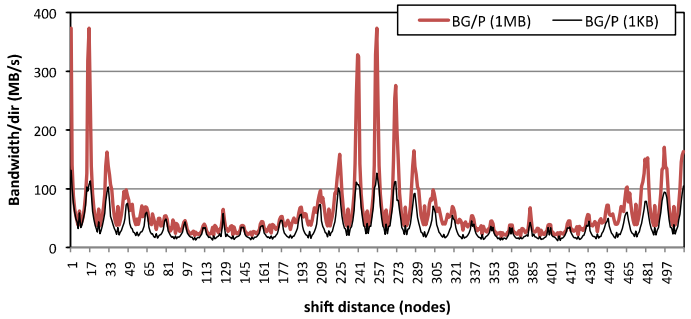
3.5 Communication Pattern Performance

The shift communication pattern was used to analyze the performance of each system's network. In this each processor P_i sends a message to P_{i+d} for all processor-cores i at a logical shift distance d apart. This is measured for a range of distances, d=1..n, where n is the number of processes. A different communication pattern results for each shift distance. Such patterns closely correspond to those that occur in regular dense grid applications when exchanging boundary data between sub-grids.

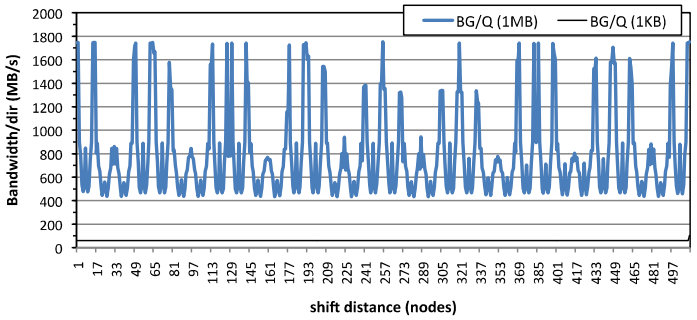
Several effects contribute to the bandwidth observed for the shift pattern as shown in Fig. 5. The mapping of the logical communication pattern onto the physical network results in different degrees of network contention that varies across machines due to their different network configurations (Table 2). Thus each pattern results in a different network contention and in different bandwidths which can be clearly seen for the large message (1MB) case. The peak bandwidth, when there is no contention in the network, corresponds to the single point-to-point peak of each system. The maximum contention corresponds to halve of the maximum topology dimension size in each of the system – this is 16 for BG/L (half of both the X and Y dimensions), eight for BG/P, and four for BG/Q (half of the A dimension). The achieved communication bandwidth shown in Fig. 5 is approximately equal to the peak divided by the contention in the network. BG/Q's higher dimensional torus network results in the lower contention factors and hence a smaller range between the max and min compared with BG/L and BG/P. It is also interesting to note that the observed bandwidth for small messages also undergoes similar contention as for large messages on BG/L and BG/P. But on BG/Q the same bandwidth is achieved no matter the contention.



(a) BG/L



(b) BG/P



(c) BG/Q

Fig. 5. Observed bandwidth per direction for the shift communication (one processor per node)

4 Case Study: NEK-Bone

4.1 Overview of NEK-Bone

NEK-Bone is a proxy application developed as part of the CESAR (Center for Exascale Simulation of Advanced Reactors) co-design center [13]. This mini-app was chosen to be modeled and studied based on our performance modeling methodology [17], in order to allow the creation of a validated model to analyze multiple architectures. While NEK-Bone is a stripped down version of the full NEK5000 developed at Argonne National Laboratory, it resembles the basic structure of the full code and

uses exactly the same communication substrate as the full application. At the heart of NEK-Bone is a Conjugate Gradient (CG) solver that uses a simplified pre-conditioner. The problem size can be configured in terms of the polynomial degree, the number of MPI processes, and the number of elements per process.

4.2 Performance Model of NEK-Bone

The normal mode of operation of NEK-Bone is that of weak-scaling in which the global problem size grows with the processor count and the size per processor remains constant. The main data structure processed is determined by two parameters: L that controls the polynomial degree of the calculation, and N the number of elements assigned to each MPI rank or process. The size of each element is L^3, and the problem size per process is $N.L^3$. The total number of processes consists of a logical 3-D arrangement of $P_X \times P_y \times P_z$. On a single process there are $N_X \times N_y \times N_z$ ($=N$) elements. The arrangement of elements in a single process is kept as close to a cube as possible.

The main inter-process communication in NEK-Bone consists of boundary exchanges between logically adjacent processor-cores using NEK's jl gather-scatter routines. The boundaries consist of a single layer of cells of the local volume that is communicated in all three dimensions. Note that the majority of the communication traffic takes place with processors that are at a logical distance of ± 1 (for X dimension boundary exchanges), $\pm P_X$ (for Y), and $\pm P_X * P_y$ (for Z). There are also communications with diagonally neighboring processors in 3-D that results in a total of 26 communications from a processor to its neighbors. A total of 6 surface (a plane of grid-points), 12 edge (a line of grid-points), and 8 corner communications result.

The performance model reflects the processing that is contained within an iteration of NEK-Bone's CG-solver: local computation, boundary communications between processors, and three one-word allreduce collective communications. The model does not predict the number of CG iterations but rather predicts the time for an iteration. The overview of the model is as follows:

$$T_{iter} = T_{compute}(P,N) + T_{collective}(P) + T_{boundary}(P,N)$$

The compute time is given by $T_{compute}=N.T_{elem}(P_{pernode})$, where the time per element, $T_{elem}(P_{pernode})$, is measured on a single node of a system while varying the number of cores per node, $P_{pernode}$, from 1 to the maximum. The collective time is given by $T_{collective}(P)=3.T_{allreduce}(P)$ where the $T_{allreduce}(P)$ is the measured time for an MPI_Allreduce of size one word on P processor-cores. The boundary communication time is taken to be the summation of the 26 boundary communication times:

$$T_{boundary(P,N)} = \sum_{i=1}^{26} \left(L(S_i) + C_i . S_i / B(S_i) \right)$$

where $L(S_i)$ and $B(S_i)$ are the effective latency and bandwidth across the system network for a message of size S_i, and C_i is the maximum contention across a communication channel in the network for communication i. The shift microbenchmark was used to determine the effective latency and bandwidth. The contention factors are determined by the number of inter-node communications that are done by the cores in a node in each of the communication stages. This is determined by the number of cores per node and by the logical arrangement of processes at a particular scale.

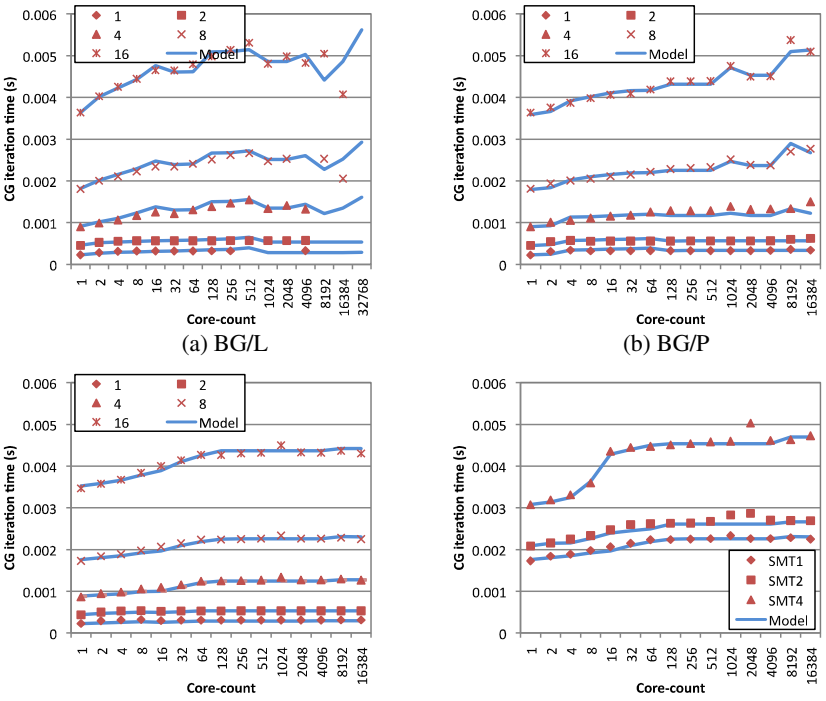
Table 3. Values for T_{elem} input to theNEK-Bone performance model

T_{elem} (μs)	BG/L	BG/P	BG/Q (SMT1)	BG/Q (SMT2)	BG/Q (SMT4)
1 process/node	227	224	220		
2 process/node	250	227	223	261	
4 process/node		231	226	270	384
8 process/node			232	282	398
16 process/node			240	292	414
32 process/node				309	450
64 process/node					545

4.3 Validation of NEK-Bone Model

The values for $T_{collective}(P)$, $L(S_i)$ and $B(S_i)$ were presented in Fig. 4 and 5 respectively. NEK-Bone was run varying the polynomial degree (L=8,10,12) and the number of elements (N=1,2,4,8,16,32) per core. In addition the number of threads per core was varied (SMT=1,2,4) on BG/Q. The time per element, $T_{elem}(P_{pernode})$, Table 3, was measured on a single node, assigning 32 elements to each processor-core and using between one and the maximum number of cores, and on BG/Q varying the SMT level.

A sub-set of the measured and modeled performance of NEK-Bone is shown in Fig. 6(a-c) for L=10, and N between 1 and 16. The performance model captures the full behavior of the code's performance with high accuracy. The mean absolute percentage prediction errors were 5.4%, 4.2%, and 2.9% on BG/L, BG/P, and BG/Q respectively.



(a) BG/L (b) BG/P

(c) BG/Q (d) BG/Q with SMT=1,2,4

Fig. 6. Measured and modeled performance of NEK-BONE (L=10)

Also shown in Fig. 6(d) is a comparison showing the use of BG/Q's Simultaneous Multi-Threading (SMT) that supports up to four threads. For this comparison the number of elements per thread was kept constant at 8. The iteration time increases only slightly between the use of SMT=1 and SMT=2 even though twice the amount of work is being done. The iteration time increases by just over 2x when using SMT=4 even though quadruple of the amount of work is done.

5 Performance Analysis

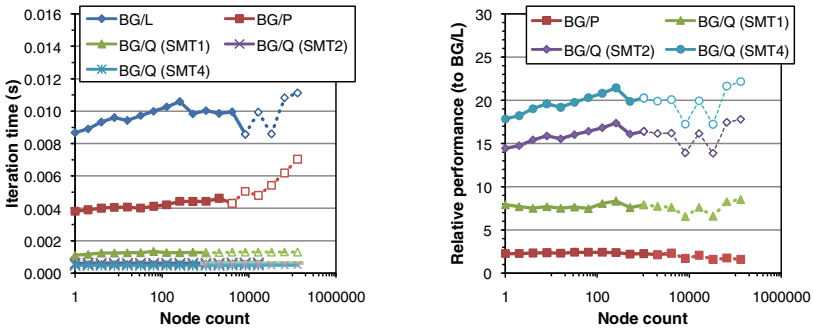
We compare the iteration time of NEK-Bone's CG solver across the three generations of Blue Gene in Fig. 7(a). In this analysis a combination of both empirical data, up to the size of the test systems, and modeled data is used allowing comparisons to be made at larger scales. The problem size per node is fixed at 64 by varying N per process on each system (N=32 on BG/L, N=16 on BG/P, N=4,2,1 on BG/Q using SMT=1,2,4 respectively). This allows for a direct comparison on a node-to-node basis across systems. L was fixed at 10. It can be seen that there is some variation in the iteration time on BG/L, especially at large node counts, corresponding to differences in the amount of network contention at certain scales. For the same reason, the iteration time on BG/P also increases at large scales. However the iteration time on BG/Q increases only slightly due to the lower contention in its higher dimensional torus.

The relative performance between BG/P and BG/Q to BG/L is shown in Fig 7(b) and contains the same scaling effects as the raw data in Fig. 7(a). The improved performance on BG/P compared to BG/L peaks at 2.8x but decreases to 1.8x at 256K nodes. The improved performance on BG/Q compared to BG/P varies between 6x and 8.5x using one thread per core. When using SMT2 or SMT4 the performance increases to ~16x and ~20x respectively. This performance comparison shows that the use of SMT on BG/Q is beneficial and whose impact persists at large-scale.

6 Related Work

This work is a first performance comparison of the three generations of IBM's Blue Gene. As far as the authors are aware there is currently no other published analysis of Blue Gene systems other than detailed descriptions of their architecture and peak capabilities. This includes [18,19]. Microbenchmarks for analyzing communication performance are widespread in terms of point-to-point messaging and collective communication. Such analysis is often coupled with application performance under varying configurations but often lacks modeling or simulation that assists in relating the performance of architectural characteristics to the observed application performance. Recent work on the empirical analysis of large-scale systems includes clusters using Intel processors [20], Cray systems [21], and system comparisons [21,23].

There are a number other works that use modeling approaches similar to our own [22,24] including that in use for the exploration of extreme scale systems e.g. [25]. Many of these approaches are often limited in their predictive accuracy, and have not been through an extensive validation process on current systems. In addition a number of simulators are in development, e.g. [27-28] that promise high accuracy for large-scale systems but often require high runtime or large resources to provide predictions.



(a) Iteration time (b) Relative performance

Fig. 7. Performance comparison of NEK-Bone across Blue Genes

7 Conclusions

The Blue Gene series of systems has seen a tremendous growth in processing capability that has resulted in significant increases in achievable performance. In this work we have analyzed all three generations: BG/L, BG/P and BG/Q using a set of micro-benchmarks to analyze individual characteristics of each system, and the NEK-Bone application using a combination of both empirical analysis and performance modeling. Our analysis has shown that there has been significant improvement in each generation especially in terms of their processing and communication capabilities though the latency for collective and point-to-point communication has only reduced slightly.

In particular the peak processing capabilities of a node has substantially increased from 5.6GF/s to 204.8 GF/s from BG/L to BG/Q, and the achievable inter-node communication bandwidth has risen from 154MB/s to 1.75GB/s for a single point-to-point communication while at the same time the number of communication channels from each node has increased from 6 (3-D torus) to 10 (5-D torus). Our analysis of NEK-Bone has shown a similar improvement of processing capabilities by a factor of 1.8x from BG/L to BG/P, and by a factor of 20x from BG/L to BG/Q.

The model for NEK-Bone not only validates the observed performance across the existing Blue Gene generations but also is being actively used to provide insights into the design of possible future systems.

Acknowledgements. This research was supported in part by DOE Office of Science through the CESAR Co-design center. We thank Rusty Lusk, Katie Heisey, and Paul Fischer at ANL, and Fred Mintzer, Dave Singer and Bob Walkup at IBM. The Pacific Northwest National Laboratory is operated by Battelle for the U.S. Department of Energy under contract DE-AC05-76RL01830.

References

1. Green500, `http://www.green500.org/`
2. Allen, F., et al.: Blue Gene: A Vision for Protein Science using a Petaflop Supercomputer. IBM Sys. J. 40(2), 310–327 (2001)

3. Almasi, G., et al.: Demonstrating the Scalability of a Molecular Dynamics Application on a Petaflops Computer. Int. J. of Parallel Programming 30(4), 317–351 (2002)
4. Almasi, G., et al.: Dissecting Cyclops: A Detailed Analysis of a Multithreaded Architecture. ACM SIGARCH Computer Architecture News 31(1), 26–38 (2003)
5. Adiga, N.R., et al.: An Overview of the Blue Gene/L Supercomputer. In: Proc. IEEE/ACM Supercomputing, SC 2002, Baltimore (2002)
6. Boyle, P.A., Chen, D., et al.: QCDOC: A 10 Teraflops Computer for Tightly-Coupled Calculations. In: Proc. IEEE/ACM Supercomputing (SC 2004), Pittsbugh (2004)
7. Chen, D., et al.: QCDOC: A 10-Teraflops Scale Computer for Lattice QCD. Nucl. Phys. B 94(1-3), 825–832 (2001)
8. Davis, K., Hoisie, A., Johnson, G., Kerbyson, D.J., Lang, M., Pakin, S., Petrini, F.: A Performance and Scalability Analysis of the BlueGene/L Architecture. In: SC 2004, Pittsburgh (2004)
9. Almasi, G., et al.: Unlocking the Performance of the BlueGene/L Supercomputer. In: Proc. IEEE/ACM Supercomputing (SC 2004), Pittsburgh (2004)
10. IBM Blue Gene Staff: Overview of the IBM Blue Gene/P Project. IBM J. Res. & Dev. (1/2), 199–220 (2008)
11. Haring, R.A., Ohmacht, M., Fox, T.W., et al.: The IBM Blue Gene/Q Compute Chip. IEEE Micro, 48–60 (2012)
12. Chen, D., Choudhury, A., Eisley, N., et al.: Looking Under the hood of the Blue Gene/Q Network. In: Proc. IEEE/ACM Supercomputing (SC 2012), Salt Lake City (2012)
13. `https://cesar.mcs.anl.gov/content/software/nekbone`
14. H.: Terascale Spectral Element Algorithms and Implementations. In: Proc. IEEE/ACM Supercomputing (SC 1999), Portland (1999)
15. P-SNAP v1.2, `http://www.c3.lanl.gov/pal/software`
16. McCalpin, J.: Memory bandwidth and machine balance in current high performance computers. In: IEEE Tech. Committee on Computer Architecture (TCCA), pp. 19–25 (1995)
17. Barker, K.J., Davis, K., Hoisie, A., Kerbyson, D.J., Lang, M., Pakin, S., Sancho, J.C.: Using performance modeling to design large-scale systems. IEEE Computer 42(11), 42–49 (2009)
18. Chen, D., Eisley, N.A., Heidelberger, P., et al.: The IBM Blue Gene/Q Interconnection Fabric. IEEE Micro 32(1), 32–43 (2012)
19. Chen, D., Eisley, N.A., Heidelberger, P., et al.: The IBM Blue Gene/Q Interconnection Network and Message Unit. In: Proc. IEEE/ACM Supercomputing, SC 2011, Seattle (2011)
20. Saini, S., Naraikin, A., Biswas, R., Barkai, D., Sandstrom, T.: Early Performance Evaluation of Nehalem Cluster Using Scientific and Engineering Applications. In: SC 2009, Portland (2009)
21. Hoisie, A., Johnson, G., Kerbyson, D.J., Lang, M., Pakin, S.: A Performance Comparison through Benchmarking and Modeling of Three Leading Supercomputers: Blue Gene/L, Red Storm, and Purple. In: Proc. IEEE/ACM Supercomputing (SC 2006), Tampa (2006)
22. Kerbyson, D.J., Hoisie, A.: Performance Modeling of the Blue Gene Architecture. In: IEEE John Vincent Atanasoff Int. Symp. on Modern Computing, Sofia, Bulgaria (2006)
23. Bhatele, A., Wesolowski, L., Bohm, E., Solomonik, E., Kale, L.V.: Understanding application performance via micro-benchmarks on three large supercomputers: Intrepid, Ranger and Jaguar. In: Int. J. of High Performance Computing Applications, IJHPCA (2010)
24. Rodrigues, R.F., et al.: The Structural Simulation Toolkit. ACM Sigmetrics Performance Evaluation Review 38(4), 37–42 (2011)
25. Zheng, G., Kakulapati, G., Kale, L.V.: BigSim: A Parallel Simulator for Performance Prediction of Extremely Large Machines. In: Proc. IPDPS, Santa Fe (2004)

Accelerators for Technical Computing: Is It Worth the Pain? A TCO Perspective

Sandra Wienke, Dieter an Mey, and Matthias S. Müller

Center for Computing and Communication, RWTH Aachen University, D-52074 Aachen
JARA – High-Performance Computing, Schinkelstr. 2, D-52062 Aachen
{wienke,anmey,mueller}@rz.rwth-aachen.de

Abstract. Nowadays, HPC systems emerge in a great variety including commodity processors with attached accelerators which promise to improve the performance per watt ratio. These heterogeneous architectures often get far more complex to employ. Therefore, a hardware purchase decision should not only take capital expenses and operational costs such as power consumption into account, but also manpower. In this work, we take a look at the total cost of ownership (TCO) that includes costs for administration and programming effort. From that, we compute the costs per program run which can be used as a comparison metric for a purchase decision. In a case study, we evaluate our approach on two real-world simulation applications on Intel Xeon architectures, NVIDIA GPUs and Intel Xeon Phis by using different programming models: OpenCL, OpenACC, OpenMP and Intel's Language Extensions for Offload.

Keywords: TCO, heterogeneous architectures, GPU, Intel Xeon Phi, programming effort, OpenCL, OpenACC, OpenMP, Intel LEO, energy efficiency.

1 Introduction

On the way to exascale computing, the HPC community is aiming at increasing system performance while keeping a tight rein on its power consumption. To this end, performance per watt has become a common metric to compare different hardware architectures and nowadays heterogeneous HPC systems – combining commodity processors with accelerators – are in front of this efficiency comparison [1].

However, relying solely on the (HPL) performance per watt evaluation and a low energy bill is not advisable for hardware purchase decision making in university computing centers. Especially, their needs concerning the applications should signify for the assessment since they influence power consumption and performance. In this case, comparing comprehensively total costs of ownership (TCO) of different systems is an appropriate approach. While TCO calculations comprise numerous parameters such as acquisition and operational costs, the manpower costs for programmers are often not taken into account. The programming effort that is needed to port a code to and fully exploit the desirable hardware may significantly differ depending on the complexity of a (heterogeneous) architecture and of its programming model.

In this paper, we make a first approach to quantify these total costs including the programming effort for different hardware architectures and programming models. Thereby,

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 330–342, 2013.

we want to investigate the question whether it is preferable to buy accelerators. Assuming a given fixed one-time budget and a homogeneous application scenario, we compute the cost per program execution for several hardware architectures and use it as a comparison metric. We illustrate our approach in a case study based on our experience at RWTH Aachen University. We look at Intel Sandy Bridge servers and compare them to nodes with an attached NVIDIA Fermi GPU or Intel Phi coprocessor. Our case study considers two real-world simulation codes from the fields of engineering and biomedicine that serve as a basis for values in programming effort, performance and power consumption. We show results for the following programming approaches: OpenMP on Intel servers, a combination of OpenMP and Intel's Language Extensions for Offload (LEO) on Intel's Xeon Phi and OpenCL and OpenACC on NVIDIA's Fermi GPU.

The paper is structured as follows: Sect. 2 covers related work. In Sect. 3, we explain our TCO perspective and introduce the cost per program run as comparison metrics between system types. In the case study in Sect. 4, we investigate whether the accelerators effort pays off. Finally, we summarize our findings in Sect. 5.

2 Related Work

In data center total ownership costs, different metrics were established. Carlyle [2] looks at node hour cost for comparison of community centers with cloud computing services, while Turner and Seader [3] propose a combination of cost per square foot and cost per watt due to recent power and cooling considerations. Applying cost per watt, Patterson et al [4] compare different data center density designs while only giving some major cost deltas. In contrast, we compute absolute total ownership costs including comprehensive capital and operational expenses (called "True TCO" by Koomey [5]) and further compare different system types by the metric "cost per program run" given a fixed investment and system lifetime. Basing on our attempt to estimate the TCO of the RWTH Aachen University's HPC equipment in [6], we additionally quantify manpower costs for operating and programming different compute nodes. While administration effort is considered in few TCO calculations (e.g. [5]), programming effort is rarely investigated as a quantified TCO aspect for computing centers. Kuck [7] elaborates on productivity issues in HPC that also include development effort and defines TCO as the sum of total cost of purchase (TCP), total cost of operation and maintenance (TCOM) and total cost of applications development (TCAD). However, his approach misses to quantify TCO in real numbers. Simultaneously, software estimation models are aware of development and maintenance effort while not putting hardware costs into the equation [8]. The emergence of accelerators in high-performance technical computing increases the TCO complexity and makes a fair comparison to nodes with commodity processors challenging. In previous works [9,10], we have seen the importance of development productivity especially on (GPU) accelerators, also dependent on the used programming model and the kind of application. CAPS' case study pamphlet [11] gives a short overview of the economics of GPU code migration and draws the conclusion that GPUs are worthwhile when gathering at least a two-fold speedup. Besides a carefully investigated TCO calculation of Intel servers and an NVIDIA Fermi GPU, we will also take a look at the just recently released Intel Xeon Phi.

3 Total Cost of Ownership

"Total cost of ownership represents the cost to the owner to purchase/build, operate and maintain a data center" [12] and comprises numerous parameters such as capital expenses, energy and maintenance costs. Defining and quantifying all parameters is a challenging task that we have just started to tackle with respect to the RWTH's high-performance computing center. Our focus is to include administration and especially programming efforts for novel heterogeneous architectures into the TCO calculation. In the following, we divide TCO in one-time and annual costs and differ between per-node and per-node-type costs that arise for each compute device or just once for each system type, respectively. Table 2 gives an overview of all components. For modeling these TCO parameters, a spreadsheet is publicly available on our webpage [13].

3.1 One-Time Costs

One-time costs comprise the initial expenses for hardware, building and infrastructure, but also for manpower. Concerning hardware costs, we look at computing facilities, whereby costs for storage and networking can be included into the infrastructure component. Manpower costs arise for the installation of an operation system (OS), the environment setup and for developing or porting user applications that leverage the investigated architecture kind. They may differ highly depending on the system type, ease of use of the programming language, availability of tools and the base knowledge of the employees. The latter makes it also challenging for fair comparisons and is an issue that should be further addressed in future. All together, we define the one-time costs C_{ot} by

$$C_{ot} = C_A \cdot n + C_B$$

where C_A is the sum of all one-time costs per node, C_B the sum of all one-time costs per node type and n is the number of nodes that can be bought by a given investment I.

3.2 Annual Costs

Annual costs aggregate expenses for maintaining the hardware, the OS, the software environment and the user applications, but energy costs got the most attention in recent exascale discussions. To this end, we have to keep in mind that power consumption rises with performance and that in an accelerator machine the host processor also uses energy. The costs per anno C_{pa} are given with C_C the sum over all annual costs per node, C_D the sum over all annual costs per node type and n the number of nodes:

$$C_{pa} = C_C \cdot n + C_D$$

Combining both cost types, we define the total cost of ownership as a function of the number of nodes n and the lifetime τ of the system:

$$\text{TCO}(n, \tau) = C_{ot} + C_{pa} \cdot \tau = (C_A + C_C \cdot \tau) \cdot n + C_B + C_D \cdot \tau \quad (1)$$

Given a fixed budget I and lifetime τ (which is between 3-6 years in most HPC centers), we can compute the number of nodes n by solving (2) for n.

$$\text{Investment } I = \text{TCO}(n, \tau) \quad (2)$$

3.3 Costs per Program Run

On the basis of the TCO (1) for each architecture type, we investigate a comparison metric that further takes the gained application performance or rather the parallel runtime into account. As a starting point, we assume that just one application runs all the time. Thus, we can compute the number of application executions for each system type n_{ex} and the costs per program run C_{ppr} for n nodes by:

$$C_{ppr}(n,\tau) = \frac{\text{TCO}(n,\tau)}{n_{ex}(\tau)\cdot n} \quad \text{with} \quad n_{ex}(\tau) = \frac{k\cdot\tau}{t_{par}}$$

Here, τ is the system's lifetime, t_{par} is the parallel runtime of the application and k denotes the system usage rate in percent. The latter is introduced to account for additional scheduling times, maintenance periods or unreliability of the system. The costs per program run C_{ppr} can serve as comparison metric for different architecture types.

One interesting aspect for comparison is the break-even point with respect to investment, i.e. what investment is needed so that one system type (and programming model) is beneficial over another (given a fixed lifetime). We can derive it by looking for zeros in the difference for costs per program run of two system types X and Y:

$$C_{pprX}(n_X,\tau) - C_{pprY}(n_Y,\tau) = 0 \tag{3}$$

Using (2), we can extract n as a function of I and substitute it into the equation above. Then, we get as break-even point I_{be}:

$$I_{be} = \frac{n_{exY}(C_{AX}+C_{CX}\tau)(C_{BY}+C_{DY}\tau) - n_{exX}(C_{AY}+C_{CY}\tau)(C_{BX}+C_{DX}\tau)}{n_{exY}(C_{AX}+C_{CX}\tau) - n_{exX}(C_{AY}+C_{CY}\tau)} \tag{4}$$

4 Case Study on Accelerators for Technical Computing

Today's hype for accelerators motivates us to evaluate their benefit (or "pain") compared to Intel servers from a TCO perspective.

4.1 Real-World Applications

For the integration of manpower efforts, performance and power consumption into the TCO calculation, we look at two different real-world simulation codes since results simply based on benchmarks like HPL can be misleading. We chose kernels from these software packages that are generally suitable for accelerators, while keeping in mind that many applications are not. The kernels are small enough so that we could implement different versions with acceptable effort.

Neuromagnetic Inverse Problem. The application *NINA* comes from the field of biomedicine, or more precisely, magnetoencephalography. The arising neuromagnetic inverse problem deals with the reconstruction of focal activity in the brain and can be

solved by means of a p-norm minimization. For this unconstrained nonlinear optimization problem, the software package of Bücker et al [14] employs first- and second-order derivatives with automatic differentiation. It is implemented primarily in MATLAB, whereas the objective function, its first- and second-order derivatives are written in C to enable parallel computing. We parallelized these three kernels that include the computations of matrix-vector products using a matrix of dimensions 128×512000. The kernels account for ~ 100 kernel code lines and 90 % of the whole application's runtime.

Simulation of Bevel Gear Cutting. The engineering application *KegelSpan* [15], written in Fortran and developed by the Laboratory for Machine Tools and Production Engineering (WZL) at RWTH Aachen University, is a 3D simulation software for the bevel gear cutting process and applied in the automotive industry. It aims at minimizing the number of expensive tool changes in the bevel gears manufacturing process by enabling a detailed tool load and wear analysis. For the optimization of manufacturing parameters, the intersection of tool and gear is computed repeatedly where each run iterates million to billion times. Although this part is the biggest hotspot in the KegelSpan package, it only accounts for approx. 25 % of the serial runtime. Since its industry costumers have had GPU hardware at their disposal anyway, we still started accelerating this kernel [9] (~ 150 lines in serial code) using the portable OpenCL. However, for our TCO calculations, we will assume that this module accounts for 90 % of the whole application runtime to illustrate our statements and not be restricted by Amdahl's law.

4.2 TCO Components

The following numbers and assumptions about the TCO components are based on our experience at the Center for Computing and Communication at RWTH Aachen University. Here, one possible scenario may belong to our integrative hosting activities that allow other RWTH members to integrate their HPC equipment into our cluster environment. We assume that an RWTH professor with a certain budget wants to buy compute nodes to accelerate his one research application (i.e. given an homogeneous application landscape). At the computing center, we want to give the professor an estimation which architecture is worth to be purchased by taking also programming effort into account.

System Types. We start by gathering results for single compute nodes and assume that results can be extrapolated to a cluster amount using (2). Furthermore, investigations that include network communication across nodes and the associated programming effort (e.g. MPI) are left for future work. The different system types (*ST*) are running Scientific Linux 6.3 and are given as follows:

ST1: As an X86 base, we take an Intel Sandy Bridge server which has widely been accepted as a cost-efficient architecture for compute services. It contains two-socket Intel Xeon E5-2650 CPUs running at 2.00 GHz with a total of 16 cores and 32 GB memory.

ST2: This accelerator architecture contains an NVIDIA Tesla C2050 (Fermi) GPU with ECC enabled. The host system consists of a 4-core Intel Westmere processor (Xeon

Table 1. Programming effort (in man-days) and kernel speedup w.r.t. the serial versions of NINA and KegelSpan. The power consumption (in watt) is taken from the whole system during the kernel execution.

		OMP-simp/ST1	OMP-vec/ST1	OCL/ST2	OpenACC/ST2	LEO/ST3
NINA	Effort	1	5	7	4	6
	Speedup	6.58	7.56	9.92	3.09	11.33
	Power	191.99	200.56	284.09	293.23	277.65
Kegel Span	Effort	0.5	3.5	5	1.5	4.5
	Speedup	15.47	22.55	46.65	47.02	44.16
	Power	166.17	155.03	249.64	227.36	230.87

E5620@2.40 GHz) with 12 GB of memory. Since we did not have access to an up-to-date NVIDIA Kepler GPU at time of writing, it will be subject of future investigations.

ST3: The second accelerator type comprises an Intel MIC coprocessor – an Intel Xeon Phi 5110P with 60 cores running at 1.053 GHz and 8 GB memory. For sound comparisons, we just assume that this machine has the same host configuration as the NVIDIA GPU system. We adapted accordingly hardware prices and power consumption, but took the real-measured kernel runtimes and speedups since the host processor does not significantly contribute to these results. In real life, the host system equals *ST1* and it contains two Intel Phis of the given type. As it is an early machine, not all settings (especially concerning energy) may be optimally configured yet.

Programming Effort. In previous works [9,10], we investigated the impact of development effort in accelerating code regions and expressed it by the number of lines of kernel code added or modified. Now, we try to quantify this effort in man-days (see Tab. 1) for inclusion into real-cost calculations and thereby put them into perspective on our way towards improving future purchase decisions. Therefore, we measured the development time of the first parallel version for both applications. It includes time for programming, debugging and analysis and therefore is also dependent on the available tools supporting the programming model. Efforts are based on a moderately-experienced programmer who already knows details on the hardware architecture and the programming paradigm. For the following implementations, we could directly apply some code lessons learned during the first implementation. Therefore, we added some approximated time to the real-measured one to be able to compute costs independently. The development days in Tab. 1 correspond to each best-effort version. We developed five parallel variants of each kernel. We started with a simple OpenMP version (*OMP-simp*) by applying OpenMP directives to the original serial code. The *OMP-vec* version includes code restructuring for (auto-)vectorization and further parallelization with OpenMP. Programming an accelerator puts much more restrictions on programmability. In the case of a GPU, the number of threads has to be very high to overcome latencies, and the brand-new Intel Xeon Phi only performs well if many threads execute highly-vectorized code. For GPUs, we tuned the application with OpenCL (*OCL*) by reducing

data transfers and using (if applicable) GPU on-chip memory, pinned memory, asynchronous execution and more. In the *OpenACC* version, we used directives to offload code regions to the GPU and tried to get the code as close as possible to the OpenCL version. For the Xeon Phi, we combined OpenMP with Intel's LEO while decreasing data transfers, applying data asynchronous movement and focusing on vectorization and vector alignment. Additional code changes must be applied to overcome small nearly-serial parts that were not an issue for tens of threads but became one for hundreds. Our GPU and Phi versions perform the compute-intensive kernels on the accelerator while the host does not contribute to performance gains. However, we will deal with estimations for full heterogeneous versions in Sect. 4.3. Going into depth in Tab. 1, we see that restructuring code for vectorization is time consuming. Especially for Xeon Phi, that is said to be easily programmable, the development effort rises as vectorization is really important to get performance. However, assuming that a highly-vectorized code version does already exist for the host, the denoted programming effort can be decreased. Furthermore, low-level GPU code development needs more manpower than parallelization on CPUs (*OMP-simp*, *OMP-vec*). Only directive-based GPU programming (with *OpenACC*) may decrease this effort.

Performance. The reported performance results include data transfers between host and device, kernel execution times and the overhead introduced by the need to adapt the data structure. The speedups shown refer to the kernel runtimes, but we use whole application speedups for the TCO calculation. For comparisons, the speedup values are given with respect to the serial version measured on *ST1*. However, in the following examinations, we compare appropriately performance to the *OMP-simp* version which uses all cores of the hardware. OpenMP (*ST1*) and Xeon Phi (*ST3*) results are gathered using the Intel compiler 13.0.1. We started 16 threads on *ST1* and 177 and 118 threads on Xeon Phi (*ST3*) for NINA and KegelSpan best-effort results, respectively. OpenCL (*ST2*) relies on CUDA toolkit 5.0 and OpenACC (*ST2*) on the PGI compiler 12.9 [1] that uses CUDA toolkit 4.1. Optimization flags are used as well (e.g. -O3 or fastmath). For both codes, the kernel speedups we can achieve on any accelerator are about 2-4x relative to the baseline *OMP-simp* implementation on *ST1*. While most performance results are as expected, we notice that NINA's OpenACC performance is rather disappointing. The reason probably lies in not yet fully-implemented OpenACC features in the PGI compiler (more details in [10]) which might be tackled in future compiler releases.

RWTH One-Time Costs. Diving into Tab. 2, the TCO components are listed based on the RWTH environment and with respect to the NINA application. Considering the hardware purchase, we use list prices for current generation servers and workstations, kindly provided by the company Bull in January 2013. For evaluations of TCO calculations with numbers from other sources, we provide the editable TCO spreadsheet [13]. In order to estimate the infrastructure costs for housing the compute devices, we express the actual one-time building costs as annual costs of 200,000€ [6] over 4 years of system lifetime. Breaking down this total annual cost per node, we divide it by a

[1] Recent PGI compiler versions (13.1 - 13.3) were not used due to a compiler problem that evokes a performance loss in our case.

Table 2. One-time and annual costs in € for the NINA application

		one-time costs C_{ot}					annual costs C_{pa}					
		per node			per node type		per node				per node type		
		HW purchase	Building/ infrastruct.	OS/env installation	OS/env. installation	Prog. effort	HW maintenance	Building/ infrastruct. per watt	OS/env maintenance	Energy	OS/env. maintenance	Compiler/ software	Application maintenance
ST1	OMP-simp	7,137	0	0	0	286	300	33	78	317	0	0	0
ST1	OMP-vec	7,137	0	0	0	1,429	300	33	78	322	0	0	0
ST2	OCL	7,713	0	0	0	2,000	324	49	78	421	0	0	0
ST2	OpenACC	7,713	0	0	0	1,143	324	49	78	506	0	0	0
ST3	LEO	9,644	0	0	0	1,714	405	49	78	465	0	0	0
		C_A			C_B		C_C				C_D		

total of 1.6 MW of energy consumption, which today is the limiting factor for housing machinery in this building [6], and multiply it by the maximum power consumption of each compute node. The initial administration effort for integrating and installing novel (accelerator) systems is high: The staff must first get to know the systems, establish operating concepts, integrate it into the existing batch scheduler, install new drivers and software and may make sure that further maintenance updates can be easily rolled out to all nodes of a system type simultaneously. If we set up a TCO calculation for a single system, such an effort would never pay off. On the other hand, it seems that university computing centers cannot avoid investigating the usability of recent accelerator architectures due to increasing power bills. Since our administration staff has already gained experiences in the past, we assume no extra one-time charges for new installations in our calculations. While transferring the programming effort that we explained previously into manpower costs, we assume the cost of one day of a full time employee (FTE) at 285.71 € in accordance to funding guidelines of the German Science Foundation [16] and the the European Commission's CORDIS [17].

RWTH Annual Costs. The annual costs per node include hardware maintenance that is provided by the vendor (Bull) and accounts for 5 % of the purchasing costs. At RWTH Aachen University, 4 administrative FTEs are running the whole compute cluster [6] and 75 % of the manpower accounts for annual maintenance. We quantify the administration effort per node by dividing the 180,000 € manpower costs [16] by the total number of nodes in our cluster (roughly 2,300) and get approximately 78 € per any kind of compute node. There is no significant additional effort per node type since a generic approach to roll out software was established during the first installation. Furthermore, we do not have any additional software or compiler costs as we buy the licenses anyway for the whole cluster (e.g. Intel or PGI compiler) or the software is free of charge (e.g. CUDA toolkit). Energy costs are dependent on the hardware, the running application, the power usage effectiveness (PUE) of the computing center and

Table 3. Costs per program run C_{ppr} given an investment of 250,000 € and a lifetime of 4 years & break-even points I_{be} in investment (see (4)) w.r.t. OMP-simp for NINA and KegelSpan.

		OMP-simp	OMP-vec	OCL	OpenACC	LEO
NINA	#nodes	24.84	24.69	22.14	21.57	18.21
	t_{par}[s]	106.87	98.92	86.10	176.72	81.01
	n_{ex}	944,265	1,020,119	1,172,090	571,034	1,245,711
	C_{ppr}[€]	0.01066	0.00993	0.00963	0.02030	0.01102
	I_{be}[€]		15,989	17,058	-	-
KegelSpan	#nodes	25.46	25.53	22.96	23.12	18.61
	t_{par}[s]	158.42	140.13	119.48	119.33	120.57
	n_{ex}	637,027	720,150	844,632	845,708	836,993
	C_{ppr}[€]	0.01542	0.01360	0.01289	0.01279	0.01605
	I_{be}[€]		7,231	7,787	1,809	-

the regional electricity cost. Here, we pay roughly 0.15 €/kWh and have an estimated PUE of 1.5 [6]. The power usage measurements were done using a Raritan Dominion PX power distribution unit on *ST1* and *ST2*. Since *ST3* is an artificial construction, we recorded the power consumption of the Xeon Phi during the kernel run and added the appropriate host consumption from *ST2*. In general, we further differ between the power consumption of the application kernel (compare Tab. 1) and the rest of the application that mostly runs sequentially. Finally, we set the costs for application maintenance to 0 € since the investigated kernels are quite small. However, one should keep in mind that this effort may increase especially for bigger codes when the chosen programming paradigm is verbose or when a lot of code restructuring is needed.

Doing the math, we compute the number of nodes that can be bought with a sample investment of 250,000 €, the number of executions n_{ex} (assuming the application runs 24/7 with a cluster usage rate of 80 %) and finally the costs per program run (Tab. 3).

4.3 Results

Based on our previous calculations, we interpret the results and perform a what-if analysis with focus on programming effort.

For both codes, NINA and KegelSpan, a simple OpenMP parallelization did not cost a lot of effort and thus increased the TCO only slightly while speeding up the compute-intensive kernels considerably. In order to investigate the cost/performance ratio of the accelerators, we take this simple OpenMP version as the baseline. Figures 1 and 2 present graphs over the varied amount of budget which show the difference in cost per program run relative to the *OMP-simp* version in percentages. A negative percentage means that the given system type is $x\,\%$ cheaper than the *OMP-simp* version and vice versa. We added results for estimated hybrid OpenCL (*OCL-hyb*) and Xeon Phi (*LEO-hyb*) solutions. The numbers on the right hand side of the figures express the limiting values for infinitely-big investments.

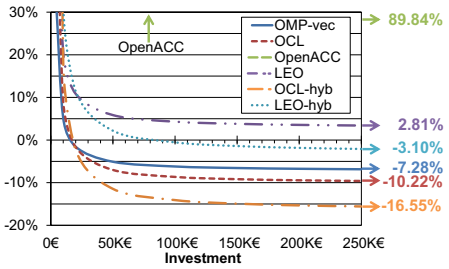


Fig. 1. NINA's cost per program run in percent relative to OMP-simp

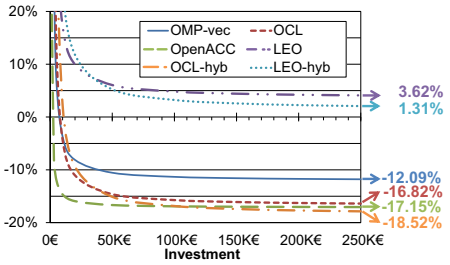


Fig. 2. KegelSpan's cost per program run in percent relative to OMP-simp

NINA. For the NINA software package (Fig. 1), the *OMP-vec* version is already worthwhile given an investment of 15,989€ or more (see Tab. 3). If more budget is available (at least 17,000€), GPU accelerators and OpenCL become profitable. For the estimated OpenCL-hybrid solution, the initial programming effort would also pay off. On the other hand, the low programming effort of the *OpenACC* version does not make up for its slowness and we would pay roughly 90 % more than the *OMP-simp* version costs per program run with an infinite investment. Looking at Intel's Xeon Phi accelerator, it is surprising that it cannot beat the simple OpenMP version when taking the total ownership costs into account (the limit amounts to $\sim 3\,\%$). However, compared to the OpenCL version, its programming effort is lower and the performance a bit better. The power consumptions plays a role, but the higher hardware purchase costs have the main impact on the TCO. For the new NVIDIA Kepler GPU, we would probably see a similar picture. The estimated *LEO-hyb* version would become at least advantageous over the *OMP-simp* version given a budget of 83,000€ or more.

KegelSpan. For KegelSpan, Fig. 2 illustrates a profitable *OMP-vec* version like for NINA. In contrast, the *OpenACC* accelerator version is already beneficial over the OpenMP simple host version given an investment of 1,800€ (see Tab. 3). This is due to low programming effort and the good performance. The other GPU accelerator versions are profitable as well while our estimated *OCL-hyb* is only slightly more beneficial than the *OpenACC* and the *OCL* version. In contrast, the Xeon Phi accelerator does not pay off. Even our hybrid estimation would run into a positive percentage of 1.3 for infinitely-big investment.

What-If Analysis. Taking the data discussed as foundation, we perform a "what-if" analysis by varying certain components of the TCO calculation.

First, we look at the impact of Amdahl's law on the KegelSpan application. We plot the break-even points in investment I_{be} (compare (4)) as a function of the kernel portion in Fig. 3. It illustrates that bigger kernel portions – and thereby higher performance – results in lower investment needed to compensate the higher initial expenses over the *OMP-simp* version. The *OMP-vec* version pays off for small kernel portions from a moderate investment on. On the contrary, the GPU variants need a kernel portion around 75 % to be beneficial at all. Then, the *OpenACC* version can require a lower budget than

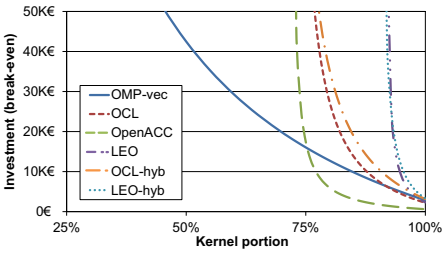


Fig. 3. Break-even points in investment over OMP-simp as function of KegelSpan's kernel portion

the *OMP-vec* version to be advantageous, whereas the OpenCL versions are roughly at the same level for big kernel portions. The Phi coprocessor variants even need 92 % kernel portions to be beneficial, i.e. the performance pays off the hardware costs. To this end, we see that the comparison of system types and programming models is also effected by Amdahl's law and that our results cannot be taken as absolute statements.

Both of our investigated kernels were moderately small. The question with bigger codes is how the higher programming effort influences the results. Given the TCO and break-even formulas, we can derive that the increase of kernel code means a proportional increase of the needed minimal investment to make a program run beneficial.

The introduced hybrid estimations assumed a certain programming effort given the optimal performance based on vectorized host and accelerator speedups. Now, we turn round the perspective and investigate whether and when a hybrid implementation would be worthwhile. Taking (3), we solve it for the development effort and thereby get the desired break-even point in man-days. We anticipate that a hybrid OpenCL implementation may need up to 146 and 162 man-days, respectively for NINA and KegelSpan, to still be beneficial over the *OMP-simp* version. In contrast, development may only take up to 28 man-days for a heterogeneous NINA Phi solution, but a hybrid Phi implementation for KegelSpan would never pay off. The latter may be canceled out by a more efficient host system.

5 Conclusion

In the context of one case study at RWTH Aachen University, we examined the benefit of accelerators in technical computing. Based on total ownership costs (TCO), we compared costs per program run for specific applications on an Intel server, NVIDIA Fermi GPU and Intel Xeon Phi coprocessor while putting human effort into the equation.

Taking a simple OpenMP version as baseline, we find that most GPU Fermi solutions pay off additional manpower efforts. Furthermore, OpenACC GPU results illustrated that the ratio of performance per development time and power consumption is interesting: Low programming effort, but low performance is expensive (see NINA application), whereas a combination with good performance rocks (compare KegelSpan results). On the other hand, results gathered on Intel's Xeon Phi were surprisingly disappointing. Here, the system acquisition costs were mainly responsible for the non cost-efficient result. Additionally, it took quite some effort to create solutions with good

performance due to vectorization tuning, despite that the Xeon Phi is said to be easily programmable. However, if highly-vectorized host implementations are available as baseline, this picture will improve (a déjà vu for elder vector computer users). Generally, host systems are usually less expensive, require less power and need less manpower so that the accelerators' performance has to compensate these three aspects. A perspective based just on performance per watt is limited.

Furthermore, our TCO model (available at [13]) allows projecting the feasibility of considered solutions such as hybrid implementations. For instance, our results show that the according effort does not always pay off (depending on hardware and performance).

In future, we will include additional programming paradigms like the upcoming OpenMP 4.0 features and new architectures like NVIDIA's Kepler GPU. We will further take network communication (MPI) into account to balance cost efficiency versus real-time constraints. Since we have assumed a homogeneous application landscape so far, future examinations will look at the impact of mixed job executions. We also want to turn our TCO analysis of experimentally-gathered data into an analytical model with predictive powers. Performance models that can predict the parallel runtime from the single-core code performance do already exist. Given the recent interest in reducing machine power envelopes, the existing power consumption models will hopefully get enhanced in the near future. Additionally, a reliable model to estimate the manpower efforts is also essential in order to improve the quality of the TCO interpretations. We started measuring the amount of effort that our students put into program development during their practical trainings in order to obtain baseline data as a first step to modeling programming productivity.

References

1. The Green 500: The Green500 List - November 2012: Heterogeneous Systems Re-Claim Green500 List Dominance (2012), `http://www.green500.org/lists/green201211`
2. Carlyle, A., Harrell, S., Smith, P.: Cost-Effective HPC: The Community or the Cloud? In: CloudCom 2010, pp. 169–176 (2010)
3. Turner, W.P., Seader, J.H.: Dollars per kw plus dollars per square foot are a better data center cost model than dollars per square foot alone. Technical report, Uptime Institute (2006)
4. Patterson, M., Costello, D., Grimm, P., Loeffler, M.: Data center TCO; a comparison of high-density and low-density spaces. Technical report, Intel Corporation (2007)
5. Koomey, J.: A Simple Model for Determining True Total Cost of Ownership for Data Centers. Technical report, Uptime Institute (2008)
6. Bischof, C., Mey, D.a., Iwainsky, C.: Brainware for green HPC. Computer Science - Research and Development 27, 227–233 (2012)
7. Kuck, D.J.: Productivity in high performance computing. Int. J. High Perform. Comput. Appl. 18(4), 489–504 (2004)
8. Galorath, D.D.: Software Total Ownership Costs: Development Is Only Job One. Software Tech. News 11(3) (2008)
9. Wienke, S., Plotnikov, D., Mey, D.a., Bischof, C., Hardjosuwito, A., Gorgels, C., Brecher, C.: Simulation of Bevel Gear Cutting with GPGPUs – Performance and Productivity. Computer Science - Research and Development 26, 165–174 (2011)

10. Wienke, S., Springer, P., Terboven, C., an Mey, D.: OpenACC — First Experiences with Real-World Applications. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds.) Euro-Par 2012. LNCS, vol. 7484, pp. 859–870. Springer, Heidelberg (2012)
11. CAPS enterprise: Case study – Code Migration to CPU-GPU Hybrid: An Economic Approach (2010)
12. Wang, L., Khan, S.: Review of performance metrics for green data centers: a taxonomy study. The Journal of Supercomputing, 1–18 (2011)
13. Wienke, S., an Mey, D., Müller, M.S.: Accelerators for Technical Computing: Is it Worth the Pain? TCO Spreadsheet (2013), `https://sharepoint.campus.rwth-aachen.de/units/rz/HPC/public/Shared%20Documents/WienkeEtAl_Accelerators-TCO-Perspective.xlsx`
14. Bücker, H.M., Beucker, R., Rupp, A.: Parallel Minimum p-Norm Solution of the Neuromagnetic Inverse Problem for Realistic Signals Using Exact Hessian-Vector Products. SIAM Journal on Scientific Computing 30(6), 2905–2921 (2008)
15. Brecher, C., Gorgels, C., Hardjosuwito, A.: Simulation based Tool Wear Analysis in Bevel Gear Cutting. In: International Conference on Gears, Düsseldorf. VDI-Berichte, vol. 2108.2, pp. 1381–1384. VDI Verlag (2010)
16. German Science Foundation (DFG): Personalmittelsätze der DFG für das Jahr 2013 (2013)
17. Community Research and Development Information Service: Audit Certificate Guidance Notes 6th Framework Programme (2005)

Evaluating Lossy Compression on Climate Data*,**

Nathanael Hübbe[1], Al Wegener[2], Julian Martin Kunkel[1], Yi Ling[2], and Thomas Ludwig[3]

[1] University of Hamburg
nathanael.huebbe@informatik.uni-hamburg.de
[2] Samplify
AWegener@samplify.com
[3] German Climate Computing Centre

Abstract. While the amount of data used by today's high-performance computing (HPC) codes is huge, HPC users have not broadly adopted data compression techniques, apparently because of a fear that compression will either unacceptably degrade data quality or that compression will be too slow to be worth the effort. In this paper, we examine the effects of three lossy compression methods (GRIB2 encoding, GRIB2 using JPEG 2000 and LZMA, and the commercial Samplify APAX algorithm) on decompressed data quality, compression ratio, and processing time. A careful evaluation of selected lossy and lossless compression methods is conducted, assessing their influence on data quality, storage requirements and performance. The differences between input and decoded datasets are described and compared for the GRIB2 and APAX compression methods. Performance is measured using the compressed file sizes and the time spent on compression and decompression. Test data consists both of 9 synthetic data exposing compression behavior and 123 climate variables output from a climate model. The benefits of lossy compression for HPC systems are described and are related to our findings on data quality.

Keywords: Data Compression, GRIB2, JPEG 2000, APAX.

1 Introduction

Climate science is a notorius producer of big data. More than 100 climate variables are typically used in modern climate models, a number that cannot meaningfully be reduced, and the simulated time spans are often decades. Data presented in scientific publications must be archived for at least ten years. Consequently, large climate computing facilities like the German Climate Computing Center make significant investments in storage systems. Despite the amount of

* This paper is partly funded by the DFG (GZ: LU 1353/5-1).
** We also thank Luis Kornblueh for providing us with the climate dataset, without which this paper would not have been possible.

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 343–356, 2013.

climate data to be stored and transfered, climate scientists have been reluctant to use data compression to reduce dataset volumes. It seems the scientists have vague fears that lossless compression would be too slow to be worthwhile and that lossy compression would unacceptably reduce the quality of the data.

In our previous paper [4], we showed that the slowest lossless compression algorithms actually achieve the best compression. In this paper we measure the effects of two lossy compression algorithms on climate variable quality while also considering lossy compression's benefits in reducing HPC system bottlenecks.

This paper is structured as follows: First, related work and the lossy compression algorithms used are described. We then discuss signal quality requirements of climate scientists and describe the different synthetic and climate variables that are used to evaluate lossy compression. We summarize the metrics to quantify differences in precision. Then our test setup is described. After summarizing our findings, we use two distinct approaches to analyze the results. First, the processing speeds of the competing algorithms is characterized. Second, approaches to obtaining acceptable climate variable quality are described. Finally, future research directions are considered.

2 Related Work

Lossless compression can be profitably used if the costs for compression and decompression are less than the costs of bandwidth and storage. For example, lossless-compressed tarballs are regularly used for source code exchange, and the SLDC [3] algorithm increases both tape drive bandwidth and capacity. However, lossless compression algorithms developed for ASCII text do not compress binary datasets, such as most HPC datasets, very well. In contrast, a small number of targeted algorithms have been developed to compress floating-point HPC data. Current research into compression of scientific data generally takes one of two approaches: Either the performance of available algorithms is evaluated on specific scientific datasets as Woodring et al. [11] have done when they applied JPEG 2000 compression to climate data, or new lossy algorithms are developed that have specific features and/or perform well for specific kinds of data.

An example for such new lossy algorithms is *isabela*, invented by Lakshminarasimhan et al. [6], [7]. Another example is the recent *sengcom* algorithm [9], which has strong similarities to the GRIB2/JPEG 2000 compression described below. Some algorithms take the multidimensionality of scientific datasets into account, such as the work by Lindstrom and Isenburg [8]. Iverson et al. [5] are exploiting data locality on unstructured grids, especially for geo-sciences. Our own last endeavor at lossless scientific data compression [4] handled diverse multidimensional datasets. MAFISC uses the standard lossless compression algorithm LZMA as a compression back-end after the MAFISC front-end transforms the data in a reversible way.

2.1 Lossy Compression in GRIB2

GRIB2 [2] is a format defined by the World Meteorological Organization that is based on self-describing messages using standardized values to identify basic

properties of the data. Examples for such properties are grid types, intended meaning of the data, and data encoding formats.

As a file format standard, the GRIB2 format itself does not specify how the data is encoded, only how the encoded data should be interpreted. This leaves a number of decisions to the authors of programs producing GRIB2 data.

GRIB data formats are based on fixed point (integer) representation that includes a conversion from the floating point representation that causes a loss of precision. The quantization parameters are selected by the encoding software according to a user choice and are kept in the header of the encoded data. In most cases, the user specifies how many bits of precision should be retained. The encoding software then scans the input data for its value range and adjusts the quantization accordingly. In this way, the user controls both signal quality and file size, resulting in lossy compression.

As provided in current implementations, GRIB2 quantization is time-adaptive since it is performed for each timeslice and elevation level seperately. The resulting quantization error places a tight upper bound on the maximum error. Finally, the quantized GRIB2 integer result can be further compressed by JPEG 2000 [1] in its lossless mode.

2.2 APAX

Figure 1 presents a block diagram of the Samplify APplication AXceleration (APAX) Encoder. The APAX algorithm encodes sequential blocks of input data elements with user-selected block size between 64 and 16,384. The signal monitor tracks the input dataset's center frequency. The attenuator multiplies each input sample by a floating-point value that, in fixed-rate mode, varies from block to block under the control of an adaptive control loop that converges to the user's target compression ratio. The redundancy remover generates derivatives of the attenuated data series and determines which of the derivatives encodes using the fewest bits. The bit packer encodes groups of 4 successive samples using a *joint exponent encoder* (JEE). JEE exploits the fact that block floating-point exponents are highly correlated. Additional APAX details are described in [10].

The APAX encoder uses a software tool called the APAX profiler that provides information about the compressibility of input datasets, and recommends

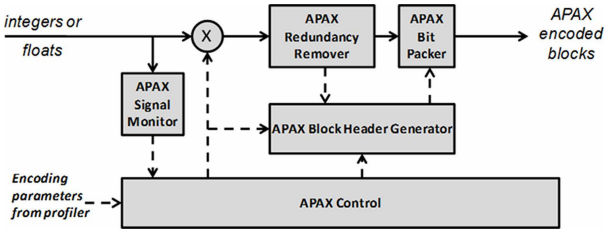


Fig. 1. APAX block diagram

a compression setting that delivers high-quality decompressed samples. Figure 2 illustrates the APAX profiler output on climate variable *ustrl*. The upper left window displays the rate-distortion graph for the input signal being profiled. The profiler suggests a Recommended Operating Point (ROP) where the correlation between the original data x and the APAX-decoded data y is 0.99999 ("five nines"). The upper right window displays metrics comparing input $x(i)$, decoded output $y(i)$, and residual or difference $d(i) = x(i) - y(i)$. The lower left window compares the input spectrum to the residual spectrum and quantifies the spectral margin at the ROP. The lower right window histograms the residuals and calculates the $2 \cdot \sigma$ (95.5%) signal-to-residual margin.

2.3 Comparing GRIB2 and APAX Control

We notice that GRIB2 and APAX are controlled in fundamentally different ways. GRIB2 users must choose the quantization level N, and whether or when to use JPEG2000. GRIB2 parameter choices directly affect the quality of decoded climate variables and the speed with which they are encoded and decoded. The APAX Profiler visually displays the rate-distortion curve on the climate variable being profiled and recommends a compression ratio, helping users in their decision process. Once the user has chosen the compression ratio, that setting is again accessed as APAX encodes that climate variable. The Profiler's default ROP ensures consistent signal quality while allowing sophisticated users to modify the ROP and to visualize and measure the new ROP's effect on signal quality.

3 Quantifying the Uncertainty

3.1 Scientific Requirements

To determine appropriate climate data signal quality requirements, one must understand that climate data will be used in two ways:

- The data is analyzed or visualized for evaluating long-term effects, such as variations in average, variance, frequency, or locality of events.
- Data may also be used to drive another model (or to keep checkpoints).

In the first case, good quality data will not introduce any significant statistical variation. No new effects should be created that were not present in the original data, and no previously visible effects should vanish. To be safe, the maximum error (worst-case scenario) should be monitored in addition to its standard deviation to check whether the error is still guaranteed to be below the required threshold. The second case cannot be evaluated as easily, because climate systems are inherently chaotic. A small change in the input data may either vanish completely or can lead to a completely different state a year later. It is impossible to consistently predict whether an error in the input data caused by lossy compression would cause different simulation effects than a random error would not have caused.

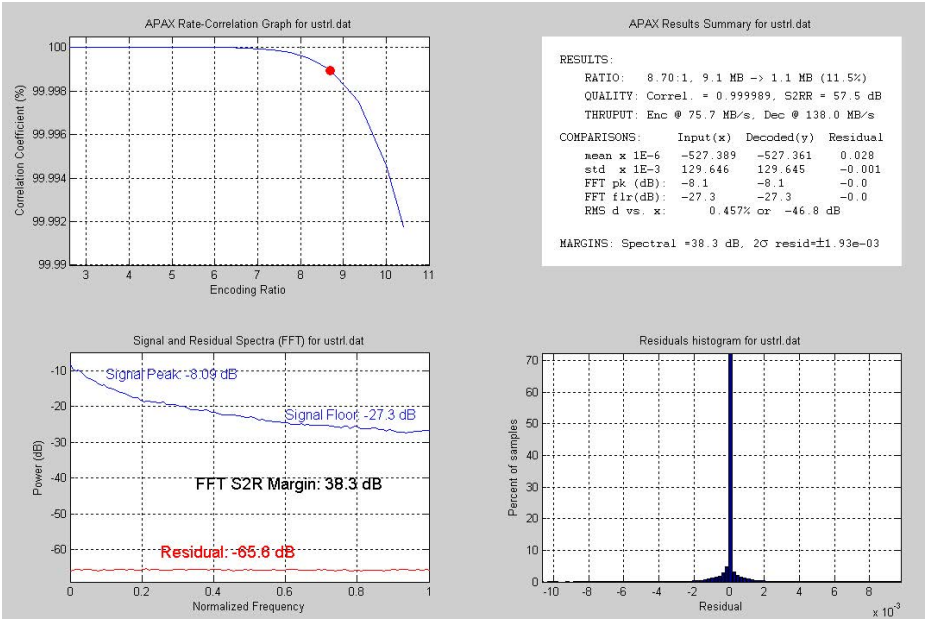


Fig. 2. APAX profiler window

In both cases, errors should not be locally correlated because such correlated errors can give rise to false positives (much in the way that overcompressed JPEG images exhibit block edges that were not present in the original image). When decompressed data drives a climate simulation, locally correlated errors are much more likely to drive the simulation into a state that would not have been reached with uncorrelated errors. In the case of checkpoints, lossy compression is usually not an option, since their goal is to allow a precise restart after a crash. Thus our signal quality metrics will measure both average and worst-case differences between the input and the decompressed datasets, as the compression ratio is varied over a range from 24/32 (75%) to 8/32 (25%).

3.2 Approach

Description of Test Cases. We have tested the the GRIB2 and APAX compressors using 9 synthetic datasets and with 123 climate output variables generated from the ECHAM climate model from the Max-Planck-Institute for Meteorology. The synthetic files are as follows:

bandlim_lowpass 1D, lowpass filtered random data.
bandlim_narrow 1D, bandpass filtered random data.
random 3D random data, flat distribution in the intervall [-1,1].
random_offset 3D random data with an offset of 1 added.

random_correlated 3D random data with 90% correlation to the mean of the three previously generated neighbours.

fractal 3D hierarchically generated data with fractal dimensionality. 2D slices resemble a mountaneous height field.

integrated 3D random data, integrated twice in all directions. Very smooth.

sines_orthogonal 3D cube with one sine per axis in superposition.

sines_random 3D cube with 100 sines of random direction and frequency in superposition.

All the 3D test cases were generated with $257 \times 257 \times 257$ grid points, producing NetCDF files with 68 MB of data. The climate model output consisted of 123 different variables, covering one month at a sample frequency of six hours (124 timesteps). The longitude/latitude grid covers the entire earth with 192×96 grid cells and 47 height levels. Some variables are expressed as 2080 spectral coefficients. The entire climate dataset contained 4.4 GB of data.

Description of Error Metrics. Let us consider the characteristics of the residual signals $r(i)$ generated by GRIB2 and APAX encoding. Since lossy compression always generates non-zero residuals $r(i)$, we should first describe the preferred characteristics of residuals. First, the magnitude of $r(i)$ should be as small as possible, in both a relative sense (minimize $r(i)/x(i)$) and also in an absolute, peak error sense (minimize $r(i)$). Second, residuals should be zero-mean, i.e. $E[r] = 0.0$. Third, residuals should be spectrally white, i.e. the residual's power spectral density $psd(r)$ should be flat from DC to Nyquist (half the sampling rate). Fourth, residuals should be uncorrelated with the signal from which they are generated. Point three and four are generally not met by lossy compression algorithms, including APAX and GRIB2. Both GRIB2 and APAX normalize floating-point input values and then quantize them to integers, so both algorithms generate residuals with similar characteristics. However, the residual characteristics are not identical, since APAX can change the quantization from block to block.

In the order in which climate dataset values are stored in memory, the standard deviation of residuals, $std(r)$, provides a direct metric of signal quality. When the residuals are spectrally white, $std(r)$ is proportional to $psd(r)$. We calculated the signal-to-residual ratio (SRR), in bits, as given in Equation 1. Since compression users are concerned both with average and worst-case signal quality, we also measured the largest residual magnitude $max(abs(r))$, and compare it to the range of input values, $max(x)$–$min(x)$, calculating our peak error metric, in bits, as given in Equation 2.

$$SRR = log_2\left(\frac{std(x)}{std(r)}\right) \tag{1}$$

$$PrecisionBits = log_2\left(\frac{max(x) - min(x)}{2 \cdot max(abs(r))}\right) \tag{2}$$

In cases where a particular climate variable contains so little numerical variation that the decoded signal is identical to the original signal, it is possible to have an "infinite" SRR. These cases are especially likely at lower compression ratios such as $N = 24$ (75%). In such cases, the residual samples are all zero and both SRR and PrecisionBits are infinite. To avoid calculations using such infinite values, we limit the maximum SRR and PrecisionBits to 50 bits.

4 Evaluation

4.1 Speed

Test Setup. Compression and decompression performance tests were run on GRIB2 and APAX using a 48-core Magny-Cours node with 128 GiB of memory and 1.9 GHz clock frequency. Since the objective was to measure the performance of the algorithms and not the performance of the disks, all input files were first copied to a RAM-disk. The compression/decompression was then performed with both the input and the output file on the RAM-disk and measured using the `time` utility. The measured GRIB2 times are the sum of the system and user times reported by `time`. Timing measurements were automated using a shell script for all measurements. APAX performance was measured using in-process timers measuring only the compression from memory buffer to memory buffer, while the GRIB2 measurements encompas the entire process, including startup times, filesystem calls and library overhead. Thus, the timings are not comparable between GRIB2 and APAX.

Results. Figure 3 compares the performance of the different algorithms. Each point quantifies the performance of one algorithm on one file. Each algorithm is represented using a colored, shaped icon. The x-axis represents compression throughput (sec/GB), while the y-axis represents compression factor.

It is interesting to see that the external LZMA compression of GRIB2 files tends to be faster than the builtin JPEG 2000 compression, which performs as slow as MAFISC in many cases. MAFISC exhibits the slowest processing times on some files. Figure 3 illustrates a roughly linear correlation between the compressability of a file and the time it takes to be encoded by GRIB2. Generally, we see a strong correlation between compressability and speed, the only exception to this is APAX, its speed solely depends on data characteristics and, to a minor degree, target compression ratio. Unfortunately, this is not visible in the graphics. Only the LZMA utility may take considerably more time on some files than for other files with similar compressability, with the LZMA based MAFISC compression this shows even more clearly. APAX throughput is about 152 MB/sec (6.58 sec/GB) for compression and 209 MB/sec (4.79 sec/GB) for decompression, measured from memory buffer to memory buffer, all other measurements include filesystem access to a Ramdisk as well. These averages are calculated from total uncompressed size and total processing time.

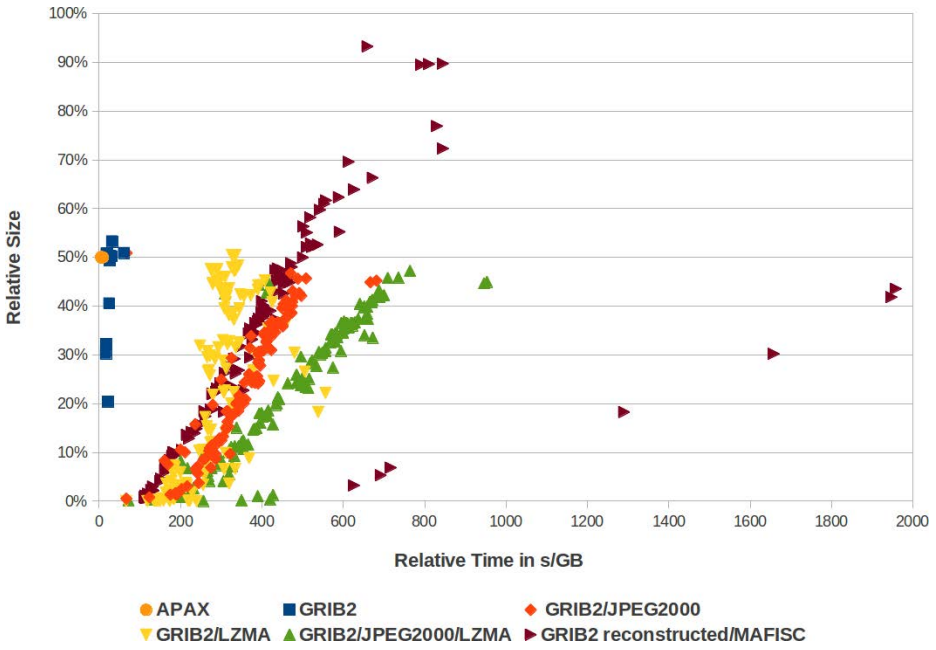


Fig. 3. Comparison of algorithm throughput (sec/GB) vs. achieved compression. Note that the measurement conditions for APAX were not the same as for the other methods.

It is clear that APAX and GRIB2 are the fastest algorithms in the field. GRIB2 and APAX have comparable compression and decompression throughput, and even though GRIB2 performs fewer calculations, APAX appears to be the faster of the two.

4.2 Compression

Apart from the compression time, the plot in Figure 3 also reveals the compression ratio of the files. It is clear that a lot of the redundancy of the data remains in the GRIB2 encoded files. This is demonstrated by strong additional lossless compression that can be achieved using JPEG 2000 or LZMA. As shown in Figure 3, the additional lossless compression benefits from JPEG2000 and LZMA are achieved at the expense of slower processing speed. Both JPEG 2000 and LZMA add between 100 and 500 s/GB to the GRIB2 encoding time for $N = 16$, in most cases, however, LZMA takes less time than JPEG 2000 compression.

Figure 4 compares the compression ratio of GRIB2/JPEG 2000, GRIB2/LZMA and GRIB2/JPEG 2000/LZMA The files were sorted according to their GRIB2/LZMA compression, the GRIB2 quality was set to 22 bits, but the results for other sizes are comparable. Figure 4 is interesting for a number of reasons:

- Neither JPEG 2000 nor LZMA can be said to be better than the other, each significantly outperforms the other on a large number of files.
- Whether JPEG 2000 or LZMA provides more compression is strongly correlated with the LZMA compressability.
- Even though the JPEG 2000 output is next to incompressible for most files, in some cases, LZMA applied on GRIB2/JPEG 2000 output still has a very strong effect. These are data sets, where a number of timeslices/height levels are identical or differ only by an offset and a factor. In these cases, the JPEG 2000 output is identical because its input is identical. LZMA finds these repeated regions in the file and achieves additional compression.

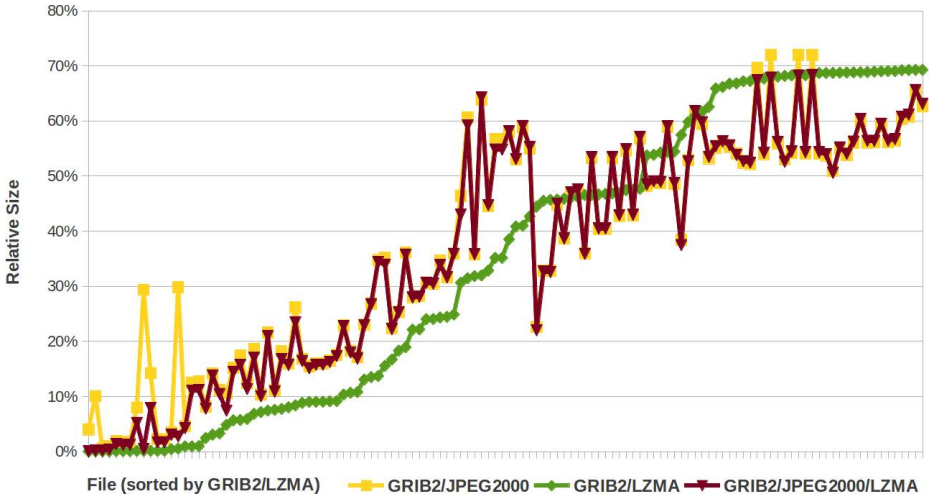
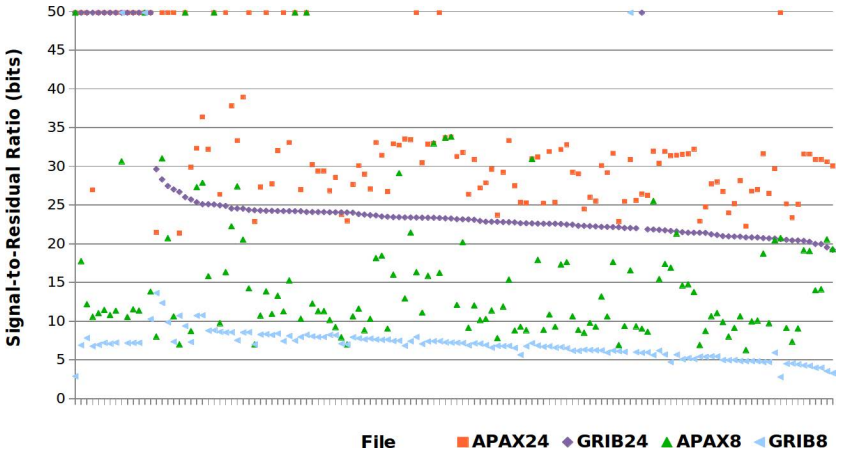


Fig. 4. Compression factor of lossless algorithms on GRIB2 data for 22 bits

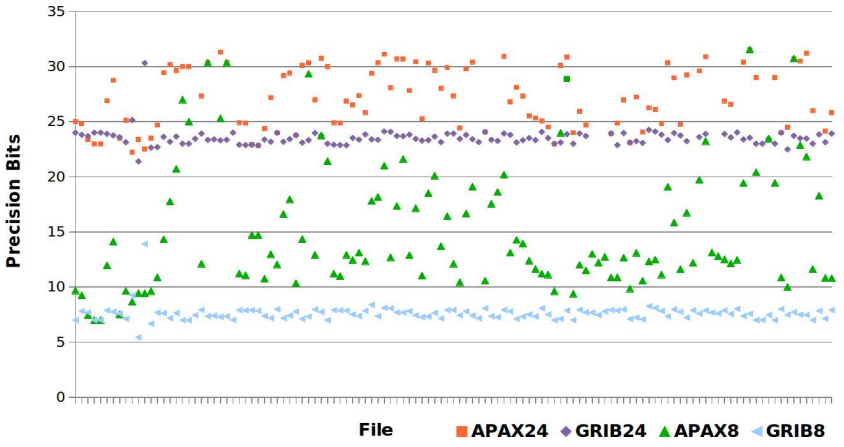
4.3 Errors

In this analysis we compare the loss in precision of compressed values between APAX and GRIB for 8 bits (4 : 1) and 24 bits (1.33 : 1) precision. Since LZMA, MAFISC and JPEG2000 are lossless, these algorithms do not alter the quality of the compressed data.

Figure 5a illustrates the SRR average signal quality metric for each of the 132 datasets. Larger SRR values indicate better decompressed signal quality. Visually, Figure 5a demonstrates that APAX SRR values generally exceed GRIB2 SRR values, with a few exceptions. Figure 5b illustrates the PrecisionBits peak error metric, measured under the same conditions as Figure 5a's SRR metric. Larger PrecisionBits values indicate better decompressed signal quality. Visually, Figure 5b demonstrates that APAX PrecisionBits values generally exceed GRIB2 values, with a few exceptions.



(a) Signal-to-Residual Ratios



(b) PrecisionBits

Fig. 5. Comparison of GRIB and APAX at N=8 and N=24

Table 1 compares the resulting signal quality on 10 climate files, when comparing GRIB2 quantization at 22 bits followed by JPEG 2000 encoding of the GRIB2 output, to APAX at the equivalent compression ratio. For instance, while GRIB2 compression of the climate variable alcov achieves a compression ratio of 68.75% (22/32), JPEG 2000 further compresses the 22-bit GRIB2 values to a compression ratio of 64%. APAX was directed to achieve the same compression ratio (64%) as GRIB2/JPEG 2000, and the quality of both results was compared. Unfortunately APAX could not be instructed to operate at a compression ratio of less than 2.9% on the alsom file, so the following considerations do not take this file into account.

Table 1. GRIB2/JPEG2000 and APAX Signal Quality Metrics at the same compression ratios. N=22 for GRIB2/JPEG2000 files.

		SRR		PrecisionBits	
file	rel. size	APAX	GRIB2/JPEG 2000	APAX	GRIB2/JPEG 2000
trads	65.4%	20.7	20.8	20.3	21.1
aclcov	64%	23.2	22.1	21.0	21.0
trafl	56.5%	20.8	21.4	20.4	22.0
trflwac	53.2%	21.5	20.9	20.2	21.8
soflwac	35.8%	21.0	22.0	18.7	22.0
wsmx	29.3%	23.7	21.6	21.8	21.8
ahfllac	28%	21.7	19.0	21.1	21.7
vdisgw	22.9%	24.6	19.6	22.9	21.9
srad0d	22.6%	13.5	21.6	12.2	21.3
alsom	2.9%/1.9%	lossless	22.8	lossless	21.5

Looking at the SRR metric, each of the two compression methods achieves lower errors for some files than the other. This changes, however, if the maximal error is taken into account, which is used for the PrecisionBits metric. With this metric, APAX only has an advantage over GRIB2/JPEG 2000 on one of nine files, for the rest it achieves at most the same precision level. This one file, however, is a file on which GRIB2/LZMA results in output only half as big as the GRIB2/JPEG 2000 result.

In addition to comparing GRIB2 and APAX signal quality at the same compression ratio, we also examined compression ratio at comparable SRR quality levels. Using APAX's fixed quality mode, APAX compression ratios vary from dataset to dataset, because APAX encodes the most compressible derivative from among three alternatives, for each input block. With a fixed quality of SRR=14 bits, APAX averaged 1.6x more compression than GRIB2 with N=16 (wich yields SRR values around 15 bits). For certain files, GRIB2 compression is improved by as much as 60:1 by JPEG 2000 post-processing, and by as much as 3000:1 by LZMA post-processing.

5 Analysis of the Differences

APAX appears to be the fastest algorithm. GRIB2 can be fast as well, but how fast it actually is depends heavily on its precision parameter: N = 8, 16 or 24 bits yields much better performance than any other setting for obvious reasons. Even though GRIB2 is the simpler algorithm, APAX appears to be faster; we believe that this is due to the fact that GRIB2 has to scan the input data twice (once to compute the data range and once to do the conversion) while APAX is a single pass algorithm, which leads to better cache usage. The GRIB2 encoder available from the Max-Planck-Institute for Meteorology is said to be a much more optimized encoder than the WMO GRIB2 encoder we used in this paper, but time did not allow us to verify these claims.

Comparing these two fast algorithms at the same compression level without LZMA or JPEG 2000 post-processing, we find that APAX signal quality exceeded GRIB2 signal quality for more than 85% of all variables in our dataset. At N = 24 / 16 / 8, APAX SRR signal quality averages 7 / 11 / 8 bits better than GRIB2, respectively. In some cases, however, GRIB2 generates better SRR and PrecisionBits metrics than APAX on the least compressible datasets that exhibit very few repeated values, including the datasets random, random_offset, sines_orthogonal, lsp, st and az0w. Since APAX needs more control bits than GRIB2 to describe what the encoder has done, these bits are consequently not available to encode the data itself, and which provide no benefit when all alternatives are equally bad. So, even if GRIB2 were as fast or faster than APAX, in this speed class APAX is the better encoding for most climate variables due to its supperior compression or data quality.

If execution speed is not as much of a concern, the GRIB2/JPEG 2000 and GRIB2/LZMA combinations come into play. While neither can outperform the other on the majority of variables, both profit from the tight error guarantees of the GRIB2 format, indicated by their better PrecisionBits results compared to APAX, and for most variables one combination clearly outperforms the other. So, while it is clear, that one of these combinations should be used when good compression is more important than speed, the decision which of the two to use should be made on a per variable basis.

6 Compression Use Cases and Benefits for HPC

The most easily obtained benefit from lossy compression of climate datasets is a significant reduction in disk file size and a corresponding increase in disk bandwidth. Compared to lossless compression, both GRIB2 and APAX lossy compression can achieve significantly higher compression ratios with acceptable quality, as described in Section 4. Both GRIB2 and APAX are fast enough to saturate typical filesystems, in HPC settings with high throughput parallel filesystems, several cores might be neccessary, though. For archiving applications where processing speed is not critical, the combination of GRIB2 and JPEG 2000 provides slightly better signal quality than APAX and is thus the preferred solution.

As climate simulation resolution improves, and as HPC core counts continue to increase, lossy compression could also be used to reduce other system bottlenecks, including PCIe, Infiniband and Ethernet links for data exchange, and DDR memory bottlenecks that store increasingly large climate datasets. In these cases, compression performance (sec/GB) becomes critical. Bus and networks speeds can reach 56 Gbps (FDR Infiniband), while sustained HPC server DDR3 memory throughput now achieves 5 GB/sec. If lossy compression could be used, simulations could complete faster as if they were using faster network and memory. As HPC core counts have increased, overall core utilization has decreased (sometimes to below 20% of peak MIPS), so significant CPU cycles could be used for compressing and decompressing datasets in DDR memory. This, of course,

would require a very tight coupling between the simulation code and the compression/decompression code.

Climate scientists will have to trade off where such *in situ* compression would add the most benefits while maintaining climate simulation results. Due to the statistical nature of climate research, input and output datasets could probably be lossy compressed. For intermediate climate simulation results, climate scientists will want to carefully monitor the numerical stability and consistency of results as they evaluate compression's artifacts. Most intermediate results will likely require lossless handling. However, even lossless compression schemes might accelerate simulations if they are fast and applied to the more compressible variables.

7 Summary and Future Work

We have compared GRIB2 (with and without optional JPEG2000 and LZMA post-processors) and the APAX lossy compression algorithms on synthetic and climate datasets. At equivalent compression ratios, APAX signal quality exceeds GRIB2 signal quality for most climate variables. On some climate datasets, GRIB2 compression ratios are improved by 60x (JPEG2000) to 3800x (LZMA). GRIB2 and APAX processing speeds are comparable to each other, and both are at least 10x faster, and often 100x faster, than when GRIB2 includes JPEG2000 or LZMA post-processing. In the future, we plan to investigate how much APAX compression would be improved by JPEG2000 and LZMA post-processing. We also plan to involve climate scientists in quantifying the acceleration in "time to results" that the fast GRIB2 and APAX algorithms could provide and analyze the required precision, by increasing HPC memory and disk capacity and bandwidth.

References

1. Christopoulos, C., Skodras, A., Ebrahimi, T.: The JPEG2000 still image coding system: an overview. IEEE Transactions on Consumer Electronics 46(4), 1103–1127 (2000)
2. Dey, C., et al.: Guide to the WMO Table Driven Code Form Used for the Representation and Exchange of Regularly Spaced Dat. In: Binary Form: FM 92 GRIB Edition 2. Tech. rep., World Meteorological Organization (2007), http://www.wmo.int/pages/prog/www/WMOCodes/Guides/GRIB/GRIB2_062006.pdf
3. ECMA: Streaming lossless data compression algorithm - (sldc), ECMA Standart 321 (2001)
4. Hübbe, N., Kunkel, J.: Reducing the HPC-Datastorage Footprint with MAFISC - Multidimensional Adaptive Filtering Improved Scientific data Compression. In: Computer Science - Research and Development. Executive Committee. Springer, Heidelberg (2012), doi:http://dx.doi.org/10.1007/s00450-012-0222-4

5. Iverson, J., Kamath, C., Karypis, G.: Fast and effective lossy compression algorithms for scientific datasets. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds.) Euro-Par 2012. LNCS, vol. 7484, pp. 843–856. Springer, Heidelberg (2012)
6. Lakshminarasimhan, S., Shah, N., Ethier, S., Klasky, S., Latham, R., Ross, R., Samatova, N.F.: Compressing the incompressible with ISABELA: In-situ reduction of spatio-temporal data. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part I. LNCS, vol. 6852, pp. 366–379. Springer, Heidelberg (2011)
7. Lakshminarasimhan, S., Shah, N., Ethier, S., Ku, S.H., Chang, C.S., Klasky, S., Latham, R., Ross, R., Samatova, N.F.: Isabela for effective in situ compression of scientific data. Concurrency and Computation: Practice and Experience (2012)
8. Lindstrom, P., Isenburg, M.: Fast and efficient compression of floating-point data. IEEE Transactions on Visualization and Computer Graphics 12(5), 1245–1250 (2006)
9. Sullivan, S.: Wavelet compression for floating point data–sengcom. Tech. rep., University Corporation for Atmospheric Research (2012), `http://www.unidata.ucar.edu/software/netcdf/papers/sengcom.pdf`
10. Wegener, A.: Adaptive compression and decompression of bandlimited signals. US patent 7,009,533 (2006)
11. Woodring, J., Mniszewski, S., Brislawn, C., DeMarle, D., Ahrens, J.: Revisiting wavelet compression for large-scale climate data using JPEG2000 and ensuring data precision. In: 2011 IEEE Symposium on Large Data Analysis and Visualization (LDAV), pp. 31–38 (2011), doi:10.1109/LDAV.2011.6092314

The Effect of Topology-Aware Process and Thread Placement on Performance and Energy

Albert Solernou, Jeyarajan Thiyagalingam, Mihai C. Duta, and Anne E. Trefethen

Oxford e-Research Centre, University of Oxford, 7 Keble Road, Oxford, OX1 3QG, UK
jeyarajan.thiyagalingam@oerc.ox.ac.uk

Abstract. Design of modern multiprocessor computer systems has become increasingly complex and renders the performance of scientific parallel applications highly sensitive to process and thread scheduling. In particular, the Non-Uniform Memory Access (NUMA), a frequent architecture solution, demands knowledge of the hardware details as well as skills that are normally beyond the average user in order to minimise memory access penalties and achieve good application performance. This situation is further complicated by the increasing use of modern heterogeneous systems involving both CPUs and accelerators, where process proximity to the accelerator strongly determines performance.

In this paper, we use a purpose-built, simple to use and extensible software library called TAPS (Topologically-Aware Process/Thread Scheduling) to evaluate the effect of topology-aware process placement on energy and runtime performance. Our evaluation used a suite of benchmarks and real-world applications showing that even simple placement schemes can provide not only a significant increase in runtime performance but also considerable energy savings (20% is typical but can be higher). Runtime and energy evaluations were carried out on a variety of multicore NUMA systems as well as on a heterogeneous system with graphic cards and a computer cluster with multicore NUMA nodes and InfiniBand interconnect.

Keywords: energy-efficient software, affinity mapping, hardware locality, process and thread placement, multi-core processing.

1 Introduction

The increasing complexity of modern High Performance Computing (HPC) systems pose a serious challenge to the performance and scalability of scientific parallel applications. The typical combination of multicore processors and Non-Uniform Memory Architecture (NUMA) renders applications susceptible to memory access latency and highly sensitive to the placement of processes and threads on the cores. This situation on multicore systems extends to the performance of parallel applications on clusters and is exacerbated by the fact that modern systems are increasingly heterogeneous, including conventional heavy-weight cores, light-weight cores and accelerators.

Operating system (OS) process and thread schedulers function around different criteria than those favoured by scientific computing, with fairness and responsiveness being very important features. Furthermore, scheduling policies may even have negative effect on application performance, for example due to thread migration. As a result, the

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 357–371, 2013.

task of optimising parallel applications is left to the developers. Conventionally, performance tuning on NUMA systems is addressed in two different ways: by managing the affinity of threads or processes [1,2] or by restructuring the parallel code to adjust the locality of the data [3]. These options demand an intimate knowledge of both the memory access patterns (or communication patterns) of the application and the underlying hardware architecture. Even then, process or thread contention limit the effectiveness of affinity mapping or restructuring and hence the application performance and scalability [4,5]. Although techniques based on code or application analysis (memory access patterns or communication patterns) may render very optimal placements, they are well beyond the skills of average parallel application developers.

In this paper, we use a purpose-built, simple and extensible library to evaluate the effect of topology-aware process and thread placement on runtime performance and energy consumption. The library, referred to as TAPS (Topologically Aware Process/Thread Scheduling), offers a robust yet simple mechanism for application developers to control the placement of threads and processes on multicore and distributed systems at the application level. The library is aware of the underlying hardware topology and enables the choice between a completely automatic process/thread placement or a guided one. TAPS is designed to co-exist with other libraries and thus can make use of additional techniques and has very negligible performance or operational overheads.

Our evaluation study presented in this paper uses a suite of benchmarks and two real-world, industry-strength applications on a representative range of HPC architectures. As mentioned before, we cover two important metrics: runtime performance and overall energy consumption. The latter is a growing concern in the context of modern high performance computing domain. Our results show that even simple binding schemes (offered by the TAPS library, in our case) can offer significant runtime and energy performance benefits over the code that leaves placement to the OS. The results were obtained on a number of multicore NUMA systems, on a heterogeneous system with graphic cards (GPUs) and on a compute cluster with multicore NUMA nodes and InfiniBand interconnect.

The rest of the paper is organised as follows. In Section 2 we present the details of the TAPS library, followed by Section 3, where we discuss the techniques we used for energy profiling of applications. This is then followed by Section 4, where we outline the details of our experimental evaluation covering benchmarks and platforms. We present and discuss our results in Section 5 and we cover similar and related work in Section 6. We then conclude the paper in Section 7 sharing some of our experience and outlining directions for further work.

2 The TAPS Library

The TAPS library is a simple yet extensible general-purpose scheduling library, designed to leverage the topological information of the underlying hardware for facilitating the control of process and thread affinity in parallel applications. The key principle of the TAPS library is to provide a high level of abstraction while providing a better control of affinity mapping to parallel application developers.

TAPS is built on top of the topology-aware HWLOC [6,7] library, which provides a portable abstraction of the hierarchical topology of modern architectures. While using HWLOC directly demands some expertise in controlling the affinity mapping, TAPS provides an extra abstraction layer, exposing the powerful features of the HWLOC in a form that is very easy to use by software developers. This abstraction is delivered by two different modes supported by TAPS: manual and automatic.

In the manual mode, library expects the affinity mapping to be supplied explicitly. This can be either given by the user or by another application. For example, it is feasible to obtain an optimal mapping through aggressive code analysis and tracing (such as data locality or communication patterns). Alternatively, the developer may have in-depth knowledge of the application behaviour and hence be able to provide an optimal mapping. Thus the manual mode is suitable for interfacing TAPS with other analysis tools — a robust means for extending the capability of TAPS.

In the absence of any analysis tools or explicit mappings, the TAPS library performs automatic placement. For this, it uses the topology information from HWLOC to map instances to cores, resulting in the same functionality as tools such as `taskset` and `cpuset`. However, unlike these tools, the placement is automatic, thus eliminating any user involvement, and scales from single node to a multi-node cluster in a transparent fashion. In automatic mode, TAPS does not perform any code analysis to establish the data locality, memory access patterns or communication patterns. Instead, process ranks and thread identifiers are extracted automatically and they are further handled as depending on the nature of the application: MPI, OpenMP, or hybrid (any combination of MPI and/or OpenMP with GPU).

If the application uses only heavy-weight CPU cores, TAPS follows two different ways to bind processes (or threads) to cores: uniform distribution and biased (or lumped) distribution. The uniform distribution assigns processes to cores evenly across the system, balancing the load as evenly as possible. In the case of lumped distribution, processes/threads are bound to cores as close to each other as possible. These mappings are derived based on information obtain through HWLOC such as number of processors in a node, total number of cores per processor and memory banks per processor. While the uniform distribution is obtained through modulo mapping, the lumped distribution is derived through in-order mapping. In both the cases, the mappings to logical cores (especially when hyper-threaded), can be controlled at will at runtime. Obviously, on a fully occupied system (when the number of threads or processes match the core count), there is no difference between both binding schemes. If the application uses GPUs, the TAPS mapping is biased towards the processor closer to the GPU. The library assumes one GPU per process, and the process is bound to the nearest die to the GPU. In the case of threaded MPI, the process is able to spawn threads which are then mapped to the cores of the same die.

All these complex details are abstracted from the user using a simple function call to TAPS (`taps_bind()`) at the beginning of the application. When TAPS is not used, the assigning of processes or threads to cores can be arbitrary and may not necessarily live closer to the GPU and hence may incur transfers through interconnects. The TAPS can be optionally customised by specifying the GPU to be used, The effect of calling to `taps_bind()` is achieving a ”static” affinity mapping, which is persistent for the

instances for the entire run of the application. The static mapping is simple but fits the reality of a large number of scientific applications adequately, in which heap memory data is allocated once and accessed repeatedly. Nevertheless, if an application demands a "dynamic" affinity mapping, either because data size changes or the number of parallel instances changes, binding can be redone at any stage in the code, with the affinity control prescribed entirely by the programmer. Hence the performance overheads due to re-binding is entirely dependent on the optimality of the new binding scheme.

3 Energy Profiling

The energy consumption of an application is the sum of energy consumption of different components in the system, such as memory, processors, data paths and so on. By integrating the instantaneous power profile of the system over the duration of the execution, it is possible to estimate the energy consumption of an application. To be more precise, the approximate energy consumption is the integrated value of the power profile less the idle energy of the system. The idle energy is simply the idle power value integrated over the duration of the execution.

The instantaneous power profile of a system is measured by using a number of power sensors. Power sensors are inserted between the power source and the concerned computer system, as shown in Figure 1. When profiling clusters, this arrangement is repeated for each node/

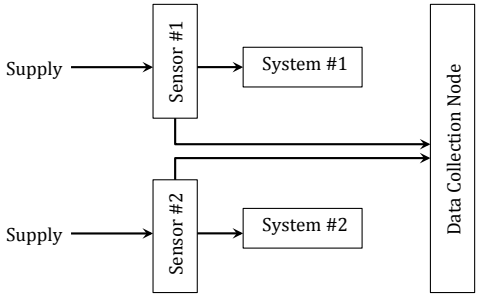


Fig. 1. A simple configuration for monitoring the energy consumption of systems

To facilitate the profiling, we used the EMPACK software [8], which provides necessary API functionalities to profile the energy consumption at code block level. The EMPACK triggers on and off appropriate sensors prior and after to executing a code block and posts the line conditions to a central data collection server. Upon completion, the library estimates the energy consumption as discussed before. In doing this, the software assumes that the idle power of the system is time invariant and the system suffers no perturbations from other processes during the execution of the concerned application. If the system has more than one power supply, the energy values are integrated

over multiple supplies as needed. Although this type of configuration can only provide overall power measurements, it has the advantage of simplicity and non-intrusiveness, thus allowing for measurements on "real-world", non-dedicated systems.

4 Evaluation

The performance and energy measurements reported in this paper were obtained on a number of representative NUMA-based HPC platforms. Table 1 gives the technical specifications of these systems, along with the details of the compilers employed. In addition to these platforms, we also used a part of a cluster — Arcus, consisting of two chassis of two nodes each, and each node with two CPUs and 16 physical cores and 64GB of RAM. Separate measurements were taken on this cluster for the compute nodes and for the Infiniband network during the execution.

Table 1. Details of the systems on which the experiments were conducted

	t2wn7	**zen**	**skyray**	**broomway**	**arcus nodes**
Processor Model	AMD Intelagos 6276	Intel Xeon HarperTown 5650	AMD Opteron Magny Cours 6128	Intel Xeon SandyBridge E52680	Intel Xeon SandyBridge E52650
Mother-board	Dell (0VKT0M)	Supermicro (X8DTG-QF)	Supermicro (H8DG6/H8DGi)	Intel (S2600CP)	Dell (0HYFFG)
Clock Speed	2.3-2.6 GHZ	1.6-2.8GHz	2.0-2.6GHz	2.7 GHz	2.0GHz
Number of Processors	4	2	2	2	2×4
Cores per Processor	16	6	8	8	8
Total Physical Cores	**64**	**12**	**16**	**16**	**16**×4
Linux kernel version	3.5.2	2.6.35	3.2.0	2.6.32	2.6.32
RAM Size	128 GB	24 GB	12 GB	64 GB	64 GB
L1/L2 Cache	16 KB / 8×2 MB	32 KB / 256 KB	64 KB / 512 KB	32 KB/ 256 KB	32 KB/ 256 KB
L3 Cache	2×8 MB	12 MB	2×6MB	20 MB	20 MB
Compilers	Open64 4.5.1-1 (AMD patch)	Intel Compilers Version 12.0.4	Open64 4.5.1-1 (AMD patch)	Intel Compilers Version 13.0.1	Intel Compilers 12.1.6
Compiler flags	-O3 -march=auto -fPIC	-O3 -xSSE4.2 -ip -mcmodel=medium	-O3 -march=auto -fPIC	-O3 -xAVX -ip -mcmodel=medium	-O3 -xAVX -ip -mcmodel=medium

We used two different groups of scientific benchmarks for our evaluation. The first software group (evaluated on all systems but the cluster) was the NAS Parallel Benchmark suite [9] (version 3.3), a set of synthetic benchmarks designed for testing HPC systems. We used a subset of the NAS suite, chosen with different memory access characteristics, different compute-to-communication ratios and sufficiently long but feasible runtimes. A summary of the benchmarks used is given in Table 2, along with the associated problem sizes (benchmark Class C). The choice of problem sizes was such that the resolution for measurement was sufficient and the capacity of the systems tested was not exceeded. There are three implementation variants (OpenMP, MPI and hybrid threaded MPI, MultiZone or MZ) and they were used independently. The second software group (evaluated on all systems and the cluster) was two independent real-world, industry-level applications:

– AirFoil: A computational fluid dynamics benchmark, which is a part of an open source library for unstructured grid computations called OP2 [10]. This MPI

Table 2. Details of the NAS benchmarks used in the paper. Note that the problem size for class C for the MPI and OpenMP (162^3) is different from the hybrid implementation.

Benchmark	Description	Grid Size	Parallelism
MG	Multigrid benchmark, long- and short-distance communication, memory intensive	512^3	MPI, OpenMP
CG	Conjugate Gradient, irregular memory access and communication	$1.5 \cdot 10^5$	MPI, OpenMP
FT	Discrete 3D fast Fourier transform, all-to-all communication	512^3 512^3	MPI, OpenMP MPI, OpenMP
BT	Block tridiagonal solution benchmark	162^3 - $480 \times 320 \times 28$ (*)	MPI, OpenMP and Hybrid
LU	Lower-Upper Gauss-Seidel solver	$480 \times 320 \times 28$	Hybrid
SP	Scalar Penta-diagonal solver	$480 \times 320 \times 28$	Hybrid

benchmark simulates the two-dimensional inviscid flow over an airfoil and each MPI process utilises multiple threads — either OpenMP threads on a multicore system or CUDA threads on GPU cards.

– Gromacs: An industrial strength computational molecular dynamics code [11], with a very large world-wide user community base. Gromacs is an MPI-based application, with the option of OpenMP multithreaded processes (proven to be very beneficial to scaling over slow networks) and processes accelerated via GPU computing. We have simulated the molecular systems that are provided with the Scalalife Validation Suite [12]. It is interesting to note that, with the same considerations put forward by this paper, Gromacs has its own process affinity control. This provides an additional route to verify our findings; the performance and energy footprint of the code instrumented with TAPS but with no own affinity control is benchmarked against Gromacs with its own affinity control. Unlike TAPS, the affinity control implemented in Gromacs is relatively simple and application specific (it corresponds to the TAPS lumped together scheme), so it is not adequate for MPI process control when GPU acceleration is involved, a feature that TAPS covers.

The last application is heavily employed in the Life Sciences, *e.g.* for screening new drugs and targets, while seeking a reduction in the associated side-effects and toxicity. These tasks are routinely associated with very large numbers of similar simulations, which perform virtual drug screening and de-novo drug design. Therefore, another reason for choosing Gromacs was its green computing relevance; any performance and power advantage demonstrated in a single run is amplified tens or hundred of times in such a series of simulations.

5 Results

The overall parameter space of our experimental evaluation cover different benchmarks on different systems with several types of parallelism on a varying number of cores. As a result, the overall result space is considerably large to present here in raw form. For this reason, we use some of them to highlight the benefits of TAPS, the lessons learnt and to perform sample analysis but summarise all the results in a condensed form in Tables 3 and 4 and Figures 5 and 6. The summary presented here are only for for fully occupied systems — a most common use scenario. It is noticeable that

majority of the benchmarks benefit from TAPS. In presenting the results, we use the legend captions *TAPS-du* and *TAPS-lt* to mean the uniform and lumped distributions of processes or threads in TAPS, respectively. Whenever we use the Gromacs-specific binding, we denote this by *GMX-binding*.

5.1 Evaluation on Multicore Systems

Figures 2 and 3 show the runtime and energy performance of the Gromacs application on the Broomway system, simulating a 40,000 atom lipid bilayer system benchmark [13], with the maximum number of processes or threads matched that of the available physical cores. In Figure 2(a), we show the variation of execution time of the pure MPI variant of Gromacs against the number of processes. Regardless of the binding scheme (none, uniform, grouped) runtime decreases as expected with the number of processes. The binding offers better performance (by over 20% at full occupancy) than the runs without any binding. The energy counterpart of the scaling results is shown in Figure 2(b), which provides the variation of energy consumption against the number of processes. The general trend is intuitive: faster executions (using a larger number of cores) are associated with a lower energy consumption. The energy savings even on a fully occupied system due to process binding is rather remarkable and exceeds 20%.

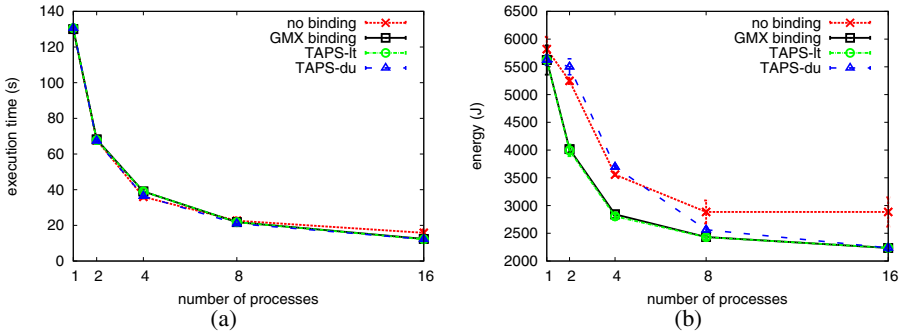


Fig. 2. The variation of runtime (a) and energy consumption (b) of the pure MPI version of Gromacs with the number of processes on the Bromway system. Both performance improvements and energy savings are assocated with TAPS binding.

Figures 3(a) and 3(b) depict similar trends as in Figure 2 but for the pure OpenMP variant of Gromacs. Again, the scaling of the threaded application is good, and faster execution correlates with lower energy consumption. Moreover, binding of threads to specific cores is beneficial and leads to both faster execution (over 30% faster on a fully occupied system) and lower energy consumption (again, by over 30%), all compared with the unbound runs. In addition to depicting typical application behaviour, Figures 2 and 3 also validate TAPS; indeed, TAPS is demonstrated to have the same effect (through the lumped binding scheme) as the Gromacs native binding.

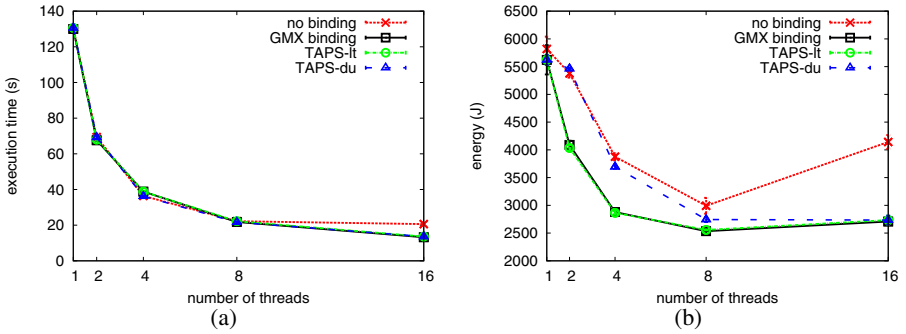


Fig. 3. The variation of runtime (a) and energy consumption (b) of the pure threaded version of Gromacs with the number of threads on the Bromway system. Both performance improvements and energy savings are assocated with TAPS binding.

The results depicted in Figures 2 and 3 are typical across all cases, with the same trends reproduced (with few exceptions) across all benchmarks on all platforms. Indeed, in Tables 3 and 4, we present the relative runtime and energy consumption gains against the no-placement scheme. While the Table 3 shows the gains for the OpenMP and MPI variants of the NAS benchmarks, Table 4 shows the same for the MultiZone (hybrid) version of NAS benchmarks and hybrid version of the applications (for a limited set of platforms). The relative gains are shown for a fully occupied system (which constitutes the typical run scenario). The ratios are taken for the mean values measured with the original benchmark (without any placement) and those measured with the instrumented code; ratios greater than unity indicate an advantage from TAPS and values less than unity indicate the loss in performance or energy consumption.

Table 3. The ratio between the respective measurements (runtime t and energy e) on the original NAS benchmarks and the TAPS instrumented versions runtime. Shown are the ratios for both the MPI and OpenMP variants of the code and indicates a benefit (value greater than unity) from using the automatic TAPS placement in most cases.

		broomway				**zen**				**t2wn7**				**skyray**			
		BT	**CG**	**FT**	**MG**	**BT**	**CG**	**FT**	**MG**	**BT**	**CG**	**FT**	**MG**	**BT**	**CG**	**FT**	**MG**
MPI	t	1.47	1.91	1.53	1.73	1.07	0.87	0.84	0.85	1.09	0.69	0.67	0.41	1.07	1.01	0.99	0.99
	e	1.31	1.61	1.32	1.47	1.03	0.81	0.79	0.73	1.15	0.84	0.80	0.55	1.07	1.01	0.99	0.99
OMP	t	1.42	1.40	1.35	1.15	1.31	1.21	1.05	1.07	1.37	1.16	0.61	0.80	1.02	1.14	1.01	1.09
	e	1.26	1.27	1.21	1.09	1.16	1.08	1.11	1.09	1.29	1.23	0.78	0.99	1.02	1.09	1.00	1.09

The exact gains in performance and energy vary depending on the platform and benchmark. For example, considering the Table 3, it is evident that the overall gains on the Broomway system is considerably higher than that on the Skyray platform.

Table 4. Relative gain in time *t* and energy *e* from TAPS in the hybrid MPI/OpenMP benchmarks. We used one process per socket, each spawned as many threads as cores available per socket.

	broomway					skyray					t2wn7
	BT	**LU**	**SP**	**gromacs**	**airfoil**	**BT**	**LU**	**SP**	**gromacs**	**airfoil**	**airfoil**
t	1.324	1.330	1.450	1.439	1.431	1.115	1.058	1.139	1.036	1.010	1.061
e	1.218	1.208	1.347	1.430	1.376	1.096	1.044	1.135	1.014	1.185	1.100

Broomway is a 16-core machine with hyper-threading enabled, resulting in 32 cores for possible bindings. With the fact that our benchmarks (both for MPI OpenMP) used only 16 physical instances, the original (unbound) version has a higher probability for imbalance and migration as the system sees 16 free (albeit logical) cores. When using TAPS, this probability and chances for runtime migrations are eliminated and thus bringing in a noticeable stability for the application, resulting in runtime improvement and energy savings and hence higher relative gains. On the other hand, on the Skyray platform, there are no free cores when all 16 single cores are fully occupied and hence reduced chances for migration and imbalance. Although using TAPS may reduce these effects somewhat further, the gains are not good as on the Broomway platform.

The gains on the Zen platform is somewhat mixed, with BT and OpenMP showing gains and the rest showing a loss in performance and energy consumption. Despite the fact that Zen is a 12 core machine with hyper-threading (additional 12 logical cores), the MPI benchmarks are unable to make use of all cores due to the nature of the benchmarks. The BT benchmark can only utilise square number of processes ($1^2, 2^2, \ldots, n^2$) while all other benchmarks demand number of processes to be power-of-two value. This limits the full occupancy for BT benchmark to be nine processes and eight for the rest. Given the architecture of the CPU (six physical cores, with two Level-3 cahes shared by groups of three cores), all the benchmarks could only run under the lumped-together scheme but this leads to a substantial imbalance to the benchmarks with eight processes.

In terms of MultiZone/Hybrid versions, as can be observed from the tables, binding always brings benefits and in particular the gains are significant for the Broomway platform (for the reasons discussed above).

It is a general belief that the faster an application is executed, the lower the energy consumption associated is and this is indeed confirmed by many of our measurements. However, this is not always the case, as illustrated in Figure 4. The variation of runtime of the pure MPI variant of the MG NAS benchmark with the number of processes on Broomway is shown. The number of processes is from one to full system occupancy, and poor scaling is observed beyond half the number of physical cores available. Regardless of the poor scaling, the full occupancy offers faster runtime than the single process. Nevertheless, the variation of energy with the number of processes indicates that full occupancy has an energy footprint higher than a single process. It is worth noticing that the minimum of energy consumption corresponds to a number of processes equal to half the total number of cores, and equally distributed across sockets. Whichever number of processes is used, using TAPS proves to be always beneficial, even with this "difficult" benchmark.

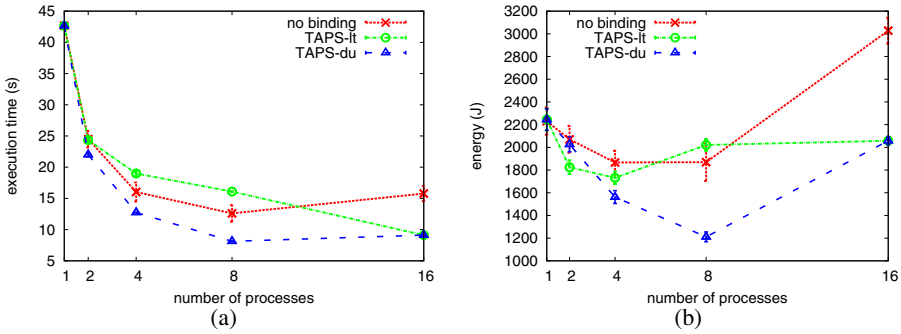


Fig. 4. The variation of runtime (a) and energy consumption (b) of the pure MPI version of the NAS MG benchmark with the number of processes, showing poor scaling beyond half the number of physical cores of the system and sub-optimal energy consumption at full occupancy. TAPS binding still has a beneficial effect.

To investigate the behaviour even further, we analysed the NAS-MG MPI application on the Broomway platform using Paraver [14], a flexible tracer able to capture both the hardware events and the MPI calls. When using eight processes (Figure 4), we observed that TAPS-lt suffers substantial L3-related cache misses compared to TAPS-du (8.5×10^5 vs. 4.9×10^5 on the most important computation phase) due to the limited size of the L3 cache memory and the synchronised access of the processes. When comparing TAPS-du from eight to 16 processes, one can see that the amount of L3 cache misses per process is very similar, but this means that the total amount of failures is doubled for the 16 processes run, and hence the speedup is flat from 8 to 16 but the amount of energy is almost doubled.

Furthermore, when the system is fully occupied (with 16 processes), the original version (where TAPS is not used) spends around 23% (of its 15 seconds) on communication compared to 8% (of the ∼8 seconds) of TAPS counterparts. Hence, besides an increased amount of L3 cache misses, the non-TAPS run shows an important imbalance amongst processes which is reflected in the performance/energy plots.

5.2 Evaluation on Heterogeneous Systems

When TAPS is used against two GPU accelerated applications (AirFoil and Gromacs) on Zen, the results are somewhat mixed. The AirFoil benchmark did not benefit from TAPS placement regardless of the number of GPUs were used. As mentioned before, the MPI-CUDA version of AirFoil benchmark uses one GPU per process and in our case we limited this up to two processes. The computation phase is entirely carried out on the GPU without any support from CPU and data exchanges are only for exchanging the halo cells of the partitioned mesh. Although the TAPS library bound the processes to the CPU cores next to the GPUs (i.e. avoiding them being lumped together), we did not see any benefits. Our interpretation is that although communication is carried on explicitly

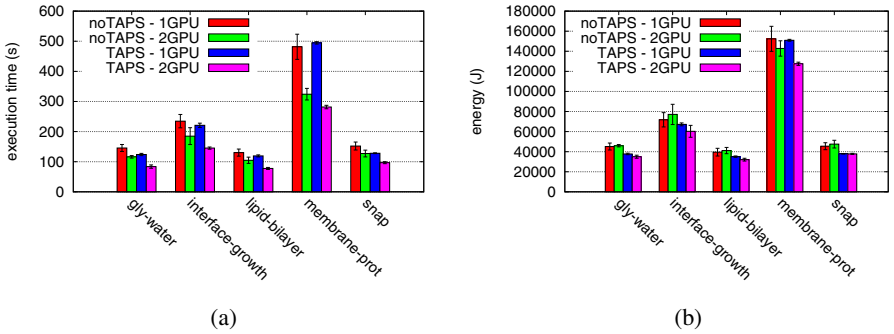


Fig. 5. The variation of runtime (a) and energy consumption (b) of the pure GPU accelerated version of Gromacs using one and two cards. Both performance improvements and energy savings are associated with TAPS binding. The performance increase on two cards is between 13% and 28%, while the energy savings are between 10% and 24%.

on the CPU side, this is only for the data exchanges between the GPUs and hence the exact location of the process in the CPU core does not matter. Instead, the results would have been different if the CPU processes were to take part in computation.

However, for the Gromacs benchmark, we did see noticeable differences in performance and energy consumption due to TAPS. Here, both CPUs and GPUs are used throughout the calculation phase in an overlapped fashion: GPU calculating the non-bonded forces while the bonded forces calculations are carried on the host. The Figure 5 shows Gromacs simulations for five different molecular systems (labelled on the horizontal axis), each with one or two MPI processes, each process using a GPU and six CPU-bound threads. When TAPS is not used, using two GPUs brings performance benefits but no energy benefits. However, using TAPS we observe a consistent gain in energy (up to 24% savings) and runtime (up to 28% reduction) for all the benchmarks. Although not presented here, another molecular system (known as virion), was too large to run on a single card, but showed an improvement of the 11% in performance (over 2089 seconds) and 7.5% in energy (over 7.8×10^5J) when using TAPS. When processes are bound to a core next to the GPU, message passing is streamlined to the specific GPU without using the interconnect between processors, which brings performance benefits and energy savings.

5.3 Evaluation on Clusters

Finally, we present the effects of placement on the cluster. As discussed in the previous Section, we measured the energy consumption of every compute node and switch associated with the computation. However, the resolution of the sensors was too low to capture the instantaneous power variations of the switch and the overall energy consumption was underestimated. Therefore, we only report the energy consumption of the nodes, with the observation that this dominates the total energy consumption associated with the runs. Figure 6 shows the variation of runtime and energy performance with the number of cluster nodes used to run the Gromacs bilayer benchmark. The general

trend is that there is good scaling and runtime decreases as expected but the energy consumption does not. Nevertheless, the binding provides at least 10% of energy savings across nodes. Another observation is the benefit of TAPS-based automatic binding (using no familiarity of the application code) matches the native binding from Gromacs; this validates TAPS yet again.

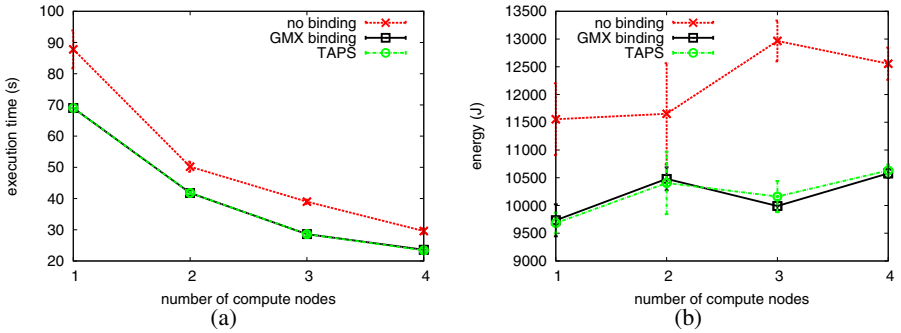


Fig. 6. The variation of runtime (a) and energy consumption (b) of the pure MPI version of Gromacs on a cluster with the number of compute nodes

6 Related Work

There exists a large body of work in affinity mapping or process placement. In [15], Kruskal and Snir formulated the placement problem for performance improvement as a multicommodity flow problem. Several vendors provide specialised libraries capable of performing process placement based on ranking of processes (for example based on [2]). The MPIPP framework uses the execution profile to schedule MPI processes in different parts of the cluster [16]. Pellegrini *et. al.* provided a graph-based partitioning algorithm [17] for mapping processes. Jeannot *et. al.* then improved this further with a tree-based algorithm called TreeMatch [1] where they accounted the hierarchical memory structures encountered in the NUMA setup (relied on HWLOC as we do). In [18] Su *et. al.* provide a runtime system for optimal thread placement in NUMA systems based on the data-locality analysis. Most of these are entirely concerned on runtime performance and it was taken granted that energy consumption is directly related to runtime performance. In [19], Su *et. al.* consider the similar problem to ours and developed a model for optimal placement of threads entirely based on memory accesses; and evaluated the model-based mapping on energy and runtime.

In contrast, our work is aimed at evaluating the impact of topology-aware placement on runtime and energy consumption. For this, we used a simple purpose-built library, TAPS, designed with portability and ease-of-use in mind and therefore expected to accommodate any of the above techniques, including the user specifying the placements. When none supplied, the library uses the topological information for providing two

automatic placement schemes - uniform and lumped distribution schemes. The library does not perform any in-depth analysis on memory access patterns, locality or communication patterns.

With respect to energy measurement and profiling, despite the need for power/energy measurement tools, only few frameworks provide capability towards capturing total energy consumption of systems. These include PowerPack [20] and EMPPACK [8]. The former provides detailed, fine-grained measurements at component level covering all system components such as memory, processor, disks, network interface and so on. However, adopting this framework for ad-hoc use may not always be practically feasible. The latter overcomes this by providing a simple framework for capturing total system energy. Both of these frameworks scale from single node to clusters and by far the most complete tools. Using the proprietary instrumentation facility from hardware vendors is another way to obtain energy measurements. For example, IBM PowerExecutive Toolkit in IBM systems (such as BlueGene/Q/L) provides a means to obtain energy measurements to systems. However, measuring the energy at the application level, especially using API is a challenging task and to our knowledge only the EMPPACK framework supports it. Other efforts include component level monitoring (e.g. [21]) and estimation techniques such as based on simulations [22,23] or analysis of power profile-based estimation techniques [24,25]. Simulation techniques rely on energy models constructed at the design phase of systems or components. The estimation techniques rely on performance logs and thus require direct measurement of performance. These techniques fail to scale across application types, across systems and at times beyond component level and therefore not very reliable as direct measurement-based techniques.

7 Conclusions

In this paper, we evaluated the impact of topology-aware process and thread placement on runtime performance and energy footprint of parallel applications on modern HPC systems. For this, we designed a simple yet extensible topology-aware scheduling library called TAPS. Using TAPS as a vehicle for topology-aware placement of processes and threads, we analysed the impact on runtime performance and energy consumption. Using a suite of parallel benchmarks and applications on a number of HPC systems (including a distributed system), we demonstrated increased performance and reduced energy footprint from process and thread scheduling in most cases. The energy savings of up to 20% in the case of Gromacs on a cluster are particularly noteworthy, as this molecular dynamic code is routinely run hundreds of time within a single drug study. Also, our evaluation include a few "difficult" cases, in which scaling up to full occupancy of a multicore system is poor and, although placements (through TAPS) may not bring any runtime benefits, they still bring notable energy savings. It is interesting to notice that in the same cases, energy savings are not directly linked to runtimes; full system occupancy (the common practice) is not optimal from the point of view of energy usage. The runtime benefits and energy savings varied considerably across platforms, but SandyBridge-EP showed remarkable benefits.

We see our work, including the TAPS, as a platform for further evaluation and development of different placement schemes. Given its simplicity of use and the very

small associated re-programming overhead, the clear overall benefits (especially in energy savings) from process and thread scheduling commend TAPS as a powerful tool. Nevertheless, the library is over-simplistic in that it both assumes a static data and process/thread configuration and ignores the memory access patterns. While reflecting the reality of many scientific applications, a static data configuration (data is created once and does not vary in size) cannot lead to a universally optimal binding scheme. More importantly, while leaving developers the possibility to guide the binding, a tool like TAPS should nevertheless offer insight into the memory access and communication patterns at runtime. As outlined in Section 6, a range of intensive analysis techniques exist to enable deriving optimal placements. In our future work, we intend to enhance the TAPS with these techniques and evaluate the relative benefits of a dynamic, analysis-informed placements against the existing static placements.

Acknowledgments. This research is supported by the ScalaLife project and by the Oxford Martin School, University of Oxford. We are also grateful to Andrew Richards of the Oxford Supercomputing Centre (OSC) and Ewan MacMahon at the Department of Physics at the University of Oxford for their support in providing access to a number of benchmarked systems presented in this paper.

References

1. Jeannot, E., Mercier, G.: Near-Optimal Placement of MPI Processes on Hierarchical NUMA Architectures. In: D'Ambra, P., Guarracino, M., Talia, D. (eds.) Euro-Par 2010, Part II. LNCS, vol. 6272, pp. 199–210. Springer, Heidelberg (2010)
2. Hoefler, T., Rabenseifner, R., Ritzdorf, H., de Supinski, B.R., Thakur, R., Traeff, J.L.: The Scalable Process Topology Interface of MPI 2.2. Concurrency and Computation: Practice and Experience 23(4), 293–310 (2010)
3. Appelbe, B., Hardnett, C., Doddapaneni, S.: Program Transformation for Locality Using Affinity Regions. In: Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing (1993)
4. Terboven, C., Mey, D.a., Schmidl, D., Jin, H., Reichstein, T.: Data and Thread Affinity in OpenMP Programs. In: Proceedings of the 2008 Workshop on Memory Access on Future Processors: a Solved Problem? MAW 2008, pp. 377–384. ACM, New York (2008)
5. Mallach, S., Gutwenger, C.: Improved scalability by using hardware-aware thread affinities. In: Keller, R., Kramer, D., Weiss, J.-P. (eds.) Facing the Multicore-Challenge. LNCS, vol. 6310, pp. 29–41. Springer, Heidelberg (2010)
6. HWLOC: Portable hardware locality, `http://www.open-mpi.org/projects/hwloc`
7. Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., Namyst, R.: hwloc: A generic framework for managing hardware affinities in HPC applications. In: 2010 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 180–186 (February 2010)
8. EMPPACK: Energy Measurement and Profiling Package, `http://www.oerc.ox.ac.uk/research/energy-efficient-computing`
9. Bailey, D., Barszcz, E., Barton, J., Browning, D., al, e.: The NAS parallel benchmarks. The International Journal of Supercomputer Applications 5(3), 63–73 (1991)

10. Giles, M., Mudalige, G., Spencer, B., Bertolli, C., Reguly, I.: Designing OP2 for GPU architectures. Journal of Parallel and Distributed Computing
11. Hess, B., Kutzner, C., van der Spoel, D., Lindahl, E.: Gromacs 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation. Journal of Chemical Theory and Computation 4(3), 435–447 (2008)
12. Scalalife: Scalable software services for life sciences, `http://www.scalalife.eu`
13. Hgberg, C., Nikitin, A.M., Lyubartsev, A.P.: Modification of the CHARMM Force Field for DMPC Lipid Bilayer. Journal of Computational Chemistry (2008)
14. Labarta, J., Girona, S., Pillet, V., Cortes, T., Gregoris, L.: DiP: a parallel program development environment. In: Fraigniaud, P., Mignotte, A., Robert, Y., Bougé, L. (eds.) Euro-Par 1996. LNCS, vol. 1124, pp. 665–674. Springer, Heidelberg (1996)
15. Kruskal, C.P., Snir, M.: Cost-Performance Tradeoffs for Interconnection Networks. Journal of Discrete Applied Mathematics 37-38, 359–385 (1992)
16. Chen, H., Chen, W., Huang, J., Robert, B., Kuhn, H.: MPIPP: An Automatic Profile-Guided Parallel Process Placement Toolset for SMP Clusters and Multiclusters. In: Proceedings of the 20th Annual International Conference on SuperComputing, ICS 2006, pp. 353–360. ACM, New York (2006)
17. Pellegrini, F.: Static Mapping by Dual Recursive Bipartitioning of Process Architecture Graphs. In: Proceedings of the Scalable High-Performance Computing Conference, pp. 486–493 (May 1994)
18. Su, C., Li, D., Nikolopoulos, D., Grove, M., Cameron, K.W., de Supinski, B.R.: Critical Path-Based Thread Placement for NUMA Systems. In: Proceedings of the Second International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems, PMBS 2011, pp. 19–20. ACM, New York (2011)
19. Su, C.Y., Li, D., Nikolopoulos, D.S., Cameron, K.W., de Supinski, B.R., Leon, E.A.: Model-Based, Memory-Centric Performance and Power Optimization on NUMA Multiprocessors. In: Proceedings of the 2012 IEEE International Symposium on Workload Characterization, San Diego, CA, pp. 163–174 (November 2012)
20. Ge, R., Feng, X., Song, S., Chang, H.-C., Li, D., Cameron, K.W.: Powerpack: Energy profiling and analysis of high-performance systems and applications. IEEE Transactions on Parallel and Distributed Systems 21(5), 658–671 (2010)
21. Brooks, D., Tiwari, V., Martonosi, M.: Wattch: a framework for architectural-level power analysis and optimizations. In: ACM SIGARCH Computer Architecture News, vol. 28, pp. 83–94. ACM (2000)
22. Wang, S.H.-S., Zhu, X., Peh, L.-S., Malik: Orion: a power-performance simulator for interconnection networks. In: Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35), pp. 294–305 (2002)
23. Zedlewski, J., Sobti, S., Garg, N., Zheng, F., Krishnamurthy, A., Wang, R.: Modeling hard-disk power consumption. In: Proceedings of the 2nd USENIX Conference on File and Storage Technologies, FAST 2003, pp. 217–230. USENIX Association, Berkeley (2003)
24. Isci, C., Martonosi, M.: Runtime power monitoring in high-end processors: Methodology and empirical data. In: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 36, pp. 93–104. IEEE Computer Society (2003)
25. Hu, C., Jiménez, D.A., Kremer, U.: Efficient program power behavior characterization. In: De Bosschere, K., Kaeli, D., Stenström, P., Whalley, D., Ungerer, T. (eds.) HiPEAC 2007. LNCS, vol. 4367, pp. 183–197. Springer, Heidelberg (2007)

TUE, a New Energy-Efficiency Metric Applied at ORNL's Jaguar

Michael K. Patterson[1,5], Stephen W. Poole[2,5], Chung-Hsing Hsu[2,5], Don Maxwell[2], William Tschudi[3,5], Henry Coles[3,5], David J. Martinez[4,5], and Natalie Bates[5]

[1] Intel Architecture Group, Intel Corporation, Dupont, Washington, USA
michael.k.patterson@intel.com
[2] Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA
[3] Lawrence Berkeley National Laboratory, Berkeley, California, USA
[4] Sandia National Laboratories, Albuquerque, New Mexico, USA
[5] Energy Efficient HPC Working Group, Anderson Island, Washington, USA

Abstract. The metric, Power Usage Effectiveness (PUE), has been successful in improving energy efficiency of data centers, but it is not perfect. One challenge is that PUE does not account for the power distribution and cooling losses inside IT equipment. This is particularly problematic in the HPC (high performance computing) space where system suppliers are moving cooling and power subsystems into or out of the cluster. This paper proposes two new metrics: ITUE (IT-power usage effectiveness), similar to PUE but "inside" the IT and TUE (total-power usage effectiveness), which combines the two for a total efficiency picture. We conclude with a demonstration of the method, and a case study of measurements at ORNL's Jaguar system. TUE provides a ratio of total energy, (internal and external support energy uses) and the specific energy used in the HPC. TUE can also be a means for comparing HPC site to HPC site.

Keywords: HPC, energy-efficiency, metrics, data center.

1 Introduction

This Whitepaper is a collaborative effort of the Metrics team of the Energy Efficient HPC Working Group (EEHPC WG). It reviews successes and issues with Power Usage Effectiveness (PUE) and explores some of the gaps in the metric. It disassembles the metric, applies the same simple logic to the IT, and then to the whole; including the IT and Infrastructure. This methodology is shown to produce two new metrics, with the higher level metric being a combination of PUE and IT-power usage effectiveness (ITUE) yielding total-power usage effectiveness (TUE). These new metrics can be used to understand the entire energy use from the utility to the silicon. It can model the entire energy stack and allow exploration of how trade-offs in the infrastructure or the IT can help change the total efficiency. Previously that total efficiency could neither be measured nor trended without these proposed metrics.

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 372–382, 2013.

2 Background

Power Usage Effectiveness (PUE), introduced in a paper by Malone and Belady [1], provides a simple metric that is used to give comparative results between data centers or of a single data center over time. The metric provides a simple way to understand the energy consumed by the infrastructure for a given IT load. In 2007 the Uptime Institute reported the average enterprise data center PUE was around 2.5. [2] This meant that the data center used 2.5X the energy needed to run the IT equipment by itself. The extra energy was used for cooling, lighting, maintaining standby power generation, and power conversion losses.

The Green Grid has written a number of White Papers since the original work [3,4]. Most recently The Green Grid, DOE, EPA, ASHRAE and others produced a white paper that represents a consensus definition including how and where to measure the metric [5]. PUE is defined as:

$$PUE = \frac{Total\ Data\ Center\ Annual\ Energy}{Total\ IT\ Annual\ Energy} \tag{1}$$

The metric has progressed in maturity and its widespread use has been responsible for the energy efficiency focus and resulting progress in energy efficiency of data center infrastructure since its definition. Admittedly it is at a very high level, a fine-grained evaluation of each term and components of the terms can be found in [6].

3 The Challenge

PUE, while very successful in driving energy efficiency of the infrastructure for data centers, is not perfect. Its advantages are its simplicity, both the math and the concept. However, it is not the be-all and end-all metric for data centers. That metric would entail computational performance and energy: a ``miles per gallon'' metric for data centers. A much improved metric would be a Data Center Productivity Index (DCPI). This would be the useful work divided by the total facility energy (DCPI=Useful Work/Total Facility Energy). Useful work is difficult to define since there are many diverse computational tasks, so today there is no definition of such a metric. An exception might exist in the HPC world where more common benchmarks and applications tend to exist. One such benchmark is LINPACK metric [5]. It is not an application but simply solves a dense system of linear equations. This benchmark generally represents only a small fraction of actual applications, but it is commonly run in most HPC systems, and is used for Top500 [6] and Green500 rankings. As stated it is a poor indicator of all but a very few workload types and therefore not really an indicator of an individual clusters productivity, but it is widely run as a benchmark. The EEHPC WG is concurrently working on how to measure energy consistently and appropriately for HPC benchmarks. With definition of a Productivity Index type of metric still well in the future, there are still other metrics which can be defined. There are two specific issues with PUE that need to be understood when using it. This paper proposes a methodology to address one of them.

One issue with PUE comes from its focus on the infrastructure. Consider a given data center with a known PUE. Assume that data center then goes through a refresh and upgrades their old IT equipment with new more efficient equipment. The new will likely provide more compute capability and possibly use less energy (IT manufacturers continue to reduce energy use at part-loads and idle, providing an overall reduction in IT energy). One interesting result here is that, if the infrastructure is left alone and runs just as it did before the new equipment was brought in, the PUE will go up. While this may be troubling to some, it actually is a non-issue. First, anytime new IT equipment is brought in, the infrastructure should be reviewed for needed changes and efficiency opportunities, this is more of an operational issue than a metric issue. The fact that the PUE went up is an indication that the infrastructure energy use did not scale with the IT load. Second, and more importantly, PUE is an infrastructure measure, to trend changes in the infrastructure over time, not to trend changes in the IT equipment. Changing the IT equipment is changing the baseline.

The second issue, the subject of this paper, is that of shifting cooling or power conversion loss. By definition, everything outside the IT is infrastructure, and everything inside is IT. As in the paragraph above, if the IT load and IT equipment remains fixed and you are only tracking your own data center energy efficiency over time, PUE can be used to guide facility operational or infrastructure efficiency improvements. The difficulty comes when infrastructure loads are moved from inside to outside the box (or vice versa). To illustrate this point consider three data centers with identical workloads and numbers of servers. Consider a data center (data center (a)) using free cooling, moving outdoor air into the building with building level fans. Then the IT level fans (considered as part of the "IT energy" in PUE) will move that cool air through the IT equipment. Now consider the neighboring data center (b). It has a different configuration with no building fans and using only the fans in the IT equipment to move the air (ramping up existing IT fans or using larger fans in the IT equipment). In this case the infrastructure load goes down, and the IT load goes up. The PUE will drop in this case. At the third data center (c) fans were removed from the IT equipment altogether and only the building fans provide air movement. Data center (c) will have the lowest IT load and a higher infrastructure load. Because of this, it will have the worst PUE of the three. In order of PUE, (b) is likely the lowest, then (a), then (c). Can we conclude that (b) is the best design and that (c) is the worst? Not at all. In fact PUE should not be used for this type of conclusion. The only valid way to determine the most energy efficient design would be to measure total energy, and we can do this because we started with identical output as an assumption. With the reality that all data centers are different in the number of servers and workloads, how would one compare the increasingly common case of infrastructure (cooling or power conversion or both) moving across the IT boundary?

4 Metric Proposal

ITUE is proposed as a possible solution. ITUE is intended to be a "PUE-type" metric for the IT equipment rather than for the data center. PUE is total energy divided by IT energy, analogously, ITUE is defined as total IT energy divided by computational energy.

$$ITUE = \frac{Total\ Energy\ into\ the\ IT\ Equipment}{Total\ Energy\ into\ the\ Compute\ Components} \quad (2)$$

As PUE identifies the infrastructure burden on the IT equipment, ITUE would identify the same for computing. Data centers have cooling devices, UPS, and PDUs, and other systems supporting the IT equipment. Similarly, IT equipment has internal fans, power supplies, and voltage regulators (VRs), etc.. The compute components can be defined as the CPU, memory, and storage, etc. The math and structure of PUE and ITUE are the same.

Now, these two metrics can be combined.

$$TUE = ITUE \times PUE \quad (3)$$

TUE is the total energy into the data center divided by the total energy to the computational components inside the IT equipment. Figure 1 illustrates the differences between PUE, ITUE, and TUE. Note that in equation 4, "IT" represents the IT equipment or everything inside the server or cluster. In equation 5 however, "IT" represents only the compute components (CPU, memory, fabric) but not cooling, power supplies or voltage regulators. Those (cooling, power supplies, and voltage regulators) are part of "IT" in equation 4. The definition of "a" through "i" in Equations 4 -6 come from Figure 1.

$$PUE = \frac{Cooling + PowerDistribution + IT_{equip}}{IT} = \frac{a+b}{d} \quad (4)$$

$$ITUE = \frac{Cooling + Pwr\ Dist + Misc + IT_{comp}}{IT} = \frac{g}{i} \quad (5)$$

$$TUE = ITUE \times PUE = \frac{a+b}{i} \quad (6)$$

Additionally, the IT equipment list in PUE would necessarily include the network switches, I/O subsystem, and storage. ITUE and TUE should also be extended to cover the full spectrum of the IT equipment in the data center. The graphics and coverage in this paper are compute centric primarily for simplicities sake and not to exclude anything in the IT suite of equipment.

Now that we have defined TUE as a function of the well understood PUE and the new ITUE we can apply it. Recall the comparisons of data centers (a), (b), and (c). The first (a) had fans in both the room and in the IT equipment. The second (b) had fans only in the IT. And the third (c) only had fans in the room, not in the IT equipment. While we cannot yet determine which uses the least energy, it is easy to see that our PUE fan energy accounting problem (where $PUE_b << PUE_a << PUE_c$) can be resolved. The mathematics of TUE do not favor one over the other as all the fans are in the numerator in all three cases. We would expect that all three TUEs to be much closer to each other than their respective PUEs, but more importantly, we can now use TUE to measure all three and to determine which data center and IT combination is actually the most energy efficient.

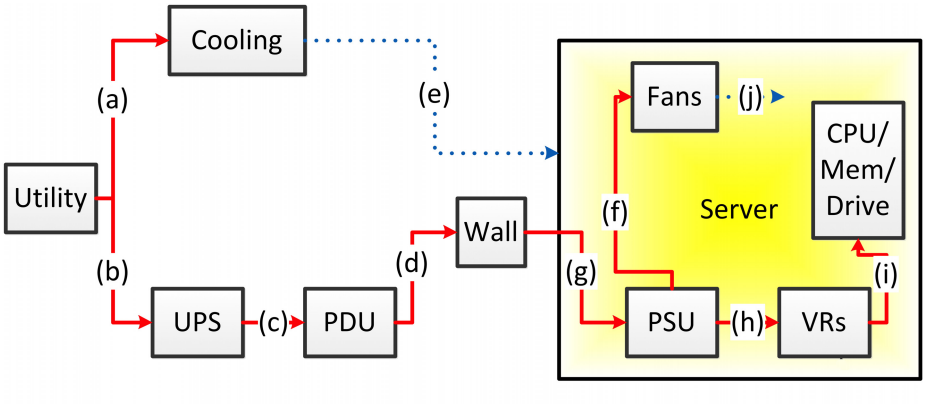


Fig. 1. Schematic of the combined Data Center and IT Equipment

Another possible methodology for developing greater use and understanding of the metric could be the analogous historical development of PUE. [5] describes a good / better / best scheme; the simplest way to get a number is to use the readout of the UPS output power as the IT inlet power. For a data center with no better way to get IT inlet, this is at least "good". "Better" is using the PDU output as IT input, with "best" being direct measure of the IT energy. Similarly, a good / better / best approach to ITUE may help in its eventual adoption. The "good" may be as simple as the energy leaving the PSU minus the fan energy for the denominator, with the PSU "in" (wall socket energy) as the numerator. "Best" would be direct measurement of component level energy consumption.

This good-better-best approach certainly applies to measurement of the value of "i" in Fig 1. While many manufacturers now measure these values and they have become critical to node and system level power management, the energy use at the component level is at best "available with a little work" depending on the suppliers manageability interface. Over time it will become more readily available for two reasons. First, if it is asked for by a growing community looking at ITUE and TUE, the eco-system will respond. This has happened already for the measurements needed for PUE. Second, as we proceed towards hard power limits in the exascale timeframe, the ability of the HPC applications to become energy aware can only happen with this data more fully exposed.

PUE's strict definition is the total ***annual*** energy divided by the IT ***annual*** energy. This is done to ensure any seasonal impacts are included in the number. Measuring PUE during the winter at a data center with extensive free cooling could skew that value significantly. Similarly, TUE and ITUE are defined as ***annual*** values as well. It may be beneficial and informative for an individual site to calculate the min or max values of these to help characterize their system (e.g. winter vs summer PUE), but for all three metrics they should only be reported as annual numbers.

The true value of ITUE is likely in the discussions around more advanced and more integrated infrastructure solutions. Difficulties with the simple concept of PUE come about when the line between infrastructure and IT are not clear. For example, many large

supercomputers come with an integrated cooling system. Some components of these systems would generally be part of a data center room infrastructure in a more standard situation, but in these large specialized systems the standard IT servers, storage, and network are also not as easily split between infrastructure and IT. TUE and ITUE used with PUE, can be useful in being able to compare different data centers.

5 Demonstration of the Metrics

Consider a data center with a PUE of 1.6. For this example assume that the data center infrastructure efficiency is independent of the specifics of the IT and its particular efficiency. Compare this to a similar second data center, each having the same number of servers. The output of each data center will be assumed equal. Data center (a) (PUE = 1.6) has servers with standard or low first-cost components, particularly the fans, power supplies, and voltage regulators. The new data center (b) servers have high efficiency power supplies and fans. The physical infrastructure of data center (b) is identical to (a). From earlier discussions we know that the PUE of data center (b) would be lower, which has been a valid criticism of the PUE metric.

A detailed platform design model [7] shows that data center (a) uses servers with a power draw of 330 W, while data center (b) servers draw only 266 W as shown in Table 1. Recall that PUE_a = 1.6 = Power+Cooling+IT/IT. If we assume 10,000 servers in the space, the IT load is 3.30 MW, and the data center infrastructure uses 1.98 MW. The new data center's PUE (with identical infrastructure) with an IT load of 2.66 MW is PUE=1.74. (The new data center could have some turn-down efficiency, but we assume not for the method's demonstrations). From this perspective, it looks like high efficiency components are a bad idea because the PUE is worse.

Table 1. Server power use by platform and component

	a) Low Eff (W)	*b) High Eff (W)*
Total Platform	330	266
PSU	58	18
VRs	56	38
Fan	18	12
Processor, Memory, Other	198	198

The platforms were analyzed using the model of [7], and fan power, PSU losses, and board level conversion losses were identified. All other loads are considered compute power or "IT"; including the processors, memory, storage, network cards, etc…So for the low efficiency servers in data center (a) we have:

$$ITUE_a = \frac{18 + 58 + 56 + 198}{198} = 1.67 \tag{8}$$

And for the high efficiency servers in (b)

$$ITUE_b = \frac{12 + 18 + 38 + 198}{198} = 1.34 \tag{9}$$

The efficient platform has the lower $ITUE_a$ of 1.34. It carries a 34% "overhead" for power and cooling losses versus 67% for the lower efficiency version ($ITUE_b$=1.67).

From here, a higher level comparison of the two data centers using TUE can be made. Table 2 shows that for total efficiency, data center (b) with the high efficiency IT equipment is more efficient. TUE_b at 2.33 is better than TUE_a at 2.67, even though PUE originally had indicated the opposite. Additionally, with our earlier premise that output from both data centers is the same, the efficiency of the two is related directly to the one with the lower energy, and as expected TUE_b (higher efficiency) correlates with the lower total site power number of 4.64 MW. If the infrastructure can scale with the IT load, the actual PUE can similarly be used to calculate TUE.

Table 2. Power and efficiency numbers of example data centers

	a) Low Eff	*b) High Eff*
Total Platform	3.31 MW	2.67 MW
Infrastructure	1.99 MW	1.99 MW
Total Site Power	5.3 MW	4.66 MW
PUE	1.6	1.74
ITUE	1.67	1.34
TUE	2.67	2.33

6 Case Study Using ITUE

In this section we apply these concepts to the Jaguar system at Oak Ridge National Laboratory.

6.1 The Jaguar Supercomputer

The Jaguar system [8] consists of 200 Cray XT5 cabinets. Each cabinet contains three backplanes, a blower for air cooling, a power supply unit, and twenty-four blades. There are 4,672 compute blades and 128 service blades in Jaguar. A compute blade consists of four compute nodes, each having two six-core 2.6 GHz AMD Opteron processors. Two 4 GB DDR2 memory modules are connected to each processor. A compute blade also has a mezzanine card to support Cray's SeaStar2+ interconnect between nodes. A service blade consists of two nodes, a mezzanine card, and two PCI risers connecting to an external file system.

Jaguar uses both air and liquid to cool the system. Jaguar's liquid-cooling system uses both water and refrigerant R-134a. Cool air is blown vertically through a cabinet from bottom to top by a single axial turbofan. As the heat reaches the top of the cabinet, it boils the refrigerant which absorbs the heat through a change of phase from liquid to gas. The gas is converted back to liquid by the chilled-water heat exchanger inside a pumping unit where the water absorbs the heat and dissipates it externally. There are 48 external liquid-cooling units (denoted as XDPs) used for Jaguar.

6.2 Energy Efficiency Analysis

The site distributes 13.8kV power to the Computer Science Building (CSB) in which Jaguar is located. Transformers at the CSB convert the power to 480 Vac, and switchboards (MSB) feed the power to Jaguar cabinets. The switchboards also provide 480 Vac connections to 48 XDPs. Inside a cabinet, the power supply unit (PSU) converts the 480 Vac power into 52 Vdc and deliver it to the blades. Each blade has an intermediate bus converter (IBC) that converts the 52 Vdc power into 12 Vdc. This power then traverses the blade and reaches the point of load (POL) next to the compute components (such as processors, memory modules, and mezzanine cards). The POL further converts the 12 Vdc power into 1.3 Vdc for the processors, and 1.8 Vdc for the memory.

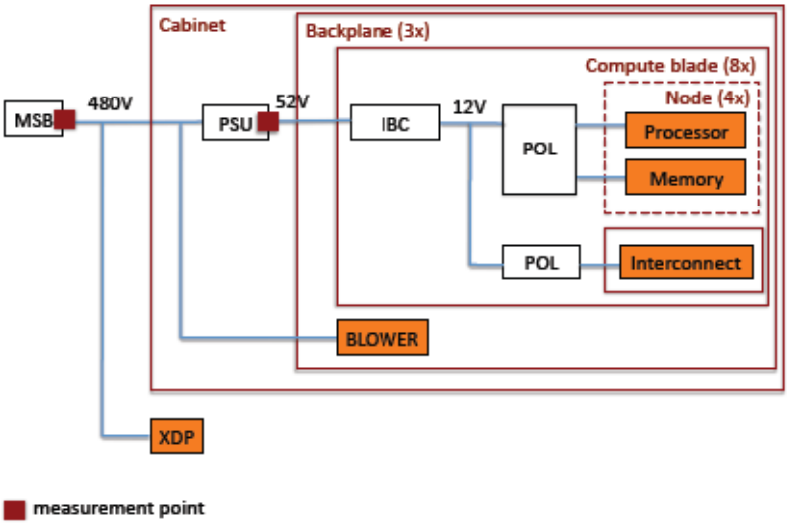


Fig. 2. Power monitoring capabilities for Jaguar

Figure 2 depicts the Jaguar power delivery network inside of a cabinet. Orange boxes represent compute components. Brown boxes indicate where the electrical power can be monitored. For Jaguar, there are two locations where we can monitor the power: One is at the output of the switchboard, and the other is at the output of the power supply unit. Apparently, the power monitoring capabilities of the Cray XT5 are limited. Power can only be monitored at the cabinet level --- not at the blade level. For January 2011, the average aggregate output power from the switchboards and from the cabinet power supply units are 5,259.56 kW and 4,209.90 kW, respectively.

To calculate Jaguar's ITUE for January 2011, the efficiency ratings of IBC and POL are needed. In fact, the best way is to be able to monitor the power draw at the outputs of IBCs and POLs. Unfortunately, Jaguar does not provide this monitoring capability. The next best way is to get the efficiency ratings from vendors. Vendors often have this data but consider them proprietary. As a result, we examine HPC systems from other vendors' public information. This is because the vendors are more

likely to use similar state-of-the-art packaging technologies for their systems. Following the methodologies around the JUGENE supercomputer [9] and the K supercomputer [10], we determine that the IBCs and POLs in Jaguar have the combined efficiency of 84%. The Cray blowers were estimated to carry a 7% penalty as reported in [11].

We calculate the Jaguar ITUE as follows. Since the average power attributed for compute is 4,209.90 kW x 84% = 3,536.32 kW, the metric value can be calculated as 5259.56 / 3536.32 = 1.49. That is, for every kW supplied for computing, there is additional 0.49 kW supplied for cooling and lost in power distribution. Using our estimate of ITUE = 1.49 and the PUE of the CSB as 1.25, the Jaguar TUE = $ITUE_J$ x PUE_{CSB} = 1.86.

7 Challenges and Future Work

Similar issues still exist in ITUE and TUE as do in PUE. These simply need to be understood and dealt with. For example, when more efficient IT equipment is installed in a data center and nothing else is done, PUE will go up (the denominator went down more than the numerator). Similarly if new lower power CPUs or DIMMs were installed in a server, ITUE (and TUE) will go up (again, the denominator went down more than the numerator).

A complicating factor of PUE and ITUE is temperature. The temperature at which the data center, as well as the IT components, operates at can significantly affect both values pushing one up and perhaps the other one down. The ability for the data center operator to pick the right temperatures is advantageous in the pursuit of overall highest efficiency (minimize TUE). Because of this, PUE or ITUE methods should not specify a temperature. However, the temperatures must be consistent. Measuring PUE at a given data center configuration with a certain temperature, then measuring ITUE at a different configuration and IT inlet temperature would render the TUE value invalid. Reporting temperatures during which PUE, ITUE, and TUE were measured would be beneficial in others understanding of the overall thermal management strategy of the data center and IT equipment.

Another issue is defining more precisely what is considered to be a compute load versus support or infrastructure loads. Certainly CPU, memory, memory controller, MIC or GPU processors are all compute. Fans, pumps, PSUs, VRs are all infrastructure. But what of disk drives? Solid state disk drives would seem to be compute, but much of a standard disk drive is spinning the disk. For consistency we suggest all storage be considered compute. Status lights are infrastructure. Baseboard or motherboard controllers are infrastructure. Using the data center level analog, the base-board controller would be the same as the building control system.

Long term, being able to measure ITUE, at least in large scale HPC systems may be a useful capability to build into the equipment, but for now the development of the concept will give us a tool with which to extend the PUE concept to the IT equipment and then to the combined infrastructure and IT installation.

A good estimate of Jaguar's TUE (1.86) and ITUE (1.49) is now published. Jaguar has been decommissioned and replaced by Titan. Work to define these values for that system are ongoing The intention is to continue this line of work, add further refinements and begin to do comparisons with other HPC sites to be able to measure the true efficiency of the site and cluster together.

8 Conclusions

The Energy Efficient High Performance Working Group has proposed two new metrics to improve the tracking and comparison of energy efficiency in data centers. PUE has been as successful as it has because of its simplicity. ITUE has been developed as a direct analog of PUE; PUE for the server. While this value is of interest the true richness comes when multiplied by PUE to get TUE for the data center. This metric surpasses the value of PUE as it now includes the IT support inefficiencies that PUE left out.

ITUE and TUE and their measurement capability will take time to develop (as did PUE), but their use can drive greater efficiency and clearer comparisons in the data center.

Acknowledgements. The work we have done for the power analysis of Jaguar is supported by the Extreme Scale Systems Center at Oak Ridge National Laboratory funded by the United States Department of Defense.

The work was performed at the Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC under Contract No. De-AC05-00OR22725.

Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

The EEHPC WG is an open collaborative group focusing on Energy Efficient High Performance Computing, partially supported the Federal Energy Management Program.

References

[1] Malone, C., Belady, C.: Metrics to Characterize Data Center & IT Equipment Energy Use. In: Proceedings of 2006 Digital Power Forum, Richardson, TX (2006)

[2] Uptime Institute, Uptime Institute 2007 Data Center Industry Survey (2007), http://www.uptimeinstitute.com

[3] The Green Grid, White Paper #6 "Green Grid Data Center Power Effieicny Metrics: PUE and DCiE (December 2008), http://www.thegreengrid.org/~/media/WhitePapers/White_Paper_6_-_PUE_and_DCiE_Eff_Metrics_30_December_2008.pdf?lang=en

[4] The Green Grid, White Paper #49-PUE: A Comprehensive Examination of the Metric (October 2, 2012), http://www.thegreengrid.org/en/Global/Content/white-papers/WP49-PUEAComprehensiveExaminationoftheMetric

[5] EPA, DOE, TGG, ASHRAE, et.al. Recommendations for Measuring and Reporting Overall Data Center Efficiency, (May 17, 2011), EPA Website: http://www.energystar.gov/ia/partners/prod_development/downloads/Data_Center_Metrics_Task_Force_Recommendations_V2.pdf?7438-21e8

[6] Top500 [Internet]. Top 500 Supercomputing Sites (c2000-2012), http://top500.org/project/linpack (cited October 31, 2012)

[7] Intel, Power Joint Engineering Team System Power Calculator (January 2013) (unpublished)

[8] Bland, A.S., Joubert, W., Kendall, R.A., Kothe, D.B., Rogers, J.H., Shipman, G.M.: Jaguar: The world's most powerful computer system – an update. Cray Users Group (May 2010)
[9] Hennecke, M., Frings, W., Homberg, W., Zitz, A., Knobloch, M., Böttiger, H.: Measuring power consumption on IBM Blue Gene/P. In: International Conference on Energy-Aware High Performance Computing (September 2011)
[10] Maeda, H., Kubo, H., Shimamori, H., Tamura, A., Wei, J.: System packaging technologies for the K computer. Fujitsu Scientific and Technical Journal 48(3), 286–294 (2012)
[11] Rogers, J., Hoehn, B., Kelley, D.: Deploying large scale XT systems at ORNL. Cray Users Group (May 2009)

iDataCool: HPC with Hot-Water Cooling and Energy Reuse

Nils Meyer[1], Manfred Ries[2], Stefan Solbrig[1], and Tilo Wettig[1]

[1] Department of Physics, University of Regensburg, 93040 Regensburg, Germany
[2] IBM Deutschland Research & Development GmbH, 71032 Böblingen, Germany

Abstract. iDataCool is an HPC architecture jointly developed by the University of Regensburg and the IBM Research and Development Lab Böblingen. It is based on IBM's iDataPlex platform, whose air-cooling solution was replaced by a custom water-cooling solution that allows for cooling water temperatures of 70°C/158°F. The system is coupled to an adsorption chiller by InvenSor that operates efficiently at these temperatures. Thus a significant portion of the energy spent on HPC can be recovered in the form of chilled water, which can then be used to cool other parts of the computing center. We describe the architecture of iDataCool and present benchmarks of the cooling performance and the energy (reuse) efficiency.

1 Introduction

According to a 2012 IDC study [1], the worldwide costs for power and cooling of IT equipment now exceed 25 billion US-$ per year and are comparable with the costs for new hardware. For this obvious financial reason, but also because of the impact on the environment, energy efficiency has become a very important concern in the IT industry. The problem can be addressed in two ways. First, every effort should be made to reduce the energy consumed by the equipment. Second, some of the energy could be reused. In this paper we will address both of these points, concentrating on the cooling part in "power and cooling". We present an innovative liquid-cooling solution for a high-performance computing (HPC) system that allows for free cooling year-round and energy reuse in the form of chilled-water generation.

A discussion of various aspects of liquid cooling with focus on high coolant temperatures can be found in Refs. [2,3]. For the following discussion, we assume that the cooling medium is water and define what we mean by "warm water" and "hot water". We consider water to be warm if its temperature is higher than the wet-bulb temperature of the ambient air even on hot days so that free cooling is always possible. In typical climates this means about 40°C/104°F. Free cooling year-round drastically reduces the cooling costs since chillers are no longer needed. Even some possibilities for energy reuse exist, e.g., the warm water could drive an underfloor heating system. We consider water to be hot if it opens up more possibilities for energy reuse, e.g., if it is hot enough to drive a radiator-based heating system or an adsorption chiller. This means at least

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 383–394, 2013.

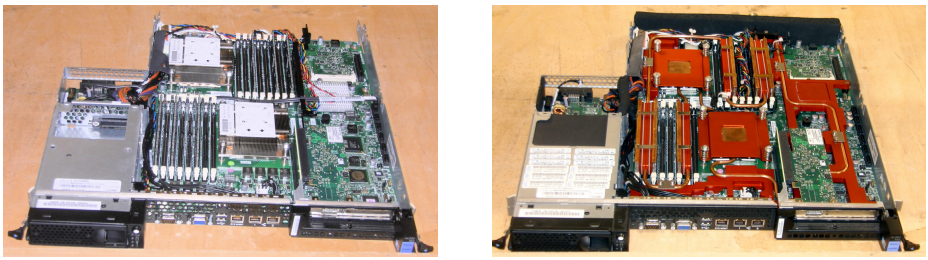


Fig. 1. Left: Original air-cooled iDataPlex dx360 M3 compute node. The power supply unit (not shown) is attached to the compute node on the top left. Right: iDataCool compute node with new water-cooling solution, consisting of a copper pipeline, copper heat sinks for processors and memory, and aluminum heat bridges. Armaflex thermal insulation is used to prevent heat from escaping into the environment.

65°C/149°F. Sustaining such cooling-water temperatures in a large system that is running in stable production mode over long periods of time is a real problem, which we claim to have solved in the project iDataCool described in this paper. The main innovation of iDataCool is the design of a low-cost processor heat sink that minimizes the temperature difference between cooling water and processor and allows for cooling-water temperatures of up to 70°C/158°F.

In the project described in this paper, the infrastructure conditions at the installation site are such that reusing energy for heating purposes is not an option. Therefore the hot water was used to drive an adsorption chiller that generates chilled water. This is another innovation of iDataCool, which is of potential interest for computing centers in hot climates.

Related projects with similar goals (i.e., hot-water cooling) are Aquasar [4] and CoolMUC [5]. Both of these are somewhat smaller in scale and run at somewhat lower temperatures. There are also a number of projects that allow for warm-water cooling as defined above, e.g., [6,7,8,9].

2 iDataCool Architecture

Before presenting our liquid-cooling solution we briefly describe the iDataCool HPC cluster, which is based on the IBM System x iDataPlex architecture [10]. It consists of three racks with 72 compute nodes each. A compute node is equipped with either two four-core Intel Xeon E5630 (44 in total) or two six-core Intel Xeon E5645 (388 in total) Westmere processors organized as a distributed shared memory system. Each node contains 24 GB of memory arranged in six 4 GB DDR3 dual in-line memory modules. The main interconnect network of iDataCool is based on QDR Infiniband, arranged in a hybrid ring/tree topology. Switched Gigabit Ethernet is used for disk I/O, system booting via NFS, and job scheduling. Every compute node is monitored and controlled by a dedicated baseboard management controller (BMC).

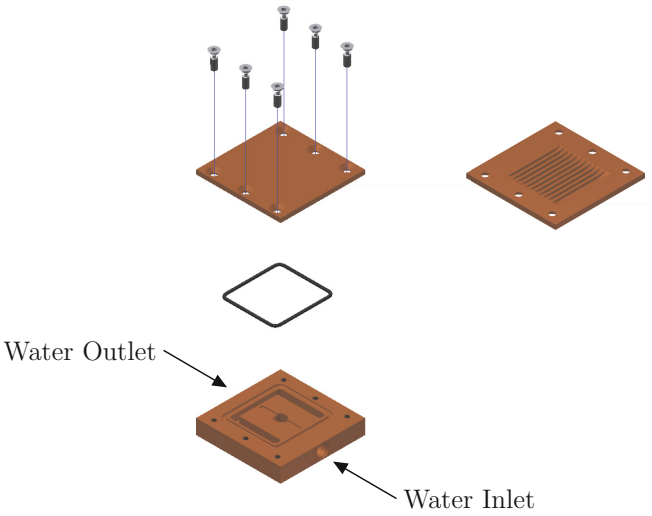


Fig. 2. Design of the iDataCool heat sink. The top part, which is attached to the processor package, is shown from both sides.

The air-cooling components of the original iDataPlex system were completely removed and replaced by a custom water-cooling solution, shown in Fig. 1, which was developed in a joint effort of the University of Regensburg and the IBM Research and Development Lab Böblingen and manufactured and installed in the machine shop of the Regensburg Physics Department. The main design drivers of the water-cooling solution were the possibility of hot-water cooling and low cost. Let us focus on the first point for now. CPUs can tolerate a certain maximum chip temperature, which depends on the specific chip used. Thus, to enable high cooling-water temperatures, the temperature difference between the compute cores and the water should be minimized. The heat transfer path can be divided into two segments. First, the heat is transferred from the cores to the package surface. Second, the heat is transferred from the package surface to the cooling water via thermal interface material and a heat sink. We have no control over the first segment but can optimize the second one, in particular through the design of the processor heat sink, which is shown in Fig. 2. Its design parameters were as follows.

- Minimize the temperature difference between coolant and processor package. This was achieved by bringing the coolant very close to the package and by using a material with high thermal conductivity, i.e., copper. (For the thermal interface material between heat sink and processor package we used Shin-Etsu X23-7783D.)
- Efficient thermal transport. This was achieved by the design shown in Fig. 2, which provides a sufficiently large interface area for heat transfer and creates turbulent flow.

- Low pressure drop. The channels shown in Fig. 2 are not microchannels but 1 mm wide. At a typical flow rate of 0.6 l/min the pressure drop is less than 0.1 bar.
- Low cost. The design shown in Fig. 2 is very simple and can be manufactured inexpensively with standard tools since the channels are rather wide. Using an O-ring and screws is simpler and more leak-proof than using glue.

The processor heat sinks are hard-soldered to a copper pipeline that provides the water flow. Other heat-critical components on the board are thermally coupled to the pipeline via copper or aluminum heat bridges and thermal interface material. These components include memory modules, Infiniband daughter card, chip set, voltage regulators, and several other chips. All of these components can tolerate higher temperatures than the processors. Different materials and designs are used to satisfy the cooling needs of these parts. E.g., the memory modules are cooled via copper heat bridges clamped to aluminum bars which embrace the cooling pipeline. Thus the memory modules can easily be replaced in the field. Proper mounting (including thermal interface material) and alignment of all components of the cooling solution is crucial for a high cooling performance.

Heat dissipation into the environment of the compute node is reduced using Armaflex thermal insulation. The only components of iDataCool that are still air-cooled are the power supply units and the network switches.

Only a minor modification of the node chassis was required to connect the cooling pipeline, via inexpensive standard water connectors, to a rack-level manifold. The nodes are connected to the manifold in a parallel fashion. The manifold is designed using the Tichelmann principle to ensure that the distance covered by the water flow, and therefore the pressure drop, is equal for all nodes. Thus the water flow rates balance themselves automatically. The manifold is attached to the backside of each rack. Armaflex thermal insulation reduces the dissipation of heat from the pipes into the computing center.

An important issue is the cost of the liquid-cooling solution. All components are made from standard materials (copper, aluminum, plastic) and were designed such that the manufacturing process is simple, and thus inexpensive. There are only six soldering joints per node (two at each heat sink, and one each at the in- and outlet), and the bending of the copper pipe can be automated by a properly designed tool. The mounting of the liquid-cooling solution (including application of thermal interface material) was somewhat time-consuming, but on an industrial scale this process could also be automated. For us the total cost of the liquid-cooling solution was about 120 Euro per node (excluding external infrastructure). While this is more expensive than an air-cooled solution, it is a small fraction of the overall cost and can be amortized quickly by the savings from free cooling and energy reuse. On an industrial scale the costs would probably be even lower.

Our sensing and monitoring facilities are described at the beginning of Sect. 4.

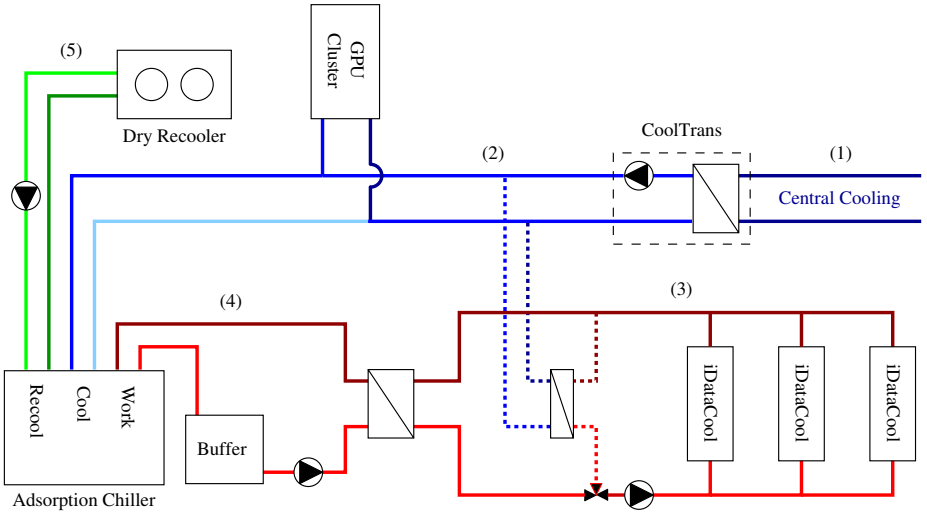


Fig. 3. Liquid-cooling installation consisting of central cooling circuit (1), primary cooling circuit (2), rack cooling circuit (3), driving circuit (4), and recooling circuit (5)

3 Infrastructure

iDataCool is installed in the computing center of Regensburg University, which was entirely air-cooled before. The liquid-cooling infrastructure for iDataCool was prepared in 2011 and completed in 2012. Since the spring of 2012 the waste heat of iDataCool drives an adsorption chiller (LTC 09 by InvenSor), which in turn generates chilled water. The LTC 09 is a so-called low-temperature chiller that works efficiently already at driving temperatures of around 65°C/149°F, see the efficiency curves in the data sheet [11]. The cooling performance of the chiller is balanced against the cooling needs of a small-sized GPU cluster that is cooled by the LTC 09. The GPU cluster has a peak power consumption of 12kW. The equipment of the GPU cluster is housed in a closed cabinet and cooled by air. The air is cooled by an air-water heat exchanger (Knürr CoolLoop [12]) that transfers the heat inside the cabinet to the water circuit.

Fig. 3 shows a schematic overview of the liquid-cooling installation. It consists of five water circuits. Each circuit is driven by a dedicated pump that keeps the water flow at a constant rate. Energy losses to the environment are reduced by thermal insulation of the hot parts of the plumbing. Special additives are used to minimize the risk of corrosion. In the following we discuss the details of the five circuits.

- The computing center is connected to the university's central cooling circuit (1) which delivers chilled water at temperatures around 8°C/46°F.
- The primary cooling circuit (2) is continuously chilled by the adsorption chiller and picks up heat from the GPU cluster. In addition, the primary

circuit can be used as an additional cooler for the iDataCool cluster, see the dotted lines in the figure. (A dry recooler would also suit this purpose.) If the water temperature exceeds 20°C/68°F the primary circuit is supported by the central cooling circuit (1), to which it is connected via a commercial heat exchanger that works autonomously (Knürr CoolTrans [13]).

- The iDataCool cluster is cooled by the rack cooling circuit (3) with hot water at outlet temperatures of up to 70°C/158°F. The waste heat of iDataCool is supplied to the driving circuit of the adsorption chiller (4) via a heat exchanger. Transfer of excess heat to the primary cooling circuit (2) allows us to keep the rack inlet temperature constant (even under change of load on the cluster). The heat transfer to primary and driving circuit is continuously regulated by a 3-way valve. The valve is automatically operated by a PID controller that determines the rack inlet temperature.
- The adsorption chiller is driven by the driving circuit (4). Temperature fluctuations in the driving circuit due to the operational characteristics of the chiller are smoothed by a buffer tank with a capacity of 800 liters. Due to proper thermal insulation there is virtually no temperature loss at the interface to the rack cooling circuit.
- The recooling circuit (5) connects to a fan-driven dry recooler that is located outside the computing center. The fans are controlled automatically by the adsorption chiller with the fan speed optimized for energy-efficient operation of the chiller. Evaporative cooling is possible in principle but has not been implemented in our setup. Freezing of the external recooling circuit is avoided by an admixture of ethylene glycol.[1]

The standard use case of the adsorption chiller is rather different from our setup. Normally the chiller drives an air-conditioning system, i.e., one specifies the desired temperature of the chilled water, and the chiller then absorbs as much heat as necessary from the driving circuit to deliver the required cooling power. In our case we want the chiller to absorb the heat from the rack circuit and to deliver as much cooling power as possible. To see how this works we now discuss in some detail the behavior of our system. The chiller is characterized by its cooling capacity $P_c^{\max}$, which is the maximum amount of heat per unit time it can remove from the cooling circuit, and by its coefficient of performance, defined as

$$\mathrm{COP} = \frac{\text{power } P_c \text{ removed from cooling circuit}}{\text{power } P_d^{\mathrm{abs}} \text{ absorbed from driving circuit}} \,.$$

All of these quantities depend (among other parameters) on the temperature T in the driving circuit [11]. Now assume that the 3-way valve in Fig. 3 completely shuts off the additional cooling path and that we turn on the iDataCool cluster with an initial water temperature of, say, 20°C/68°F. At $T < 55$°C/131°F

[1] Since driving and recooling circuit are connected in the chiller, we also have glycol in the driving circuit. This is the reason for the heat exchanger between rack and driving circuit.

the adsorption chiller is in standby mode and thus absorbs no heat from the cluster. As a result, the temperature in the rack circuit increases until it goes above 55°C/131°F and the chiller turns on.[2] What happens then depends on the temperature dependence of the function $P_d^{\max}(T) = P_c^{\max}(T)/\mathrm{COP}(T)$, which is the maximum power that can be removed from the driving circuit of the chiller. This function depends on the parameters of the chiller. A certain power P_d is transferred from the rack circuit to the driving circuit. If $P_d^{\max}(T) < P_d$ the temperature keeps going up. If $P_d^{\max}(T)$ intersects P_d at some $T = T_{\mathrm{eq}}$, the system settles into equilibrium at that temperature. If P_d is larger than the maximum of $P_d^{\max}(T)$, we have to employ the additional cooling mechanism via the 3-way valve to remove the rest of the heat and to keep the rack circuit at a well-defined temperature. The parameters of our system are such that for $T = 60 \ldots 70$°C/$140 \ldots 158$°F the value of $P_d^{\max}(T)$ is almost equal to, but slightly smaller than, the power transferred from the rack circuit to the driving circuit at maximum load of the cluster. Thus the system is almost in equilibrium and only a very small amount of additional cooling is necessary.

Our setup also solves two redundancy issues: (i) Should the adsorption chiller fail to absorb all the heat from the iDataCool cluster, additional cooling is provided by the primary cooling circuit, which may be supported by the central cooling circuit. (ii) Should the adsorption chiller fail to provide enough cooling power to the GPU cluster, again the central cooling circuit comes to the rescue.

4 Measurements

In this section we present a number of measurements and benchmarks performed on the iDataCool system. We first describe our sensing and monitoring facilities. The liquid-cooling installation is constantly monitored, and relevant system parameters are logged electronically. On the node level, we read out the individual processor core temperatures from chip-internal sensors, we estimate the water in- and outlet temperature of each node using the original air-flow temperature sensors (which we attached to the copper pipe), and we monitor the DC power consumption of each node. On the cluster level, we measure the in- and outlet temperature and the AC power consumption of the 3-rack installation. Our instrumentation also allows us to determine the combined AC power consumption of the iDataCool cluster, the GPU cluster, water pumps, the adsorption chiller, and the dry recooler. To determine the flow rates in the different water circuits we use various kinds of flow meters. As for the accuracy of our equipment, we estimate the node-level temperature sensors to be accurate to about 1°C/2°F, while the cluster-level temperature sensors, which are in direct contact with the water, are specified to have an accuracy of 0.2°C/0.4°F. The ultrasonic flow meter for the rack cooling circuit is specified to have an accuracy of 1%, while the flow meters for the other circuits are much simpler and only about 10% accurate.

[2] In our system the thermal contact between rack circuit and driving circuit is very good so that the driving temperature T equals the outlet temperature of the rack.

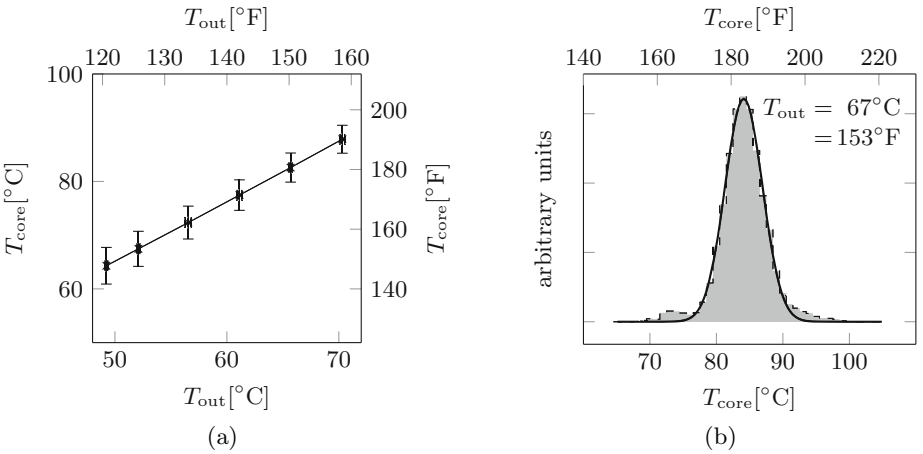


Fig. 4. (a) Core temperatures of 13 compute nodes under `stress` and (b) core temperature distribution of the cluster in production mode. In both plots only six-core E5645 processors are included. The vertical error bars in (a) are the standard deviations after averaging over time and nodes.

When plotting quantities as a function of the cooling-water temperature we have the choice of using the rack inlet or outlet temperature. We chose the outlet temperature T_{out} since this is the quantity of interest for energy reuse purposes. The difference between inlet and outlet temperature can be controlled by adjusting the water flow rate and is about 5°C/9°F in our system.[3] For constant rack inlet temperature, T_{out} fluctuates slightly depending on the load of the cluster and on the control parameters. In the figures, the horizontal error bars in the T_{out} direction reflect these fluctuations in time.

Some of our measurements [those presented in Figs. 4(a), 5(a), and 6(a)] were taken on a subset of 13 randomly selected nodes (six-core E5645 processors at 2.4 GHz with Turbo Boost disabled) running a well-defined load (the standard `stress` tool [14]). The other measurements were taken on the whole iDataCool system running in production mode, i.e., various jobs of different sizes and with different computing and communication requirements are scheduled and executed by the batch queueing system.

In Fig. 4(a) we show the average compute core temperature as a function of the outlet temperature. The average difference between core and water temperature increases slightly, from 15°C/59°F to 17.5°C/63.5°F, over the range of temperatures considered. The error bars are rather large, indicating a large variation between nodes. A histogram of core temperatures for $T_{\mathrm{out}} = 67$°C/153°F is shown in Fig. 4(b). The solid line is a Gaussian fit centered at 84°C/183°F with $\sigma = 2.8$°C/5.0°F. The small bump at the low end of the histogram is due

[3] At constant water flow rate this temperature difference decreases somewhat with the outlet temperature since the system is not perfectly insulated from the environment. At higher temperatures more heat is lost to the air.

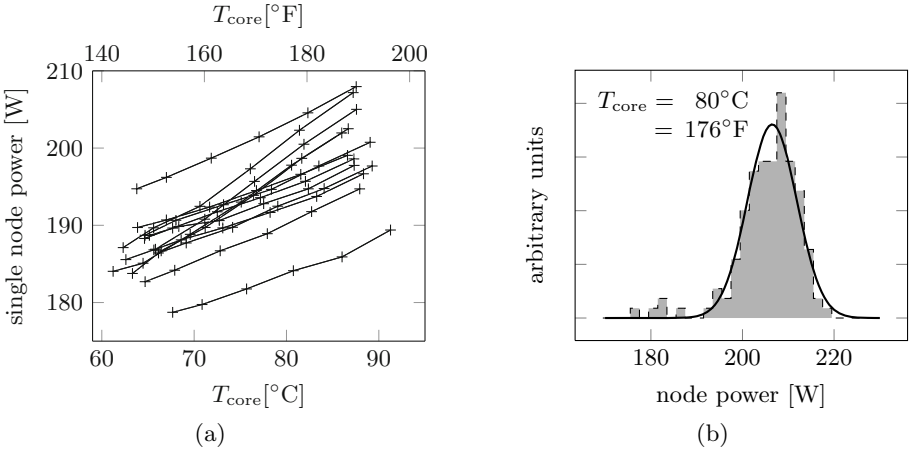


Fig. 5. (a) Node power consumption of 13 compute nodes under `stress` and (b) node power distribution of the cluster in production mode. In both plots only six-core E5645 processors are included.

to idle nodes that have a much lower core temperature. Our interpretation of the large spread visible in Fig. 4(b) is that it is mainly due to the first segment of the heat transfer path described in Sect. 2, over which we have no control, while we can control the second part very carefully. Nevertheless, this spread is a real problem if we aim, with energy reuse in mind, for a high outlet temperature. The cores throttle at about 100°C/212°F,[4] so the outlet temperature is limited by the core with the largest difference between core and outlet temperature. In our system this largest difference is below 30°C/54°F so that we can safely run at $T_{\mathrm{out}} \leq 70$°C/158°F. If we desired higher temperatures we could sort out the "bad" chips and run them at lower temperature in a separate system. The high end of the histogram in Fig. 4(b) indicates that we could perhaps gain another 5°C/9°F in this way.

The power consumption per node also shows large fluctuations. In Fig. 5(a) we present the DC power consumption of 13 nodes vs. their average core temperature. To quantify the spread we measure the DC power on most six-core nodes for various temperatures, interpolate to 80°C/176°F, and then construct a histogram of the interpolated node power, see Fig. 5(b). The solid line is a Gaussian fit centered at 206W with $\sigma = 5.4$W. We see that the individual CPUs vary greatly in their power consumption even for the same coolant temperature. We again attribute most of these variations to the manufacturing process of the chips, not to our liquid-cooling solution.

With higher cooling-water temperatures the power consumption of the nodes increases, which has a negative effect on the total energy reuse efficiency of the system. To quantify this effect we plot in Fig. 6(a) the relative average increase of the node power consumption, which is about 7% when going from

[4] Note that there are other processors that throttle already at much lower temperatures. Such processors are obviously not suitable for cooling with hot water.

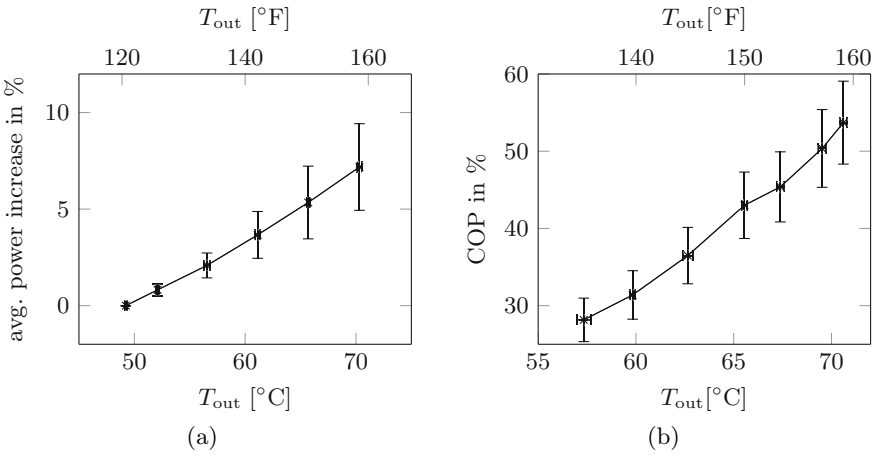


Fig. 6. (a) Relative node power increase for 13 nodes with six-core E5645 processors and (b) COP of adsorption chiller. The vertical error bars in (a) are the standard deviations after averaging over nodes, while the vertical error bars in (b) reflect the 10% accuracy of the flow meters.

49°C/120°F to 70°C/158°F. This should be compared to the efficiency gain of the adsorption chiller, which is quantified by the COP defined in Sect. 3 and shown in Fig. 6(b). The temperature in the last plot starts at 57°C/135°F since the adsorption chiller is in standby mode for lower temperatures. When going from 57°C/135°F to 70°C/158°F the COP increases by 90%, while the node power consumption increases by only 5%. Thus the energy reuse efficiency dramatically improves when running at higher temperatures.

In Fig. 7(a) we show the fraction of the electric power delivered to the cluster that is transferred to the water in the rack circuit. We observe that this fraction drastically decreases with temperature. The reason for this decrease is the imperfect thermal insulation of the iDataCool racks from the environment.[5] A higher temperature difference between rack and air implies that more energy is lost to the air. The lesson from this figure is that in future hot-water cooling designs serious attention should be paid to the thermal insulation of the rack already in the early planning stages. In Fig. 7(b) we show the fraction of electric power that is transferred to the driving circuit of the adsorption chiller, i.e., P_d/P_{electric}, as a function of the coolant temperature in the rack circuit.[6] The increase shows that higher coolant temperatures in the rack circuit lead to a

[5] We did make serious insulation efforts, but since we retrofitted an existing system we were limited in what we could do.

[6] The energy balance in the rack circuit is $P_r = P_d + P_{\text{add}} + P_{\text{loss}}$, where P_r = heat-in-water $\times$ P_{electric}, P_{add} is the additional cooling power from the primary cooling circuit, and P_{loss} is the heat per unit time that is lost to the environment due to imperfect thermal insulation of the plumbing. P_{add} is small at high temperatures, see Sect. 3. The fact that the numbers in Fig. 7(b) are significantly lower than those in Fig. 7(a) thus implies that for our system P_{loss} is rather large.

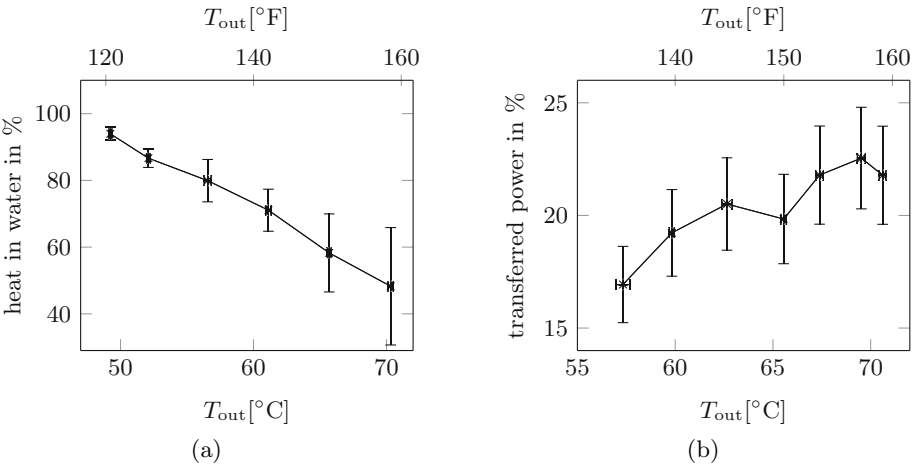


Fig. 7. (a) Heat-in-water fraction and (b) transferred power. The vertical error bars in (a) combine temporal fluctuations of the inlet- and outlet coolant temperatures and the flow, while the vertical error bars in (b) reflect the 10% accuracy of the flow meters.

better utilization of the chiller, i.e., for our system the increase of the chiller effectiveness with T_{out} outweighs the reduced heat in water.

We do not show plots of the fraction of energy reused (or, equivalently, of the energy reuse efficiency) since the cooling capacity of our chiller is not high enough to convert all heat from the iDataCool system to chilled water. The fraction of energy that could be reused (e.g., by adding another chiller) can be computed by multiplying the numbers in Figs. 6(b) and 7(a) and is on the order of 25% for $T = 60 \ldots 70$°C/$140 \ldots 158$°F. With better thermal insulation this fraction could increase by almost a factor of two at $T = 70$°C/158°F, see Fig. 7(a).

5 Conclusions

We have demonstrated that, by employing a sophisticated but low-cost water-cooling solution, it is possible to cool a large compute cluster in stable production mode with water outlet temperatures of up to 70°C/158°F. At such temperatures a significant fraction of the energy consumed by the cluster can be reused. In the iDataCool system the waste heat from the cluster drives an adsorption chiller that operates efficiently above 65°C/149°F. The minor increase in power consumption of the nodes due to the higher temperature is more than offset by the dramatic increase in the COP of the chiller.

The main problem of the iDataCool system is the imperfect thermal insulation, which leads to a serious loss of heat to the environment and decreases the amount of energy that can be reused. In future designs this problem should be attacked from the very start. Our numbers indicate that with better thermal insulation almost 50% of the energy can be recovered in the form of chilled water. Of course, other opportunities for energy reuse exist where an even higher

fraction of the energy can be recovered, e.g., by heating. However, at some sites heating may not be an option or not necessary at all, in which case the generation of chilled water, which can be used to cool other parts of the computing center, is an attractive possibility.

Finally, an important issue is the effect of high water temperatures on the reliability of electronic components, and in general on the long-term stability of the system. iDataCool gives us a unique opportunity to study this issue (except for hard disks since the iDataCool nodes are diskless). We cannot predict the future, but after more than one year of cooling with hot water we have not yet observed any negative effects.

Acknowledgments. We thank J. Marschall (IBM Germany) and S. Heybrock, B. Mendl, A. Schäfer, and M. Wimmer (University of Regensburg) for their contributions to the project, and A. Auweter and H. Huber (LRZ Garching) for helpful discussions. We also thank the machine shop of the Regensburg Physics Department and InvenSor for technical support. The iDataCool project was funded by the German Research Foundation (DFG), the German state of Bavaria, and IBM.

References

1. IDC White Paper #231528, Server Transition Alternatives: A Business Value View Focusing on Operating Costs (2012)
2. Coles, H., Elsworth, M., Martinez, D.J.: "Hot" for Warm Water Cooling. In: SC 2011 State of the Practice Reports. ACM, New York (2011)
3. Meyer, N., et al.: Data centre infrastructure requirements. European Exascale project DEEP (2012)
4. Zimmermann, S., et al.: Aquasar: A hot water cooled data center with direct energy reuse. Energy 43, 237 (2012)
5. Auweter, A., Huber, H.: Direct warm water cooled Linux Cluster Munich: CoolMUC. inSiDE 10, 26 (2012)
6. Baier, H., et al.: QPACE: Power-efficient parallel architecture based on IBM PowerXCell 8i. Computer Science - Research and Development 25, 149 (2010)
7. Eurotech, AURORA Liquid Cooling Solution, `http://www.eurotech.com/aurora`
8. Hughes, P.: Liquid Cooling for Servers, Server Design Summit. Clustered Systems Company (2011)
9. LRZ, SuperMUC Petascale System, `http://www.lrz.de/supermuc`
10. IBM System x iDataPlex dx360 M3, `http://www-03.ibm.com/systems/x/hardware/rack/dx360m3/index.html`
11. InvenSor GmbH Germany, LTC 09 adsorption chiller data sheet, `http://www.invensor.com/en/pdf/Datasheet_Invensor_LTC_09.pdf`
12. Knürr AG Germany, CoolLoop data sheet, `http://www.emersonnetworkpower.com/en-EMEA/Brands/Knurr/Documents/en/brochures/CoolLoop.pdf`
13. Knürr AG Germany, CoolTrans data sheet, `http://www.emersonnetworkpower.com/en-EMEA/Brands/Knurr/Documents/en/brochures/CoolTrans.pdf`
14. `http://weather.ou.edu/~apw/projects/stress`

Pre-execution Data Prefetching with Inter-thread I/O Scheduling

Yue Zhao, Kenji Yoshigoe, and Mengjun Xie

Department of Computer Science, University of Arkansas at Little Rock
2801 S. University Avenue, Little Rock, Arkansas 72204, USA
{yxzhao,kxyoshigoe,mxxie}@ualr.edu

Abstract. With the rate of computing power growing much faster than that of storage I/O access, parallel applications suffer more from I/O latency. I/O prefetching is effective in hiding I/O latency. However, existing I/O prefetching techniques are conservative and their effectiveness is limited. Recently, a more aggressive prefetching approach named pre-execution prefetching [19] has been proposed. In this paper, we first identify the drawback of this pre-execution prefetching approach, and then propose a new method to overcome the drawback by scheduling the I/O operations between the main thread and the prefetching thread. By careful I/O scheduling, our approach further extends the computation and I/O concurrency and avoids the I/O competition within one process. The results of extensive experiments, including experiments on real-life applications such as big matrix manipulation and Hill encryption, demonstrate the benefits of the proposed approach.

1 Introduction

Parallel applications execution suffers from large latency of I/O accesses. The poor I/O performance has been attributed as a critical cause of the low sustained performance of parallel systems ([1], [2]). In order to improve I/O performance numerous works have been conducted. However, their effectiveness and practicability are limited by their inherent drawbacks.

A remarkable advancement in I/O parallelism ([5], [6], [7], [18]) has been achieved. However, this advancement in I/O parallelism is accompanied with a much more expeditious development of parallel processing both on hardware and software, so it is still not capable of reducing the I/O latency effectively. The Adaptable IO System (ADIOS) ([22], [23]) and non-blocking I/O [21] can gain a high I/O performance improvement but they require application modification ([21], [24]). The effectiveness of collective I/O and data sieving ([8], [9]) is application dependent. Due to the inherent nature of applications, there are still many small I/O requests that cannot be eliminated [19]. Studies [3] and [4] use data compression scheme to reduce the amount of I/O traffic. However, limited by the data condensability, compression rate and extra overhead on the system management, the exploitation of data compression approach in practice is restricted. Traditional prefetching strategies ([10], [11], [12], [13], [14], [15],

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 395–407, 2013.

[16], [17]) are conservative and most cannot guarantee the prefetching accuracy and timeliness.

Considering computing power is plenty but data access is the bottleneck and most of existing I/O prefetching techniques are conservative and their effectiveness is limited, Chen et al. [19] proposed a pre-execution I/O prefetching approach. Pre-execution I/O prefetching approach is promising in reducing I/O access latency and it can convert original applications to prefetching version automatically. Following this direction, in order to overcome the limitation due to read after write (RAW) dependency and further extend the computation and I/O concurrency, Zhao et al. [20] proposed a parallel pre-execution prefetching (PPP) approach. However, both [19] and [20] do not pay attention to the relationship among the I/O accesses conducted by diverse threads. And they failed to further extend the I/O and computation concurrency by carefully coordinating the I/O accesses. Our work aims to resolve this issue by developing a new approach named pre-execution prefetching with inter-thread I/O scheduling (PPIS). With PPIS we extend the computation and I/O concurrency while avoiding the I/O competition caused by multiple concurrent I/O operations requested by the main thread and prefetching thread in one process.

The rest of the paper is organized as follows. Section 2 describes the motivation of this work. Section 3 presents PPIS. Section 4 details the experiment designs and results. Section 5 concludes this paper and states our future work.

2 Motivation

In [19], Chen et al. proposed a pre-execution prefetching approach (PP). The basic idea is to pre-execute a portion of code on each process to identify future I/O references, and then fetch the data closer to CPU in advance in order to overlap the computation and I/O access.

PP approach aims to overlap the computation and I/O access by creating a pre-execution prefetching thread (PT) to work with the main thread (MT) in parallel. However, a portion of I/O accesses requested by PT may be overlapped with MT's I/O accesses when MT has I/O operations such as writes and those reads that cannot be conducted by PT early enough. In other words, this portion of pre-executed I/O accesses fails to be hidden by computation. This issue will result in a series of adverse effects. First, it diminishes the degree of the parallelism between computation and I/O, which affects the effectiveness of pre-execution prefetching. Second, the I/O resource competition between the simultaneous I/O accesses of MT and PT can delay MT's I/O access, which goes against the purpose of pre-execution prefetching to accelerate the execution of the original program. Third and most importantly, PP does not take into account the global I/O network and file system source competition. Simply launching more concurrent I/O requests within local process will result in high I/O competition even I/O congestion in the whole system, and end up overwhelming the network and the file system, which not just limits the scalability of the pre-execution prefetching, but makes the prefetcher counter-productive as well.

Figure 1 illustrates an application scenario and shows how it runs under normal execution mode and PP mode, where the size of each operated segment represents time duration. This application scenario is typical in real applications such as big matrix manipulation and big file encryption where the process in normal execution mode sequentially processes a large volume of data. For each piece of data, data reading, computation, and writing are executed in sequence. Under PP execution mode, the process contains two threads, MT and PT. Since I/O access is the focus of PT and in this scenario dominates PT's execution time, we can safely ignore the time incurred by computation conducted by PT in Fig. 1. As under PP mode, PT is designed to do data prefetching as fast as possible, I/O overlap between PT and MT is easy to occur. The scenario in Fig. 1 shows that under PP mode, a high portion of I/O accesses of PT overlaps with that of MT. They are R_2 overlapping with R_1 and R_4 overlapping with W_1. Only the I/O operation R_3 is successfully overlapped with the computation of MT. In this scenario, the computation and I/O access concurrency achieved by PP is much limited. Worse, the I/O overlap between MT and PT results in the I/O resource competition, which makes the I/O access latency (R_1 , R_2 , W_1 and R_4) longer, and then delays the normal execution of MT. When the impact induced by prefetching operations conducted by other processes is taken into account the outcome will be even worse. In case hundreds of processes are employed for a large computing job, which is common for high-performance computing applications, doubled number of concurrent I/O accesses induced by PP can even cause I/O congestion.

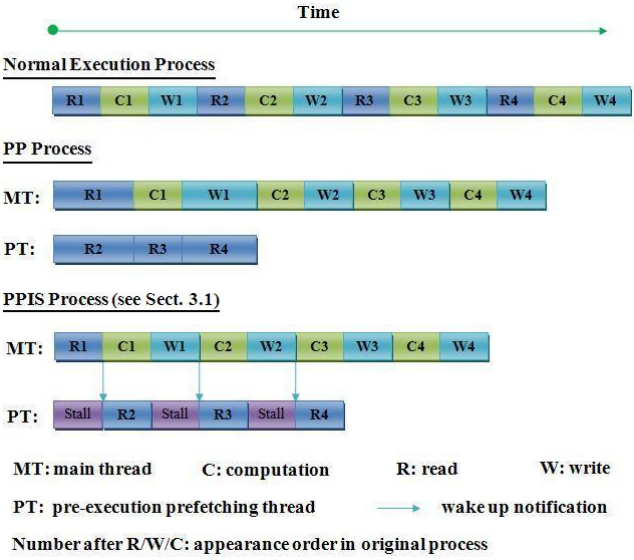


Fig. 1. Hiding I/O Latency with PP and PPIS

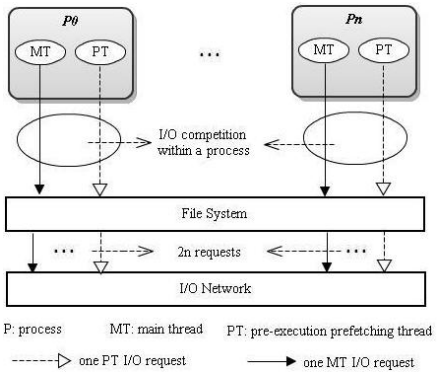


Fig. 2. I/O Workflow of PP

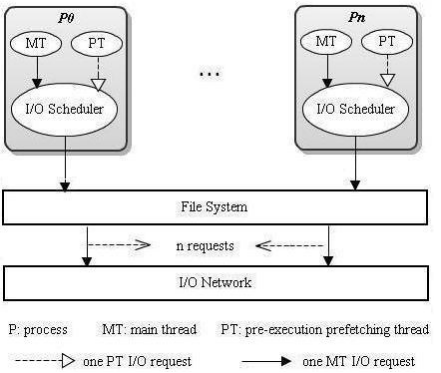


Fig. 3. I/O Workflow of PPIS

3 Pre-execution Prefetching with Inter-thread I/O Scheduling

3.1 Description

We propose a new approach, pre-execution prefetching with inter-thread I/O scheduling (PPIS), to further improve the pre-execution prefetching strategy. The benefit of our approach is twofold. First, we extend the degree of computation and I/O concurrency of a parallel application, and further hide the I/O latency. Second, we improve the scalability of the pre-execution prefetching by avoiding multiple concurrent I/O operations conducted within each process.

In PPIS, we assign a higher priority to MT's I/O accesses to make sure they gain the maximal system I/O resources. Concretely, only when MT is not performing I/O operation can PT launch an I/O operation. In case MT needs to perform I/O operations while at this moment PT is still doing I/O prefetching, we suspend PT's prefetching and record its current prefetching status information, for example the identifier of the block that has just been completely prefetched. After MT finishes its I/O access, MT notifies PT to continue prefetching. By scheduling the I/O accesses of MT and PT in a coordinate manner, PPIS can maximize the parallelism of I/O access and computation, and meanwhile avoid the I/O competition within one process. Figure 2 and Fig. 3 show the I/O workflow of threads when both MT and PT are conducting I/O operation under PP and PPIS scenarios respectively.

The advantages of PPIS over PP are illustrated in Fig. 1, which compares how PPIS and PP progress in the scenario mentioned in Sect. 2. In this ideal case, PPIS maximizes PT's I/O access and MT's computation concurrency. Moreover, it avoids the I/O competition between the two threads, which not only optimizes the completion time of MT's I/O access, but also avoids the potential I/O congestion caused by PP when the number of concurrent processes is large.

3.2 Implementation

Software Stack. We employ MPI protocol and its parallel API to actualize the execution of parallel applications. In order to implement PT co-working with MT within each process, we adopt the POSIX Threads (Pthreads) multi-threaded programming standard. We conduct the parallel file system access through the ROMIO MPI-IO implementation in Open MPI. Figure 4 shows the software stack to implement our approach.

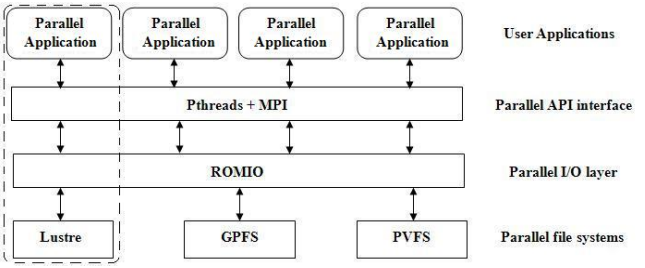


Fig. 4. Software Stack (Module inside the dashed line box represents the experiment environment used in Sect. 4)

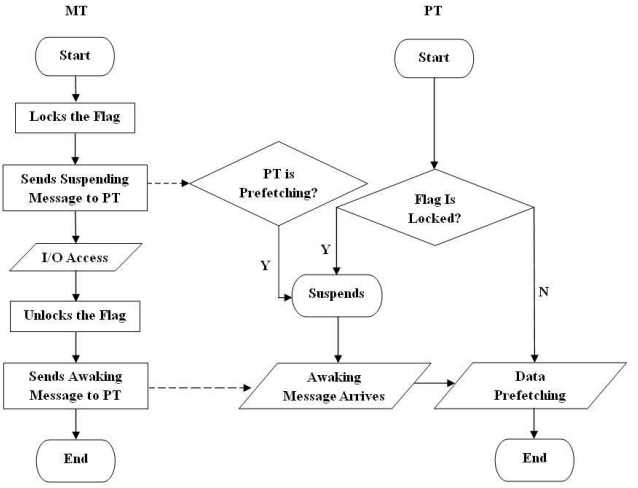


Fig. 5. Threads I/O Related Behavior of PPIS

Threads I/O Related Behavior to Implement I/O Scheduling. We employ a condition variable and Pthreads inter-thread message passing mechanism to accomplish the inter-thread I/O scheduling. The condition variable is used as a flag managed by MT. Initially the flag is set as unlocked. When MT starts

performing I/O access it first locks the flag. Locked flag indicates that PT cannot launch any I/O accesses. Otherwise, the prefetching is allowed. When PT encounters a read function, it has to check the flag's status first. If the flag is locked, then PT goes into the suspend status. When MT finishes its I/O access it unlocks the flag and sends a notification to wake up and allow PT to prefetch data into cache. Figure 5 shows the I/O related behavior of each thread in PPIS.

3.3 Analysis

Optimal Analysis. PP introduces extra I/O accesses over normal execution. In the worst scenario, all MT and PT I/O accesses collide and result in I/O congestion. Then the original application execution can be delayed infinitely. By introducing I/O scheduling PPIS not only avoids this issue but also further extends the degree of I/O access and computation concurrency of a parallel program. In this section we analyze the optimal speedup achieved by PPIS over normal execution.

Table 1. Notations

N_C	the number of segments of computation in the original application
N_W	the number of segments of write operation in the original application
N_R	the number of segments of read operation in the original application
T_{Normal}	the total execution time of the application under normal execution
T_{PPIS}	the total execution time of the application under PPIS mode
$T_{TH\_OP\_SN\_MO}$	The execution time of a certain operation (TH: thread; OP: operation; SN: serial number; Mode: execution mode (e.g., $T_{MT\_R\_i\_Normal}$ refers to the execution time of MT's ith segment of read under normal execution mode))
$N_{TH\_OP\_MO}$	The number of segments of a certain operation (e.g., $N_{PT\_R\_PPIS}$ refers to the number of segments of read conducted by PT under PPIS mode)

The total execution time of the program under a certain mode is actually MT's execution time, so

$$T_{Mode} = \sum_{i=1}^{N_{MT\_R\_Mode}} T_{MT\_R\_i\_Mode} + \sum_{j=1}^{N_{MT\_W\_Mode}} T_{MT\_W\_j\_Mode} + \sum_{k=1}^{N_{MT\_C\_Mode}} T_{MT\_C\_k\_Mode}. \quad (1)$$

Here, $Mode \in \{Normal, PP, PPIS\}$. Under normal execution mode MT conducts all the operations exactly identical to how the original application does. And under PPIS all write and computation operations are conducted by MT with the same progress time as those under normal execution mode assuming the I/O

accesses of other processes in the system are not disturbed by the prefetching. Thus,

$$T_{Normal} = \sum_{i=1}^{N_R} T_{MT\_R\_i\_Normal} + \sum_{j=1}^{N_W} T_{MT\_W\_j\_Normal} + \sum_{k=1}^{N_C} T_{MT\_C\_k\_Normal}. \quad (2)$$

$$T_{PPIS} = \sum_{i=1}^{N_{MT\_R\_PPIS}} T_{MT\_R\_i\_PPIS} + \sum_{j=1}^{N_W} T_{MT\_W\_j\_Normal} + \sum_{k=1}^{N_C} T_{MT\_C\_k\_Normal}. \quad (3)$$

In the optimal case, all the data to be read in the application is prefetched by PT with completely overlapping with the computation of MT, then,

$$T_{PPIS} = \sum_{j=1}^{N_W} T_{MT\_W\_j\_Normal} + \sum_{k=1}^{N_C} T_{MT\_C\_k\_Normal}. \quad (4)$$

The speedup achieved by PPIS over the normal execution mode is:

$$\begin{aligned} &Speedup_{(PPIS/Normal)} \\ &= T_{Normal}/T_{PPIS} \\ &= \frac{\sum_{i=1}^{N_R} T_{MT\_R\_i\_Normal} + \sum_{j=1}^{N_W} T_{MT\_W\_j\_Normal} + \sum_{k=1}^{N_C} T_{MT\_C\_k\_Normal}}{\sum_{j=1}^{N_W} T_{MT\_W\_j\_Normal} + \sum_{k=1}^{N_C} T_{MT\_C\_k\_Normal}} \\ &<= \frac{\sum_{i=1}^{N_R} T_{MT\_R\_i\_Normal} + \sum_{k=1}^{N_C} T_{MT\_C\_k\_Normal}}{\sum_{k=1}^{N_C} T_{MT\_C\_k\_Normal}}. \end{aligned} \quad (5)$$

To hide all the read latency by computation, there must be

$$\sum_{i=1}^{N_R} T_{MT\_R\_i\_Normal} < \sum_{k=1}^{N_C} T_{MT\_C\_k\_Normal}. \quad (6)$$

With

$$\lim \sum_{k=1}^{N_C} T_{MT\_C\_k\_Normal} = \sum_{i=1}^{N_R} T_{MT\_R\_i\_Normal}. \quad (7)$$

So,

$$Speedup_{PPIS/Normal} <= 2 \sum_{k=1}^{N_C} T_{MT\_C\_k\_Normal} / \sum_{k=1}^{N_C} T_{MT\_C\_k\_Normal} = 2. \quad (8)$$

Namely,

$$\max\{Speedup_{(PPIS/Normal)}\} = 2. \quad (9)$$

So, in the optimal scenario, PPIS can hide all the read latency suffered by an application and achieve 50% total execution time reduction of that application over normal execution.

Cost. Over the existing pre-execution prefetching approach, in which I/O related operations conducted by PT do not involve communication with other processes in general [19], PPIS requires an inter-thread I/O scheduling, which is quite light-weight in cost. Throughout the implementation of the I/O scheduling, only a condition variable and several inter-thread messages are added to the existing pre-execution prefetching implementation. Also, the messages are thread control messages with no additional data transported, which are quite small in size. Thus, the overhead caused by I/O scheduling is negligible, especially, when it is compared to the huge workload of parallel applications. The cost does not impact the effectiveness of PPIS, which is also verified by the performance improvement achieved by PPIS as shown in Sect. 4.

Correctness. First, the existing pre-execution prefetching approach can guarantee the correctness of the original program. Second, the I/O scheduling between PT and MT only deals with the prefetching time of PT, so it does not affect the logical behavior and accuracy of MT. Thus, the MT in a program running with and without PPIS will logically behave identically. In summary, PPIS does not affect the correctness of the original program.

Thread Safety. First, only one global variable is added on top of PP to implement PPIS. Since only MT has the access to write it, there is no concurrent read/write by multiple threads on this global variable. Thus, no additional thread safety risk is induced to the existing pre-execution prefetching. Second, by introducing a prefetching file pointer as a hidden file offset pointer within the non-transparent MPI file handle object in order to track the prefetching thread file offset, the thread safety can be guaranteed naturally by PP [19]. Therefore, PPIS is thread-safe.

4 Experiment

Our experiments were conducted on a 66-node 528 processors Linux-based cluster. Each compute node has 16 GB of RAM and 2 CPU sockets, each with quad-core Intel Xeon 2.66GHz CPU. Depending on the number of processes in experiment, we used the subset of this cluster with size ranging from 1 to 16 compute nodes. We dynamically assign the buffer size as demand in each node, which can be large enough for our experiments as each node has 16 GB of RAM. Software environment refers to the dashed line box in Fig. 4.

4.1 Design

We evaluated the benefits of our approach on big matrix searching operation and Hill encryption application respectively. The former was tested under the system with light I/O workload. The later was tested under the system with light and heavy I/O workload, respectively. For the light I/O workload, only our experiment benchmark accessed the disk in the system. To achieve a heavy workload

environment, we conducted multiple simultaneous large file I/O operations in the system. We evaluated the results with three metrics, total execution time, aggregate sustained bandwidth and I/O latency, which are the most important performance metrics in practice.

Experiment #1: Big Matrix Searching. Big matrix searching is the fundamental operation of many real parallel applications. In this experiment we conducted searches in a big integer matrix, which is 4GB in size, to find its top 30 maximum items. The matrix was split into 4 sub-matrices with equal size to process in sequence.

Experiment #2: Hill Encryption. Hill encryption is a real application to encrypt data with Hill cipher, in which the key is a matrix. When the plaintext data is large it will be partitioned into smaller chunks, and then these chunks are encrypted in sequence. In this experiment we encrypted a big file of 6GB with the key matrix size was set as 100 by 100. First, we tested the case in which the chunk size is 2GB. Then we further tested the I/O latency reduction achieved by PPIS over normal execution mode under different chunk sizes.

4.2 Results

Experiment #1. Figure 6 shows the total execution time results. The execution time under the PPIS mode is reduced in all the cases showed in Fig.6 compared to normal execution and PP, respectively. Over normal execution and PP the maximal reduction is 28.6% and 28.2% when the number of processes is 16 and 64, respectively. As a reference, Fig. 6 also shows the application's execution time under the theoretically optimal scenario, in which the I/O latency would be completely masked. The computation dominates the whole application when the number of processes is small (e.g., 1 and 2). Thus, even if a large amount of I/O latency was hidden by PPIS the reduction percentage of the whole application execution time is quite marginal. For large applications which run

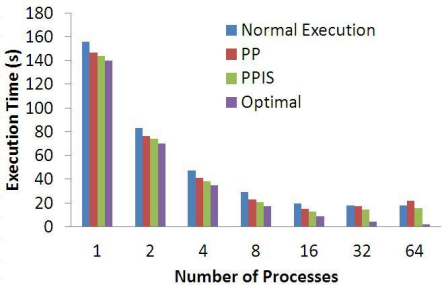


Fig. 6. Execution Time

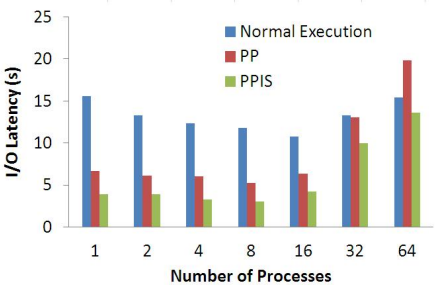


Fig. 7. I/O Latency

tens of days or even months, the reduction of the execution time is significant as the amount of I/O time being hidden would constitute tens of hours of the applications. When the number of the processes is large the computing workload assigned to each process is quite low, which limits the amount of I/O latency hidden by computation. Thus, high execution time reduction percentage under PPIS mode can be achieved with moderate number of parallelisms as observed in [20]. In Fig. 6, when the number of processes exceeds 16 and 32 respectively, the execution time under the PP mode starts to increase. With the number of processes is 64, it is even larger than that of the normal execution. PPIS, on the other hand, can achieve execution time reduction compared to the other two modes in all the cases.

Figure 7 shows the corresponding results of I/O latency during the whole execution of the application. In most of cases, a considerable I/O latency reduction percentage has been achieved by PPIS over the other two modes with the maximal reduction being 75.1% over normal execution mode when the number of processes is 1 and 54.3% over PP mode when the number of processes is 4. Most importantly, the PPIS outperforms the normal and PP modes for all process sizes being evaluated while the normal and PP modes outperform/underperform over one another at certain process sizes being evaluated.

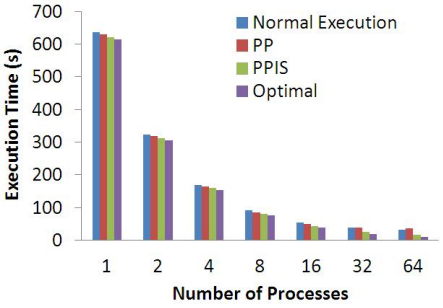


Fig. 8. Execution Time (light I/O)

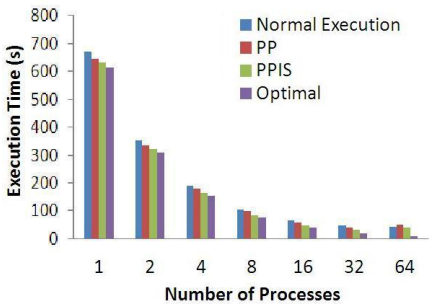


Fig. 9. Execution Time (heavy I/O)

Experiment #2. Figure 8 and Fig. 9 show the execution time of the Hill encryption with the chunk size is 2GB. Likewise, PPIS achieves execution time reduction in all cases shown in the figures compared to normal execution and PP, respectively. Under the light workload system, the best execution time reduction achieved by PPIS is 47.4% and 54.6% over normal execution and PP with the number of processes is 32 and 64, respectively. Under the heavy workload system, the corresponding results turn out to be 27.6% and 24.0% with the number of processes is 16 and 64, respectively. Besides, better scalability was achieved by PPIS compared to PP in both circumstances.

Figure 10 and Fig. 11 show the I/O latency during the whole encryption process. Quite high I/O latency reduction percentage acquired by PPIS is observed with the maximal reduction being 67.0% over normal execution and 62.9% over PP under the number of processes being 1 and 32, respectively.

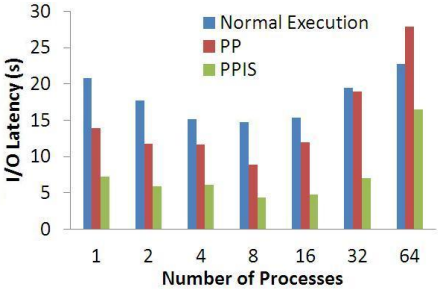


Fig. 10. I/O Latency (light I/O)

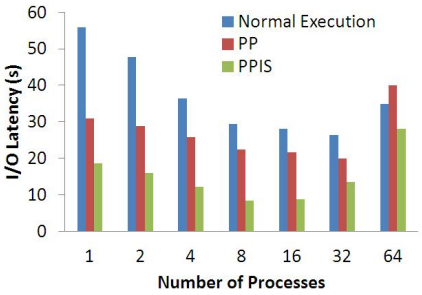


Fig. 11. I/O Latency (heavy I/O)

Figure 12 shows the I/O latency reduction achieved by PPIS over normal execution mode under light I/O workload as the chunk size changes. In some cases, the I/O latency reduction is close to 100%. It demonstrates that in these cases PPIS can almost hide the entire I/O latency of the Hill encryption application by scheduling the pre-executed I/O operation to strictly overlap with computation. When the number of process is larger than or equal to 32, the reduction drops. The reason is that the computation workload assigned to each process is too small to hide all the I/O latency.

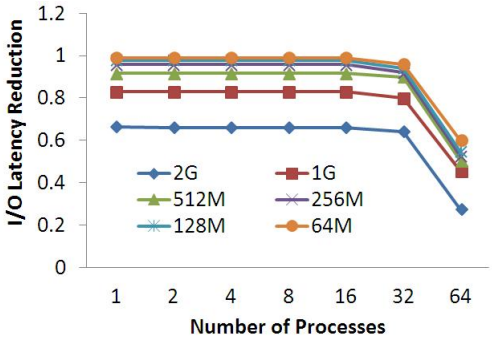


Fig. 12. I/O Latency Reduction

In experiments #1 and #2 when the number of processes is larger such as 128 or 256 the advantage of PPIS in terms of scalability is more remarkable. For instance, the execution time reduction achieved by PPIS compared to PP for 256 processes under the heavy I/O workload system in Experiment #2 is 42.1%. It is larger than those under the number of processes being 128 and 64, which are 31% and 24.0%, respectively. The effectiveness of PPIS under heavy workload system also indicates that PPIS benefits the application, which is running simultaneously with other multiple applications in the system as well.

5 Conclusion and Future Work

Parallel applications continue to suffer from I/O latency. In this study, by proposing PPIS approach, we enhanced the existing pre-execution prefetching strategy to further hide I/O latency. Meanwhile, the new pre-execution prefetching approach is more scalable. The main contribution of this study is that we employed the active scheduling or careful coordination on the normal and pre-executed I/O accesses to maximum the overlap between the pre-executed I/O accesses and computation, which is the first work to the best of our knowledge in this research direction. Compared to the existing pre-execution prefetching approach PPIS extends the degree of computation and I/O concurrency, and also avoids the I/O congestion caused by multiple I/O operations requested by one process synchronously. The extensive evaluation results, including one from Hill encryption as a real-life application, have verified that the proposed approach has more potential and better scalability to hide I/O access delay than the existing approach. In order to further decrease or avoid impact on all processes in system introduced by prefetching our future work is to schedule the normal and pre-executed I/O accesses in inter-process level.

Acknowledgement. This work is supported in part by the National Science Foundation under Grant CRI CNS-0855248, Grant EPS-0701890, Grant EPS-0918970, and Grant MRI CNS-0619069.

References

1. Chen, Y., Sun, X.H., Thakur, R., Roth, P.C., Gropp, W.: LACIO: a new collective I/O strategy for parallel I/O systems. In: Proc. of IEEE IPDPS 2011, pp. 794–804 (2011)
2. Sun, X.-H., Chen, Y., Wu, M.: Scalability of heterogeneous computing. In: Proc. of 34th International Conference on Parallel Processing (2005)
3. Makatos, T., Klonatos, Y., Marazakis, M., Flouris, M.D., Bilas, A.: Using transparent compression to improve ssd-based i/o caches. In: Proc. of the ACM EuroSys 2010, pp. 1–14 (2010)
4. Welton, B., Kimpe, D., Cope, J., Patrick, C., Iskra, K., Ross, R.: Improving I/O forwarding throughput with data compression. In: Proc. of IEEE CLUSTER 2011, pp. 438–445 (2011)
5. `http://www.oracle.com/us/products/serversstorage/storage/storage-software/031855.htm` (accessed: March 09, 2010)
6. Ligon, W., Ross, R.: Parallel I/O and the parallel virtual file system. In: Beowulf Cluster Computing with Linux, Cambridge, MA, pp. 493–534 (2003)
7. Schmuck, F., Haskin, R.: GPFS: A shared-disk file system for large computing clusters. In: Proc. of the 1st USENIX Conference on File and Storage Technologies (2002)
8. Reed, D.: Scalable Input/Output: achieving system balance. The MIT Press (2003)
9. Thakur, R., Gropp, W., Lusk, E.: Data sieving and collective I/O in ROMIO. In: Proc. of the 7th Symposium on the Frontiers of Massively Parallel Computation (1999)

10. Ding, X., Jiang, S., Chen, F., Davis, K., Zhang, X.: DiskSeen: exploiting disk layout and access history to enhance I/O prefetch. In: Proc. of USENIX Annual Technical Conference (2007)
11. Kotz, D.F., Ellis, C.S.: Prefetching in file systems for MIMD multiprocessors. IEEE Transactions on Parallel and Distributed Systems 1(2) (1990)
12. May, J.: Parallel I/O for high performance computing. Morgan Kaufmann Publishing (2001)
13. Papathanasiou, A., Scott, M.: Aggressive Prefetching: an idea whose time has come. In: Proc. of the 10th Workshop on Hot Topics in Operating Systems (2005)
14. Patterson, R.H.: Informed prefetching and caching, Carnegie Mellon Ph.D. Dissertation, CMU-CS-97-204 (1997)
15. Son, S.W., Kandemir, M., Karakoy, M., Chakrabarti, D.: A compiler-directed data prefetching scheme for chip multiprocessors. In: Proc. of the 14th Symposium on Principles and Practice of Parallel Programming, pp. 209–218 (2009)
16. Ravichandran, N., Paris, J.F.: Making early predictions of file accesses. In: Proc. of the 4th Int. Inf. Telecommun. Technol., pp. 122–129 (2005)
17. Brown, A.D., Mowry, T.C., Krieger, O.: Compiler-based I/O prefetching for out of- core applications. ACM Transactions on Computer Systems 19(2) (2001)
18. Chen, Y., Byna, S., Sun, X.-H., Thakur, R., Gropp, W.: Exploring parallel I/O concurrency with speculative prefetching. In: Proc. of the ICPP (2008)
19. Chen, Y., Byna, S., Sun, X.H., Thakur, R., Gropp, W.: Hiding I/O latency with pre-execution prefetching for parallel applications. In: Proc. of SC (2008)
20. Zhao, Y., Yoshigoe, K.: Hiding I/O latency with parallel pre-execution prefetching. In: Proc. of the 24th IASTED International Conference on Parallel and Distributed Computing and Systems, PDCS 2012 (2012)
21. Buettner, D., Kunkel, J., Ludwig, T.: Using non-blocking I/O operations in high performance computing to reduce execution times. In: Ropo, M., Westerholm, J., Dongarra, J. (eds.) PVM/MPI. LNCS, vol. 5759, pp. 134–142. Springer, Heidelberg (2009)
22. Lofstead, J.F., Klasky, S., Schwan, K., Podhorszki, N., Jin, C.: Flexible io and integration for scientific codes through the adaptable io system (adios). In: Proc. of the 6th International Workshop on Challenges of Large Applications in Distributed Environments, pp. 15–24 (2008)
23. Jin, C., Klasky, S., Hodson, S., Yu, W., Lofstead, J., Abbasi, H., Schwan, K., Wolf, M., Liao, W., Choudhary, A., Parashar, M., Docan, C., Oldfield, R.: Adaptive io system (adios). In: Cray User's Group (2008)
24. Bent, J., Gibson, G., Grider, G., McClelland, B., Nowoczynski, P., Nunez, J., Polte, M., Wingate, M.: Plfs: A checkpoint filesystem for parallel applications. In: Proc. of SC 2009 (2009)

A Semantics-Aware I/O Interface for High Performance Computing

Michael Kuhn

University of Hamburg
michael.kuhn@informatik.uni-hamburg.de

Abstract. File systems as well as I/O libraries offer interfaces which can be used to interact with them, albeit on different levels of abstraction. While an interface's syntax simply describes the available operations, its semantics determine how these operations behave and which assumptions developers can make about them. There are several different interface standards in existence, some of them dating back decades and having been designed for local file systems. Examples are the POSIX standard for file system interfaces and the MPI-I/O standard for MPI-based I/O.

Most file systems implement a POSIX-compliant interface to improve portability. While the syntactical part of the interface is usually not modified in any way, the semantics are often relaxed to reach maximum performance. However, this can lead to subtly different behavior on different file systems, which in turn can cause application misbehavior that is hard to track down.

On the other hand, providing only fixed semantics also makes it very hard to achieve optimal performance for different use cases. An additional problem is the fact that the underlying file system does not have any information about the semantics offered in higher levels of the I/O stack. While currently available interfaces do not allow application developers to influence the I/O semantics, applications could benefit greatly from the possibility of being able to adapt the I/O semantics at runtime.

The work we present in this paper includes the design of our semantics-aware I/O interface and a prototypical file system developed to support the interface's features. Using the proposed I/O interface, application developers can specify their applications' I/O behavior by providing semantical information. The general goal is an interface where developers can specify *what* operations should do and *how* they should behave – leaving the actual realization and possible optimizations to the underlying file system. Due to the unique requirements of the proposed I/O interface, the file system prototype is designed from scratch. However, it uses suitable existing technologies to keep the implementation overhead low.

The new I/O interface and file system prototype are evaluated using parallel metadata benchmarks. Using a single metadata server, they deliver a sustained performance of up to 50,000 lookup and 20,000 create operations per second, which is comparable to – and in some cases, better than – other well-established parallel distributed file systems.

Keywords: Distributed File Systems, I/O Interfaces, I/O Semantics.

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 408–421, 2013.

1 Introduction

High performance computing is an increasingly important tool for scientific computing. It is used to conduct large-scale computations and simulations of complex systems from basically all branches of the natural and technical sciences, such as meteorology, climatology, particle physics, biology, medicine and computational fluid dynamics. These computations and simulations are usually realized in the form of parallel applications. They use threads, message passing or a combination of both to distribute and speed up the computational work across a supercomputer. Additionally, high performance computing is invaluable in analyzing the large amounts of data produced by such applications.

An important aspect is high performance I/O, because storing and retrieving such large amounts of data can greatly affect the overall performance of these applications. A common access pattern produced by these applications involves many parallel processes, each performing non-overlapping access to a shared file.

File systems provide an abstraction layer between the applications and the actual storage hardware, such that application developers do not have to worry about the organizational layout or technology of the underlying storage hardware. Distributed file systems usually stripe data across several storage devices to improve both storage capacity as well as throughput. Parallel file systems allow multiple clients to access the same data simultaneously. Consequently, most file systems used in high performance computing are parallel distributed file systems. Two of the most widely-used file systems today are Lustre [3] and GPFS [17].

Parallel distributed file systems provide one or more I/O interfaces which can be used to access data within the file system. Additional interfaces are available in the form of libraries. Popular choices include POSIX, MPI-I/O, NetCDF and HDF5. Almost all the I/O interfaces found in high performance computing today offer simple byte- or element-oriented access to data and thus do not have any a priori information about what kind of accesses the applications perform. Even though there are some notable exceptions such as ADIOS or NetCDF, even the more advanced I/O interfaces do not offer support to specify additional semantical information about the applications' behavior and requirements. Due to this lack of knowledge about application behavior, optimizations are often based on heuristic assumptions which may or may not reflect the actual behavior.

While the I/O interface defines which I/O operations are available, the I/O semantics describe and define the behavior of these operations. Usually each I/O interface is accompanied by a set of I/O semantics, tailored to this specific interface. The POSIX I/O semantics are probably the most widely-used semantics, even in high performance computing. However, due to being designed for traditional local file systems, they impose unnecessary restrictions on today's parallel distributed file systems. One of these restrictions are the very strict consistency requirements which can lead to performance bottlenecks in distributed environments.

Performing I/O efficiently is becoming an increasingly important problem. While CPU speed and HDD capacity continue to increase by roughly a factor

of 1,000 every 10 years [26,25], the speed of HDDs grows much slower: Early HDDs in 1989 delivered about 0,5 MB/s, while current HDDs manage around 150 MB/s [24]. This corresponds to a 300-fold increase of throughput over the last (almost) 25 years. Even newer technologies such as SSDs only offer throughputs of around 600 MB/s, resulting in a total speedup of 1,200. For comparison, over the same period of time, the computational power increased by a factor of 1,000,000.

There are several ways to compensate for this fact: Increasing the efficiency of I/O, using novel storage technologies or simply buying more storage hardware. The JULEA project aims to increase the efficiency of I/O by providing a new semantics-aware I/O interface which should allow applications to make the most of the available storage hardware. It allows specifying the semantics of I/O operations at runtime and supports batch operations to increase performance. The overall goal is to allow the application developer to specify the desired behavior and leave the actual realization to the I/O system.

This paper is structured as follows: The current state of the art with regards to I/O interfaces and semantics is presented in Section 2. In Section 3, the design of our new semantics-aware I/O interface is elaborated. A preliminary evaluation is given in Section 4. Our design is then compared with other related work in Section 5, followed by a conclusion and some ideas for future work in Section 6.

2 State of the Art

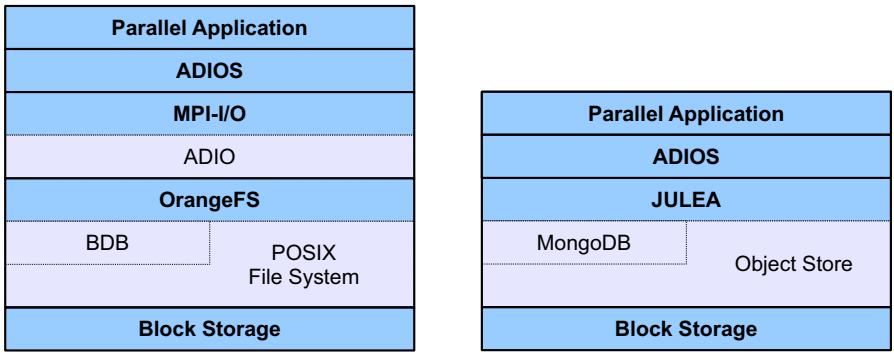
Currently, I/O systems have a strongly layered concept. One major problem with this approach is the fact that the lower layers do not have any information about the upper ones. Due to this, each layer has to perform its own optimizations to be able to use the I/O system's full potential. An example of such an I/O stack can be seen in Figure 1a. The different interfaces such as ADIOS, MPI-I/O and POSIX will be explained below. While the upper layers usually provide more comfort and abstraction, the performance yield might be lower. Therefore, the lower, more difficult-to-use layers are often used directly to harness the I/O system's full potential.

An additional problem is the fact that it is currently not possible to hand semantical information down through the I/O stack. To ease the development of codes in need of high performance I/O it would be very beneficial to provide easy-to-use interfaces that still provide adequate performance.

2.1 I/O Interfaces

Each I/O interface is usually accompanied by its own set of I/O semantics, which are tailored specifically to this interface. A description of the most common I/O interfaces and their corresponding semantics follows.

The **POSIX** I/O interface has been originally designed for use in local file systems. Its first formal specification dates back to 1988, when it was included in POSIX.1. Asynchronous/synchronous I/O was added in POSIX.1b from 1993.



(a) Strongly-layered traditional I/O system (b) JULEA I/O layers

Fig. 1. Comparison of traditional and JULEA I/O stacks

This interface is very widely used, even in parallel distributed file systems, and thus provides excellent portability.

The original interface did not offer ways to specify semantical information about the accesses or the data. A feature added in POSIX.1-2001 is called `posix_fadvise()` and allows announcing the pattern which will be used to access the data. However, this does not change the semantics of any following I/O operations. It typically used to increase the readahead window (`POSIX_FADV_SEQUENTIAL`), disable readahead (`POSIX_FADV_RANDOM`), or to populate (`POSIX_FADV_WILLNEED`) and free (`POSIX_FADV_DONTNEED`) the cache.

The **MPI-I/O** interface was introduced in the MPI-2.0 standard in 1997 [11] and offers support for parallel I/O. It provides an I/O middleware which abstracts from the actual underlying file system – the popular ROMIO implementation uses the so-called ADIO layer which includes support and optimizations for POSIX, NFS, OrangeFS and many others. The MPI-I/O interface uses the existing MPI infrastructure of MPI datatypes to access data within files.

The actual interface looks very much like the POSIX interface using file handles to access files. MPI-I/O offers support for different file access modes, which can be specified at file-open time. The MPI standard specifies several access modes [12]. However, the only to access modes which can be considered semantical information are `MPI_MODE_UNIQUE_OPEN` and `MPI_MODE_SEQUENTIAL` as these give information about *how* the file is going to be accessed.

ADIOS [10] provides a high-level I/O interface that abstracts from the usual byte- or element-oriented access as found in POSIX or MPI-I/O. It outsources the actual I/O configuration into an external XML file which can be used to describe which data structures should be accessed and to automatically generate C or Fortran code. Due to this automatically generated code, the application developer does not need to directly interact with the underlying I/O middleware or file system.

There are a number of other I/O interfaces like SIONlib [5], NetCDF [15] and HDF5 [21] which focus on solving performance problems or offering additional features such as annotated data storage. As they also do not offer semantical information to be specified, they are only mentioned briefly here.

2.2 I/O Semantics

In the following, the most common I/O semantics are presented and potential shortcomings are highlighted.

POSIX I/O features very strict consistency requirements. For example, `write` operations have to be atomic and have to be visible to other clients immediately after the system call returns. While this might be relatively easy to support in local file systems, it can pose a serious bottleneck in parallel distributed file systems, because it effectively prohibits client-side caching from being used and requires additional locking. The semantics can only be changed in a very limited fashion. For example, the `strictatime`, `relatime` and `noatime` options change the file system's behavior regarding the last access timestamp, which can have an impact on performance. Additional options for `async` and `sync` are also available. However, all of these options can only be specified on a per-mount basis and have to be fixed at mount time – that is, they can not be modified by users under normal circumstances.

The **NFS** protocol provides close-to-open cache consistency by default, which implies that changes performed by a client are only written back to the server when the client closes the modified file. However, NFS offers limited support for changing this behavior: By mounting NFS using the `cto` or `nocto` options close-to-open cache coherence semantics can be switched on or off respectively. Additionally, the `async` and `sync` options can be used to modify the behavior of write operations: While `async` causes writes to only be propagated to the server when necessary[1], `sync` will cause system calls to only return when the data has been flushed to the server. Additional mount options are available to modify the caching behavior of attributes and directory entries. However, as in the POSIX case, these options can only be specified at mount time by the administrator.

MPI-I/O's consistency requirements are less strict than those defined by POSIX [19,4]. By default, MPI-I/O guarantees that non-overlapping write operations will be handled correctly and that changes are immediately visible only to the writing process itself. Other processes first have to synchronize their view of the file to see the changes. For use-cases requiring stricter consistency semantics, MPI-I/O offers the so-called *atomic mode*. The atomic mode specifies that changes will be visible to all process within the same communicator instantly. This can be difficult to achieve, because MPI-I/O allows non-contiguous operations and parallel distributed file systems can stripe single write operations over multiple servers [16,8]. However, apart from the configurable atomic mode, MPI-I/O does not offer any other means of changing the semantics. MPI-I/O

[1] Possible reasons include memory pressure and (un)locking, synchronizing or closing a file.

implementations are free to offer so-called *hints*, which are mainly used to control things like buffer sizes and participating processes. However, because hints are optional, different implementations are free to ignore them [20].

3 Design

As previously shown, the interfaces and semantics currently used for distributed file systems are suboptimal because they are either not well-adapted for the requirements and demands found in high performance computing today or do not allow fine-grained semantical information to be specified [14,18,22]. In this paper, we demonstrate a new I/O interface as well as a file system prototype called JULEA. It has been implemented from scratch to be suited specifically for the requirements found in high performance computing.

3.1 Layers

The intended general architecture of the JULEA I/O stack is illustrated in Figure 1b and features less layers than the traditional one in Figure 1a. This allows concentrating all optimizations into a single layer, reducing the implementation and runtime overhead. An important design goal is to remove the duplication of functionality found in the traditional I/O stack. For example, path lookup should only be performed on the uppermost layer. This can be achieved by eliminating the underlying POSIX file systems and using a suitable object store, which allows objects to be accessed directly using a unique ID. JULEA supports multiple storage backends such as existing POSIX file systems as well as object stores. This allows JULEA to always use the best-suited backend while maintaining compatibility with a wide range of software environments. For the metadata part, we decided to use an existing NoSQL database system called MongoDB [1]. The only remaining deficiency is MongoDB's dependency on an underlying POSIX file system, which we are currently investigating.

3.2 File System Namespace

Traditional file systems allow deeply-nested directory structures. To avoid the overhead caused by this, only a restricted, relatively flat hierarchical namespace is supported. While our approach might be unsuited for a general purpose file system, we explicitly focus on specific use-cases that are commonly found in high performance computing.

It is divided into *stores*, *collections*, and *items*. Each store can contain multiple collections which in turn can contain multiple items. Additionally, items feature a very reduced set of metadata. For example, unimportant information like the time of the last access has been omitted. The goal of these changes is to minimize the overhead during normal file system operation. For example, in traditional POSIX file systems, each component of the potentially deeply-nested path has to be checked for each access. This requires reading its associated metadata,

checking permissions, etc., which usually happens sequentially. Additionally, in distributed file systems these operations can be very costly if many (relatively small) network messages are involved.

3.3 Interface

The interface has been designed from scratch to offer simplicity of use while still meeting the requirements of high performance and semantics-awareness. Two major features are the ability to specify semantical information and to batch operations.

It is possible to specify additional information equivalent to the coarse-grained statement "this is a checkpoint" or the more fine-grained "this operation requires strict consistency semantics". This allows the file system to tune operations for specific applications by itself. Additionally, it is possible to emulate well-established semantics as well as mixing different semantics within one application.

All accesses to the file systems are done via so-called *batches*. Each batch can consist of multiple operations. For example, multiple items can be created or different offsets within an item can be accessed in one batch. It is also possible to combine different kinds of operations within one batch. For example, one batch might create a collection and several items within it, and write data to each one. Because the file system has knowledge about all operations within one batch, more elaborate optimizations can be performed. The advantages of this approach will be evaluated in Section 4.

The pseudo code found in Listing 1.1 shows an example of how the interface generally works. A new batch using the POSIX semantics (line 1) as well as a store, collection and item are created (lines 2–4). Afterwards, the collection is added to the store (line 6) and the item is added to the collection (line 7). Additionally, some data is written to the item (line 8). Finally, the batch is executed (line 10) which in turn executes all three operations with the given semantics.

Listing 1.1. JULEA pseudo code

```
1  batch = new Batch(POSIX_SEMANTICS);
2  store = new Store("test");
3  collection = new Collection("test");
4  item = new Item("test");
5
6  batch.add(store.add(collection));
7  batch.add(collection.add(item));
8  batch.add(item.write(...));
9
10 batch.execute();
```

3.4 Semantics

The JULEA interface allows many aspects of the semantics of file system operations to be changed at runtime on a per-batch basis. Several key areas of the semantics have been identified as important to provide opportunities for optimizations. Support for atomicity, concurrency, consistency, persistency and safety has already been designed, while possible data manipulation and security aspects are still in the planning stage.

Other ideas include prompting the file system to store multiple copies of file data and metadata, and to compress or encrypt it on-the-fly. The security policy could be changed depending on the file system environment, enabling or disabling more strict permission checks.

Detailed information about the different semantics together with possible configurations is given in the following list.

- **Atomicity:** The atomicity semantics can be used to specify whether accesses should be atomic, that is, whether it is possible for clients to see intermediate states of operations. For example, a single write operation spanning two servers might have already reached one of them but not the other. If atomic accesses are enabled, other clients will be unable to see this inconsistent state.
- **Concurrency:** The concurrency semantics can be used to specify whether concurrent accesses will take place and, if so, how they will look like. This can be used to enable or disable locking as needed.
- **Consistency:** The consistency semantics can be used to specify if and when clients will see modifications performed by other clients. This can be used to enable client-side read caching whenever possible.
- **Persistency:** The persistency semantics can be used to specify if and when data must be written to persistent storage. This can be used to enable client-side write caching whenever possible. For example, temporary data can be cached more aggressively and does not necessarily need to be written to persistent storage at all. This can be especially advantageous when different levels of storage such as node-local SSDs are available.
- **Safety:** The safety semantics can be used to specify how safely data should be handled. For example, this can be used to disable waiting for the server's acknowledgment when sending unimportant data. On the other hand, it can be used to make sure that important data will survive a system failure by flushing it to the storage devices immediately.
- **Templates:** Semantics templates can be used to provide templates for specific use-cases. For example, it can be used to to mimic the current POSIX semantics as closely as possible, and to tune the semantics for application input or output, which can be handled differently.

3.5 Architecture

JULEA provides a user-space library which can be linked to applications, allowing them to use the JULEA I/O interface. Additionally, a user-space daemon handles storing the file data on the I/O servers. The library communicates

with both the JULEA daemons and the MongoDB servers running on the data and metadata servers, respectively. By providing all functionality in user-space, JULEA is largely independent of the used operating system and kernel, and can be easily ported to new software environments.

Operations within one batch are aggregated and sent to the appropriate daemons in as few messages as possible to decrease network overhead. There are also plans to reorder and merge operations, which can help to further increase efficiency. The semantics specified for each batch are used to internally modify the behavior of I/O operations. While most of this semantical information is only needed by the client, it can also be transferred to the daemons when necessary.

For example, the safety semantics are always sent to the daemons, which can use this information to avoid sending back unneeded replies to the clients. The atomicity and concurrency semantics can be used to decide whether locking is necessary. Traditional interfaces do not have access to such information and therefore have to make pessimistic assumptions, which might force them to always handle the worst-case scenario. Since application developers know the access patterns of their applications, they can easily specify such information. This can be very beneficial, because lockless access to shared files can improve performance dramatically. The additional semantical information can also be used to reduce locking overhead on the metadata servers by making sure that specific metadata such as the file size and modification time are only stored explicitly for non-parallel workloads. Since highly parallel workloads would cause metadata update storms, such information is better computed on-the-fly whenever it is needed in these cases.

4 Evaluation

Our prototype was built to provide a reference implementation of our new I/O interface. It has built-in support for tracing client and server activities in various formats such as OTF [7] and HDTrace [13]. This can be used to visualize the inner workings and can be very helpful when debugging errors or searching for performance issues. To evaluate the benefits of our implementation we have extended the fileop benchmark – which is part of IOzone [27] – to support MPI and to also use the native interfaces of OrangeFS and JULEA. Only the most interesting metadata-heavy operations (`mkdir`, `rmdir`, `create`, `stat` and `delete`) were benchmarked. All results are averaged over at least five runs.

Figure 2 shows results for Lustre, OrangeFS and JULEA[2]. fileop was configured to run with a varying number of MPI processes on 1–5 nodes with up to 12 processes per node, each process working in its own directory/collection. OrangeFS and JULEA were tested using their native interfaces, while the POSIX interface was used for Lustre. All file systems were configured to provide one data and metadata server, and used their default configuration – apart from OrangeFS, where `TroveSyncMeta` was set to `no` to disable synchronous metadata operations. OrangeFS and JULEA were run on dual-socket machines with two

[2] Note the logarithmic y-axis and different scaling for each of the subfigures.

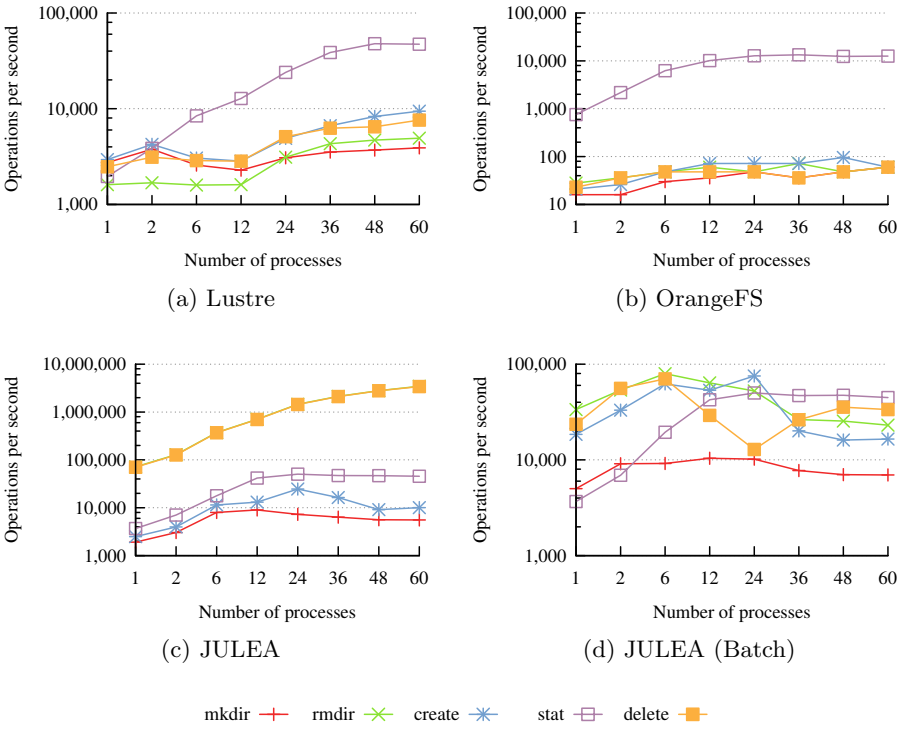


Fig. 2. Comparison of Lustre, OrangeFS and JULEA metadata performance

Intel Xeon X5650 processors and 12 GB of main memory each. Due to different operating system requirements, Lustre was run on single-socket machines with one Intel Xeon E31275 processor and 16 GB of main memory each. The evaluation was carried out using the `ldiskfs` backend for Lustre, an underlying `ext4` file system for OrangeFS and JULEA, and MongoDB 2.2.2 for JULEA. All data was stored on 7,200 RPM HDDs.

The results for Luste are shown in Figure 2a. It is interesting to note that the performance of all operations except for `stat` does not improve if more than 1 or 2 client processes per node are used. For these operations, there is practically no performance difference for the configurations using 1–12 processes. However, performance does increase when processes are distributed across more client nodes. Using 60 client processes, the `mkdir` and `rmdir` operations reach a maximum of around 4,000 and 5,000 operations per second, while the `create` and `delete` operations reach approximately 9,500 and 7,500 operations per second. `stat` scales well up to 36–48 client processes, where the curve begins to flatten, reaching a maximum of roughly 48,000 operations per second.

Figure 2b shows the results for OrangeFS. The `mkdir`, `rmdir`, `create` and `delete` operations deliver 20–30 operations per second when using a single client process. Increasing the number of processes does not significantly improve the results, resulting in a maximum performance of 75–100 operations per second when using 60 client processes. The `stat` operation performs significantly better with 750 operations per second when using a single client process. It also scales well up to 12 processes, where the curve flattens, reaching a maximum of 13,000 operations per second when using 24–60 processes.

The results for JULEA using individual operations is shown in Figure 2c. The `mkdir` and `create` operations deliver 2,000 and 2,500 operations per second using only a single client process. They scale well up to 12 and 24 processes respectively, where they reach their maximum performance of 9,000 and 25,000 operations per second. Increasing the number of processes further causes the performance to drop to around 50%. The `stat` operation starts out with 3,500 operations per second, reaches its maximum of 50,000 operations per second when using 24 processes and stays at this level when further increasing the number of processes. The `rmdir` and `delete` operations show suspiciously high performance of up to 3,400,000 operations per second, which might be caused by the MongoDB query returning before the actual remove operation has finished. However, this initial assumption still has to be investigated.

Figure 2d shows the results for JULEA using batches to aggregate operations. The `mkdir` operation provides only slightly better performance than in the previous case, peaking at 10,000 operations per second. However, fewer client processes are required to reach maximum performance, with only two processes needed to obtain 9,000 operations per second. The `create` operation performs significantly better, starting at 18,500 operations per second and reaching a maximum of 75,000 operations per second with 24 client processes. While performance drops to 20–25% when using 36 processes and more, it is still faster by a factor of 2 when compared to using individual operations. `status` behaves exactly the same as in the previous case. The `rmdir` and `delete` operations now deliver more realistic performance numbers. However, both operations reach a maximum with 6 client processes, dropping steadily after that. As already noted, both operations still have to be investigated more closely. Especially the `delete` operation's steep drop from 6–24 processes warrants a closer examination.

Overall, JULEA provides metadata performance comparable to other established parallel distributed file systems. More investigation and tweaking of the MongoDB configuration will be required to eliminate the performance drop-off with larger amounts of client processes. We are currently also working on supporting sharded configurations of MongoDB which we expect to increase performance even further.

5 Related Work

MosaStore [2] is a versatile storage system which is configurable at application deployment time and thus allows application-specific optimizations. This

approach is similar to the JULEA approach, however, MosaStore provides a storage system bound to specific applications instead of a globally shared one. Additionally, the storage system can not be reconfigured at runtime and keeps the traditional POSIX I/O interface.

The authors present a configurable security approach in [6] which allows using scavenged storage systems consisting of unused workstation hardware in trusted, partially trusted and untrusted environments in a secure way. While JULEA does not use scavenged storage hardware, the cited work shows that configurable security can be achieved with relatively low overhead. This could also be supported in JULEA to cater to different security requirements.

A new file system approach is presented in [9] that eliminates the current need for many small accesses to get the metadata of all path components during path lookup. By using the hashed file path to directly look up the related data and metadata, this can be reduced to only require one read operation per file access. While this can significantly increase small file performance, renaming of parent directories causes all child hashes to change which might lead to a lot of computational overhead. The JULEA interface does not use hashed path lookups for this reason, but implements a relatively flat namespace to reduce lookup overhead.

CAPFS [23] introduces a new content-addressable file store that allows users to define data consistency semantics at runtime. While providing a client-side plug-in API allows users to implement their own consistency policies, CAPFS is limited to tuning the consistency of file data and keeps the traditional POSIX interface. Additionally, the consistency semantics can only be changed on a per-file basis.

6 Conclusions and Future Work

In this paper, we have presented the design and implementation of our novel semantics-aware I/O interface and prototypical file system. It provides an interface and semantics suited for high performance computing, and aims to reduce redundant functionalities currently found within the I/O stack. Unlike similar approaches, JULEA allows fine-grained specification of the I/O semantics required by the application on a per-operation basis. By merely specifying the I/O requirements and leaving the realization and potential optimizations to the underlying file system, the I/O system can be tailored better to the actual hardware, improving efficiency. Additional features like the batching of operations allow further optimizations and provide potential for experimenting with other novel ideas regarding the handling of I/O.

One such idea, which fits naturally into the concept of batches, but is more oriented towards programming efficiency, is to implement transaction support. This would allow the application developer to specify how errors should be handled. For example, the file system could be told to automatically revert to the previous state in case of an error, making error handling and subsequent cleanup tremendously more easy.

Additional evaluations focusing on other aspects of the file system such as data performance and the influence of different semantics on a number of use-cases have already been planned and will follow soon. Early benchmarks suggest that shared file access can especially benefit from being able to specify semantical information about the access patterns. In some cases, the strict atomicity and consistency requirements enforced by the traditional POSIX semantics can cause performance drops of a factor of 100 and more. To evaluate the potential benefits with realistic use-cases we also plan to port an existing numerical application to the JULEA I/O interface in the future.

References

1. 10gen, Inc.: MongoDB (2012), http://www.mongodb.org/ (last accessed: February 2013)
2. Al-Kiswany, S., Gharaibeh, A., Ripeanu, M.: The Case for a Versatile Storage System. SIGOPS Oper. Syst. Rev. (January 2010)
3. Cluster File Systems, Inc.: Lustre: A Scalable, High-Performance File System (November 2002), http://www.cse.buffalo.edu/faculty/tkosar/cse710/papers/lustre-whitepaper.pdf (last accessed: February 2013)
4. Corbett, P., Feitelson, D., Fineberg, S., Hsu, Y., Nitzberg, B., Prost, J.P., Snir, M., Traversat, B., Wong, P.: Overview of the MPI-IO Parallel I/O Interface. In: IPPS 1995 Workshop on Input/Output in Parallel and Distributed Systems (April 1995)
5. Frings, W., Wolf, F., Petkov, V.: Scalable massively parallel I/O to task-local files. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC 2009 (2009)
6. Gharaibeh, A., Al-Kiswany, S., Ripeanu, M.: Configurable security for scavenged storage systems. In: Proceedings of the 4th ACM International Workshop on Storage Security and Survivability, Storage (2008)
7. Knüpfer, A., Brendel, R., Brunst, H., Mix, H., Nagel, W.E.: Introducing the Open Trace Format (OTF). In: Alexandrov, V.N., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2006. LNCS, vol. 3992, pp. 526–533. Springer, Heidelberg (2006), http://dx.doi.org/10.1007/11758525_71
8. Latham, R., Ross, R., Thakur, R.: Implementing MPI-IO Atomic Mode and Shared File Pointers Using MPI One-Sided Communication. Int. J. High Perform. Comput. Appl. (May 2007)
9. Lensing, P., Meister, D., Brinkmann, A.: hashFS: Applying Hashing to Optimize File Systems for Small File Reads. In: Proceedings of the 2010 International Workshop on Storage Network Architecture and Parallel I/Os, SNAPI 2010 (2010)
10. Lofstead, J.F., Klasky, S., Schwan, K., Podhorszki, N., Jin, C.: Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In: Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments, CLADE 2008 (June 2008)
11. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard. Version 3.0 (September 2012), http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf (last accessed: February 2013)
12. Message Passing Interface Forum: Opening a File (February 2013), http://www.mpi-forum.org/docs/mpi22-report/node265.htm (last accessed: February 2013)

13. Minartz, T., Molka, D., Kunkel, J., Knobloch, M., Kuhn, M., Ludwig, T.: Tool Environments to Measure Power Consumption and Computational Performance, ch. 31. Chapman and Hall/CRC Press Taylor and Francis Group (2012)
14. Patil, S., Gibson, G.A., Ganger, G.R., Lopez, J., Polte, M., Tantisiroj, W., Xiao, L.: In search of an API for scalable file systems: Under the table or above it? In: Proceedings of the 2009 Conference on Hot Topics in Cloud Computing, HotCloud 2009 (2009)
15. Rew, R., Davis, G.: Data Management: NetCDF: an Interface for Scientific Data Access. IEEE Comput. Graph. Appl. (July 1990)
16. Ross, R., Latham, R., Gropp, W., Thakur, R., Toonen, B.: Implementing MPI-IO atomic mode without file system support. In: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid, CCGRID 2005 (2005)
17. Schmuck, F., Haskin, R.: GPFS: A Shared-Disk File System for Large Computing Clusters. In: Proceedings of the 1st USENIX Conference on File and Storage Technologies, FAST 2002 (2002)
18. Sehrish, S.: Improving Performance and Programmer Productivity for I/O-Intensive High Performance Computing Applications. Phd thesis, School of Electrical Engineering and Computer Science in the College of Engineering and Computer Science at the University of Central Florida (2010)
19. Sterling, T., Lusk, E., Gropp, W. (eds.): Beowulf Cluster Computing with Linux, 2nd edn. MIT Press (2003)
20. Thakur, R., Ross, R., Lusk, E., Gropp, W., Latham, R.: Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation (April 2010), http://www.mcs.anl.gov/research/projects/romio/doc/users-guide.pdf (last accessed: February 2013)
21. The HDF Group: Hierarchical data format version 5 (2000-2010), http://www.hdfgroup.org/HDF5 (last accessed: February 2013)
22. Vilayannur, M., Lang, S., Ross, R., Klundt, R., Ward, L.: Extending the POSIX I/O Interface: A Parallel File System Perspective. Tech. Rep. ANL/MCS-TM-302 (October 2008)
23. Vilayannur, M., Nath, P., Sivasubramaniam, A.: Providing Tunable Consistency for a Parallel File Store. In: Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies, FAST 2005, vol. 4 (2005)
24. Wikipedia: Festplattenlaufwerk – Geschwindigkeit (February 2013), http://de.wikipedia.org/wiki/Festplattenlaufwerk#Geschwindigkeit (last accessed: February 2013)
25. Wikipedia: Mark Kryder – Kryder's Law (February 2013), http://en.wikipedia.org/wiki/Mark_Kryder#Kryder.27s_Law (last accessed: February 2013)
26. Wikipedia: TOP500 (February 2013), http://en.wikipedia.org/wiki/TOP500 (last accessed: February 2013)
27. Norcott, W.D., Capps, D.: IOzone Filesystem Benchmark (2006), http://www.iozone.org/ (last accessed: February 2013)

Towards Self-optimization in HPC I/O*

Michaela Zimmer, Julian Martin Kunkel, and Thomas Ludwig

University of Hamburg, Germany
michaela.zimmer@informatik.uni-hamburg.de

Abstract. Performance analysis and optimization of high-performance I/O systems is a daunting task. Mainly, this is due to the overwhelmingly complex interplay of internal processes while executing application programs. Unfortunately, there is a lack of monitoring tools to reduce this complexity to a bearable level. For these reasons, the project Scalable I/O for Extreme Performance (SIOX) aims to provide a versatile environment for recording system activities and learning from this information. While still under development, SIOX will ultimately assist in locating and diagnosing performance problems and automatically suggest and apply performance optimizations.

The SIOX knowledge path is concerned with the analysis and utilization of data describing the cause-and-effect chain recorded via the monitoring path. In this paper, we present our refined modular design of the knowledge path. This includes a description of logical components and their interfaces, details about extracting, storing and retrieving abstract activity patterns, a concept for tying knowledge to these patterns, and the integration of machine learning. Each of these tasks is illustrated through examples. The feasibility of our design is further demonstrated with an internal component for anomaly detection, permitting intelligent monitoring to limit the SIOX system's impact on system resources.

Keywords: Parallel I/O, Machine Learning, Self-Optimization.

1 Introduction

While processor performance has been blessed with continual growth according to Moore's Law for decades now, performance increases of persistent storage media fall short of this by several orders of magnitude. To bridge this gap, I/O systems for high-performance computing (HPC) in particular have had to grow horizontally, requiring ever more layers of management infrastructure to control the ensuing complexity. Diagnosing such a system has become a task to challenge even experts. Parametrizing it for optimum performance requires intimate knowledge of every component, its optimization parameters and strategies and the interplay emerging when dozens to tens of thousands of them are combined.

* We want to express our gratitude to the „Deutsches Zentrum für Luft- und Raumfahrt e.V." as responsible project agency and to the „Bundesministerium für Bildung und Forschung" for the financial support under grant 01 IH 11008 A-C.

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 422–434, 2013.

The vision of autonomous computing, as laid out by Kephart and Chess [1], promised to curb this complexity by marshalling the system itself to share into the effort. The SIOX Project [2] was initiated to realize that vision with respect to self-optimizing HPC-I/O systems. Continually monitoring performance and overhead, an I/O system instrumented for SIOX will autonomously detect problems and infer advantageous settings such as MPI hints, RAID stripe sizes and possible interactions between them. By adjusting its own level of reflexive activity to the situation, it will secure a net positive impact on overall efficiency.

This paper illustrates the structure of and techniques used in the knowledge processing sub-system enabling SIOX to achieve these goals.

In Section 2, we survey some other approaches to the problem, highlighting the differences to SIOX. We briefly outline the SIOX design with some definitions and an overview of the components involved in Section 3. Our main focus lies on Section 4, where we introduce the SIOX knowledge path, its modules and the concepts realized in its operation. Section 5 describes how the knowledge path can use modules for intelligent monitoring, before we conclude with a summary and some thoughts on future work in Section 6.

2 Related Work

Early approaches to system self-management relied on the direct classification of system state or behaviour to automatically diagnose problems or even enact optimization policies.

A typical proponent is the work of Madhyastha and Reed [3], comparing classification of I/O access patterns by feed-forward neural networks and by hidden Markov models. As results, higher level application I/O patterns are inferred and looked up in a table to determine the file system policy to set for the next accesses. The table, however, has to be supplied by an administrator implementing his heuristics.

Later approaches are marked by schemes to persist their results. Holding these in a database, problem analysis benefits from past diagnostic efforts, possibly even applying known repair actions to recognized problems. Here, we can divide systems according to whether they observe system state, recording metrics, or behaviour, tracing program execution.

Of those relying on program traces, Modani et al. [4] employ the call stacks reported by system failures as a search index to classify presumptive root causes.

`Magpie`, a system by Barham et al. [5], traces events under Windows, merging them according to pre-defined schemas specifying event relationships. Their causal chains are reconstructed, attributed to external requests via temporal joins over the event stream and clustered into models for the various types of workload observed. Deviations will point to anomalies deserving human attention.

Yuan et al. [6] combine system state and system behavior to identify the root causes of recurring problems. Tracing the system calls generated under Windows XP, they use support vector machines to classify the event sequences.

A presumptive root cause is identified, leaving the sequence – if flagged by a human as accurately diagnosed – available as eventual new training case for the classifier. The root cause description may include repair instructions, which, in some cases can be applied automatically.

Of the systems focussing on system metrics, `Cluebox` by Sandeed et al. [7] analyses logs for anomalies, pointing out the system counters most likely involved in the problem by principal feature analysis, ranking by decision trees and subsequent clustering.Expected latencies can now be predicted for new loads, detecting not only anomalies but also the counters most significantly deviating from par. No direct tracing or causal inference are needed, but once again, only hints for administrators are produced.

Cohen et al. [8] build on their previous work on *metric attribution*, identifying the low-level system metrics most significant for given classes of high-level system states using Tree-Augmented Bayesian Networks [9]. They collect system state information and combine the attributed metrics into *signatures*, clustering those belonging to the same problem class into *syndromes*.

In `Fa`, Duan et al. [10] define a system's base state by service level objectives; compliance constitutes health, violation failure. A robust data base of failure signatures is constructed from periodically sampled system metrics. New data is first classified against failure data annotated by a human, then, if necessary, against data clusters from healthy system states.

The one thing all of these systems have in common is the need for human intervention to benefit from the results, to apply the solutions to the problems identified or prepare the automated responses that some of the authors hint at.

While the work on invasive programming by Bungartz et al. (e.g. [11]) aims to automatically acquire and release resources according to the system's current requirements and capabilities, and can thus be regarded targeted at a similar problem as SIOX, it is mainly concerned with resource management. In contrast, SIOX will enable the existent management mechanisms to adapt their parameters to the system's current state and workload. As its scope is also limited to the I/O subsystem, the results of SIOX and invasive programming may very well complement each other in any system implementations.

3 The SIOX Approach

In comparison to described related work, SIOX covers full HPC I/O systems and aims to be applicable at all granularities and portable to platforms and across middleware and file systems, in a flexible and extensible hierarchy accommodating both bespoke heuristics and generic machine learning modules.

Under SIOX, a system will collect information on I/O activities on all instrumented levels, as well as relevant system information and metrics. An example I/O stack and the integration of SIOX is sketched in Figure 1, a scenario for potential activities is illustrated in Figure 2.

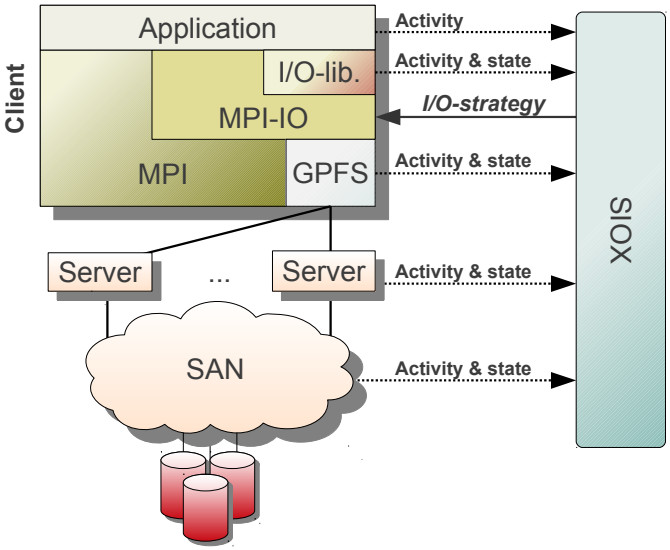


Fig. 1. Integration of SIOX into a traditional I/O-stack

Continuous intelligent monitoring, possibly adapting to problem type, will detect more than mere service level violations; instead, locally diverse anomaly conditions will be able to trigger fully automated (and learned) responses rather than provide mere pointers for human intervention. For this, SIOX combines on-line monitoring with off-line learning, joining state- to behaviour-based attributes and comparison to signatures as well as to a base-line. The recorded information will be analysed off-line to create and update a knowledge base holding optimized parameter suggestions for common or critical situations. During on-line operations, these parameters may be queried and used as pre-defined responses whenever such a situation occurs. Furthermore, the choice of responses to every situation is diverse, ranging from a mere log-level adjudication and detailed reports to facilitate human administration right to automated optimizations and problem solution strategies.

To cover the enormous multitude of possible hardware and software components that may be part of today's HPC system, SIOX takes an abstract view, as first suggested by Kunkel and Ludwig [12]. By virtue of this I/O path model (IOPm), a minimal set of components making up the system can be determined and classified according to their functionality, such as cache, block storage, network or an address translation of objects. A model graph of the system can be constructed covering the minimal cause-and-effect chain from application to storage device. Components may need to be represented by more than one of these elements, but this scheme warrants that any and all can be described by a very limited set of generic categories.

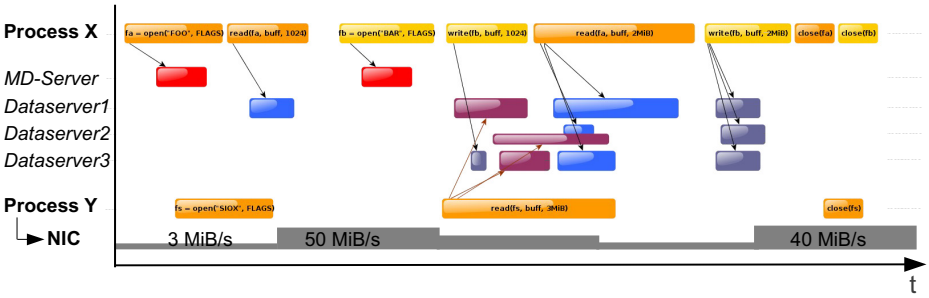


Fig. 2. Activity timelines of two processes and four file system servers – one system metric is provided for the entity executing process Y. Details for server activity and intermediate high-level I/O libraries are omitted.

3.1 Definitions

The literature on computer systems is extensive, with many terms being used ambiguously. We therefore define some terms we will refer to in the following:

Component. A hardware or software entity, such as a network switch, a hard disk drive, an application or a library.

Entity. A logical subunit of a component aware of SIOX, using SIOX interfaces or reporting monitoring data to it. Its extent is defined according to its functionality, such as a software layer in a library or a cache in a server.

Activity. A single, elementary operation on a single entity, possibly bundled with parameters, attributes and metrics pertaining to it. For example, an HDF5 `write()`, ATAPI `read()`, POSIX `fadvise()` or setting an MPI hint.

System Information. The state of a component and the whole system, depending on the hardware characteristics and executed activities. It consists of **dynamic** information describing the system state, e.g., utilization of the components, and **static** information about hardware and software, such as device types, available resources and performance characteristics.

Metric. A measurable or derivable quantity describing an aspect of the system, a component, an entity or an activity on any of the former, such as the number of Bytes written per second. An **Activity Metric** is a metric tied at report time to a specific activity, such as the execution time of a call. A **System Metric** is a metric that cannot be accurately assigned to a single activity, though usually influenced by them.

System Statistics. Data derived or regularly sampled from system metrics. Some system metrics can only be measured periodically, either because the system only provides the difference over that interval, or because the value changes so fast that recording every variation would prove prohibitive. Examples are network and disk throughput and CPU utilization.

Pattern. A set of activities linked by closeness in executing entity, time or causal relation. Also, a formal representation of such a set like a regular expression.

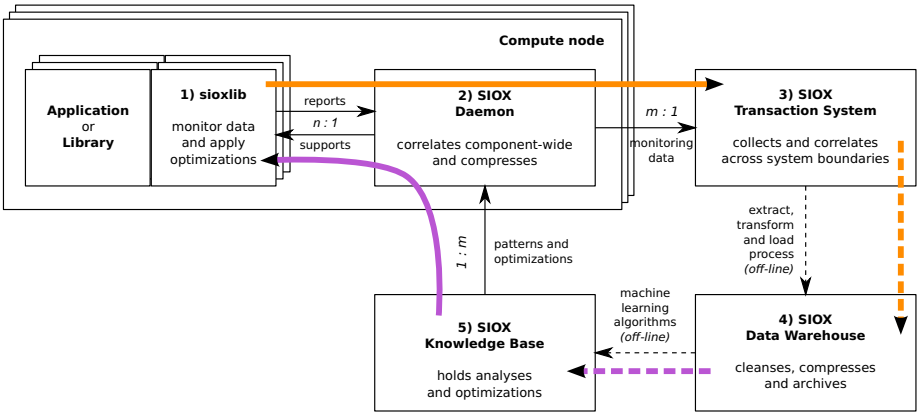


Fig. 3. The main components of the SIOX system and their cardinality. Monitoring data moves along the orange path; the knowledge path is shown in violet.

Situation. The current system state as observable by SIOX, including activities being executed and system metrics.

History. A limited record of recent situations, including a sliding window of previous activities and observed performance statistics.

Optimization Strategy. A scheme detailing how operations are executed by an entity, while not altering its functionality; e.g., a cache strategy. The strategies supported, if any, depend on the individual entity. The parameters for an optimization strategy, once chosen, influence operational performance according to the system's characteristics. **Optimization** is the choice of and selection of parameters for one or more optimization strategies.

Log-Level. The level of detail of its history an entity reports to SIOX. A higher log-level will supply more details but also require more system resources.

Response. Actions that may be taken upon certain situations occurring; they may also depend on the history. Typical examples are re-configuring an optimization or adjusting the log-level, both of which may also cascade through the entity's dependencies.

3.2 Components of the SIOX System

As first detailed in Wiedemann et al. [2], the SIOX system will be comprised of five primary components as shown in Figure 3:

1. The *sioxlib* linked to every component instrumented to work with SIOX.
2. One SIOX *daemon* per compute node. It acquires system information and aggregates and pre-processes activity logs, performance data and metrics. In a hierarchical set-up, additional daemons may serve as concentrator nodes to allow for scalability (see Section 4.4).

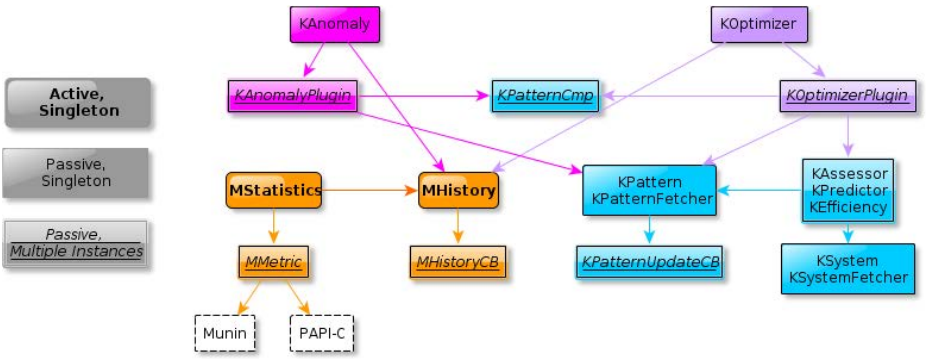


Fig. 4. SIOX modules involved in the knowledge path

3. A *transaction system* to collect, revise and concentrate the data submitted.
4. A *data warehouse* for long-time archiving of the monitoring data.
5. A *knowledge base* to hold the system information, as well as knowledge extracted from the data warehouse by off-line machine learning processes.

4 The SIOX Knowledge Path

The SIOX knowledge path (see Figure 3) begins at the data warehouse, holding representations of every activity logged, its parameters and the activity metrics that resulted from executing it. Off-line machine learning processes generalize activity sequences with comparable characteristics into patterns and potential responses to them (see Section 4.2). The results are stored in the knowledge base, together with records on system statistics for heuristics to assess observed vs. potential behaviour. A tree-shaped hierarchy of concentrator nodes connects knowledge base and entities to maintain full scalability (see Section 4.4). For every entity, the knowledge base will also hold information about the patterns most likely to occur there, those most likely to allow for effective optimizations, and those most warranting further investigation. Based on this, every entity will keep a local cache, updated regularly, of the patterns most crucial to its operation; this will conserve bandwidth and vastly improve response time. During operations, entities will match the cached patterns against situational and historic information to optimize their behaviour and control logging (see Section 5).

4.1 Modules and Interfaces

To allow for maximum flexibility while maintaining manageability, the SIOX knowledge path is built on a modular design. Not only does this permit us to replace a module with another implementation even after starting `sioxlib`, but many modules also allow concurrent loading of multiple plug-ins, all of which might excel at different jobs. Of the modules involved in the knowledge path (shown in Figure 4), two take lead roles: `KOptimizer` and `KAnomalyDecision`.

`KOptimizer` directs various plug-ins implementing the `KOptimizerPlugin` interface, observing the entity's recent activity history and injecting a response if a relevant pattern occurs. Each of these plug-ins may be tailored to optimize certain aspects of the entity's operation, specializing on certain heuristics for a group of activities or a class of patterns; details are given in Section 4.2.

`KAnomaly` orchestrates various `KAnomalyPlugin` implementations, each of which will monitor the entity's operational metrics or activities to detect exceptional (good or bad) performance (see Section 5). For example, a plug-in might compare an activity's execution time with a model of system characteristics to estimate its performance. Should any of these report anomalous behaviour, the entity's log-level will be adjusted and relevant parts of the history will be dispatched along the monitoring path.

Plug-ins that implement one of the interfaces `KAssessor`, `KPredictor` or `KEfficiency` are basic building blocks performing evaluations on activities and higher granularities. They differ in the scope of information used for evaluation, as described in Section 4.3. By our modular design, an `KOptimizerPlugin` may use any evaluator, easing development and maintainability significantly.

The decision which, if any, pattern matches a given sequence of activities best will be delegated to `KPatternCmp`. Therewith, each plug-in can have its own matching rule, to rely on only certain activities, such as file I/O. `KSystem` manages an entity's knowledge about general system characteristics, such as neighbourhood topology and hardware specifications, accessing the knowledge base via `KSystemFetcher`. Likewise, `KPattern` administers the entity's local cache of patterns and responses regularly, updated via `KPatternFetcher`.

`MHistory` provides access to an entity's sliding history. Certain events, such as a new activity being entered into the history or the history reaching capacity, will trigger callback functions previously registered. The latter follow the form laid down in the `MHistoryCB` interface. Plug-ins for `KAnomaly` and `KOptimizer` will subscribe and automatically be called when the situation changes.

The statistical performance monitor for system metrics, encapsulated in `MStatistics`, performs regular updates on system statistics as supplied by the module `MMetric` and delivers the information to `MHistory`. To provide the metrics themselves, it in turn may interface various tools such as `PAPI-C`, or use existing plug-ins from Munin[1].

4.2 From Observation to Pattern and Response

The modules responsible for pattern creation and processing come in pairs of an *off-line machine learning plug-in* (OMLP) and an on-line module implementing `KAnomalyPlugin` and/or `KOptimizerPlugin`.

The OMLP will employ data mining algorithms to extract interesting (read: having performed exceptionally badly or well) sequences of activities and their pertinent metric data from the data warehouse, cluster them and create a pattern representation of the set selected. It may simply concatenate the symbols

[1] `http://munin-monitoring.org/`

`open`(a, "F") `read`(a, 1024) `open`(b, "B") `write`(b, 1024) `read`(a, 2MiB) `write`(b, 2MiB) `close`(a) `close`(b)

(a) Observed activities (timing information omitted)

pattern	advice
Sr()Sr()Sr()	seq & willneed(size)
O(ext="nc")	willneed(0, 20 KiB)
O(ext="dat")	noReuse & random
Rw(size < 4K){5}	noReuse & random

(b) Table for an `fadvise()` plug-in

pattern	buffer-size
O()	4 MiB
W(size < 2 KiB){5}	1 MiB
W(size < 4 MiB) W(size < 4 MiB)	20 MiB
W(size ≥ 100 MiB)	direct-write

(c) Table for a write-behind plug-in

Fig. 5. Exemplary patterns including key/value pairs in brackets and responses for two optimization plug-ins. Usage of symbols and key/value pairs are the responsibility of OMLP and the plug-ins.

representing the single activities and collect any additional attributes in a list of key-value pairs. It may first transform the sequence of activities, merging subsequences into single symbols or filtering some, deriving new attributes from the original ones in the process. As this process will run off-line, it does not influence a production system.

The result, in any case, will be a table of symbol strings in a form resembling regular expressions, and their attributes as key-value pairs attached to each symbol. Responses will be encoded as key-value pairs and appended to its list. This table will be personal to the OMLP and its on-line plug-ins, stored in the knowledge base and propagated to entities as applicable.

Figure 5 has some illustrative examples: In (a), activities observed at file level are shown; in (b) and (c), patterns and responses are listed for two plug-ins. One controls `fadvise()` while the other manages the size for a write-behind buffer (assuming such controls exist for the deployed parallel file system). Both plug-ins filter observed activities and translate them into symbols and relevant attributes. Timings are not given but are part of the attributes observed. The first plug-in converts sequential accesses to the symbols *Sr* and *Sw* for read and write, respectively. Whenever three sequential reads are observed, Line 1 in its table encodes the response *seq* and *willneed*(*size*) which translates to a `FADV_SEQUENTIAL` of the total file and an `FADV_WILLNEED` which pre-fetches the same amount of data as previously accessed. This plug-in also allows usage of the file extension to restrict matching patterns. The write-behind plug-in could use ranges to match defined file sizes, and prioritize patterns further down in the table. It dynamically adapts to the record size. When a file is first opened, Line 1 sets a default buffer size of 4 MiB; for 5 small write accesses, Line 2 reduces the buffer size. While these are simple examples, they demonstrate the power of the concept.

A plug-in registers a function with the `KOptimizer` or `KAnomaly` modules for these entities, to be invoked in a pre-defined order whenever a defined condition is met, for example, when new activity or metric is reported. It then compares current situation and history to its table, finds the best matches – if any – and decides which of those with attached responses will see them enacted.

Any entity or higher-order sub-system may employ many of these plug-in pairs, each working in turn on its own pattern table and each implementing a different specialization or heuristic. A simple pair concerned only with `MPI_File_open()` and optimizing them by setting appropriate hints is just as possible as a general catch-all pair, forming actual regular expressions over the set of all activity types possible at this entity. This allows for quick implementation of highly specialized heuristics as well as for classical machine learning algorithms which will even determine activities and attributes of interest.

4.3 Assessing Activities

To ease assessing system performance, we decided to employ three distinct basic function classes. Though each can operate on activities, patterns, components and other granularities right up to the whole system itself, usually utilizing information generated by its more specific variants, we will concentrate on activity evaluators to demonstrate the concepts.

Assessors resort to the local situation and history to evaluate a completed activity's perceived performance impact, returning a performance metric.
Example: A very simple model for file access or data transfer, computes an estimate for a device's transfer rate:

$$f_{\text{assess}}(\text{Device, Job}) = \frac{\text{Time(Job)}}{\text{Size(Job)}}$$

Predictors forecast an incomplete action's performance given the current situation and history. Its return value has to be comparable to an assessor's, thus following the same rules. Predictors' main usage is to estimate the benefit of possible responses for optimization plug-ins.
Example: The most simple and important one, the *historical predictor*, is an extrapolation of all assessments of the pattern's previous occurrences under comparable system states and loads.

Efficiency evaluators relate the action's actual performance as given by an assessor to the best performance possible on the given system and may take present conditions into account, e.g. faults in hardware. They compute a real number from the interval $[0; 1]$, but use additional information about the pattern's performance distribution to return one of $\{-1; 0; +1\}$, signifying exceptionally bad, reasonable and exceptionally good performance, respectively.
Example:

$$f_{\text{efficiency}} = \begin{cases} +1 & \text{if} \quad 0.9 < \text{eff} \\ 0 & \text{if} \quad 0.2 \leq \text{eff} \leq 0.9 \\ -1 & \text{if} \quad \text{eff} < 0.2 \end{cases}$$

with

$$\text{eff} = \frac{f_{\text{assess}}(\text{Device, Job})}{\text{SequentialTransferRate(Device)}},$$

which will classify any observed transfer rate estimated at more than 90% or less than 20% of the maximum as exceptional, but more complex functions with more than three value ranges are well possible.

Like their base-level cousins, higher-order evaluators may compute simple weighted averages of their component activities' results, or follow any more sophisticated scheme.

4.4 Scalable Data Transport

To allow for the degree of scalability required in high-performance computing, SIOX implements a hierarchical subdivision scheme where daemons can act as concentrators, collecting, forwarding and disseminating data as needed. An example topology is illustrated in Figure 6. Daemons cache the information most important to a compute node, observing the local stream of activities as reported via the `sioxlib` and choosing and enacting any responses suggested by the knowledge available to them. They also keep a sliding history window of the activities, performance data and system statistics for each of their assigned entities.

Concentrators are similar but also function as local executive for the subtree of nodes assigned to them, just with more complex patterns combined of their child nodes' patterns. Additionally, each of the concentrators can deploy `KAnomaly` and `KOptimizer` plugins that operate on data generated by multiple entities and evaluate patterns spanning them.

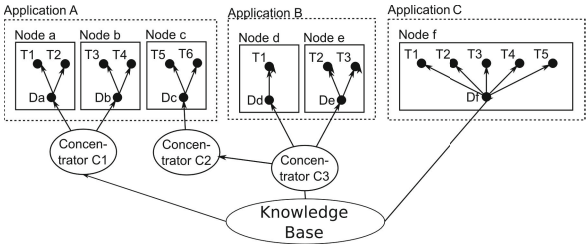


Fig. 6. Scalable data transport. SIOX-aware applications run on several compute nodes and fetch information from the knowledge base. Intermediate nodes in the graph cache fetched information to increase scalability.

5 Intelligent Monitoring

Every entity needs to determine which data about the current or past activities to report along the monitoring path, how much of the history to retain and and for how long, which system statistics to collect, at what interval to sample them and

what additional derived metrics to compute. As transferring all reported activity over the network and into the transaction database has a huge performance impact, it is imperative to restrict the logging to exceptional behaviour. For SIOX to provide detailed logs on anomalies while remaining unobtrusive otherwise, it must be able to react to developments in system stability – quickly, sensitive to developments spanning whole sub-systems, and with a fine local granularity. Our scheme for intelligent monitoring allows for three ways to influence logging:

- A user can control logging on each component to manually correct situations for which the history window is not sufficient or for bootstrapping the system.
- Neighbouring entities such as the local daemon may request a change in log-level to propagate alerts and allow pattern analysis across components.
- Most importantly, the various `KAnomalyPlugin` implementations may flag anomalies, influencing log-levels in the process. Of course, a plug-in detecting a return to normal behaviour will use the same mechanism to give the all-clear and decrease the log-level again.

As described in Section 4.1, the `KAnomaly` module permits implementations of `KAnomalyPlugin` to register as callback functions, calling them in a pre-defined order whenever a new activity is reported. Each of them may inspect current activity, situation and history to decide whether they constitute an anomaly, i.e., unusually good or bad system performance.

A typical example would be a system metric entering exceptional range, though defining these often is anything but trivial. A low processor load, for instance, may signify either a balanced system coping well with its workload, or a severe unbalance, leaving some components idling and others congested.

Should any plug-in flag the current state, it will cause the history to be dispatched along the monitoring path for later analysis. Additionally, it may adjust local log-levels, which may even be propagated to other entities.

The usual fate of the data thus recorded is to serve as a lesson to SIOX. Distilled into patterns (see Section 4.2) bearing the response "raise log-level" – the other typical example for anomaly detection – it will alert SIOX whenever the conditions leading up to the error are observed. This greatly benefits the analysis of recurring failures, as every instance provides some valuable new insights into the problem's preconditions and, ultimately, cause. In this sense, SIOX will intelligently direct its own analytical efforts where they are needed most.

6 Conclusion and Future Work

Our vision for the SIOX project is a system that will collect and analyse activity patterns and performance metrics in order to assess current and possible system performance, locate and diagnose problems and suggest solutions and improvements. In this paper, we have described the design of and information flow along the SIOX knowledge path. We have shown its logical layout and how its modules interact to perform their various tasks. Examples for our approach to knowledge representation have been given to demonstrate its applicability and a scheme for intelligent logging presented.

Opportunities for future work abound. Different platforms pose different problems even though SIOX is designed to be portable to all major operating and file systems. In the consortium, we are currently working on a first prototype for GPFS and MPI-IO which will rely on simple plug-ins. However, the various modules imaginable for pattern creation and matching, for anomaly detection and optimization as well as the machine learning algorithms offer a rich field for researchers and system administrators alike.

References

1. Kephart, J.O., Chess, D.M.: The Vision of Autonomic Computing. Computer 36(1), 41–50 (2003)
2. Wiedemann, M.C., Kunkel, J.M., Zimmer, M., Ludwig, T., Resch, M., Bönisch, T., Wang, X., Chut, A., Aguilera, A., Nagel, W.E., Kluge, M., Mickler, H.: Towards I/O Analysis of HPC Systems and a Generic Architecture to Collect Access Patterns. Computer Science - Research and Development 1, 1–11 (2012)
3. Madhyastha, T.M., Reed, D.A.: Learning to Classify Parallel Input/Output Access Patterns. IEEE Transactions on Parallel and Distributed Systems 13(8), 802–813 (2002)
4. Modani, N., Gupta, R., Lohman, G., Syeda-Mahmood, T., Mignet, L.: Automatically Identifying Known Software Problems. In: 2007 IEEE 23rd International Conference on Data Engineering Workshop, pp. 433–441 (April 2007)
5. Barham, P., Donnelly, A., Isaacs, R., Mortier, R.: Using Magpie for Request Extraction and Workload Modelling. In: Proceedings of the 6th Symposium on Opearting Systems Design and Implementation, vol. 6, pp. 259–272 (2004)
6. Yuan, C., Lao, N., Wen, J.-R., Li, J., Zhang, Z., Wang, Y.-M., Ma, W.-Y.: Automated Known Problem Diagnosis with Event Traces. In: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006, EuroSys 2006, pp. 375–388. ACM, New York (2006)
7. Sandeep, S.R., Swapna, M., Niranjan, T., Susarla, S., Nandi, S.: CLUEBOX: a Performance Log Analyzer for Automated Troubleshooting. In: Proceedings of the First USENIX Conference on Analysis of System Logs, WASL 2008. USENIX Association, Berkeley (2008)
8. Cohen, I., Zhang, S., Goldszmidt, M., Symons, J., Kelly, T., Fox, A.: Capturing, Indexing, Clustering, and Retrieving System History. SIGOPS Oper. Syst. Rev. 39(5), 105–118 (2005)
9. Cohen, I., Goldszmidt, M., Kelly, T., Symons, J., Chase, J.S.: Correlating Instrumentation Data to System States: a Building Block for Automated Diagnosis and Control. In: Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation, OSDI 2004, vol. 6. USENIX Association, Berkeley (2004)
10. Duan, S.S., Babu, Munagala, K.: Fa: A System for Automating Failure Diagnosis. In: IEEE 25th International Conference on Data Engineering, ICDE 2009, March 29-April 2, pp. 1012–1023 (2009)
11. Bader, M., Bungartz, H.J., Gerndt, M., Hollmann, A., Weidendorfer, J.: Invasive programming as a concept for HPC. In: Proc. of the 10th IASTED Int. Conf. on Parallel and Distr. Comp. and Netw., PDCN (2011)
12. Kunkel, J., Ludwig, T.: IOPm – Modeling the I/O Path with a Functional Representation of Parallel File System and Hardware Architecture. In: PDP 2012, Munich Network Management Team. IEEE (2012)

Using GPFS to Manage NVRAM-Based Storage Cache

Salem El Sayed[1], Stephan Graf[1], Michael Hennecke[2], Dirk Pleiter[1], Georg Schwarz[1], Heiko Schick[3], and Michael Stephan[1]

[1] JSC, Forschungszentrum Jülich, 52425 Jülich, Germany
[2] IBM Deutschland GmbH, 40474 Düsseldorf, Germany
[3] IBM Deutschland Research & Development GmbH, 71032 Böblingen, Germany

Abstract. I/O performance of large-scale HPC systems grows at a significantly slower rate than compute performance. In this article we investigate architectural options and technologies for a tiered storage system to mitigate this problem. Using GPFS and flash memory cards a prototype is implemented and evaluated. We compare performance numbers obtained by running synthetic benchmarks on a petascale Blue Gene/Q system connected to our prototype. Based on these results an assessment of the architecture and technology is performed.

1 Introduction

For very large high-performance computing (HPC) systems it has become a major challenge to maintain a reasonable balance of compute performance and performance of the I/O sub-system. In practice, this gap is growing and systems are moving away from Amdahl's rule of thumb for a balanced performance ratio, namely a bit of I/O per second for each instruction per second (see [1] for an updated version). Bandwidth is only one metric which describes the capability of an I/O sub-system. Additionally, capacity and access rates, i.e. number of I/O requests per second which can be served, have to be taken into account. While today's technology allows to build high capacity storage systems, reaching high access rates is even more challenging than improving the bandwidth.

These trends of technology are increasingly in conflict with application demands in computational sciences, in particular as these are becoming more I/O intensive [2]. Using traditional technologies these trends are not expected to reverse. Therefore, the design of the I/O sub-system will be a key issue which needs to be addressed for future exascale architectures.

Today, storage systems attached to HPC facilities typically comprise an aggregation of spinning disks. These are hosted in external file servers which are connected via a commodity network to the compute system. This design has the advantage that it allows to integrate a huge number of disks and to ensure the availability of multiple data paths for resilience.

The extensive use of disks is largely driven by costs. Disk technology has improved dramatically over a long period of time in terms of capacity versus

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 435–446, 2013.

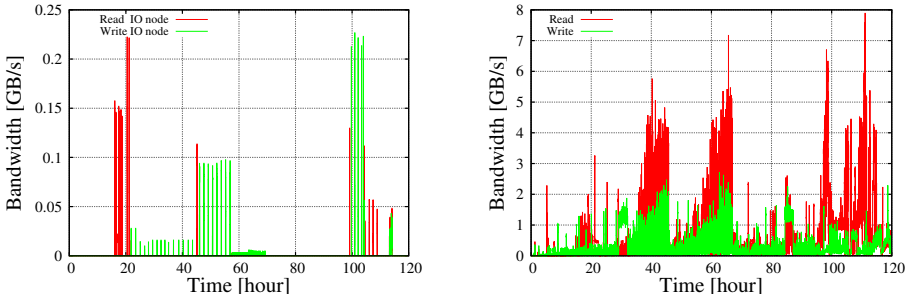


Fig. 1. Read and write bandwidth averaged over 120 s as a function of time. The left pane shows the measurements on a single Blue Gene/P I/O node while the right pane shows the results from all 600 I/O nodes of the JUGENE system.

costs. Bandwidth per disk and access rates, however, increase at a much slower rate (see, e.g., [3]) than compute performance. High bandwidth can thus only be achieved by increasing the number of disks within a single I/O sub-system, with load being suitably distributed. For HPC systems meanwhile the number of disks started to be mainly determined by bandwidth and not by capacity requirements. Scaling of disks is, however, limited by cost and power budget as well as the exponentially increasing risk of failures and data corruption. Using todays disk technology to meet exascale bandwidth requirements of 60 TByte/s [4] would exceed the currently mandated power budget of 20 MW, Whereas satisfying exascale bandwidth with flash memory and satisfying exascale capacity with a disk tier is possible at an "affordable" power consumption.

To meet future demands it is therefore necessary to consider other non-volatile storage technologies and explore new designs for the I/O sub-system architecture. Promising opportunities arise from storage devices based on flash memory. Compared to disk technologies they feature order(s) of magnitude higher bandwidth and access rates. The main disadvantage is the poor capacity versus costs ratio, but device capacity is slowly increasing.

In this article we explore an architecture where we integrate flash memory devices into IBM's† General Parallel File System (GPFS†) to implement an intermediate layer between compute nodes and disk-based file servers. GPFS's Information Lifecycle Management (ILM) [5] is used to manage the tiered storage such that this additional complexity is hidden from the user. The key feature of GPFS which we exploit is the option to define groups for different kind of storage devices, known as GPFS storage pools. Furthermore, the GPFS policy engine is used to manage the available storage and handle data movement between different storage pools.

This approach is motivated by the observation that applications running on HPC systems as they are operated at Jülich Supercomputing Centre (JSC) tend to start bursts of I/O operations.[1] In Fig. 1 we show the read and write

[1] Such behaviour has also been reported elsewhere in the literature, see, e.g., [6].

bandwidth measured on JSC's petascale Blue Gene/P† facility JUGENE. On each I/O node the bandwidth has been determined from the GPFS counters which have been retrieved every 120 s.

If we assume an intermediate storage layer being available providing a much higher I/O bandwidth to the compute nodes (but an unchanged bandwidth towards the external storage) it is possible to reduce the time needed for I/O. Alternatively, it is also possible to lower the total power consumption by providing the original bandwidth through flash, and reduce the bandwidth of the disk storage. This requires mechanisms to stage data kept in the external storage system before it is read by the application, as well as to cache data generated by the application before it is written to the external storage. All I/O operations related to staging are supposed to be executed asynchronously with respect to the application. The fast intermediate storage layer can also be used for out-of-core computations where data is temporarily moved out of main memory. Another use case is check-pointing. In both cases we assume the intermediate storage layer to be large enough to hold all data such that migration of the data to the external storage is avoided.

The key contributions of this paper are:

1. We designed and implemented a tiered storage system where non-volatile flash memory is used to realize a fast cache layer and where resources and data transfer are managed by GPFS ILM features.
2. We provide results for synthetic benchmarks which were executed on a petascale Blue Gene/Q system connected to the small-scale prototype storage cluster. We compare results obtained using different flash memory cards.
3. Finally, we perform a performance and usability analysis for such a tiered storage architecture. We use I/O statistics collected on a Blue Gene/P system as input for a simple model for using the system as a fast write cache.

2 Related Work

In recent years a number of papers have been published which investigate the use of fast storage technologies for staging I/O data, as well as different software architectures to manage such a tiered storage architecture.

In [7] part of the nodes' volatile main memory is used to create a temporary parallel file system. The performance of their RAMDISK nominally increases at the same rate as the number of nodes used by the application is increased. The authors implemented their own mechanisms to stage the data before a job starts and after the job completed. Data staging is controlled by a scheduler which is coupled to the systems resource manager (here SLURM).

The DataStager architecture [8, 9] uses the local volatile memory to keep buffers needed to implement asynchronous write operations. No file system is used to manage the buffer space, instead a concept of data objects is introduced which are created under application control. After being notified, DataStager

processes running on separate nodes manage the data transport, i.e. data transport is server directed and data is pulled whenever sufficient resources are available.

To overcome power and cost constrains of DRAM based staging, in [10] the use of NVRAM is advocated. Here a future node design scenario is evaluated comprising Active NVRAM, i.e. NVRAM plus a low-power compute element. The latter allows for asynchronous processing of data chunks committed by the application. Final data may be flushed to external disk storage. Data transport and resource management is thus largely controlled by the application.

3 Background

NAND flash memory belongs to the class of non-volatile memory technologies. Here we only consider Single-Level Cell (SLC) NAND flash, which features the highest number of write cycles. SLC flash chips currently have an endurance of up to O(100,000) erase/write cycles. At device level the problem of failing memory chips is significantly mitigated by wear-leveling and RAID. A large number of write cycles is critical when using flash memory devices for HPC systems. The advantage of flash memory devices with respect to standard disks are significantly higher bandwidth and orders of magnitude higher I/O operation rates due to significantly lower access latencies, at much lower power consumption.

GPFS is a flexible, scalable parallel file system architecture which allows to distribute both data and metadata. The smallest building block, a *Network Shared Disk (NSD)*, which can be any storage device, is dedicated for data only, metadata only, or both. In this work we exploit several features of GPFS. First, we make use of *storage pools* which allows to group storage elements. Pools are traditionally used to handle different types or usage modes of disks (and tapes). In the context of tiered storage this feature can be used to group the flash storage into one and the slower disk storage into another pool. Second, a policy engine provides the means to let GPFS manage the placement of new files and staging of files. This engine is controlled by a simple set of rules.

4 GPFS-Based Staging

In this paper we investigate a setup where flash memory cards are integrated into a persistent GPFS instance. GPFS features are used (1) to steer initial file placement, (2) to manage migration from flash to external, disk-based storage, and (3) to stage files from external storage to flash memory. The user therefore continues to access a standard file system.

We organise external disk storage and flash memory into a `disk` and `flash` pool, respectively. Then we use the GPFS policy engine to manage automatic data staging. In Fig. 2 we show our policy rules with the parameters defined in Table 1. These rules control when the GPFS events listed in Table 1 are thrown, the numerical values should be set according to operational characteristics of an HPC system's workload. Files are created in the storage pool `flash` as long as

```
RULE SET POOL 'flash' LIMIT(f_max)
RULE SET POOL 'disk'
RULE MIGRATE
     FROM POOL 'flash'
        THRESHOLD(f_start,f_stop)
        WEIGHT(CURRENT_TIMESTAMP-ACCESS_TIME)
     TO POOL 'disk'
```

Fig. 2. The GPFS policy rules for the target file system defines the placement and migration rule for the files' data blocks

filling does not exceed the threshold $f_{\max}$. When filling exceeds $f_{\max}$ the second rule applies which allows files to be created in the pool `disk`. This fall-back mechanism is foreseen to avoid writes failing when the pool `flash` is full but there is free space in the storage pool `disk`.

The third rule manages migration of data from the pool `flash` to `disk`. If the filling of the pool `flash` exceeds the threshold f_{start} the event `lowDiskSpace` is thrown, and automatic data migration is initiated. Migration stops once filling of pool `flash` drops below the limit f_{stop} (where $f_{\text{stop}} < f_{\text{start}}$). For GPFS to decide which files to migrate first, a weight factor is assigned to each file, which we have chosen such that least recently accessed files are migrated first. Migration is controlled by a callback function which is bound to the events `lowDiskSpace` and `noDiskSpace`. At any time these events occur, the installed policy rule for the according file system are re-evaluated. The GPFS parameter `noSpaceEventInterval` defines at which intervals the events `lowDiskSpace` and `noDiscSpace` may occur (it defaults to 120 seconds).

Finally, to steer staging of files to flash memory we propose the implementation of a prefetch mechanism which is either triggered by the user application or the system's resource manager (like in [7]). Technically this can be realized by using the command `mmchattr -P flash <filename>`. To avoid this file being automatically moved back to the pool `disk`, the last access time has to be updated, too. A possible way to achieve this is to create a library which will offer a function like `prefetch(<filename>)`. This library also has to ensure that only one rank of the application is in charge.

Table 1. Policy rule parameters and the values used in this paper

Parameter	Description	Value	Event
$f_{\max}$	Maximal filling (in percent)	90	`noDiskSpace`
f_{start}	Filling limit which starts migration (in percent)	30	`lowDiskSpace`
f_{stop}	Filling limit which stops migration (in percent)	5	–

The optimal choice of the parameters f_{start}, f_{stop} and $f_{\max}$ depends on how the system is used. For larger values of $f_{\max}$ the probability of files being opened for writing in the slow disk pool reduces. Large values for f_{start} and f_{stop} may cause migration to start too late or finish too early and thus increase the risk of the pool `flash` becoming full. On the other hand, small values would lead to early migration which could have bad effects on read performance in case the application tries to access the data for reading after migration.

5 Test Setup

5.1 Prototype Configuration

Our prototype I/O system *JUNIORS* (JUliche Novel IO Research System) consists of IBM x3650 M3 servers with two hex-core Intel Xeon X5650 CPUs (running at 2.67 GHz) and 48 GBytes DDR3 main memory. For the tests reported here we use up to 6 nodes. Each node is equipped with 2 flash memory cards either from Fusion-io† or Texas Memory Systems† (TMS). The most important performance parameters of these devices as reported by the corresponding vendor have been collected in Table 2. Note that these numbers only give an indication of the achievable performance. Each node is equipped with 2 dual-port 10-GbE adapters. The number of ports has been chosen such that the nodes nominal bandwidth to the flash memory and the network is roughly balanced. Channel bonding is applied to reduce the number of logical network interfaces per node.

The Ethernet network connects the prototype I/O system to the peta-scale Blue Gene/Q system JUQUEEN and the peta-scale disk storage system JUST. To use the massively parallel compute system for generating load, we mount our experimental GPFS file system on the Blue Gene/Q I/O nodes. For an overview of the prototype system and the Ethernet interconnect see Fig. 3 (left pane).

On each node we installed the operating system RHEL6.1 (64bit) and GPFS version 3.5.0.7. For the *Fusion-io ioDrive Duo* we used the driver/utility software version 3.1.5 including the firmware v7.0.0 revision 107322. For the *TMS RamSan-70* cards we used the driver/utility software version 3.6.0.11.

To monitor data flow within the system we designed a simple monitoring infrastructure consisting of different sensors as shown in Fig. 3 (right pane). The tools `netstat` and `iostat` are used to sense the data flow between node and network as well as through the Linux† block layer, respectively. Vendor specific

Table 2. Manufacturer hardware specification of the flash memory cards[2]

	Fusion-io ioDrive Duo SLC†	TMS RamSan-70†
Capacity	320 GByte	450 GByte
Read/write bandwidth [GByte/s]	1.5 / 1.5	1.25 / 0.9
Read/write IOPS	261,000 / 262,000	300,000 / 220,000

[2] For the RamSan-70 we report the performance numbers for 4 kBytes block size.

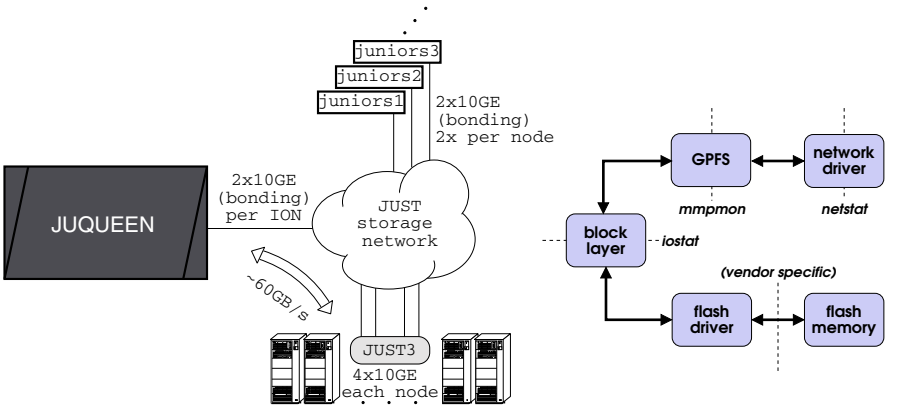


Fig. 3. The left pane shows a schematic view of the I/O prototype system and the network through which it is connected to a compute system as well as a storage system. The right pane illustrates the data flow monitoring infrastructure.

tools are used to sense the amount of data written (read) to (from) the flash memory device. GPFS statistics is collected using the mmpmon facility.

5.2 Benchmark Definitions

To test the performance we used three different benchmarks. For network bandwidth measurements we used the micro-benchmark `nsdperf`, a tool which is part of GPFS. It allows to define a set of server and clients and then generate the network traffic which real write and read operations would cause, without actually performing any disk I/O.

For sequential I/O bandwidth measurements we used the SIONlib [11] benchmark tool `partest`, which generates a load which is similar to what one would expect from HPC applications. These often perform task-local I/O where each job rank creates it's own file. If the number of ranks is large then the time needed to handle file system metadata will become large. This is a software effect which is not mitigated by using faster storage devices like flash memory cards. SIONlib is a parallel I/O library that addresses this problem by transparently mapping a large number of task-local files onto a small number of physical files. To benchmark our setup we run this test repeatedly using 32,768 processes, distributed over 512 Blue Gene/Q nodes. During each iteration 2 TBytes of data is written in total.

To measure bandwidth as well as I/O operation (IOP) rates we furthermore use the synthetic benchmark tool IOR [12] to generate a parallel I/O workload. It is highly configurable and supports various interfaces and access patterns and thus allows mimicking the I/O workload of different applications. We used 512 Blue Gene/Q nodes with one task per node which all performed random I/O operations on task-local files opened via the POSIX interface, with the flag `O_DIRECT` set to minimize caching effects. In total 128 GBytes are written and read using transfers of size 4 kBytes.

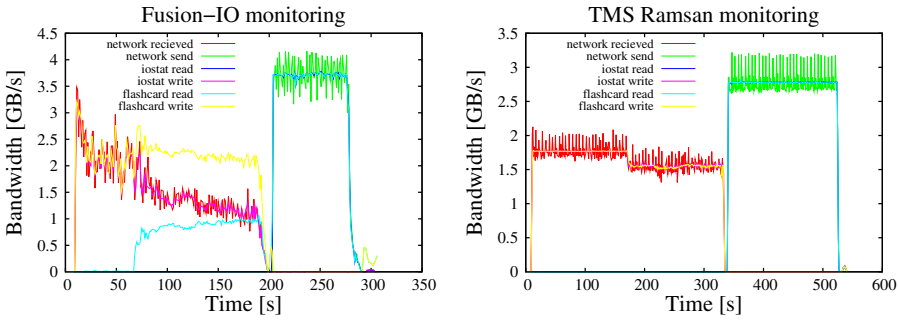


Fig. 4. Read and write bandwidth as a function of time for the SIONlib benchmark using flash memory cards from Fusion-io (left) or TMS (right). Both figures shows the I/O monitoring of one JUNIORS node (2 flash cards).

6 Evaluation

We start the performance evaluation by testing the network bandwidth between the JUQUEEN system, where the load is generated, and the prototype I/O system JUNIORS using `nsdperf`. For these tests we used 16 Blue Gene/Q IO nodes as clients and 2 JUNIORS nodes as servers. We found a bandwidth of 8.1 GByte/s for emulated writes and 9.7 for reads. Comparing these results with the nominal parameters of the flash cards listed in Table 2 indicates that our network provides sufficient bandwidth to balance the bandwidth to 2 flash cards per node.

For the following benchmarks each JUNIORS server was configured as an NSD server within a GPFS file system which was mounted by JUQUEEN I/O nodes. Using 16 clients we measured the I/O bandwidth for sequential access using `partest`. The observed read bandwidth is 12.5 Gbyte/s using 4 JUNIORS server and 8 Fusion-io flash cards, slightly more than we would expect from the vendors' specifications. However, a significant different behaviour is observed for writing where we observe a drop of the performance of more than 40% from 6.5 to 3.7 GByte/s after a short period of writing. To investigate the cause for this behaviour we analyse the data flow information from our monitoring system. The sensor values collected on 1 out of 4 nodes is plotted in Fig. 4 (left pane). The benchmark first performs a write and then a read cycle. We first notice that the amount of data passing the network device is consistent with the amount of data transferred over the operating systems block device layer. At the beginning of the initial write cycle the amount of data received via the network agrees with the amount of data written to the flash card. However, after 67 s of writing we observe a drop in the bandwidth of received data while at the same time the flash card reports read operations. We observe that the amount of data passing the block device layer towards the processor remains zero. Furthermore, the amount of data written to the flash card agrees with the amount of data received via the network plus the amount of data read from the flash cards. We therefore conclude

that the drop in write performance is caused by read and write operations from and to the flash card initiated by the flash card driver.

Let us now consider the performance obtained when using the flash cards from TMS. Here we used 2 JUNIORS servers with 2 flash cards each. We observe a read (write) bandwidth of 5.7 (3.2) GByte/s. This corresponds to 114% (90%) of the read (write) bandwidth one would naively expect from the vendor's performance specification. As can be seen from Fig. 4 the write bandwidth is sustained during the whole test.

Next, we evaluated the I/O operation (IOPS) rate which we can obtain on our prototype for a random access pattern generated by IOR. We again used the setup with 2 JUNIORS servers and 4 TMS cards. For comparison we repeated the benchmark run using the large-capacity storage system JUST, where about 5,000 disks are added to a single GPFS file system. In Table 3 the mean IOPS values for read and write are listed. We observe the IOPS rate on our prototype I/O system to be significantly higher than on the standard, disk-based storage system. For a fair comparison it has to be taken into account that the storage system JUST has been optimized for streaming I/O with disks organized into RAID6 arrays, with large stripes of 4 MBytes. Furthermore, the prototype and compute system are slightly closer in terms of network hops. The results on the prototype are an order of magnitude smaller than one would naively expect from the vendor's specification (see Table 2). These numbers we could only reproduce when performing local raw flash device access without a file system.

In the final step we defined two storage pools in the GPFS file system. For the pool `flash` we used 4 TMS cards in 2 JUNIORS nodes. the pool `disk` was implemented with 6 Fusion-io cards in 3 other servers, because there were no disks in the JUNIORS cluster. These pools were all configured as `dataOnly`, while a third pool on yet another flash disk is used for metadata. The GPFS policy placement and a migration rule are defined as shown in Fig. 2 and Table 1. To evaluate the setup we use the SIONlib benchmark `partest`, which uses POSIX I/O routines, as well as the IOR benchmark, which we configured such that MPI-IO is used, to create 1.5 TBytes of data.

In Fig. 5 we show the data throughput at the different sensors for the different benchmarks as a function of time. For the two different benchmarks no significant differences are observed. We obtained similar results for IOR using the POSIX instead of the MPI-IO interface. The SIONlib benchmark starts with a write cycle followed by a read cycle, which here starts and ends about 500 s and 800 s after starting the test, respectively. Initially all data is placed in pool `flash`. After 260 s GPFS started the migration process. There is only a small impact

Table 3. IOPS comparison between two JUNIORS nodes using 4 *TMS RamSan-70* cards and the classical scratch file system using about 5,000 disks

storage system	write IOPS (mean)	read IOPS (mean)
JUNIORS	52234	122123
JUST	20456	69712

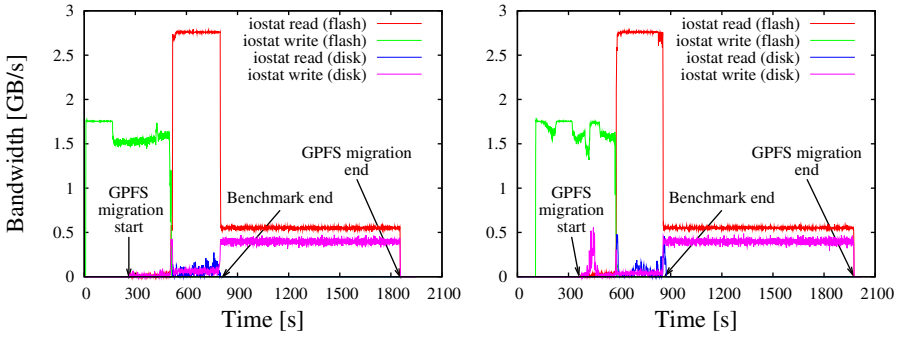


Fig. 5. Read and write bandwidth (per node) as a function of time for the SIONlib benchmark using the POSIX interface (left) and the IOR benchmark using the MPI-IO interface (right)

seen on the I/O performance. After the benchmark run ended it took up to 17 minutes to finish the migration.

7 Performance Analysis

To assess the potential of this architecture being used exclusively as a write cache, it is instructive to consider a simple model. Let us denote the time required to perform the computations and (synchronous) I/O operations by t_{comp} and t_{io}, respectively, and assume $t_{\text{comp}} > t_{\text{io}}$. While the application is computing, data can be staged between disk and flash pool. Therefore, it is reasonable to choose the bandwidth between the compute system and the staging area $y = t_{\text{comp}}/t_{\text{io}}$ times larger than between staging area and the external storage. As a result the time for I/O should reduce by a factor $1/y$ and therefore the overall execution time should reduce by a factor $(t_{\text{comp}} + t_{\text{io}}/y)/(t_{\text{comp}} + t_{\text{io}}) = (y + 1/y)/(y + 1)$. In this simple model the performance gain for the overall system would be up to 17% for $y = 1 + \sqrt{2}$. Since the storage subsystem accounts only for a fraction of the overall costs, this could significantly improve overall cost efficiency.

To investigate this further we implemented a simple simulation model as shown in Fig. 6. The model mimics the GPFS policy rules used for the JUNIORS prototype. The behaviour is controlled by the policy parameters f_{start}, f_{stop} and f_{max} as well as a set of bandwidth parameters. For each data path a different bandwidth parameter is foreseen. The bandwidth along the data path connecting processing device and pool `flash` as well as the pools `flash` and `disk` may change when data migration is started, like it is observed for our prototype. All bandwidth parameters are chosen such that the performance of our prototype is resembled. Note that the bandwidth is assumed not to depend on I/O patterns. This is a simplification, as sustainable bandwidth for a disk-based system depends heavily on the I/O request sizes. One big advantage of flash storage is that it can sustain close to peak bandwidth for much small I/O request sizes (and file system block sizes). From the JUGENE I/O statistics (see Fig. 1) we

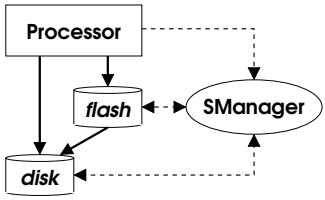


Fig. 6. Schematic view on the simulation model consisting of a processing device, a storage manager and 2 storage pools, `flash` and `disk`

extract the amount of data written and the time between write operations. With the additional pool `flash` disabled the execution times obtained from the model and the real execution times agree within 10 %. When enabling the write cache, the simulated execution time reduces at a 1 % level. This result is consistent with above model analysis as for the monitored time period $t_{\mathrm{comp}}/t_{\mathrm{io}} \gtrsim 300$, i.e. the system was not used by applications which are I/O bound. This we consider typical for current HPC systems since the performance penalty when executing a significant amount of I/O operations is large.

8 Discussion and Conclusions

In this paper we evaluated the functionality and performance of an I/O prototype system comprising flash memory cards in addition to a disk pool. We could demonstrate that the system is capable of sustaining high write and read bandwidth to and from the flash cards using a massively-parallel Blue Gene/Q system to generate the load. Taking into account that in standard user operation mode I/O operations occur in bursts we investigated how such an I/O architecture could be used to realise a tiered storage system where flash memory is used for staging data. We demonstrated how such a system can be managed using the policy rule mechanism of GPFS.

Our results indicate that for loads extracted from current I/O statistics some gain is to be expected just using the architecture as a write cache. The statistics are however biased as I/O bound applications are hardly using the system for performance reasons. Transferring data sequentially using large blocks does not meet the requirements of many scientific applications. A performance assessment of the proposed prefetching mechanism from disk to flash could not be carried out in the scope of this paper. It requires extensions to workflow managers and other software tools which make it easy for the user to provide information which files will be accessed for reading before or while executing a job (see RAMDISK [7] for a possible solution).

The considered hierarchical storage system comprising non-volatile memory has an even higher potential for performance improvements when the application's I/O performance is mainly limited by the *rate* at which read and write requests can be performed. For our prototype system we show that a high IOPS

rate can be achieved. We therefore expect our tiered storage system, e.g., to be particular efficient when being used for multi-pass analysis applications performing a large number of small random I/O operations.

Acknowledgements. Results presented in this paper are partially based on the project "MPP Exascale system I/O concepts", a joined project of IBM, CSCS and Jülich Supercomputing Centre (JSC) which was partly supported by the EU grant 261557 (PRACE-1IP). We would like to thank all members of this project, in particular H. El-Harake, W. Homberg and O. Mextorf, for their contributions.

References

[1] Gray, J., Shenoy, P.: Rules of Thumb in Data Engineering, pp. 3–10 (2000)
[2] Bell, G., Gray, J., Szalay, A.: Petascale computational systems. Computer 39(1), 110–112 (2006)
[3] Hitachi, `https://www1.hgst.com/hdd/technolo/overview/storagetechchart.html` (accessed: January 26, 2013)
[4] Stevens, R., White, A., et al. (2010), `http://www.exascale.org/mediawiki/images/d/db/planningforexascaleapps-steven.pdf` (accessed: January 26, 2013)
[5] Mueller-Wicke, D., Mueller, C.: TSM for Space Management for UNIX – GPFS Integration (2010)
[6] Miller, E.L., Katz, R.H.: Input/output Behavior of Supercomputing Applications. In: SC 1991, pp. 567–576. ACM, New York (1991)
[7] Wickberg, T., Carothers, C.: The RAMDISK Storage Accelerator: a Method of Accelerating I/O Performance on HPC Systems using RAMDISKs. In: ROSS 2012, pp. 5:1–5:8. ACM, New York (2012)
[8] Abbasi, H., Wolf, M., Eisenhauer, G., Klasky, S., Schwan, K., Zheng, F.: DataStager: Scalable Data Staging Services for Petascale Applications. In: HPDC 2009, pp. 39–48. ACM, New York (2009)
[9] Abbasi, H., Wolf, M., Eisenhauer, G., Klasky, S., Schwan, K., Zheng, F.: DataStager: Scalable Data Staging Services for Petascale Applications. Cluster Computing 13(3), 277–290 (2010)
[10] Kannan, S., Gavrilovska, A., Schwan, K., Milojicic, D., Talwar, V.: Using active NVRAM for I/O staging. In: PDAC 2011, pp. 15–22. ACM, New York (2011)
[11] Frings, W., Wolf, F., Petkov, V.: Scalable Massively Parallel I/O to Task-local Files. In: SC 2009, pp. 17:1–17:11. ACM, New York (2009)
[12] Borrill, J., Oliker, L., Shalf, J., Shan, H.: Investigation of Leading HPC I/O Performance Using a Scientific-application Derived Benchmark. In: SC 2007, pp. 10:1–10:12. ACM, New York (2007)

VM-MAD: A Cloud/Cluster Software for Service-Oriented Academic Environments

Tyanko Aleksiev[2], Simon Barkow-Oesterreicher[1], Peter Kunszt[3], Sergio Maffioletti[2], Riccardo Murri[2], and Christian Panse[1]

[1] Functional Genomics Center Zürich
ETH Zürich / Universität Zürich
Winterthurerstrasse 190, CH-8057 Zürich, Switzerland
cp@fgcz.ethz.ch, simon.barkow@fgcz.uzh.ch
[2] Grid Computing Competence Center
Universität Zürich
Winterthurerstrasse 190, CH-8057 Zürich, Switzerland
{tyanko.alexiev,riccardo.murri}@gmail.com,
sergio.maffioletti@gc3.uzh.ch
[3] SystemsX, ETH Zürich
Clausiusstrasse 45, CH-8006 Zürich, Switzerland
peter.kunszt@systemsx.ch

Abstract. The availability of powerful computing hardware in IaaS clouds makes cloud computing attractive also for computational workloads that were up to now almost exclusively run on HPC clusters.

In this paper we present the VM-MAD *Orchestrator* software: an open source framework for cloudbursting Linux-based HPC clusters into IaaS clouds but also computational grids. The *Orchestrator* is completely modular, allowing flexible configurations of cloudbursting policies. It can be used with any batch system or cloud infrastructure, dynamically extending the cluster when needed. A distinctive feature of our framework is that the policies can be tested and tuned in a simulation mode based on historical or synthetic cluster accounting data.

In the paper we also describe how the VM-MAD *Orchestrator* was used in a production environment at the Functional Genomics Center Zurich to speed up the analysis of mass spectrometry-based protein data by cloudbursting to the Amazon Elastic Compute Cloud. The advantages of this hybrid system are shown with a large evaluation run using about hundred large Elastic Compute Cloud (EC2) nodes.

1 Introduction

Recent years have seen great advances in virtualization technologies, to the point that it is now possible to run computationally-heavy workloads on completely virtualized infrastructures. Starting with Amazon EC2, commodity on-demand virtualized compute infrastructures[1] have become affordable to anyone. They

[1] Commonly referred to as "Infrastructure-as-a-Service (IaaS) clouds".

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 447–461, 2013.

include virtualized compute and storage hardware, dedicated networking and a software stack entirely under control of the end-user.

Therefore, the use of virtualized computational infrastructures has become very appealing to smaller research groups: it is now possible to access large computational resources without the need to buy and maintain a corresponding hardware infrastructure.

Today, emerging computational disciplines (e.g., Bioinformatics, Medical informatics) are showing usage patterns that do not fit well in the traditional High-Performance Computing (HPC) model of few individual jobs making use of the entire infrastructure through massively parallel programming. Their model is to submit a very large number of small jobs in bursts to analyze the relevant data, and then post-process the results to get a statistical overview or model prediction. Their need for computational resources in terms of CPU hours is similar to the massively parallel HPC use-cases but without the need for low-latency networks for MPI communication. HPC resource providers, who need to support such user communities with transient "peak" workloads, cannot afford to plan the infrastructure for peak usage, as it would be underutilized for most of the time. At the same time, they do not want to see a negative impact on the traditional HPC cluster users either. Therefore, exploitation of cloudbursting to IaaS clouds for HPC is interesting also to small and mid-sized facilities.

The term "cloudbursting" describes the ability of a local computational resource facility to dynamically add virtual machine instances from IaaS providers to their local resource, extending it in size elastically as needed. Cloudbursting improves application throughput and response time as seen by the user. It is an efficient technique for dynamic HPC resources expansion and peak workload offloading.

Cloudbursting also allows to add cluster nodes to the local resource that extends it with new abilities to the benefit of the users. For example, it is possible to extend the local cluster with virtual nodes enabling Hadoop workloads, or special GPU workloads that are not supported locally.

In this paper we present the Virtual Machines Management and Advanced Deployment (VM-MAD) *Orchestrator* software: an open source framework for cloudbursting Linux-based HPC clusters into IaaS clouds. The VM-MAD *Orchestrator* is completely modular, allowing flexible configurations of cloudbursting policies in the Python programming language. It can be used with any batch-queuing cluster system or cloud infrastructure, dynamically extending the cluster when needed. The policies can be tested and tuned by using the VM-MAD *Orchestrator* in simulation mode, based on historical or synthetic cluster accounting data.

The paper is organized as follows. We first discuss the design goals of the VM-MAD *Orchestrator* and the architecture we devised to implement them (Section 2). In section 3 we take a more in-depth look at the implementation and discuss how cloudbursting policies are configured in VM-MAD. As a real-world use case example, we report on the usage of the VM-MAD *Orchestrator* to run some special ensemble jobs on the bioinformatics cluster at the

Functional Genomics Center Zurich (Section 4). Finally, we survey similar and concurrently-developed solutions (Section 5) and outline some conclusions and possible future developments (Section 6).

1.1 List of Acronyms

API	Application Programming Interface
CPU	Central Processing Unit
EC2	Elastic Compute Cloud
ECU	EC2 Compute Unit
ETHZ	*Eidgenössische Technische Hochschule Zürich*, Swiss Federal Institute of Technology Zurich
FGCZ	Functional Genomics Center Zurich
HPC	High-Performance Computing
IaaS	Infrastructure-as-a-Service
IBM	International Business Machines
LSF	Load Sharing Facility (a batch-queuing system)
SMSCG	Swiss Multi-Science Computational Grid
UZH	University of Zurich
VM	Virtual Machine
VM-MAD	Virtual Machines Management and Advanced Deployment (the project described in this paper)
VPN	Virtual Private Network

2 Overall Design and Architecture

The stated goal of the VM-MAD project was to build a stable software service that could be used on existing production-grade HPC cluster infrastructures to dynamically add computing power during peak loads, and to automatically revert to using only local processing facilities when the "rush hour" is over. This elastic "cloudbursting" feature should have as little impact as possible on the current usage patterns of HPC clusters; ideally, nothing should change in the HPC users' experience but the system would automatically launch cloud-based Virtual Machines (VMs) and schedule jobs that would otherwise not be possible or take too much time or resources out of the cluster.

2.1 Implementation Requirements

Early in the development process, we realized that achieving these goals entails dealing with large heterogeneity.

First of all, we would need to accommodate different batch-queuing systems, even if we are restricting ourselves to the HPC clusters in use at the University of Zurich (UZH) and the *Eidgenössische Technische Hochschule Zürich* (ETHZ). While they all share the same workflow and interaction models, details of the submission Application Programming Interface (API) vary greatly. This ruled

out the possibility of implementing the VM-MAD cloudbursting software as an extension package for a particular batch system implementation. Instead we decided to interact with the batch system via the available command-line tools.

The second very important consideration was the actual definition of what is meant by "peak load", i.e., under what conditions the computing power should be extended using VM instances from the cloud, and what kinds of jobs can be run on the elastic part of the infrastructure. Defining peak load is a subtle matter of local policy. Any choice of a domain-specific language would have constrained the range of supported policies and therefore limited the applicability of the VM-MAD cloudbursting system. We chose instead to allow the definition of the local policy as a set of functions written in the Python programming language [17]: now a decision can be taken on the basis of *all* the data available to the cloudbursting software (see details in Section 3).

Finally, the "cloud" ecosystem is currently very dynamic. For our software to be useful even just in the next few years, it needs to be able to interface to different IaaS cloud infrastructures.

Based on these requirements we opted for a completely modular architecture: the VM-MAD software is a *framework* for building cloudbursting scripts, perfectly adapted to the peculiarities of each HPC installation. It is not a ready-made add-on for a particular product that a systems administrator can deploy with just a few touches to a configuration file.[2]

2.2 Architecture Overview

Our solution to the "cloudbursting" problem, as outlined in the previous sections, is to build a tool which can be run as an add-on to existing batch systems. Our "Orchestrator" software implements the additional services needed to link the batch system with an elastic IaaS infrastructure. It' runs in the background[3] and performs the following tasks:

a. Monitor the jobs queued in the batch system, and select those that could run on a cloud-based VMs;
b. Start and shut down VM instances;
c. Add and remove VMs compute nodes to and from the cluster.

It is very important to remark that points *a.* and *b.* involve taking decisions according to *configurable policies and metrics*. The cluster system administrator is responsible for these policies and metrics.

Figure 1 shows the interaction of the software components involved in a cloudbursting scenario under control of a VM-MAD Orchestrator.

[2] This is similar to the current situation for the HPC scheduling softwares: low-maintenance schedulers are also limited in configurability and functionality, while those that are flexible enough to implement complex policies are often custom-built or have a rich and detailed configuration language.
[3] It is a "daemon" in the UNIX terminology.

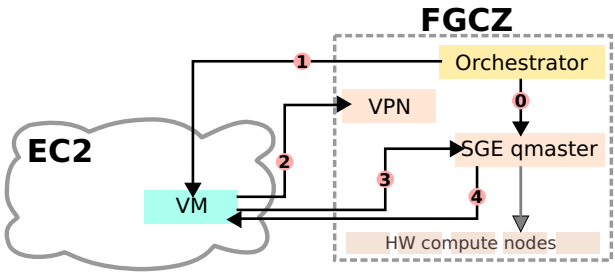


Fig. 1. Interaction of parts in a cloudbursting scenario. *(0)* The Orchestrator monitors the batch system state and determines when a new compute node is needed. *(1)* A new VM is started. *(2)* The VM connects back to the batch system network via VPN. *(3)* The VM is added to the cluster as a compute node. *(4)* The batch system can now start jobs on the VM.

(0) The Orchestrator monitors the batch system state and determines that —by the local policy definition— a new compute node is needed. (For example, the number of queued jobs that could be executed in a cloud-based VM exceeds a certain threshold.)
(1) The Orchestrator consults the cloud state and the local policy, and determines that the current set of cloud-based resources is insufficient. It therefore contacts the cloud provider via its network API and starts a new VM.
(2) The new VM connects back to the batch system network via a VPN. This requires that the VM image has been previously prepared by the cluster systems administrator: it should contain a the portion of the cluster execution environment that is necessary for running jobs destined to the cloud and the preconfigured VPN software to connect back to the "home" network.
(3) The Orchestrator adds the new VM to the cluster as a compute node, reconfiguring the batch system scheduler on the fly. All properties of this node are registered with the scheduler and jobs requesting those properties can be scheduled on the new cloud-based nodes.
(4) The batch system scheduler can now start jobs on the VM. It should be noted that the Orchestrator has a passive role with regards to scheduling computational jobs in the cloud: all it does is to start new VMs that satisfy the job requirements, and lets the batch system scheduler use those for actually running a job.
(5) When the Orchestrator detects that the amount of cloud-based resources exceeds the current needs (as defined by local policy), it shuts down the unneeded VMs.

3 Implementation Overview

The VM-MAD cloudbursting framework is implemented as a library package written in the Python [17] programming language. The code is written in an object-oriented style; the basic components of the framework are Python classes.

The *Orchestrator* object is the core of the framework: it implements the main loop and performs housekeeping of the shared data structures. The *Orchestrator* is a singleton: only one single instance should be monitoring a given batch system. An *Orchestrator* instance must be adapted to the cluster setup by initializing it with a *SchedInfo* and a *Provider* instance.

SchedInfo objects are responsible for interacting with the batch system scheduler, especially for gathering information about the running/queued jobs and the available compute nodes. New batch-queuing systems can be supported by creating an appropriate *SchedInfo* subclass.

Provider objects are responsible for interacting with a remote IaaS cloud system and starting/stopping virtual machines.

The cloudbursting policy is defined by subclassing the *Orchestrator* object and overriding well-defined methods that decide whether a job is a candidate for cloud execution, or what type of virtual machine should be started.

It should be noted that this simple component architecture allows a great deal of flexibility: for instance, a *Provider* instance needs not interface to a cloud provider, but can also request nodes from a peer cluster or Grid infrastructure.[4] Likewise, the *SchedInfo* component does not need to read information from a live batch system: the standard VM-MAD software distribution includes components for replaying job information from a batch system accounting file, which can be used for simulating the effect of cloudbursting policies over historical data, see section 4.3. It also includes components to generate random workloads to be used for testing of the system and available infrastructure.

3.1 Policy Definition

"Orchestrator policies" are criteria that govern decisions on whether:

1. a given job can run on cloud-based virtualized hardware;
2. a new VMs should be started to extend the current virtualized computational resource pool;
3. a running VMs should be stopped, shrinking the current virtualized resource pool.

For each of these decisions, a method is provided in the *Orchestrator* class that should return a *True/False* value based on the evaluation of available data. Systems administrators should override the default implementation to implement their chosen criteria.

Example: Policy on Jobs Eligible to Run on Virtualized Hardware. The decision whether a certain job can run on cloud-based resources is taken by the *is_cloud_candidate* method. This method is called once for each new job

[4] This has actually been done in the course of the VM-MAD benchmarks, by starting VMs [14,3] on the Swiss Multi-Science Computational Grid (SMSCG) [18] computational grid infrastructure.

that appears in the batch system queues and returns *True* if that job is eligible for cloudbursting. The default implementation always returns *False*, so that no job accidentally triggers the spawning of cloud-based VMs.

For example, the following code would implement a policy where only jobs that have been submitted to a special "cloud" queue trigger cloudbursting of compute resources:

```
1 def is_cloud_candidate(self, job):
2   return (job.queue == 'cloud.q')
```

The *job* record passed as argument to the *is_cloud_candidate* method contains all the information that the batch system scheduler provides via its queue-listing command (e.g., `qstat` on Sun/Oracle Grid Engine).

Example: Policy on Starting New Compute Resources. The decision on whether new cloud-based resources should be requested is taken by the *is_new_vm_needed* method. This method is called at each iteration of the Orchestrator's main loop. It has access to all the internal data structures, in particular the list of jobs eligible to run on cloud-based hardware (*self.candidates*) and the list of cloud-based VMs that have already been started by the Orchestrator (*self.vms*). By default, this method always returns *False*, so that cloud-based VMs are never spawned; this is a safety measure to avoid that non-configured Orchestrators start spawning VMs: since usage of cloud-based resources usually comes at a cost, it is entirely the administrator's task to decide when and how to initiate cloudbursting.

For example, the following code implements a policy where new cloud-based VMs are started if the number of queued candidate jobs is greater than double the number of VMs required to run them:

```
1 def is_new_vm_needed(self):
2   if len(self.candidates) > 2*len(self.vms):
3     return True
4   else:
5     return False
```

Example: Policy on Stopping Cloud-Based Compute Resources. At every iteration of the Orchestrator's main loop, a decision will also be taken on whether an idle VM (i.e., one that is not currently running any job) should be stopped. Since booting a cloud-based VM can take up to a few minutes' time, and many cloud infrastructure bill usage in hourly increments, it makes sense to try to re-use already-started VMs instead of starting new ones. The *can_vm_be_stopped* method is there exactly for this purpose: change the default Orchestrator behavior, which is to stop a VM as soon as it turns idle.

For example, the following code implements a policy where a VM is allowed to be idle for 10 minutes before it is stopped by the Orchestrator:

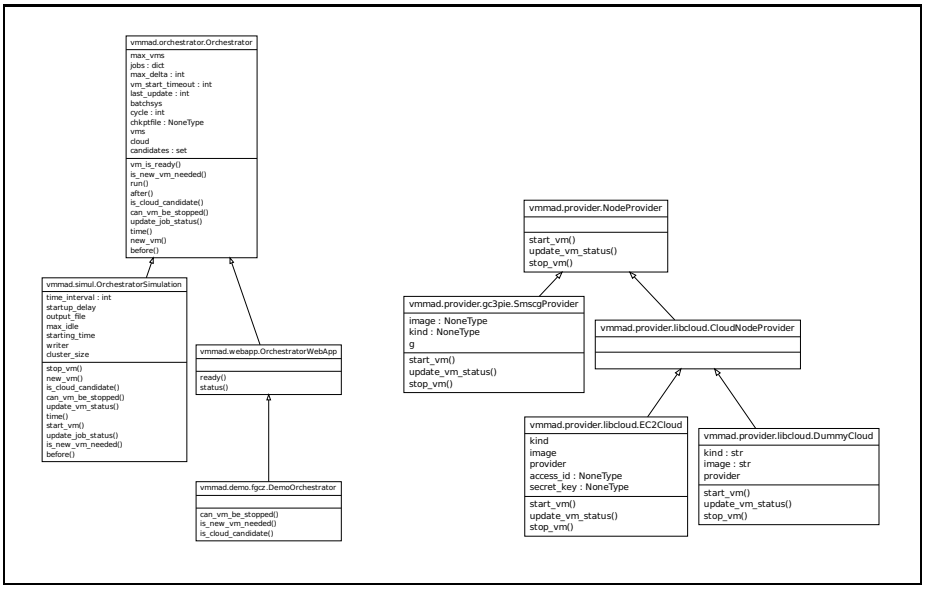


Fig. 2. *Left:* UML class diagram of the *Orchestrator* and its derived classes. The root of the hierarchy is the `vmmad.orchestrator.Orchestrator` class, which implements the main daemon loop and the core infrastructure for the VM-MAD functionality. Two derived classes are shown: the `vmmad.simul.OrchestratorSimulation` class is used to simulate running the VM-MAD software on historical accounting data; the `vmmad.demo.fgcz.DemoOrchestrator` is a an actual implementation of the VM-MAD Orchestrator for use on the FGCZ cluster. Note that `vmmad.demo.fgcz.DemoOrchestrator` derives from `vmmad.orchestrator.Orchestrator` through `vmmad.webapp.OrchestratorWebApp`, which implements a web interface for *Orchestrator* status reporting. *Right:* UML class diagram of the cloud interface classes. The root class `vmmad.provider.NodeProvider` defines the programming interface to which other classes must conform. Classes in the `vmmad.provider.libcloud` package implement interfaces to different IaaS cloud stacks using the Apache LibCloud library. The `vmmad.provider.gc3pie.SmscgProvider` draws nodes from clusters participating in the SMSCG computational grid infrastructure; it is an example of how VM-MAD can be interfaced to non-cloud infrastructures.

```
def can_vm_be_stopped(self, vm):
  TIMEOUT = 10*60 # 10 minutes
  if vm.last_idle > TIMEOUT:
    return True
  else:
    return False
```

4 Application and Testing

For testing we have chosen an area we have a lot of expertise in. Identifying proteins in a biological sample with the help of large computer systems is a common application in the life sciences which behaviour is well studied so that we have enough experience with all parameters and configuration details, e.g. memory consumption, input-output, and stability.

4.1 Test Case: Analyzing Mass Spectrometric Related Protein Data

The processing of mass spectrometry data can be challenging as it involves several computationally demanding algorithmic steps. Examples are the peptide spectrum assignment of mass spectrometry data to identify proteins in a biological sample, as well as the detection and identification of post-translational modifications of proteins. Both tasks can be computed simultaneously and can easily occupy hundreds of Central Processing Units (CPUs) for several days.

With every new mass spectrometer, the amount of measured data increases and the local computing infrastructures would need to be extended accordingly. However, these computing resources are only needed for a short period of time. The computation demand varies widely with the actual measurement type and the corresponding data set size. To be able to meet also larger use-cases, the available local cluster would need to be very large and powerful, but then it would be mostly under-utilized. Therefore such large use-cases are not feasible as currently the capacity cannot be extended on demand.

Large-scale so-called "shotgun experiments" with complex samples from, e.g., human or fruit fly involve about ten thousand proteins. The peptide spectrum matches for our test were computed with the SEQUEST and OMSSA search algorithm [9,10]. To benchmark the VM-MAD Orchestrator, we have run the search on both the local HPC cluster facility at the UZH, as well as on the Amazon EC2 Cloud computing resources in the Amazon region US-East. To avoid denial of service like failures on the cloud system, e.g., during file server and authentication operations we started our virtual machines in a staggered manner with a delay of 60 seconds. In order to avoid problems like hanging processes, that might be caused by a high latency of the network connection (e.g. for accessing a network filesystem), each of our compute jobs is responsible for dealing with its own input and output data.

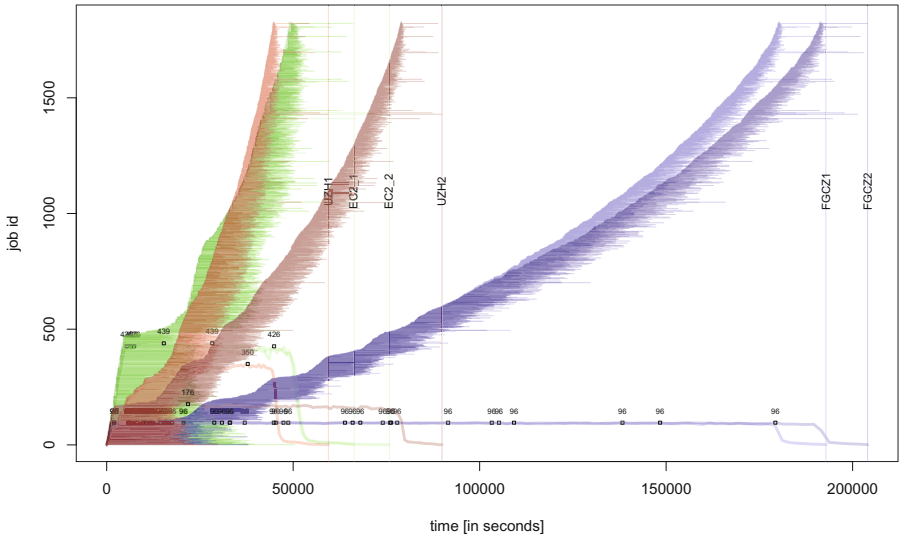


Fig. 3. Overview of the benchmark. On the utilization graph each horizontal line indicates the start and end of each job. The graph shows that the lines for the two jobs runs on the cloud (green) have almost the same length (we cannot distinguish 2 green branches) while the running times on the cluster nodes (red and blue) differ more significantly (the two repetitions are clearly distinguishable). This can be explained by the variable queue status of the cluster nodes because of other users using the cluster at the same time. Also, it takes much longer to run through all jobs on the limited FGCZ cluster (blue). The lines in the lower part of the graphic show the total number of concurrently running jobs. The squares on those lines indicate the maxima on the respective system.

4.2 Effectiveness: Benchmark of a Real World Data Set

As a test data set we used a large scale proteomics *Drosophila* (fruit fly) experiment [4] consisting of 1800 (LC)-MS/MS runs, having a peptide mass window of 3 Dalton, and 8474960 tandem mass spectra. We identified 498000 redundant peptides, 72281 distinct peptides, and 9124 proteins using the peptide spectrum match parameters as described in [4]. The data input volume is approximately 0.3TB split into 1800 jobs. The whole experiment data and the graphics are included in the *cloudUtil* R-package [16]. In our benchmark we compare three compute systems: the small cluster at the FGCZ consisting of around 100 CPUs, a larger system as part of the Schroedinger cluster of the UZH, and a virtual cluster on Amazon EC2. For benchmarking we recorded network bandwidth, CPU performance (compute time) and robustness on all systems. An overview of the experiment and a relative comparison of each compute job can be seen in the utilization plot on Figure 3.

The box plots [5] in Figure 4 show a comparison of the run time and the network throughput on three compute systems having two repetitions. One EC2

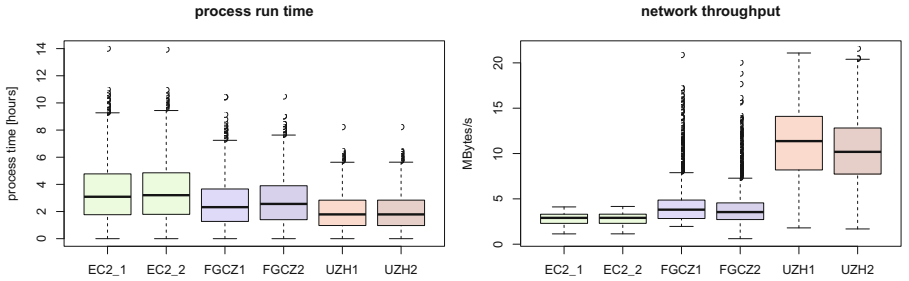


Fig. 4. Comparison run time and copy input I/O – The box plots [5] display the job run time distributions of the two repetitions of all three compute systems (left) and the copy I/O network throughput (right).

Compute Unit (ECU) provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor. The FGCZ cluster is based on Intel Xeon CPU E5450 3.0GHz and the UZH cluster is based on Intel Xeon CPU 5500.

4.3 Simulation of LRMS Accounting Data

For demonstrating the effectiveness of the Orchestrator software and to study the behavior of the Orchestrator policies for different hardware scenarios (i.e. number of nodes) and different accounting data we have implemented a simulation mode policy. If this policy is used the *Orchestrator* takes the batch-system accounting information and the cloud configuration file as input. The accounting file of the LRMS contains the ordered start time-stamps of every compute job and its corresponding run time. The configuration parameters are the time step argument (in seconds), the start time of the simulation, the maximum number of available hosts. Instead of orchestrating real nodes the Orchestrator writes all decisions about starting or stopping virtual machines to an output file. The visualization in Figure 5 shows the simulated state of the LRMS queue and the status of the VMs over time for different simulation runs. In particular, the plot on the bottom of Figure 5 displays the output of the following command line:

$$simul.py\ --time\text{-}interval\ 30\ --start\text{-}time\ '2008\text{-}12\text{-}16\_02{:}13{:}50\_CET'\ \backslash$$
$$--max\text{-}vms\ 512\ --cluster\text{-}size\ 100\ --csv\text{-}file\ accounting.csv$$

The simulation mode can also be used for determining the optimal number of compute nodes for a given task; see. e.g., Figure 5.

5 Related Work

Cloudbursting, as a compute model where local resources elastically allocate cloud instances for improving application throughput/response time, was first proposed by Amazon's Jeff Barr [7]. There is a variety of different mechanisms for cloudbursting an on-premise computational cluster to an external cloud provider:

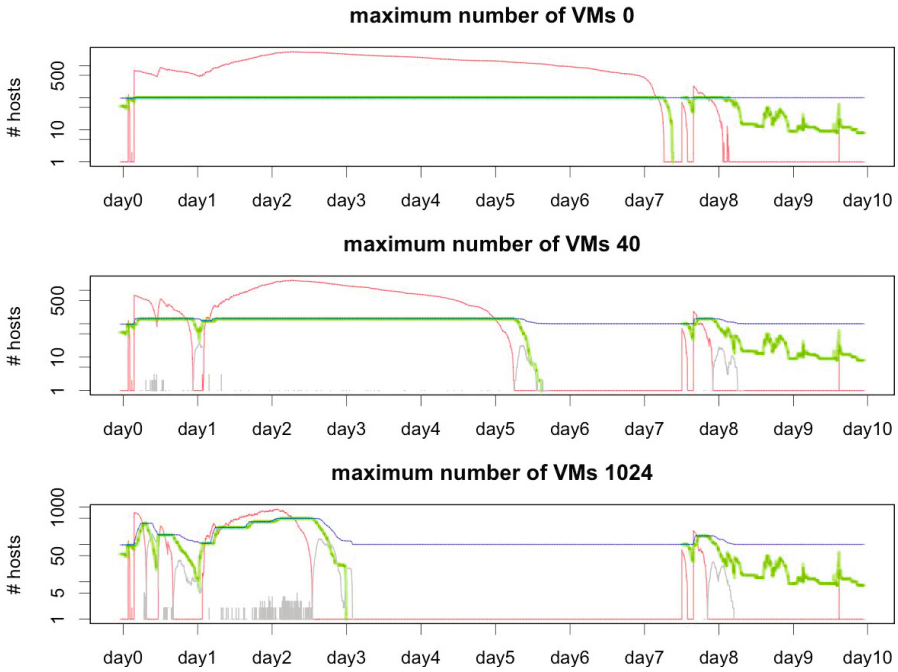


Fig. 5. The graphics show the simulation of 10 days of FGCZ batch cluster accounting data. The vertical axis showing the number of hosts/jobs is $\log_{10}$-scaled. The colored lines have the following meaning: *(red)* pending jobs; *(green)* running jobs; *(blue)* available nodes to the cluster (100 plus VMs); *(grey)* idle VMs. The upper simulation run corresponds to the FGCZ setup of 100 CPUs. For the simulation depicted in the lower graphic we added on demand up to 40 and 1024 VMs. For the computation we have submitted the described proteomics data set. It can be seen that with increasing number of VMs the overall compute time can be reduced to several days.

the most common derives from the HTCondor glide-in model [8] that is used to add a machine running on an external provider to an existing HTCondor pool. HTCondor glide-in configures a remote resource such that it reports to and joins the local HTCondor pool. This is the technology used for example by CycleComputing.com.

Inspired by this model, workload management systems that do support cloudbursting, like Sun/Oracle Grid Engine [19], Moab [13], or the HTCondor CloudScheduler [6], allow to start a pre-configured virtual instance, that can reside on an external cloud provider, and let it join the pool of resources they control. While VM-MAD takes an open approach in providing cloudbursting capabilities that could be adapted to virtually any workload management system thanks to its plug-in based approach, Grid Engine and Moab do provide a vendor-specific solution based on policies and configurations that cannot be applied nor ported to other similar systems.

Another approach in supporting cloudbursting is provided by the Multi-Cluster [12] solution from IBM's Platform Computing. An existing on-premise Load Sharing Facility (LSF)-controlled cluster could be extended by starting an entire LSF cluster on a cloud provider and use Multi-Cluster to federate them The main limitation of these approaches is the lack of an automatic system to start and control an LSF cluster on a cloud provider.

In terms of cloudbursting out of applications, Software-as-a-Service solutions make use of IaaS clouds to assure their workloads are scaled properly. An example in the life science domain is the Galaxy CloudMan project [1,2]. Here the Galaxy portal makes use of cloud resources to extend the support for selected computational workflows. The CloudMan system, that is tightly coupled to the Galaxy portal, provides a pre-selected set of tools and services as well as the possibility of deploying own software tools and integrate them through a web interface. While Galaxy targets the sequencing community, with ProteoCloud [15], there exists also a cloud computing pipeline for proteomics applications but it does not feature automatic cloudburst functionality.

In contrast to the these specialized frameworks, our approach is not limited to life-science applications. Any community-specific portal that is already capable of using on-premise computational clusters could seamlessly profit from clouds by deploying the VM-MAD *Orchestrator*.

An example of cloudbursting from an on-premise cloud infrastructure to an external provider is brought by the Seagull project [11]. Seagull dynamically decides which running applications can be moved from the on-premise cloud infrastructure to the configured external provider, using an Intelligent Placement module based on a placement algorithm that picks those applications to move that free up the most units of local resources relative to their cost of running in the cloud using a pre-defined cost function. To reduce cloudbursting latency (due to the copying of the disk image corresponding to the selected running application), Seagull performs pre-copying by transferring an incremental snapshot of a virtual machine's disk-state to the cloud. Seagull focuses on cloud-to-cloud cloudbursting features, whereas VM-MAD allows to cloudburst a batch-controlled computational cluster; Seagull takes autonomous decisions on what running applications to migrate live; on the other hand, VM-MAD has a simple policy module to determine whether to launch new appliances on the connected cloud provider.

6 Conclusions and Future Work

In this work we have described the architecture, the implementation, and an application use case of the *Orchestrator* cloudbursting software framework developed by the the VM-MAD project. The *Orchestrator* allows an existing compute cluster to be extended, burst into the cloud based on a highly configurable set of local policies. We have successfully extended local clusters with Amazon EC2 instances. In the future we also want to run Hadoop applications on the virtual part of the cluster, enabling MapReduce applications for our local users. Also,

connecting to non-public clouds, e.g., in-house OpenStack IaaS is possible, as well as to extend to other batch cluster systems.

The *Orchestrator* is a modular framework that can be interfaced with any batch-queuing system and IaaS cloud infrastructure. An interesting consequence of this modularity is that the *Orchestrator* can also be run in *simulation mode*, to allow testing cloudbursting policies against historical accounting data and evaluate the most cost-effective one. We will try to optimize the usage of historical data in the future to suggest good predefined policies to system administrators.

As a test and benchmark, we have used the VM-MAD *Orchestrator* for reprocessing a large set of proteomics data; the performance data collected show that commercial IaaS clouds can already deliver computational and network performance comparable to what is offered by a small in-house cluster, and are thus suitable for offloading peak computational workloads. We will also use the same concept with other workloads, like computational chemistry and structural biology, but also in the domains of geography and finance.

References

1. Afgan, E., Baker, D., Coraor, N., Chapman, B., Nekrutenko, A., Taylor, J.: Galaxy CloudMan: delivering cloud compute clusters. BMC Bioinformatics Supplements 12, S4 (2010)
2. Afgan, E., Baker, D., Team, T.G., Nekrutenko, A., Taylor, J.: A reference model for deploying applications in virtualized environments. Concurrency Computat.: Pract. Exper. 24, 1349–1361 (2012)
3. AppPot (January 2013), `http://apppot.googlecode.com/`
4. Brunner, E., et al.: A high-quality catalog of the drosophila melanogaster proteome. Nature Biotechnology 25(5), 576–583 (2007)
5. Cleveland, W.S.: Visualizing Data. Hobart Press, Summit (1993)
6. Cloud scheduler — your htc jobs on the cloud (January 2013), `http://cloudscheduler.org`
7. Cloudbursting, hybrid application hosting (2008), `http://aws.typepad.com/aws/2008/08/cloudbursting-.html`
8. HTCondor glide-in, `http://www.uscms.org/SoftwareComputing/Grid/WMS/glideinWMS/`
9. Eng, J.K., McCormack, A.L., Yates III, J.R.: An approach to correlate tandem mass spectral data of peptides with amino acid sequences in a protein database. Journal of the American Society for Mass Spectrometry 5 (1994)
10. Geer, L.Y., Markey, S.P., Kowalak, J.A., Wagner, L., Xu, M., Maynard, D.M., Yang, X., Shi, W., Bryant, S.H.: Open Mass Spectrometry Search Algorithm. eprint arXiv:q-bio/0406002 (June 2004)
11. Guo, T., Sharma, U., Wood, T., Sahu, S., Shenoy, P.: Seagull: intelligent cloud bursting for enterprise applications. In: Proceedings of the 2012 USENIX Conference on Annual Technical Conference, USENIX ATC 2012, p. 33. USENIX Association, Berkeley (2012)
12. IBM Platform Computing - Multicluster, `http://www-03.ibm.com/systems/technicalcomputing/platformcomputing/products/symphony/index.html`
13. Moab Cloud solution, `http://www.adaptivecomputing.com/home/cloud/`

14. Murri, R., Maffioletti, S.: AppPot: bridging the Grid and Cloud worlds. In: EGI Community Forum, PoS(EGICF12-EMITC2)004 (2012), http://pos.sissa.it/
15. Muth, T., Peters, J., Blackburn, J., Rapp, E., Martens, L.: ProteoCloud: A full-featured open source proteomics cloud computing pipeline. Journal of Proteomics (January 2013)
16. Panse, C., Qeli, E.: cloudUtil: Cloud Util Plots, R package version 0.1.9 (2012), http://CRAN.R-project.org/package=cloudUtil
17. Python Programming Language — Official Website (January 2013), http://www.python.org/
18. Swiss multi-science computing grid (January 2013), http://www.smscg.ch/
19. Sun/Oracle Grid Engine Cloud, http://www.oracle.com/technetwork/oem/cloud-mgmt/index.html

Federating HPC Access via SAML: Towards a Plug-and-Play Solution

Jens Köhler, Michael Simon, Martin Nussbaumer, and Hannes Hartenstein

Karlsruhe Institute of Technology (KIT), Steinbuch Centre for Computing (SCC), Karlsruhe, Germany
{jens.koehler,simon,nussbaumer,hartenstein}@kit.edu

Abstract. Many potential users hesitate to use HPC resources due to sometimes complex procedures that are necessary to get access. Furthermore, HPC providers need up-to-date identity information to make correct access control decisions. Federated identity management addresses both issues by enforcing access control based on the users' familiar accounts at their home organizations. SAML-based federations consisting of home organizations and web-services are already established, but the integration of non web-based services such as HPC resources is not trivial due to the absence of a browser as a user client or missing trust between web-portals and HPC resources. In this paper, we propose a concept that enables non web-based services to join SAML-based federations. From the service's point-of-view, our approach is transparent and appears to be a local LDAP directory. From the federations point-of-view, our approach can be integrated like an ordinary SAML service provider. Due to this separation of concerns, integration effort is considerably reduced. Furthermore, we will show how our approach can be extended to enable federated access to semi-trusted web-portals.

1 Introduction

The demand of many scientific communities for grid, cluster or high performance computing (HPC) resources in general is growing. However, many potential users hesitate to use existing computing resources due to complex registration processes to get access to the resources. Many HPC facilities require the users to physically identify themselves at the facility to create a new user account with additional credentials to be memorized. Furthermore, to access grid resources the users often have to manage cryptographic certificates instead of being able to use their "standard" accounts. From a different perspective, a major drawback of current access management approaches is potentially stale identity information at the HPC provider. For instance, consider an affiliate of a university that created an account (or a certificate) to access HPC resources. If the affiliate loses it's affiliation it might also lose the right to access the HPC resources. However, since the created service-local account (or the certificate) might be independent of the home-organizational account at the university, the former affiliate is still able to access the resources.

J.M. Kunkel, T. Ludwig, and H.W. Meuer (Eds.): ISC 2013, LNCS 7905, pp. 462–473, 2013.

The concept of federated identity management can be used to couple home-organizational and service-local accounts and to enable users to access HPC resources via their approved account at their (trusted) home organization. As the authentication of users is performed by the identity provider component hosted by the home organization, users can use their already established credentials in a single sign-on (SSO) fashion. Furthermore, the authorization decision of the HPC provider can be based on assertions issued by the home organization, allowing for active checking whether the user still has the properties required to access the service. For web-based services, federations based on the Security Assertion Markup Language (SAML) standard [Hug2005] are already well established. Integrating HPC resources with these existing federations would benefit both the users in terms of usability and the service providers being relieved from the overhead of managing identity data.

In this paper, we show how existing operational non web-based services can be connected to established SAML federations. Our main focus lies on minimizing the integration effort with existing service deployments and the usability from a user's perspective. We pursuit a *separation of concerns approach*: from a service provider perspective our approach can be used like a common local *Lightweight Directory Access Protocol* (LDAP) server, from the SAML federation's perspective our approach cannot be distinguished from an ordinary SAML service provider (see Figure 1). Thus, given that LDAP servers already are prevalent components for local user management [Li2011], integration effort is considerably reduced. Furthermore, we introduce a concept to federate access to semi-trusted web portals, i.e., web portals that must not get access to long-term user credentials. Our approach neither requires the user to manage certificates nor does it require a trusted proxy certificate store.

Our approach allows the following setup to manage user accounts and access rights: The home organization manages the user's home organizational account by maintaining the user's identity data and setting authorization tokens as appropriate. For instance, a university might only allow a subset of persons to use HPC resources by setting an authorization token `HPCflag` to `true`. The service provider manages a service-local account for each user that contains service-local authorization tokens and service specific attributes such as the users home-directory. For instance, some HPC facilities require users to describe their intended use of the service and use this description to approve the user for certain resources by setting the according authorization tokens. These service-local authorization tokens are additionally utilized to the authorization tokens provided by the user's home organization to make an authorization decision. Thus, both the account managed by the home organization and the service-local account managed by the service provider have an impact on the authorization decision.

The contributions of this paper are:

- An **concept to *federate* access to arbitrary operational services** that rely on a generic LDAP interface for user authentication and authorization.
- A **concept to *federate* access to semi-trusted web portals** that can be used to access HPC resources.

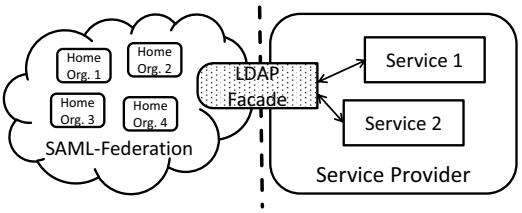


Fig. 1. Separation of concerns

– An **evaluation** of the proposed concept in terms of ease of integration and usability.

The paper is structured as follows: First, we present related work in Section 2 and define our trust model(s) in Section 3. We introduce preliminary work in Section 4, present our advanced contributions in Section 5 and show how web-portals can be integrated in Section 6. Finally, the concepts are evaluated in Section 7 and the paper is concluded in Section 8.

2 Related Work

Many previous proposals focused on integrating GridShib[1] to federate access to grid portals. GridShib enables user authorization based on attributes issued by the home organization of the user [Bar2006] [Gri2007]. However, authentication on the grid itself is still based on certificates that have to be maintained. GridShib is already used within TeraGrid [Bas2010] and the UK National Grid Service [Spe2006] to federate access to grid web portals. Both approaches utilize MyProxy[Nov2001] repositories to store and retrieve the users' (proxy) certificates that are necessary to submit jobs to the grids on behalf of the user. In contrast to our approach, users still have to go through (complex) certificate registration procedures and have to upload the certificate to the MyProxy repository manually. In [Wan2009] [Mur2011] enhancements that automatically generate and manage certificates after the user authenticated to the web portal in a federated manner via SAML are proposed. However, a potential attacker that compromises either a home organization or the central MyProxy repository can get access to credentials that yield access to the grid. Our approach does not need a central credential repository and only has to rely on the trustworthiness of the users' home organizations. Furthermore, contrasting to the introduced contributions, we show how to federate access via web portals *and* console access (e.g., via SSH).

Similar to our work, Project Moonshot[2] also has the aim to federate access to non web-based services. However, Project Moonshot focusses on integrating the services in *eduroam*[Mil2008] federations. The use of SAML to retrieve attributes

[1] `http://gridshib.globus.org/`
[2] `http://project-moonshot.org`

on users from the home organizations is optional and additionally requires an established SAML federation besides the eduroam federation. Furthermore to federate SSH services, both modified SSH clients and modified SSH servers are needed. Project Moonshot aims at establishing those modified software components as "standard" components of major operating systems. Until this happens, however, integrating the approach into operational services is hard. Our approach only requires an established SAML federation. No modified SSH server is necessary and modified SSH clients are optional.

We presented preliminary work to this paper in [Koe2012] under the name FACIUS. While FACIUS also enables non web-based services to join SAML federations, it was limited to services supporting the pluggable authentication module interface. The concepts in this paper extend previous work by allowing to federate arbitrary services based on an LDAP facade that can be used like a local LDAP directory and by addressing the case of federating access to semi-trusted web portals.

3 Trust Model

Traditional service clients (e.g., an SSH client) implement user authentication against the service provider rather than the home organization. Thus, unmodified clients inherently send the users password to the service provider which uses the password, for instance, to authenticate the user against an LDAP server. User passwords can be used to impersonate users, steal data or make unauthorized use of ressources and are therefore highly sensitive. As service providers are not entirely trusted in all scenarios, we differentiate between two trust models:

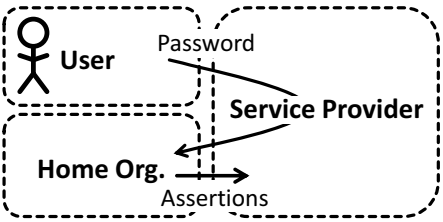


Fig. 2. Utilization of the home-organizational password in the full trust model

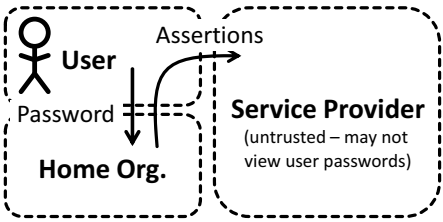


Fig. 3. Utilization of the home-organizational password in the limited trust model

– **Full Trust:** The service providers are trusted in the sense that they are treated as if they are part of the home-organization. In particular, service providers may view user passwords. Our approach leverages that and enables the users to log on to a service provider via the password of the home organizational account using traditional, unmodified service clients. A high level overview of the password utilization in the full trust model is shown in

Figure 2. The user client sends the users password to the service provider in the same way it would in the unfederated case. The service provider then uses the password to authenticate the user against the home organization and retrieves *Assertions* that contain both authorization tokens and identity information in return.

– **Limited Trust:** The service providers are untrusted in the sense that they may not be able to observe user passwords (which would enable them to impersonate users). However, they may still access the user's identity information and authorization tokens that are issued by the home-organization. Should a limited trust model be assumed for a given scenario, unmodified service clients can still be used if the user authenticates by other means than the home organizational password (e.g., via challenge response public-key authentication or a service provider specific password). To authenticate via the home organizational password in the limited trust model, however, modified service clients are necessary that pass the password to the home organization instead of the service provider. A high level overview of the password utilization in the full trust model is shown in Figure 3. The modified service client passes the user's password to the home organization and forwards the issued Assertions to the service provider to allow for an authorization decision.

Thus, while our approach can also be applied in the limited trust model, it offers a higher user convenience in the full trust model. A use-case for the full trust model is the bwIDM project from which this work originated. In this project, both the identity providers and the service providers, are hosted by regionally connected universities of the state of Baden-Württemberg (Germany) that trust each other in handling user credentials, but have distinct identity management systems.

4 The FACIUS Approach

The FACIUS approach enables users to get federated access to non web-based services. It focusses on SAML federations and therefore makes extensive use of several SAML profiles [Hug2005] that constitute use-cases defined within the SAML standard. SAML profiles are already implemented in conventional software like Shibboleth[3] or simpleSAMLphp[4]. The profiles FACIUS relies on are:

– **WebSSO:** A profile that allows users to get federated service access via their web-browser that makes active use of HTTP-redirects.
– **Enhanced Client or Proxy (ECP):** A profile that can be implemented to federate access via clients that do not offer the features of a web-browser.
– **AssertionQuery:** A profile to enable services to query the home organization for assertions on users in an offline fashion, i.e., without user participation. In particular, this profile can be used to request up-to-date assertions to

[3] `http://shibboleth.net/`
[4] `http://simplesamlphp.org/`

decide whether a user that authenticated via credentials not known the home organization such as deployed SSH-publickeys still fulfills the conditions to use the service or not.

FACIUS allows users to register for a local account at the service by authenticating at a *registration web-application* and accepting policies of the service provider such as acceptable-use-policies. The authentication is performed in a federated manner via the WebSSO profile. The established local account contains service specific information such as the UID or the home-directory of the user.

The ECP profile is used to enable the users to log on in a federated manner using non web-based service clients (e.g., an SSH client). The AssertionQuery profile can be used by the service provider to retrieve up-to-date assertions for users authenticating via credentials that are not known to the home organization (e.g., deployed SSH-publickeys) and for deprovisioning purposes, i.e., to check whether a local account can be deleted because of a user not being authorized to use the service anymore. FACIUS implements the main parts of the ECP/AssertionQuery logic in a pluggable authentication module (PAM) that constitutes a prevalent interface on UNIX systems. In this paper we will extend the approach to be integratable with an even bigger variety of services that are LDAP compatible and show how the logic can be implemented in an *LDAP facade*, i.e., a component that provides an LDAP interface.

5 LDAP Facade

The LDAP facade we propose in this paper integrates the FACIUS logic including the web registration application and offers a generic LDAP interface to services that should be federated. The LDAP facade constitutes a single component that needs to be installed at the service provider. It makes use of the *local accounts* that have been previously established during the user registration at the registration web-application. Services can use the LDAP interface like a traditional LDAP server. The general login process with an unmodified service client is depicted in Figure 4. To enable a secure login via the users' home organizational passwords in the limited trust model, a modified service client is needed (cf. Figure 5).

Using an **unmodified client** (cf. Figure 4), the user enters her/his username and password at the client (e.g., an SSH client) which passes the credentials to the login node of the service provider (e.g., an SSH server) (1). The login node then authenticates the user against the LDAP interface of the LDAP facade in an ordinary fashion via LDAP bind (2). The LDAP facade uses the passed credentials to authenticate against the *identity provider* component (IdP) of the home organization of the user on behalf of the user and retrieves assertions on the user via the SAML ECP profile (3). Based on these assertions, an authorization decision can be made, the local account data of the user can be retrieved (4) and the decision along with the account data can be passed to the login node (5) which grants access to the user.

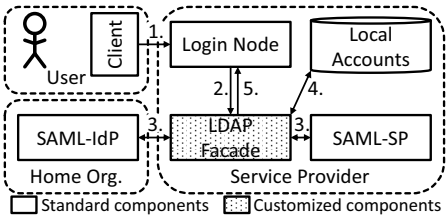


Fig. 4. General login workflow (except via home organizational password in the limited trust model)

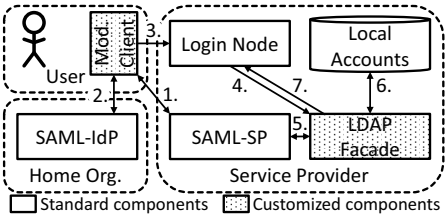


Fig. 5. Login via home organizational password in the limited trust model

While this process is considered secure in the *full trust* model, passing the user credentials to the service provider violates the *limited trust* model. However, an unmodified service client can still be used in the *limited trust* model if the user is authenticated by the service provider rather than the home organization. This can be done either by authenticating via challenge response (e.g., via a public SSH-key that has been deployed during the registration) or via a password that is service provider specific and set during the registration of the local account. In case the user authenticates to the service provider, to retrieve recent user attributes from the home organization, the AssertionQuery profile of SAML is utilized in step 3 in Figure 4.

To enable users to use the passwords of their home organization account to authenticate in the *limited trust* model, they have to use a **modified client** (cf. Figure 5). A modified client can execute the ECP logic itself and retrieves a SAML authentication request from the SAML-SP (1). The password is then send to the home organization (2), which issues assertions that are passed to the login node (e.g., an SSH server) together with the username (3). To enable this without modifying the login node, the assertions can be wrapped into the expected password if the login node supports long passwords. The login node then performs an ordinary LDAP bind with the LDAP facade and passes the username as well as the wrapped assertions (4). The LDAP facade authenticates the user by passing the assertions to the SAML-SP that decrypts and verifies them (5). Based on the contained attributes, the LDAP facade makes an authorization decision, retrieves the local user account (6) and passes the decision along with the account data to the login node (7).

6 Integration of Web-Portals

Federating access to web portals that act as a proxy between the user and the HPC resources has to be further investigated, as in many cases the HPC computing resources do not trust portals entirely. This can be due to the complex and potentially error prone structure of the portal code or due to third parties being able to integrate custom code into the portal. To limit the impact of a compromised portal, as of now, short-lived credentials (e.g., self-signed, fast

expiring certificates) are issued by users and passed to the portal. These short-lived credentials assert that the user successfully authenticated her-/himself at a specific point in time and can be used by the portal to access the computing resources if this point in time is not past longer than a certain timespan. However, using self-signed certificates as short lived credentials requires the user to manage certificates or delegate this task to a trusted party (cf. MyProxy [Nov2001]).

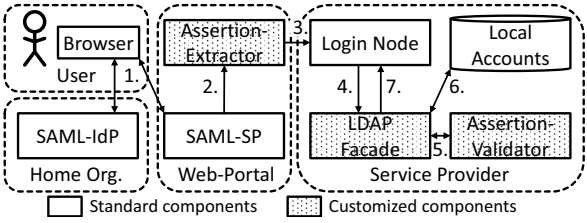


Fig. 6. Login via web-portal

Instead of requiring the user to sign short-lived credentials, our approach enables secure web-portal access by relying on SAML assertions that contain a timestamp and that are signed by the IdP. Thus, assertions that assert a successful authentication of the user at the IdP constitute short-lived credentials. Therefore, HPC access via portals can be federated using SAML as depicted in Figure 6: First, the user is authenticated via an ordinary SAML-WebSSO login, i.e., when accessing the portal, the user is redirected to her/his IdP to authenticate and is then redirected back to the SP carrying assertions issued by the IdP asserting the successful authentication (1). After the *Assertion-Extractor* extracted and decrypted the assertions from the SAML response of the IdP (2), the decrypted assertions can be used as short-lived credential to authenticate to the computing resources (3). This can be done without modifying the Login-Node (e.g. an SSH server) by wrapping the assertions into the submitted password. To verify the passed credentials, the Login-Node performs an LDAP bind against the LDAP facade using the transmitted username and password (4). In turn, the LDAP facade extracts the assertion from the password and passes it to the *Assertion-Validator* (5) which checks if the assertion's timestamp is up-to-date and verifies the IdP's signature. If the assertion could be successfully validated, the LDAP facade retrieves the local state of the user (6), signals the Login-Node that the user has been successfully authenticated and passes the local state (7). Finally, the Login-Node grants access to the local account of the user that has been extracted from the local state. The computing resources can itself use the conveyed assertions to access other computing resources on behalf of the user. Thus, even job-traveling can be realized (as long as the assertion is deemed up-to-date).

Using this concept to federate semi-trusted web portals, only the patterned components in Figure 6 need be implemented. It is not necessary to adapt existing SAML-IdPs or SAML-SPs that support the WebSSO SAML profile. Furthermore, the Login-Node of the computing resources (e.g., the SSH server) does

not have to be adapted if it supports authentication against an LDAP directory and allows for long passwords.

7 Evaluation

Regarding **deployability**, our approach does not require any software adaption on login nodes of the service provider that can authenticate users against a common LDAP server. In some cases, a service local identity management system such as an LDAP server that contains local accounts may already exist. There are two ways to seamlessly integrate the LDAP facade:

- by migrating all contents of the existing LDAP directory into the underlying database of the LDAP-Facade where the local user accounts are stored (see Figure 7). In our current implementation this database is an LDAP directory itself which facilitates migration. The LDAP facade is capable of distinguishing whether an account is linked to a home organization account or not.
- by linking the LDAP facade to the existing LDAP server via LDAP proxying (see Figure 8). For instance, with OpenLDAP this is possible using the slapd-meta backend[5]. Thus, users that do not exist in the existing LDAP directory are automatically authenticated against the LDAP facade. For LDAP servers that do not support this functionality, an OpenLDAP server can be set up that acts as a proxy for both, the LDAP-Facade *and* the existing LDAP.

In both cases, offering federated access to resources is completely transparent for the login nodes of the service provider and no additional software components are required.

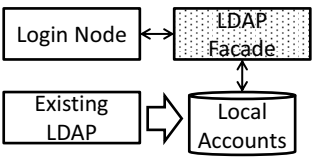


Fig. 7. Integration of the LDAP facade via account migration

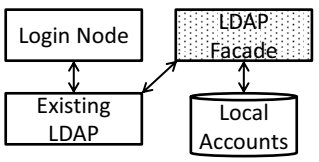


Fig. 8. Integration of the LDAP facade via LDAP proxying

From the federations point-of-view, the service provider can be integrated into the federation in an ordinary fashion. Home organizations of the federation do not have to adapt their identity provider components and only have to support the WebSSO, ECP and AssertionQuery profiles as defined in the SAML standard. For instance, identity providers using the Shibboleth framework support all three of them. From a user's perspective, it is possible to use familiar, unmodified service clients in most cases, as Table 1 illustrates. The only exception is the

[5] `http://linux.die.net/man/5/slapd-meta`

case of user logon via password of the home organization using non web-based clients. If federation partners are not trustworthy enough to receive user passwords of the home organization (limited trust model), clients that traditionally transmit credentials to the service provider rather than the home organization have to be modified to enable a secure logon.

Table 1. Login scenarios supported by unmodified service clients

	Full trust	Limited trust
Login via home org. password (non web-client)	✓	mod. client needed
Login via home org. password (web-portal)	✓	✓
Login via service spec. password.	✓	✓
Login via public-key challenge-response	✓	✓

To deploy our approach also for semi-trusted web-portals, the Assertion-Extractor component needs to be deployed by the web-portal to extract the authentication assertions from the result of the WebSSO login and use them to authenticate to the service provider on behalf of the user. The service provider itself, however, does not need to be adapted, as the necessary logic to validate the passed assertions can be integrated into the LDAP facade.

We used the LDAP facade to federate access to the *Large Store Data Facility* (bwLSDF)[6], a file storage service for the universities of the state of Baden-Württemberg. The requirement to also support users that are not members of a university was satisfied by integrating the LDAP facade via LDAP proxying (see Figure 8). Thus, besides supporting users that register via the web-application in a federative manner, service-local user accounts that are not connected to any home organization can also be created in an LDAP server by the bwLSDF administrators. As the members of the bwIDM federation in which bwLSDF participates as a service provider are trusted, neither the login node (an IBM Scale Out Network Attached Storage[7] in this case) nor the user clients needed to be modified to federate access via the LDAP facade. Each university can individually manage which users may access bwLSDF via authorization tokens in the home- organizational user accounts. For instance, guest users are not allowed to use the service. Fine grained authorization tokens can be issued by bwLSDF in the service-local accounts to manage user groups and their access rights.

Regarding **provisioning** of local accounts, our concepts allow services to offer a registration process that is highly usable compared to some traditional registration workflows for HPC resources. For instance, if a service provider trusts the home organization of a user to have correctly identified the user, users do not have to identify themselves locally to get a local account at the service. Thus, all that is necessary to get a local account at a service from the perspective of the user is to log on to the web-registration application (using their familiar

[6] `http://www.bw-grid.de/bwservices/bwlsdf/`
[7] `http://www-03.ibm.com/systems/de/storage/network/sonas/`

home credentials). To **deprovision** stale accounts once they are no longer needed due to users losing the access rights to the service, our approach enables service providers to query the home organization for up-to-date assertions on the users' attributes in regular intervals without user-interaction.

With regards to **legal aspects**, the web-registration application enables service providers to request the user's consent to certain policies before a local account is created. Furthermore, with regards to privacy laws, it is possible for the identity provider component of the home organization to request the user's consent before releasing user specific data to the service provider. For instance, this can be accomplished by using plugins like *uApprove*[8] for Shibboleth identity providers.

In terms of **maintenance effort**, we relied upon existing frameworks such as ApacheDS[9], the OpenSAML library[10], Shibboleth and simpleSAMLphp to implement the LDAP facade. The complex SAML logic to build SAML requests and validate SAML responses as well as the implementation of the LDAP protocol and the backend data store is not part of the custom code of our approach. As the existing frameworks are already maintained independently, only our light weighted custom code has to be monitored and maintained.

While we feel that usability is enhanced considerably with the omission of user certificates, some features of (grid-)middleware in operation that rely on certificates cannot be used anymore. For instance, using the Globus framework[11], job-traveling between multiple sites and third-party copy, i.e., moving data between sites directly, require a short lived certificate to enable a site to pass jobs/data to another site on behalf of the user. However, to our knowledge, in small sized HPC facilities these features might be rarely used, making our approach especially attractive for this type of HPC resources. Furthermore, job-traveling and third-party copy can be supported by our approach similarly to web-portal access by using the (short-lived) assertions of the home organizations as a token that can be passed between multiple sites.

8 Conclusion

In this paper, we proposed a concept to seamlessly join non web-based services that rely on intra-organizational LDAP servers into SAML federations. Furthermore, we showed how the concept can be extended to allow users to access these services via semi-trusted web-portals. The concepts have been evaluated to be highly integratable into existing service deployments and reach a high degree of usability, as users do not need modified clients to access the service in most use-cases. We are already using the concepts to federate access to storage resources within the bwLSDF and bwIDM projects and plan to apply it on the Tier-3 HPC infrastructure of the state of Baden-Württemberg.

[8] `http://www.switch.ch/aai/support/tools/uApprove.html`
[9] `http://directory.apache.org/apacheds/`
[10] `http://www.opensaml.org/`
[11] `http://www.globus.org/`

Acknowledgements. The results presented in this paper are based on the work within the project bwIDM that aims at providing identity management solutions to ease the access to non web-based services provided by universities of the state of Baden-Württemberg, Germany. The authors would like to thank the bwIDM project partners as well as the Ministry of Science, Research, and the Arts of the State of Baden-Württemberg (further information: http://www.bwidm.de/). Furthermore, we like to thank the anonymous reviewers for their very constructive and helpful comments.

References

[Koe2012] Köhler, J., Labitzke, S., Simon, M., Nussbaumer, M., Hartenstein, H.: Facius: An easy-to-deploy SAML-based approach to federate non web-based services. In: Proc. of the 11th IEEE Int. Conf. on Trust, Security and Privacy in Computing and Communications, TrustCom (2012)

[Bar2006] Barton, T., Basney, J., Freeman, T., Scavo, T., Siebenlist, F., Welch, V., Ananthakrishnan, R., Baker, B., Goode, M., Keahey, K.: Identity Federation and Attribute-based Authorization through the Globus Toolkit, Shibboleth, GridShib, and MyProxy. In: 5th Annual PKI R&D Workshop (2006)

[Bas2010] Basney, J., Fleury, T., Welch, V.: Federated Login to TeraGrid. In: Proc. of the 9th Symposium on Identity and Trust on the Internet, IDTRUST (2010)

[Spe2006] Spence, D., Geddes, N., Jensen, J., Richards, A., Viljoen, M., Martin, A., Dovey, M., Norman, M., Tang, K., Trefethen, A., Wallom, D., Allan, R., Meredith, D.: ShibGrid: Shibboleth Access for the UK National Grid Service. In: Proc. of the IEEE Int. Conf. on e-Science and Grid Computing, e-Science (2006)

[Wan2009] Wang, X.D., Jones, M., Jensen, J., Richards, A., Wallom, D., Ma, T., Frank, R., Spence, D., Young, S., Devereux, C., Geddes, N.: Shibboleth Access for Resources on the National Grid Service (SARoNGS). In: Proc. of the Int. Conf. on Information Assurance and Security, IAS (2009)

[Gri2007] Grimm, C., Groeper, R., Makedanz, S., Pfeiffenberger, H., Gietz, P., Haase, M., Schiffers, M., Ziegler, D.W.: Trust Issues in Shibboleth-Enabled Federated Grid Authentication and Authorization Infrastructures Supporting Multiple Grid Middleware. In: Proc. of the 3rd Int. Conf. on e-Science and Grid Computing, e-Science (2007)

[Hug2005] Hughes, J., Cantor, S., Hodges, J., Hirsch, F., Mishra, P., Philpott, R., Maler, E.: Profiles for the OASIS Security Assertion Markup Language (SAML) V2.0. OASIS Std. (2005)

[Mur2011] Murri, R., Kunszt, P.Z., Maffioletti, S., Tschopp, V.: GridCertLib: A Single Sign-on Solution for Grid Web Applications and Portals. Journal of Grid Computing 9(4), 441–453 (2011)

[Mil2008] Milinovic, M., Rauschenbach, J., Winter, S., Florio, L., Simonsen, D., Howlett, J.: Deliverable DS5.1.1: eduroam Service Definition and Implementation Plan. GÉANT2. Tech. Rep. (2008)

[Nov2001] Novotny, J., Tuecke, S., Welch, V.: An online credential repository for the Grid: MyProxy. In: Proc. of the 10th IEEE Int. Symp. on High Performance Distributed Computing, HPDC (2001)

[Li2011] Li, X., Palit, H., Foo, Y.S., Hung, T.: Building an HPC-as-a-Service Toolkit for User-interactive HPC services in the Cloud. In: Proc. of the IEEE Workshops of the Int. Conf. on Advanced Information Networking and Applications, WAINA (2011)

Author Index