Supercomputing Frontiers and Innovations

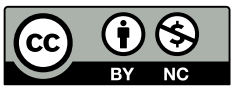
2015, Vol. 2, No. 1

Scope

- Enabling technologies for high performance computing
- Future generation supercomputer architectures
- Extreme-scale concepts beyond conventional practices including exascale
- Parallel programming models, interfaces, languages, libraries, and tools
- Supercomputer applications and algorithms
- Distributed operating systems, kernels, supervisors, and virtualization for highly scalable computing
- Scalable runtime systems software
- Methods and means of supercomputer system management, administration, and monitoring
- Mass storage systems, protocols, and allocation
- Energy and power minimization for very large deployed computers
- Resilience, reliability, and fault tolerance for future generation highly parallel computing systems
- Parallel performance and correctness debugging
- Scientific visualization for massive data and computing both external and in situ
- Education in high performance computing and computational science

Editorial Board

Contents



DOI: 10.14529/jsfi150101

AlgoWiki: an Open Encyclopedia of Parallel Algorithmic Features

Vladimir V. Voevodin[1,2], *Alexander S. Antonov*[1,3], *Jack Dongarra*[4,5]

The main goal of this project is to formalize the mapping of algorithms onto the architecture of parallel computing systems. The basic idea is that features of algorithms are independent of any computing system. A detailed description of a given algorithm with a special emphasis on its parallel properties is made once, and after that it can be used repeatedly for various implementations of the algorithm on different computing platforms. Machine-dependent, part of this work is devoted to describing features of algorithms implementation for different parallel architectures. The proposed description of algorithms includes many non-trivial features such as: parallel algorithm complexity, resource of parallelism and its properties, features of the informational graph, computational cost of algorithms, data locality analysis as well as analysis of scalability potential, and many others. Descriptions of algorithms form the basis of AlgoWiki, which allows for collaboration with the computing community in order to produce different implementations and achieve improvement. Project website: `http://algowiki-project.org/en/`.

Keywords: algorithm structure, resource of parallelism, parallel computing, efficiency, performance, supercomputers, scalability, data locality, encyclopedia of algorithmic features.

Introduction

Computers evolve quickly, and there have been at least six generations of computing architecture over the last forty years that caused the need for radical changes in software. Vector computers, vector-parallel, massive parallel computers, shared-memory nodes, clusters of shared-memory computers, computers with accelerators. . . The computing community has survived this, but in each evolution, the software basically had to be rewritten from scratch as each new generation of machines required singling out new properties in the algorithms, and that was reflected in the software.

Alas, there is no reason to hope that the situation will change for the better in the future. Vendors are already considering various prospective architectures, featuring light and/or heavy computing cores, accelerators of various types, SIMD and data-flow processing concepts. In this situation, codes will yet again have to be rewritten in order to utilize the full potential of future computers. It's an endless process that — understandably — doesn't make software developers any happier.

However, the situation isn't quite hopeless. Indeed, new computing systems require a full review of the legacy code. But the algorithms themselves don't change; only the requirements that new computers present to the structure of algorithms and programs change. To support vector computing, data parallelism needs to be built into the innermost loops within a program. The key concern in ensuring the efficient usage of massive parallel computers is to find a representation of an algorithm whereby a large number of computing nodes can work independently from each other, minimizing data exchange. This is true for every generation of parallel com-

[1] Moscow State University, Moscow, Russia
[2] voevodin@parallel.ru
[3] asa@parallel.ru
[4] University of Tennessee, Knoxville, USA
[5] dongarra@eecs.utk.edu

puting systems: the new architecture requires taking a new look at the properties of existing algorithms, in order to find the most efficient way of implementing them.

There are two facts that matter in this situation: the algorithms themselves don't change much, and their properties do not depend on the computing system. This means that once algorithm's properties have been described in detail, this information can be used repeatedly for any computing platform — existing today or in the future.

The idea of a deep *a priori* analysis of properties of algorithms and their implementation formed the basis for the AlgoWiki project. The main purpose of the project is to present a description of the fundamental properties of various algorithms giving a complete understanding of both their theoretical potential and the particular aspects of their implementation for various classes of parallel computing systems.

The description of all algorithms in AlgoWiki consists of two parts. The first part describes algorithms and their properties. The second part is dedicated to describing particular aspects of their implementation on various computing platforms. This division is made intentionally, to highlight the machine-independent properties of algorithms which determine their potential and the quality of their implementations on parallel computing systems, and to describe them separately from a number of issues related to the subsequent stages of programming and executing the resulting programs.

AlgoWiki provides an exhaustive description of an algorithm. In addition to classical algorithm properties such as serial complexity, AlgoWiki also presents additional information, which together provides a complete description of the algorithm: its parallel complexity, parallel structure, determinacy, data locality, performance and scalability estimates, communication profiles for specific implementations, and many others.

In AlgoWiki, details matter. Classical basic algorithms must be supplemented with their important practical modifications. For example, AlgoWiki contains a description of a basic point-structured real version of Cholesky factorization for a dense symmetrical positive-definite matrix. In practice, modifications of the algorithm are just as important: a block-structured version, a version for a dense complex-Hermitian matrix (both point-structured and block-structured), versions of the algorithm for sparse matrices, etc. It is also important to consider the use of Cholesky factorization in iterative methods, etc.

Equally important are the details related to particular aspects of an algorithm's implementation on specific parallel computing platforms: multi-core processors, SMP, clusters, accelerators, using vector processing and so on. In many cases it is necessary to go one step lower, describing the implementation of an algorithm, for example, not just for a specific accelerator, but to single out relatively important individual cases, such as GPU and Xeon Phi. At the same time, when we provide data about execution time, performance and scalability, we are not only laying out some estimates of the possible implementation quality of a given algorithm on a specific computer, but also setting the foundation for comparative analysis of various computing platforms with regards to the algorithms provided in AlgoWiki.

The outcome of the AlgoWiki project is an open encyclopedia of algorithm properties and the particular aspects of their computer implementation. It was started with a focus on using Wiki technologies, enabling the entire computing community to collaborate on algorithm descriptions. Currently, the encyclopedia is actively being expanded by outside experts; a multi-lingual version is also in the works which will eventually become the main version. The pilot version of the encyclopedia is available at `http://algowiki-project.org/en/`.

1. Background and Related Work

Today, efficiency and parallelism support throughout the entire supercomputing software stack are the central topics in all supercomputing forums worldwide. The same issues are being discussed within major international projects that formulate key challenges and prepare roadmaps of supercomputing software development for decades to come, e.g., International Exascale Software Project [1], Big Data and Extreme Computing [2], European Exascale Software Initiative I and II [3].

In many cases, research on the process of mapping algorithms onto parallel computing system architecture comes down to studying types of algorithm structures which are used as building blocks for a number of programs in many different subject areas. For example, a group of researchers at Berkeley are using the term "motifs" (previously "dwarfs") to describe methods that use certain common templates for computations and communications [4, 5]. Sandia National Laboratories runs the Mantevo project aimed at studying mini-applications, which represent computational kernels for various scientific and engineering applications [6, 7]. A similar approach is used by the TORCH project [8–10].

Studies of parallel algorithm properties have been conducted for a long time, starting with this fundamental work [11], and a rather wide range of algorithms has been considered in the work [12]. In many cases the authors stick to one specific subject area; for example, parallel algorithms for linear algebra are reviewed in the works [13–15].

In this project, the key aspect is using an algorithm graph (also known as an information graph or dependency graph in literature) for identifying and utilizing algorithm properties. The idea of true information dependency and its usage for software transformation was described in [16], and laid the foundation for a theory of studying the properties of algorithms and programs developed in [17–19].

There are no direct analogues to AlgoWiki — an open encyclopedia of algorithm properties being built within this project. While there are a number of projects attempting to classify and describe properties, and write up implementations of various algorithms [20–23], none of them follows a unified predefined structure to describe all relevant algorithm properties and parallel implementations for various target architectures.

2. A Description of Algorithm Structure and Properties

All fundamentally important issues affecting the efficiency of the resulting parallel programs must be reflected in the description of the properties and structure of the algorithms. With this in mind, an algorithm description structure was developed, which formed the basis for the AlgoWiki encyclopedia. The encyclopedia offers standardized elements for different sections and recommendations for building them, so that descriptions of different algorithms could easily be compared.

Sections 3 and 4 of this paper describe the two parts that form the description of each algorithm within AlgoWiki. Structure of these sections repeats exactly the structure of the description (ten subsections for the Part I and seven subsections for the Part II).

3. AlgoWiki Encyclopedia: "PART I. Algorithm Structure and Properties"

Algorithm properties are independent of the computing system, and, in this regard, this part of AlgoWiki is an important thing by itself. An algorithm description is only prepared once; afterwards, it is used repeatedly for implementation within various computing environments. Even though this part is dedicated to the machine-independent properties of algorithms, things to consider during the implementation process or links to respective items in the second part of AlgoWiki are acceptable here.

3.1. General description of algorithms

This section contains a general description of the problems the algorithm is intended to address. The description shows specific features of the algorithm itself and objects it works with.

3.2. Mathematical description of algorithms

A mathematical description of the problem to be addressed is presented as a combination of formulas, as it is commonly described in textbooks. The description must be sufficient for an unambiguous understanding of the description by a person who is familiar with mathematics.

3.3. Computational kernel of algorithms

The computational kernel (the part of algorithm that takes up most of the processing time) is separated and described here. If an algorithm has more than one computational kernel, each one is described separately.

3.4. Macro structure of algorithms

If an algorithm relies on other algorithms as its constituent parts, this must be specified in this section. If it makes sense to provide the further description of the algorithm in the form of its macro structure, the structure of macro operations is described here. Typical macro operations include finding the sum of vector elements, dot product, matrix-vector multiplication, solving a system of linear equations of small rank, calculating the value of a function at a specific point, searching for the minimum value in an array, matrix transposition, reverse matrix calculation, and many others.

The macro structure description is very useful in practice. The parallel structure for most algorithms is best seen at the macro level, while reflection of all operations would clutter the picture.

The choice of macro operations is not fixed; by grouping macro operations in different ways it is possible to emphasize different properties of the algorithms. In this regard, several macro structure options can be shown here to provide an additional aspect of its overall structure.

3.5. A description of algorithms' serial implementation

This section describes the steps that need to be performed in the serial implementation of this algorithm. To a certain degree, this section is redundant, as the mathematical description

already contains all the necessary information. However, it is still useful; the implementation of the algorithm is clearly laid out, helping to unambiguously interpret the properties and estimates presented below.

3.6. Serial complexity of algorithms

This section of the algorithm description shows an estimate of its serial complexity, i.e., the number of operations that need to be performed if the algorithm is executed serially (in accordance with section 3.5). For different algorithms, the meaning of an operation used to evaluate its complexity can vary greatly. This can include operations on real numbers, integers, bit operations, memory reads, array element updating, elementary functions, macro operations, etc. LU factorization is dominated by arithmetic operations on real numbers, while only memory reads and writes are important for matrix transposition; this must be reflected in the description.

For example, the complexity of a vector elements pairwise summation is $n-1$. The complexity of fast Fourier transformation (Cooley-Tukey algorithm) for vector lengths equals a power of two — $n \log_2 n$ complex addition operations and $(n \log_2 n)/2$ complex multiplication operations. The complexity of a basic Cholesky factorization algorithm (point-structured version for a dense symmetrical and positive-definite matrix) is n square root calculations, $n(n-1)/2$ division operations, and $(n^3 - n)/6$ multiplication and addition (subtraction) operations each.

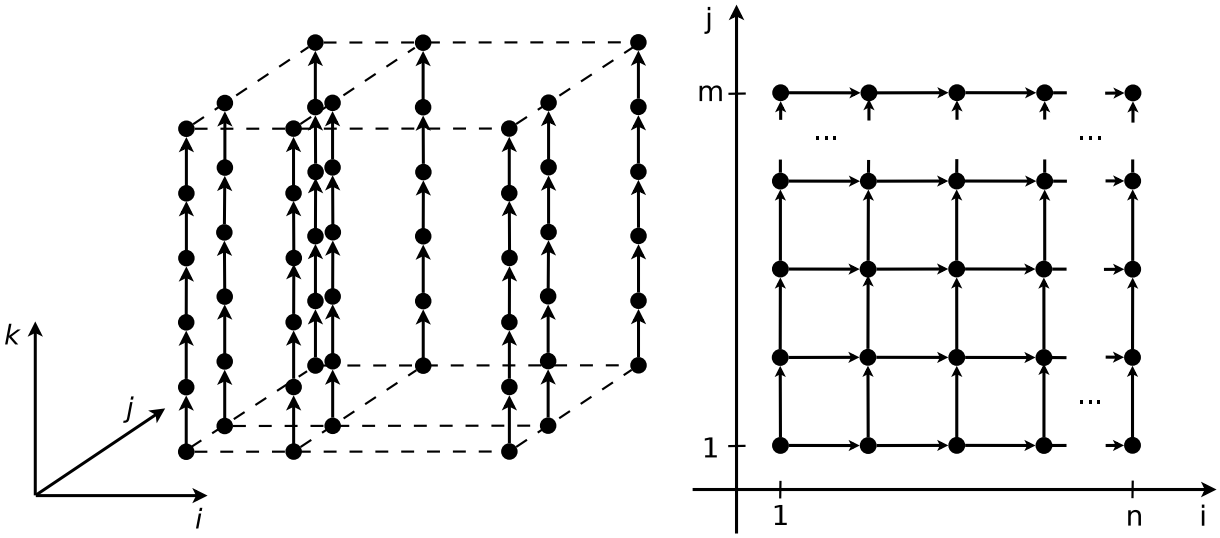
3.7. Information graph

This is a very important part of the description. This is where one can show (see) the parallel structure of the algorithm, for which there is a description and a picture of its information graph [19]. There are many interesting options for reflecting the information structure of algorithms in this section. Some algorithms require showing a maximum-detail structure, while using only the macro structure is sufficient for others. A lot of information is available in various projections of the information graph, which make its regular components stand out, while minimizing insignificant details.

Overall, the task of displaying an algorithm graph is non-trivial. To begin with, the graph is potentially endless, as its number of vertices and edges is determined by the values of external variables, which can be very large. Situations like this can usually be saved by the "similarity" approach, which makes graphs for different values of an external variable "similar": in most cases it is enough to present a relatively small-sized graph, and the graphs for other values will be "exactly the same". In practice, it isn't always so simple, and one needs to be very careful.

Further, an algorithm graph is potentially a multi-dimensional object. The most natural system for placing the vertices and edges of the information graph is based on the nesting loops in the algorithm implementation. If the nesting loop level does not exceed three, the graph can be placed in the traditional three-dimensional space; more complex loop constructs with a nesting level of 4 or more require special graph display and presentation methods.

Fig. 1 shows the information structure of a matrix multiplication algorithm and of an algorithm for solving a system of linear algebraic equations with a block-structured bidiagonal matrix. A more complex example is shown in fig. 2, it demonstrates the information structure of a Cholesky algorithm with input and output data.



(a) (b)

Figure 1. Information structure of a matrix multiplication algorithm (a) and of an algorithm for solving a system of linear algebraic equations with a block-structured bidiagonal matrix (b)

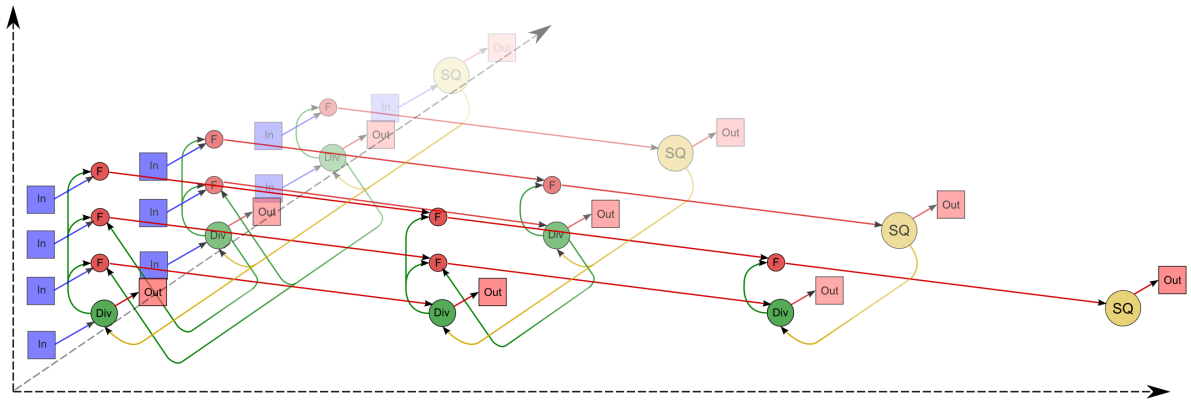


Figure 2. Information structure of a Cholesky algorithm with input and output data: SQ is the square-root operation, F is the operation a-bc, Div is division, In and Out indicate input and output data

3.8. Describing the resource parallelism of algorithms

This section shows an estimate of the algorithm's parallel complexity: the number of steps it takes to execute the algorithm assuming an infinite number of processors (functional units, computing nodes, cores, etc.). The parallel complexity of an algorithm is understood as the height of its canonical parallel form [19]. It is necessary to indicate in terms of which operations the estimate is provided. It is also necessary to describe the balance of parallel steps by the number and type of operations, which determines the layer width in the canonical parallel form and the composition of operations within the layers.

Parallelism in algorithms frequently has a natural hierarchical structure. This fact is highly useful in practice and must be reflected in the description. As a rule, such hierarchical parallelism structures are well reflected in the serial implementation of the algorithm through the program's loop profile, which can well be used to reflect the parallelism resource.

Describing the parallelism resource for an algorithm requires specifying key parallel branches in terms of finite and mass parallelism. The parallelism resource isn't always expressed in simple terms, e.g., through coordinate parallelism; the skewed parallelism resource is equally important. Unlike coordinate parallelism, skewed parallelism is much harder to use in practice, but it is an important option to keep in mind, as sometimes it is the only option. A good illustration is the algorithm structure shown in fig. 1b: there is no coordinate parallelism, but skewed parallelism is there, and using it reduces the complexity from $n \times m$ in serial execution to $(n + m - 1)$ in the parallel version.

For example, let's look at the algorithms, for which serial complexity has been considered in section 3.6. The parallel complexity for vector elements pairwise summation is $\log_2 n$, with the number of operations at each level decreasing from $n/2$ to 1. The parallel complexity of the fast Fourier transformation (Cooley-Tukey algorithm) for vector lengths equal to a power of two is $\log_2 n$. The parallel complexity of a basic Cholesky factorization algorithm (point-structured version for a dense symmetrical and positive-definite matrix) is n steps for square root calculations, $(n - 1)$ steps for division operations and $(n - 1)$ steps for multiplication and addition operations.

3.9. Input/output data description

This section contains a description of the structure, features and properties of the algorithm's input and output data: vectors, matrices, scalars, arrays, a dense or sparse data structure, and their total amount.

3.10. Algorithm properties

This section describes other properties of the algorithm that are worth considering in the implementation process. As noted above, there is no connection to any specific computing platform, but since implementation issues always prevail in a project, there is a distinct need to discuss additional properties of an algorithm.

The *computational power* of an algorithm is the ratio of the total number of operations to the total amount of input and output data. Despite its simplicity, this ratio is exceptionally useful in practice: the higher the computational power, the less the overhead cost of moving data for processing, e.g., on a co-processor, accelerator, or another node within a cluster. For example, the computational power of the dot product operation is 1; the computational power of multiplying two square matrices is $2n/3$.

The issue of utmost importance is the *algorithm stability*. Anything related to this notion, particularly the stability estimate, must be described in this section.

The *balance* of the computing process can be viewed from different perspectives. This includes the balance between different types of operations, particularly arithmetic operations (addition, multiplication, division) together or between arithmetic operations and memory access operations. This also includes the balance of operations between parallel branches of the algorithm.

The *determinacy of an algorithm* is also important in practice, and it is understood as the uniformity of the computing process. From this point of view, the classical multiplication of dense matrices is a highly deterministic algorithm, as its structure, given a fixed matrix size, does not depend on the elements of the input matrices. Multiplying sparse matrices, where matrices are

stored in a special format, is no longer deterministic: data locality depends on the structure of the input matrices. An iteration algorithm with precision-based exit is also not deterministic, as the number of iterations (and therefore the number of operations) changes depending on the input data.

A serious issue affecting the indeterminacy of a parallel program is a change in the order of execution for associative operations. A typical example is the use of global MPI operations, e.g., finding the sum of elements in a distributed array. An MPI runtime system dynamically chooses the order of operations, assuming associativity; as a result, rounding errors change between runs, leading to changes in the final output of the program. This is a serious and quite common issue today in massive parallel systems, which translates to lack of reproducible results in parallel program execution. This feature is characteristic for the second part of AlgoWiki, dedicated to algorithm implementations. But the issue is quite important and the respective considerations should be mentioned here as well.

It should be noted that in some cases, a lack of determinacy can be "rectified" by introducing the respective macro operations, which makes the structure both more deterministic and more understandable.

"Long" edges in the information graph are a sign of potential challenges in placing data in the computer's memory hierarchy during the program execution. On the one hand, edge length depends on the specific coordinate system chosen for placing graph vertices, so long as edges can disappear in a different coordinate system (but that can lead to even more long edges appearing elsewhere). On the other hand, regardless of the coordinate system, their presence can signal the need to store data for a long time at a specific hierarchy level, which imposes additional restrictions on the efficiency of the algorithm implementation. One reason for long edges is a scalar value broadcasted over all iterations of a specific loop: in this case long edges do not cause any serious issues in practice.

Compact packing of the information graph is of interest in developing specialized processors or implementing an algorithm on FPGAs; it can also be included in this section.

4. AlgoWiki Encyclopedia: "PART II. Software Implementation of Algorithms"

The Part II of the algorithm description in AlgoWiki deals with all of the components of the implementation process for an algorithm described in Part I. Both the serial and parallel implementations of an algorithm are considered. This part shows the connection between the properties of the programs implementing the algorithm and the features of the computer architecture they are executed on. Data and computation locality are explained, and the scalability and efficiency of parallel software are described along with the performance achieved with a given program. This is also the place to discuss specific aspects of implementation for different computing architecture classes and to provide references to implementations in existing libraries.

4.1. Features of algorithms' serial implementation

This section describes the aspects and variations of implementing an algorithm within a serial program that affect its efficiency. In particular, it makes sense to mention the existence of block-structured versions of the algorithm implementation, further describing the prospective advantages and drawbacks of this approach. Another important aspect is related to the options

for organizing work with data, variations on data structures, temporary arrays and other similar issues. The required parallelism resource and memory amount for different implementation options need to be specified.

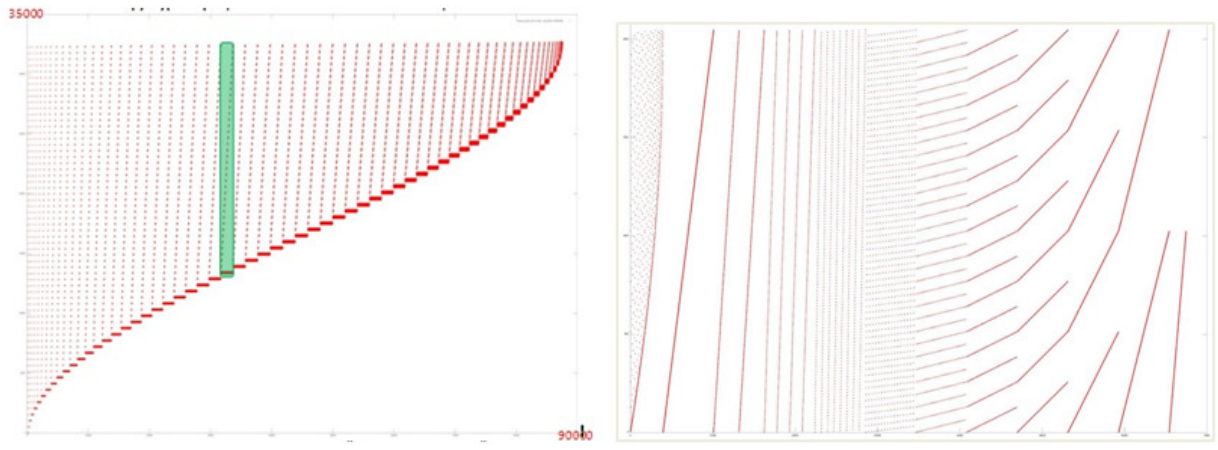
Another important feature is a description of the precision of operations within the algorithm. In practice, it is rarely necessary to perform all arithmetic operations on real numbers with 64-bit floating point arithmetic (double precision), as it doesn't affect the stability of the algorithm or the precision of the results obtained. In this case, if most operations can be performed on a float single precision data, and some fragments require switching to the double, this must be specified here.

Based on information from section 3.8, when describing the serial version of the program, it is worth noting the possibility of equivalent transformaions for programs implementing this algorithm. For example, parallelism of iterations of the innermost loop is normally used for vectorization. However, in some cases this parallelism can be "moved" up the nested loops, which enables more efficient implementation of this algorithm on multi-processor multi-core SMP computers.

4.2. A description of data and computation locality

The issues of data and computation locality are rarely studied in practice, but locality is what affects the program execution efficiency on modern computing platforms. This section provides an estimate of data and computation locality in the program, considering *both the temporal and spacial locality*. Positive and negative locality-related facts should be mentioned, along with what situations could arise under what circumstances. This section also specifies how locality changes when passing from a serial to a parallel implementation. Key patterns in the program's interaction with memory are identified here. Note any possible interrelation between the programming language used and the degree of locality in the resulting programs.

Separately, memory access profiles are specified for computational kernels and key fragments. Fig. 3 shows memory access profiles for programs implementing Cholesky factorization and fast Fourier transformation, which illustrates the difference between the locality properties of the two algorithms.



(a) (b)

Figure 3. Memory access profiles for programs implementing Cholesky factorization (a) and fast Fourier transformation (b)

4.3. Possible methods and considerations for parallel implementation of algorithms

This is a rather big section that must describe key facts and statements that define a parallel program. These can include:

- a hierarchically presented parallelism resource based on the program's loop structure and program call graph;
- a combination (hierarchy) of mass parallelism and finite parallelism;
- possible ways to distribute operations between processes/threads;
- possible strategies to distribute data;
- an estimate of the number of operations, amount and number of data transfers (both the total number and the share for each parallel process);
- an estimate of data locality, etc.

This section should also include recommendations or comments regarding the algorithm's implementation using various parallel programming technologies: MPI, OpenMP, CUDA, or using vectorization directives.

4.4. Scalability of algorithms and their implementation

This section is intended to show the algorithm's scalability limits on various platforms. The impact of barrier synchronization points, global operations, data gather/scatter operations, estimates of strong or weak scalability for the algorithm and its implementations.

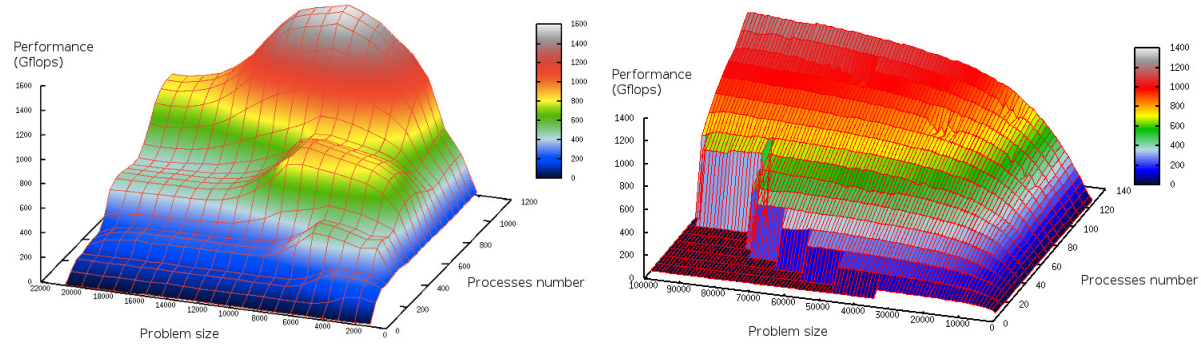
An algorithm's scalability determines the properties of the algorithm regardless of the computer being used. It shows how much the algorithm enables the computer's actual performance to increase with a potentially infinite increase in the number of processors. The scalability of parallel programs is determined in connection with a specific computer and shows how much the performance of this computer running this program can increase when utilizing more computing power.

The central idea of this section is to show the actual scalability of the program implementing a given algorithm on various computing platforms, depending on the number of processors and the size of the problem. It is important to understand a correlation between the number of processors and the size of the problem, so as to reflect all features in behavior of the parallel program, particularly achieving maximum performance, and more subtle effects arising, for example, from the algorithm's block structure or memory hierarchy.

Fig. 4a shows the scalability of a classical matrix multiplication algorithm, depending on the number of processes and the size of the problem. The chart shows visible areas with greater performance, reflecting cache memory levels. Fig. 4b shows the scalability of the Linpack benchmark.

4.5. Dynamic characteristics and efficiency of algorithm implementation

This is a rather large section of AlgoWiki, as evaluating an algorithm's efficiency requires a comprehensive approach and careful analysis of all steps — from the computer architecture to the algorithm itself. Efficiency is understood rather broadly in this section: this includes the efficiency of program parallelization, and the efficiency of program execution relative to the peak performance of computing systems.



(a) (b)

Figure 4. Scalability of programs implementing classical matrix multiplication (a) and Linpack benchmark (b)

In addition to the actual performance, all major reasons should be described that limit further increase in the performance of a given parallel program on a specific computing platform. This is not an easy task, as there is no commonly accepted methodology or respective tools that facilitate such an analysis at the present time. One needs to estimate and describe the efficiency of interaction with the memory subsystem (features of the program's memory access profile), the efficiency of using a resource of parallelism built into the algorithm, the efficiency of interconnect usage (features of the communication profile), the efficiency of input/output operations, etc. Sometimes overall program efficiency characteristics are sufficient; in some cases it is important to show lower-level system monitoring data, such as CPU load, cache misses, Infiniband network usage intensity, etc.

In our studies of parallel programs for the AlgoWiki project, we use Job Digest [24], a tool for building reports on program performance. Fig. 5 shows some results of the execution of a program implementing ten iterations of a Cholesky decomposition for dense real positive-definite matrices.

Fig. 5a shows that, during the runtime of the program, the processor usage level is about 50%. That is a good result for programs executed without the usage of the Hyper-threading technology. Fig. 5b shows the number of floating point operations per second during the Cholesky decomposition exection time. To the end of each iteration, the number of opertations increases intensively. From fig. 5c it follows that the number of L1 cache-misses is large enough (about 25 millions per second on the average for all nodes). Fig. 5d shows that the interconnest IB) is intensively used at each interation. To the end of each iteration, the data transfer intensity increases significantly. Overall, the data obtained from the monitoring system allows one to come to the conclusion that this program was working in an efficient and stable manner. The memory and communication environment usage is intensive, which can lead to an efficiency reduction with an increase of the matrix order or the number of processors in use.

4.6. Conclusions for different classes of computer architecture

This section should contain recommendations on implementing the algorithm for various architecture classes. If the architecture of a specific computer or platform has any specific features affecting the implementation efficiency, this must be noted here.

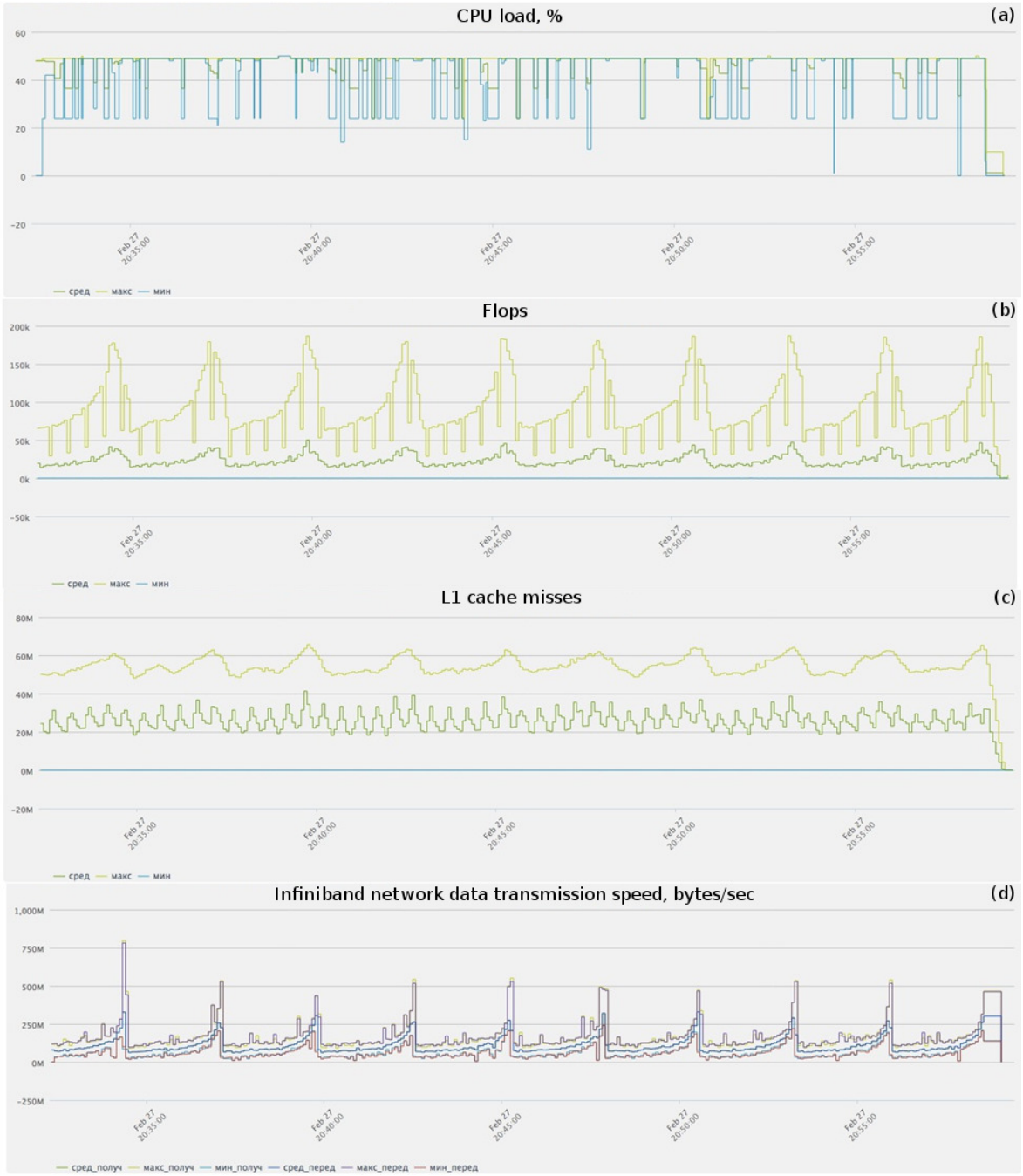


Figure 5. Ten iterations of a Cholesky decomposition for dense real positive-definite matrices: CPU load during program run (a); number of floating point operations per second (b); L1 cache misses (c); Infiniband network data transmission speed, bytes/sec (d)

It is important to point out both positive and negative facts with regards to specific classes of computers. Possible optimization techniques or even "tips and tricks" in writing programs for a target architecture class can be described.

4.7. Existing implementations of algorithms

For many algorithm-computer pairs, good implementations have already been developed which can and should be used in practice. This section is here to provide references to existing

serial and parallel implementations of an algorithm that are available for use today. It indicates whether an implementation is open-source or proprietary, what type of license it is distributed under, the distributive location and any other available descriptions. If there is any information on the particular features, strengths and/or weaknesses of various implementations, these can be pointed out here.

Conclusion

AlgoWiki Encyclopedia is an exceptionally large-scale project, which, despite its youth, is actively being developed today and has a number of areas for future development. Materials from the encyclopedia can be used efficiently for a comparative analysis of computing platforms over different applications. We are talking about a direct extension of the methodology used in the Top500 list, which is currently based on just the Linpack test, and by which it is criticized by many researchers. With AlgoWiki, an additional item can be included in Part II for each algorithm to present performance data for various supercomputing platforms. This extension will come naturally for AlgoWiki, and given the encyclopedia's open nature and potential for community contribution, it can become a supplement to the Top500 list, expandable to any algorithm.

An important issue is using the AlgoWiki algorithm information structure in the education process. It is not enough to know the mathematical description, it is vital to understand the structure and special features of every basic step, from formulating the algorithm to its execution. This knowledge is vital in the supercomputing world where everything must be done with the ideas of supercomputing co-design. But this is also necessary for today's ordinary computers, where even smartphones and tablets have become parallel. All of the problems of parallel computing have become important everywhere — from supercomputers to mobile gadgets; this is what brought the AlgoWiki project forward.

The results were obtained with the financial support of the Russian Science Foundation, Agreement N 14-11-00190. The research presented in Sections 4.4 and 4.5 is supported by the Russian Foundation for Basic Research, grant N 13-07-00790.

References

1. Dongarra, J., Beckman, P., Moore, T., Aerts, P., Aloisio, G., Andre, J.C., Barkai, D., Berthou, J.Y., Boku, T., Braunschweig, B. and others: The International Exascale Software Project roadmap // International Journal of High Performance Computing Applications, Vol. 25, No. 1, p.3–60 (2011). DOI: 10.1177/1094342010391989.

2. Big Data and Extreme-scale Computing (BDEC), http://www.exascale.org

3. EESI project — The European Exascale Software Initiative, http://www.eesi-project.eu

4. Asanovi, K., Bodik R., Demmel J., Keaveny T., Keutzer K., Kubiatowicz J. D., et al.:

The Parallel Computing Laboratory at U.C. Berkeley: A Research Agenda Based on the Berkeley View (2008)

5. Asanovi, K., Bodik R., Catanzaro B., Gebis J. J., Husbands P., Keutzer K., et al.: The Landscape of Parallel Computing Research: A View from Berkeley (2006)

6. Christesen C.: Mantevo Views. A Flexible System for Gathering and Analyzing Data for the Mantevo Project. Thesis, College of St. Benedict/St. John's University (2007)

7. Heroux, M.A., Doerfler, D.W., Crozier, P.S., Willenbring, J.M., Edwards, H.C., Williams, A., Rajan, M., Keiter, E.R., Thornquist, H.K., Numrich, R.W.: Improving Performance via Mini-applications. Sandia National Laboratories, Report SAND2009-5574 (2009). DOI: 10.2172/993908.

8. Kaiser, A., Williams, S., Madduri, K., Ibrahim, K., Bailey, D., Demmel, J., Strohmaier, E.: TORCH Computational Reference Kernels: A Testbed for Computer Science Research, Lawrence Berkeley National Laboratory, Paper LBNL-4172E (2010). DOI: 10.2172/1004197.

9. Kaiser, A., Williams, S., Madduri, K., Ibrahim, K., Bailey, D., Demmel, J., Strohmaier, E.: A Principled Kernel Testbed for Hardware/Software Co-Design Research, Proceedings of the 2nd USENIX Workshop on Hot Topics in Parallelism (HotPar) (2010). DOI: 10.2172/983482.

10. Strohmaier, E., Williams, S., Kaiser, A., Madduri, K., Ibrahim, K., Bailey, D., Demmel, J.: A Kernel Testbed for Parallel Architecture, Language, and Performance Research, International Conference of Numerical Analysis and Applied Mathematics (ICNAAM) (2010)

11. Knuth, D.: The Art of Computer Programming, Volumes 1-4A Boxed Set 3rd Edition (Reading, Massachusetts: Addison-Wesley (2011)

12. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes 3rd Edition: The Art of Scientific Computing. WH Press. Cambridge university press (2007). DOI: 10.1145/1874391.187410.

13. Ortega, J.M.: Introduction to Parallel and Vector Solution of Linear Systems, Plenum Press, New York, USA (1988). DOI: 10.1007/978-1-4899-2112-3_1.

14. Barrett, R., Berry, M., Chan, T.F., Demmel, J., Donato, J., Dongarra, J, Eijkhout, V., Pozo, R., Romine, C., Van der Vorst, H.: Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition, SIAM, Philadelphia (1994). DOI: 10.1137/1.9781611971538.

15. Saad, Y.: Iterative methods for sparse linear systems, 2nd ed., Society for Industrial and Applied Mathematics, Philadelphia (2003)

16. Polychronopoulos, C.D.: Compiler optimizations for enhancing parallelism and their impact on architecture design. IEEE Trans. on Computers, Vol.37, N.8 (1988). DOI: 10.1109/12.2249.

17. Voevodin, V.V.: Mathematical foundations of parallel computing, World Scientific Publishing Co., Series in computer science. Vol.33. 343 p. (1992)

18. Voevodin, V.V.: Information structure of sequential programs. Russ. J of Num. An. and Math Modelling. Vol.10, N3. 279–286 (1995)

19. Voevodin, V.V., Voevodin, Vl.V.: Parallel computing. BHV-St.Petersburg, 608 p. (in Russian) (2002)

20. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, http://www.netlib.org/linalg/html_templates/Templates.html

21. A Library of Parallel Algorithms, http://www.cs.cmu.edu/s̃candal/nesl/algorithms.html

22. Scientific and educational Internet site of RCC on numerical analysis, http://num-anal.srcc.msu.ru/ (in Russian)

23. Wikipedia, List of algorithms, https://en.wikipedia.org/wiki/List_of_algorithms

24. Adinets A.V., Bryzgalov P.A., Voevodin Vad.V., Zhumatii S.A., Nikitenko D.A., Stefanov K.S.: Job Digest: an approach to dynamic analysis of job characteristics on supercomputers, Numerical methods and programming: Advanced Computing, Vol. 13, Section 2, Pp. 160–166 (2012)

Received March 15, 2015.

.

DOI: 10.14529/jsfi150102

Applications for Ultrascale Computing

***Lars Ailo Bongo*[1], *Raimondas Čiegis*[2], *Neki Frasheri*[3], *Jing Gong*[4], *Dragi Kimovski*[5], *Peter Kropf*[6], *Svetozar Margenov*[7], *Milan Mihajlović*[8], *Maya Neytcheva* [9], *Thomas Rauber*[10], *Gudula Rünger*[11], *Roman Trobec*[12], *Roel Wuyts*[13], *Roman Wyrzykowski*[14]**

Studies of real complex physical and engineering problems represented by multiscale and multiphysics computer simulations have an increasing demand for computing power. The demand is driven by the increasing scales and complexity of the scientific problems investigated or the time constraints. Ultrascale computing systems could offer the computing power required to solve these problems. Future ultrascale systems will be large-scale complex computing systems combining technologies from high performance computing, distributed systems, big data, and cloud computing. The challenge of developing and programming complex algorithms that can efficiently perform on such systems is twofold. Firstly, the complex computer simulations have to be either developed from scratch, or redesigned in order to yield high performance, while retaining correct functional behaviour. Secondly, ultrascale computing systems impose a number of non-functional cross-cutting concerns, such as fault tolerance or energy consumption, which can significantly impact the deployment of applications on large complex computing systems. This article discusses the state-of-the-art of programming for current and future large-scale computing systems with an emphasis on complex applications. We derive a number of requirements regarding programming and execution support by studying several computationally demanding applications that the authors are currently developing and discuss their potential and necessary upgrades for ultrascale execution on ultrascale facilities.

Keywords: sustainable ultrascale systems, impact factors on applications, multiscale and multiphysics applications, computational modelling.

Introduction

A glance over the historical development of computational science shows that software and hardware developments have always been driven by the need for a continual growth. For software this is a continuously increasing growth in complexity of algorithms, of data sizes and processing requirements; for hardware these have been and are the technological inventions providing increasing computing power and storage capabilities. Topics such as High Performance Computing (HPC), distributed systems, big data and cloud computing are well-established domains of software and hardware development reflecting this tendency. In the near future, the growth in algorithmic complexity, data volumes to be processed, and available computing power is expected

[1]University of Tromsø, Tromsø, Norway
[2]Vilnius Gediminas Technical University, Lithuania
[3]University of Tirana, Albania
[4]Royal University of Technology, Stockholm, Sweden
[5]University for Information Science and Technology "St. Paul the Apostle", Ohrid, Macedonia
[6]University of Neuchâtel, Switzerland
[7]Institute of Information and Communication Technologies, Sofia, Bulgaria
[8]The University of Manchester, UK
[9]Uppsala University, Sweden
[10]University Bayreuth, Germany
[11]Technical University Chemnitz, Germany
[12]Jožef Stefan Institute, Ljubljana, Slovenia
[13]imec, Leuven, Belgium and KU Leuven, Belgium
[14]Czestochowa University of Technology, Poland

to reach extreme scales. Successful handling of this growth and maintaining performance on such scales requires the existing and the emerging hardware and software aspects and concerns to be re-evaluated and adapted to the new paradigms. The generic term ultrascale computing captures all these efforts and challenges.

Ultrascale computing systems are expected to have the form of large-scale complex systems comprising parallel and distributed components as well as heterogeneous processors, e.g. enhanced by Graphical Processing Units (GPU), Field Programmable Gate Arrays (FPGA) or other types of accelerators. These compute facilities might be provided by cloud architectures that are made available to scientific computing communities in an effort to achieve scientific cloud computing [85]. Large-scale scientific simulations often have to deal with large volumes of data which may come from multiple sources and are diverse, complex, and have massive scale, requiring the big data techniques to be included. Due to the complexity of ultrascale systems, their efficient usage is a challenging task which exceeds the effort for programming and maintaining the HPC systems that are available today. Consequently, adequate support for developing software on different levels is needed to properly exploit the hardware potential offered by ultrascale systems.

The experience with current HPC systems has shown that some of the available application codes and also some of the well-developed algorithms are not suitable for the hardware, since their internal structure and behavior lacks a high degree of parallelism and flexibility [30, 89]. This situation is expected to be even more critical for ultrascale computing. An important starting point for investigating the potential of ultrascale computing is to identify algorithms, applications, and services amenable to ultrascale systems. In addition, the requirements that need to be fulfilled to port applications to ultrascale systems have to be identified. This will enable the development of new applications that will conform to these requirements and recommendations.

In this article, we discuss a large variety of aspects which might be crucial for application codes to be suitable for ultrascale computing systems and to exploit the compute power for achieving a sufficiently high performance and scalability of those applications on emerging ultrascale platforms. In this context, we highlight the issues that are important for migrating existing parallel applications to ultrascale platforms. Application areas amenable to ultrascale computing include earth sciences, astrophysics, chemistry such as molecular dynamics, material sciences, life sciences such as analysis of short-read sequencing data, health science, high energy physics such as QCD, fluid dynamics, coupled multiscale and multiphysics methods. In addition, diverse applications for analysing large and heterogeneous data sets in social, financial, and industrial contexts are candidate areas for ultrascale computing. To illustrate the significance of these issues, we first review the current state-of-the-art in HPC execution of a typical multi-scale simulation with separated spatio-temporal scales. We then proceed with a review of several real parallel applications that the authors of this article are working on and discuss the likely impact that the ultrascale execution paradigm will have on these applications.

Ultrascale computing offers a way to provide sufficient computing resources for a persistent increase in problem sizes and parameter sets needed to process increasingly larger computational tasks in a required amount of time. It is general consent that applications will have to be re-designed or re-programmed substantially in order to perform efficiently on the heterogeneous hardware and to exploit fully the available technology of ultrascale computing systems [26]. This might require new data structures, new algorithms, or even new mathematics. New programming models for flexible coding and performance adaptation as well as more abstract and

advanced programming interfaces and domain-specific languages might be the key components for delivering highly sustainable and scalable applications. However, each redesign of an application for ultrascale computing systems must take into consideration cross-cutting issues, which include resilience and fault tolerance mechanisms, handling of data I/O, especially for a growing amount of data on geographically distributed systems (big data), power management and energy efficiency as well as programmability and portability with respect to the underlying ultrascale hardware system.

This article discusses the challenges for achieving ultrascale performance from an application perspective. The rest of the article is structured as follows: In Section 1 we discuss hardware as well as software development and program execution issues related to ultrascale computing. Then, in Section 2 we consider a number of specific applications from different areas and discuss their potential and requirements for ultrascale computing. Section 2.2.7 concludes the article.

1. Hardware and software issues for programming ultrascale computing applications

Designing applications for ultrascale computing systems includes a multitude of different interacting challenges. Firstly, this involves programming of a functionally correct application software that provides correct simulation results. In this context, it is useful that the application is based on a developed mathematical formalism of the scientific problem. A numerical computation problem, for example, can benefit from algorithms that have proven asymptotic convergence and provide error estimates. Secondly, the ultrascale application software has to fulfill non-functional properties, including a large variety of criteria ensuring software efficiency.

There is a whole range of hardware and software issues and challenges associated with computing on ultrascale platforms. From user's requirements point of view we emphasize time constraints (closely related to the total execution time and the reservation of HPC resources), the energy consumption, resilience (measured as the average time between consecutive failures within the system), and the impact of speed, security and quality of service of the interconnection networks. From the user involvement perspective, the productivity, i.e., the effort required to develop an ultrascale application (either from scratch or adapting an existing application to a new computing platform) is relevant. This section discusses a number of hardware and software properties that are expected to be of utmost importance for ultrascale computing.

1.1. Hardware and infrastructure issues

The emergence of new hardware platforms aimed at achieving ultrascale performance involves new heterogeneous technologies, such as accelerators (GPUs, FPGAs, or many integration core (MIC) architectures), as well as techniques for reducing energy consumption or network enhancements.

1.1.1. Power consumption and energy efficiency

The reduction and control of the energy consumption per flop (floating-point operation) is essential for achieving sustainable ultrascale computing. Energy-efficient processors with features such as power gating or DVFS (dynamic voltage frequency scaling) are designed to enable a reduction of the energy consumption at hardware level. These energy-saving hardware features

are supported at different software levels. At the system software level, appropriate runtime systems can be developed to map computational parts of an application to hardware resources such that the overall energy consumption is reduced. The runtime systems are based on suitable load balancing and scheduling methods with the goal to minimize the resulting energy consumption. Access to energy-minimizing runtime systems at the application level opens an opportunity for developing energy-aware ultrascale applications.

As a basis for an energy-efficient mapping of computations to hardware resources, suitable energy models are required that capture the power and energy behavior of application programs [83]. These models have to take the computational characteristics of an application into consideration to provide good estimates for its power consumption on a target architecture. The availability of suitable energy models is an important requirement to address the energy behavior of ultrascale systems from the application perspective. Accurate energy models depend on the identification of the key influencing factors, which is still an open research question.

At the application level, it can be observed that different applications may lead to a different power consumptions, depending on the computational behavior of the application and the resulting usage of hardware resources [84]. For ultrascale systems, the memory access and communication patterns will definitely play an important role. The redesign and reimplementation of the algorithms/codes for ultrascale applications need to address the problem of extensive communication patterns. In addition, the use of specialised architectures such as FPGAs, which are known to have favourable flop/Joule ratios [46], can be considered for reimplementing some frequently used kernels from scientific codes, for example, parts of the linear algebra routines.

1.1.2. Sustainable data storage and data management

Ultrascale applications have a wide variety of data storage and management requirements. The execution time of traditional HPC applications is typically dominated by floating-point computations, i.e. they are computationally intensive. An emerging class of ultrascale applications are big data applications [50, 92, 95], for which the application performance is determined by the performance of data access operations. Such applications are essential in fields such as genomics, astronomy, high-energy physics, or data analysis in web-scale companies. Some complex applications also combine computationally intensive and data intensive parts. Examples from the life sciences domain include machine learning techniques that process, among other inputs, high-resolution images.

Computationally intensive applications are typically executed on hardware platforms with a centralized network-attached high capacity storage system. Such platforms may restrict the choice of resource allocation for the distributed multiscale applications to be described in Section 2. Applications transfer data from the storage system to the compute nodes, execute the computations, and write the results back to the storage system. The data transfer may be transparent to the applications, or exposed through a library for parallel I/O such as MPI-IO. Since the execution time is computation bound, the I/O bandwidth does not significantly influence the application performance. In contrast, data intensive applications require much higher I/O bandwidth. To achieve high aggregated I/O bandwidth, the storage can be distributed among the compute nodes such that the data to be processed is read directly from a local disk. The Hadoop Distributed File System [90] (an implementation of the Google File System design [35]) provides such a distributed storage system. Current distributed storage systems such as Spanner [22] provide data management services for data stored on geographically separated sites.

Data-intensive applications may be implemented using programming models such as MapReduce [25] or Spark [104], high-level languages such as Pig [76], SQL-like languages such as HiveQL [96], or libraries such as Mahout [78]. In addition, applications may require services such as fault-tolerance [25], random I/O [21], low-latency operations [59], iterative computations [104], incremental computations [39], transaction support [22], or secure data storage.

1.1.3. Self-configurability

FPGA accelerators, being reconfigurable, offer some desirable possibilities for introducing self-configurability in ultrascale platforms, allowing a flexible re-adjustment of the hardware features to match the requirements of a particular application. Compared to general-purpose computers of equivalent performance, FPGAs are characterized by a better performance to power consumption ratio and lower cost. The combination of a general-purpose processor and application-specific processors synthesized in a reconfigurable logic with a structure utilizing features of the executed algorithms allows an increase of the overall performance by orders of magnitude.

However, FPGA-based accelerators and reconfigurable computer systems (that use FPGAs as a processing unit) face some typical problems: (1) The process of coding applications requires a special program to perform computing tasks load-balanced between the general-purpose computer and the FPGAs. (2) FPGAs require designing application-specific processor soft-cores; (3) FPGAs are only effective for certain classes of problems and data patterns, for which the application-specific processor soft-cores have originally been developed. Problems related to designing heterogeneous computer systems with hardware accelerators, are discussed in [71].

1.1.4. Interconnection networks

The interconnection network (ICN) is a critical element of every high performance computing system. It strongly determines its overall performance as well as the development and the operating costs. Enabling future advances in ultrascale computing requires the development of efficient, flexible, and highly scalable ICNs. The main responsibility of an interconnection network is to provide fast and reliable data transfer with respect to point-to-point and collective communication. The requirements for communication performance of the network imply high and stable maximal bandwidth and low latency. In a contemporary parallel system, the ICN connects hundreds of thousands of computing nodes. As the amount of computing nodes increases, the communication traffic and the resulting latency can rise dramatically, resulting in a degradation of system's computational performance. In order to overcome the scalability limitations, the developers usually implement enhanced interconnection networks based on high radix switches and specially tuned up topologies, routing algorithms and flow control mechanisms. Furthermore, the operating system and the management of the communication between the processes are highly optimized to efficiently utilize the communicational resources. From user's point of view, the underlying structure and characteristics of the network are known and can be used for optimization of the parallel code. In a sense, it is of paramount importance for the resulting performance to take the structure of the ICN into account when developing a specific code for ultrascale systems.

Due to their direct influence on speed-up and scalability, and consequently the run-time and power consumption of applications, ICNs play an important role in cooperating and coordinating

ultrascale computing systems. Today the ICN's performance has the same relevance as the performance of the CPU because the execution time depends on both communication time and computation time. The efficiency of most realistic parallel applications is determined to a large extent by the architecture's ICN. Matching the application communication patterns to the architecture of the ICN can shorten the overall execution time and increase the number of processors that can be efficiently exploited, which both leads to a higher ultimate speedup. The throughput, performance, low latency and quality of service of the ICNs are crucial for achieving scalability and good performance when applications are run on federated HPC resources (see, for example the multi-scale model described in Section 2). The performance of the ICNs for realistic multiscale applications has been studied in detail in [23].

The performance of ICNs depends on many factors with the three most relevant ones being topology, routing, and flow-control algorithms. The routing and flow-control algorithms have advanced to a state where efficient techniques are already developed and used [24]. Many network topologies have already been present since the dawn of parallel computing and are still widely used. With contemporary standards like Infiniband, vendors and end-users do not have the possibility to alter the routing and flow-control. Recent initiatives in the Network Functions Virtualization (NFV) support software-based virtual implementations of networking devices [28] such as switches, routers, firewalls, traffic analyzers, load balancers, etc. NFV can be easily combined with the concept of Software Defined Networking (SDN), which improves the performance and manageability of network functions. The end users can apply SDN to define a number of different topologies, based on an anticipated usage.

A further step towards a performance increase is made possible by an improved ICN topology or by innovative technological approaches in optical networking, which could solve current ICN bottlenecks such as message latency and non-efficient collective communication. New approaches in Networks on Chips (NoC) with high-level node radices will be considered as an option for further improvement of the performance on the chip level. It is expected that ICNs will be able to adapt dynamically to the current application in some optimal way in the near future. It is important to analyze the applications requirements regarding the currently used ICN technologies in HPC parallel systems [97], focusing on ICNs used in the present top-level systems. Based on past and present technology trends it is also relevant to establish several proposals for future development of ICNs that are expected to fit better the needs of high-performance parallel computer applications [98] or to be specifically tailored to exascale applications.

Most of the applications from Section 2 are sensitive to the ICN performance because of underlying matrix operations and advanced data structure with complex, often non-local, data manipulation. For example, well-known parallel performance degradations that can be observed for computations with sparse matrices, see Section 2.2.6, could be overcome in part by developing data traffic models that try to optimize those for frequently used sparse matrix kernels. These computations are inherently problematic on parallel architectures due to their low computation to communication ratios.

1.1.5. Cloud computing

The cloud is a type of parallel and distributed system consisting of a collection of interconnected and virtualized compute facilities, which are shared between users and can be used by a service mechanism. Thus, the cloud can play an important role for ultrascale computing. The term cloud computing refers both to the hardware and software providing the services and to

the application software provided as a service over the internet [11]. A network of parallel and distributed compute facilities belonging to different administrative domains has already been used in grid computing, which has been successful in scientific applications usually provided as task applications, scientific workflows, or MPI applications. However, grid computing has also some disadvantages, such as applications not fitting to those programming models or issues of being unable to get access to grid infrastructures. Cloud computing aims at solving these problems by, e.g., providing the entire computing stack from hardware to the application level and a pay by use basis.

The use of cloud computing for scientific application and high performance is being discussed and investigated recently. Programming models comprise task models, thread models, MapReduce models, scientific workflows, actor models or the parameter sweep model (PSM) [100]. Popular infrastructures include Amazon EC2, Google App Engine, Microsoft Azure, or Manjrasoft Aneka. The performance and cost are of specific interest. The Amazon EC2 infrastructure is evaluated for the scientific code FEFF [85] and for several other scientific workflows [53]. Open-source cloud computing solutions, including Eucalyptus, CloudStack and OpenNebula, have been studied with respect to their use and performance in geosciences [45]. A recent study on the application of cloud computing to scientific workflows discusses data intensive applications [13]. The active scientific discussion of this topic shows that there are very promising approaches and that cloud computing may play a very important role for scientific applications in ultrascale computing. The integration of cloud computing and big data is a necessary next step and first studies are given in [10].

1.2. Program Execution Issues

Program execution issues comprise properties such as scalability, resilience, security, integrity and privacy of data, which are connected with properties of the actual processing of applications.

1.2.1. Scalability

Throughout the development history of parallel systems, scalability was one of the pivotal features, and will remain so when moving to ultrascales. Two aspects of scalability have to be mentioned. The first one is the scalability of the hardware resources, ensuring that adding more such resources will allow the maintainance of the system's properties and operability. The second aspect is the software scalability, measured as the effect of increased computational and communication load as well as increased input and output, which can sustain a near-peak performance. A scalable system should allow an upscaling from few-variable small data sets on current parallel systems to many-variable large data sets on future ultrascale systems while maintaining high level system's efficiency.

Two principal types of hardware scalability can be distinguished: *scale-out scalability* means that more compute nodes are added to a system, and *scale-up* scalabilty means that more resources are added to a node, where the previously mentioned heterogeneity at micro-level can play a role [72]. Scalability raises difficult challenges as well in the case of substantial increase of the size of databases where strong consistency might be needed to be replaced by the weaker eventual consistency.

Software scalability captures the behavior of an application on a hardware system with substantial computing resources on which the application should exhibit satisfactory efficiency [74]. For HPC systems, software scalability is typically related to the execution time and the resulting speedup when running the application. We distinguish between *strong scaling* or *fixed size scalability*, capturing the scaling behavior of a problem of fixed size running on increasing resources, and *weak scaling*, capturing the scaling behavior when the problem size is increased in line with the number of computational resources, thus keeping the computational load per resource, e.g. the CPU core, roughly constant. However, weak scaling is much more relevant for ultrascale execution. Many applications, such as those presented in Section 2, would struggle to achieve very good strong scaling beyond hundreds of processors, but with a suitable load per processor may weakly scale way beyond that. Software scalability is especially important, since the resources of the ultrascale systems should be used efficiently without a significant re-writing of the application code when porting it to another hardware platform.

As very different heterogeneous system architectures are expected to become available in the near future, software adaptivity also plays an important role in the context of ultrascale systems. Ideally, the application software should be able to adapt automatically or with minimal changes to a new execution situation on a new architecture and a good scalability may ease this process. Scalability for ultrascale systems requires novel or improved approaches, such as task-based approaches or Network Functions Virtualization and Software Defined Networks as previously discussed in Section 1.1.4.

1.2.2. Resilience

Efficient utilisation of ultrascale computer architectures in scientific computing is restricted by possible deficiencies in the availability and reliability of the resources. To handle hardware failures, the software needs fault-tolerance features, such as checkpointing and rollback facilities. In particular, the possibility of unavailable resources requires that an application has frequent checkpoints for synchronisation and correctness verification. For example in the case of a multiscale model described in Section 2, the information exchange points between the models are the natural choice for checkpointing. Changing of the runtime system configuration caused by failing processing elements are usually difficult to handle by deterministic numerical algorithms and require a complete restart of the application. This can be alleviated to some extent by checkpointing. Efficient mapping of numerical algorithms to high-end architectures should allow for robust execution in the presence of hardware failures, either by the ability to take preemptive actions before a failure affects a running application, or, by creating a (hardware/software) fault-tolerant version of the algorithm, capable of recovering the solution within a timescale that is much shorter than that of re-running the entire application. An example is a fault-tolerant multigrid solver for exascale computation [47]. In addition, the computational error introduced by this process should be mathematically bounded, e.g. easy to measure and control. Some algorithms, such as iterative solution methods, can handle a certain amount of (non-repetitive) hardware failures and regain consistence without requiring any additional hardware features, see [19] for a more detailed discussion.

1.2.3. Security, integrity and privacy of data

Security, integrity and privacy of data are essential in the development of state-of-the-art ultrascale computing systems, in particular if these are cloud-based. The intrinsic heterogeneity of such systems induces flexible and ever-changing structures in the sense of geographical and logical distribution of the hardware resources, thus allowing scalability of the system to a very large size. Unfortunately, this architectural concept requires numerous remote data transfers. The problem is aggravated when we take into account that nodes distributed on different geographical points could be instructed to execute simultaneously several codes on a given data set, which is frequently locally stored.

The transfer and storage of data poses security threats, which includes risks involved in the transfer itself and security risks connected with the enormous scaling of the system. Each time a new node is connected to the system, the security risks could increase, as the new node could be infected by an ill-intended code or it could be a Trojan node. Moreover, frequent data transfers could be intercepted and the data could be stolen and used for unwanted purposes. The data security, especially in today's big data world, still remains an open problem. An interesting concept that is yet to be investigated is to consider whether computations could be performed on coded data.

1.3. Program Development Issues

Program development may constitute a significant effort, especially for ultrascale systems, and effective support for reducing program development time is important. In this section, we discuss the related issues of programmability, portability, and productivity in more detail.

1.3.1. Programmability

Programmability of highly parallel, heterogeneous computing systems is a major concern for potential applications. It expresses the ability to implement an application in such a way that ultrascale computer systems can be efficiently exploited to ensure its high-performance execution and good utilisation of computer resources. The term programmability includes the requirement for portability, since parts of or the entire application may have to be ported to newer and larger hardware platforms. Well-established and standardized programming models, such as MPI or OpenMP, as well as portable libraries or simulation tools are important to support portability. In the context of grid applications, such as a distributed multi-scale problem discussed in Section 2, various flexible coupling tools have been developed to couple existing (parallel) single-scale models. Such coupling tools should be flexible and generic as much as possible, thus minimising the development effort and maximising software reusability [14, 16].

However, porting codes are usually not enough to achieve high performance and a redesign or sometimes even a reimplementation of an application might be required. The complex task of redesigning an application has to be supported by a programming environment and a concise programming model for ultrascale applications. Such a programming model should provide an abstract view of the coarse (top-down) structure of an application. The specific way to support programmability is still to be investigated and proposed solutions may be application-specific. For example, a formalism based on complex automata was developed for the design of multiscale models, and a markup language, called MML has been designed to allow their formal description [15, 44]. The implementations based on the abstract view may include many well-

known subsolutions and the inclusion of standards, such as MPI, seems reasonable [82]. The requirement for the support of programmability will be investigated for specific applications in Section 2. Programmability is also strongly related to productivity in code design for sustainable ultrascale computing.

Task-based programming approaches in combination with suitable runtime systems can support the software adaptivity of applications, since the task-based approach allows a hardware-independent formulation of the application and the mapping of tasks to hardware resources can be performed by a runtime system so that the resources are efficiently used. The task-based programming decouples the specification of application's computations from the actual mapping to the computing resources. The runtime system can dynamically map tasks that are ready for execution to the computing resources, thus providing a dynamic load balancing that can adapt to the current execution situation of the hardware platform. This can help to enable an efficient use of the computing resources and a good overall scalability of the application, provided that enough tasks are available for execution at each point in time during the execution of the application. An example of a high-level runtime system that allows the allocation of federated HPC resources for distributed component applications, e.g. a loosely coupled multiscale model, is the Application Hosting Environment (AHE) [38]. Task-based approaches can be used with single-processor tasks [58], where each task is executed by a single execution unit, or multiprocessor tasks, where each task can be executed by multiple execution units in parallel [80, 81]. In the latter case, the actual number of execution units can be adapted to the execution situation at the time of task execution. Task-based approaches can also be used for balancing load between CPU and GPU by providing tasks in different versions for different platforms such as CPU or GPU and assigning a suitable version to the platforms with free resources [66].

The task-based programming paradigm is a promising approach for developing scalable solvers for ultrascale computers. However, it has to be taken into account that a considerable number of large real world applications are dealing with parallel algorithms for models based on partial differential equations (PDE). Parallelization of such algorithms is based on the paradigm of data parallelism. It is important to investigate whether the existing parallel algorithms can be redesigned by using the task-based templates (e.g. Monte-Carlo methods).

1.3.2. Portability

The portability of parallel codes and algorithms is an essential issue for any specialist involved in parallel computing and applications. There are two aspects of portability: portability of functionality and portability of efficiency. Clearly, the portability issues are mainly connected to selection, definition and continuous improvement of programming languages and standards such as MPI, OpenMP and hybrid programming MPI+OpenMP. These have served as programming models very efficiently for the last 20 years. Now the situation is changing and new parallel architectures such as manycores GPU require new ideas and tools, e.g. CUDA. Interesting approaches in this direction include several new languages that are based on a Partitioned Global Address Space (PGAS) concept which uses a global address space that is logically partitioned between the resources such that each resource has a portion of the address space attached to it to support the locality of memory reference. Languages, such as Cilk, that are based on the concept of tasks or task-based libraries also provide a useful abstraction that can support the portability to new hardware systems, see the discussion in Section 1.2.1.

Another way to provide portability is to use special libraries and templates or macros, well adapted to some specialized types of problems. This is often done for stencil driven algorithms. Examples of such libraries and toolkits are PETSc, LAPACK, ML, Hypre, CVODE [3, 4, 6–8]. These software packages are expected to be highly scalable and adaptable when used in ultrascale systems incorporating different architecture. Due to the portability of the libraries, their use can increase the portability of application codes. There is an interaction between the development of scientific libraries and the usage of new programming approaches for the development of these libraries in the sense that the usage of specific programming approaches influences the characteristics and the efficiency of the libraries. The third way to provide portability is to use simulation tools such as OpenFOAM, COMSOL, ANSYS [1, 2, 5], where the portability of solvers is guaranteed by the developers of the software packages.

1.3.3. Productivity

Productivity refers to the efficiency of implementing applications on specific architectures. The resulting application should be functional and reasonably efficient. Productivity for ultrascale systems involves the effort required to extend existing (parallel) applications to the new ultrascale systems such that the available resources are sufficiently well exploited. It is clear that a complete re-architecturing and re-implementation of the application with a new programming model should be avoided if possible. For large applications, this would be a huge effort even if large code blocks could be re-used. However, it would be unreasonable to expect that no change in the software will be required in order to use it efficiently on an ultrascale system. A preferred scenario involves applications that can be adapted with minor changes to the new platforms. Another aspect of productivity is the extensibility of the application code to include new features and functionalities without negative effect on its scalability and efficiency.

The reimplementation effort needs to be sustainable in the sense of making an application reasonably efficient on various ultrascale architectures. The use of novel task-based runtime systems which decouple the concerns of the computation specification and its mapping to computational resources can be an important step towards an increase in software development productivity for ultrascale systems. Productivity is inherently intertwined with other requirements discussed in previous subsections. In particular, a good portability as well as a good scalability of an application code leads to a higher productivity.

Related to productivity are radically new cross-platform software development and performance analysis tools aiming at increasing the capabilities of the codes to take an advantage of the phenomenal power of ultrascale computing platforms. Examples of highly scalable debuggers that target exa- and ultrascale systems are TotalView and Allinea, that provide troubleshooting for a wide variety of applications including serial, parallel, multi-threaded, multiprocess, and remote applications.

Performance analysis of parallel codes is already increasingly difficult at existing scales. Therefore, novel paradigms and techniques to measure, track, analyse and visualize performance data are necessary to be developed, in order to facilitate faster and more intuitive analysis of a wide range of gathered performance data, including execution time, memory system behavior, power consumption and resiliency to faults.

2. Some specific applications and implementation requirements

In this section, specific applications are investigated concerning their requirements for exascale execution, as identified in Section 1. After discussing a prototype multisclae application in Subsection 2.1, we present several specific applications in Subsection 2.2.

2.1. A prototype multiscale application

Processes and phenomena of interest in many scientific disciplines involve a complex mix of sub-systems that may operate on inherently different spatio-temporal scales. To model accurately the behaviour of such systems one needs to develop a scheme which would capture with sufficient detail the contributions of each sub-model, and couple them seamlessly into a global, computationally feasible system. Such models are commonly referred to as *multiscale models*. Multiscale modelling has numerous applications, including astrophysics (simulation of thermonuclear processes in stars and galaxies [34]), biology (studies of live organisms, spanning from genome to the entire population [88]), high energy physics (modelling of the fusion process and nuclear reactors [41]), engineering (simulations of structures, devices and chemical processes [69]), environmental science (climate modelling, weather prediction [57]), and material science (nano-composites [93]), to name a few.

The main components of a multiscale method are the single scale sub-models, the scale bridging techniques, and the deployment strategies. To emphasize the challenges related to performing computer simulations of a multiscale problem on a ultrascale computing facility we restrict our attention to a model application involving two single scale sub-models with well separated spatio-temporal scales. The single scale sub-models are assumed to be implemented as parallel legacy codes (e.g. using MPI or OpenMP). Putting the single scale sub-models together is the main algorithmic and software engineering challenge in multiscale modelling. This process is referred to as the *scale bridging*. Domain-specific techniques, such as sampling, projection, lifting, homogenisation (coarse graining), micro-macro coupling are the instances of scale bridging techniques. In terms of the coupling strategies, we can distinguish between *tight* coupling and *loose* coupling. Tight coupling is a feasible alternative when spatio-temporal scales of the sub-models are close, or partially overlap. In such cases, monolithic coupling may be an advantageous option [40]. Loose coupling strategies are effective for multiscale models with sub-models operating on well separated scales. This approach is more flexible in terms of the reuse of legacy codes for single scale sub-models and their deployment on distributed HPC resources.

At the methodology level there is a number of challenges that are awaiting answers, for example, finding generic theories and formalisms for model coupling, defining the minimal set of conservation laws for scale bridging, formulating mathematically rigorous theories for multiscale modelling, including the error analysis [43]. At the implementation level, scale bridging is usually handled by the software middleware, commonly referred to as *coupling framework* [38]. In terms of the coupling patterns, we can distinguish *acyclically coupled* simulations, in which the sub-models (codes) are run sequentially, with the output of one model serving as an input of the other (i.e. the sub-models are not mutually dependent during the execution), or *cyclically coupled*, where a mutual interdependency (a feedback loop) between the sub-models exist. In the latter case the sub-models can be run either sequentially or concurrently.

Concerning the mapping of computational tasks to a specific architecture, distributed computing strategies are of particular interest for achieving ultrascale performance. An efficient

mapping of sub-models depends on the software systems that handle the advanced reservation of federated computational resources, monitor data transfers, and provide easy to use GUI on a server. An example of such a system is the Application Hosting Environment (AHE) [105].

Next, we discuss how the issues covered in Section 1 affect the execution of a distributed multiscale application. In terms of the hardware issues, mentioned in Section 1.1, power consumption and energy efficiency can be addressed by deploying single scale applications on the architectures most suitable for the underlying algorithms (see the discussion in [14]). In this context, the use of hardware accelerators for certain sub-tasks within single scale sub-models can be beneficial, providing that the legacy codes support such extensions. Based on the existing experience, the impact of the interconnections speed on the performance of distributed multiscale applications is well documented (see [23]) and no drastic change of the behaviour is forseen. If cross-site applications with significant data traffic are executed, or the applications have requirements for interactive graphical rendering and steering, the existence of high bandwidth, low latency dedicated network interconnects between the HPC sites is one of the crucial factors for achieving high-performance execution.

In terms of program execution issues, scalability of a multiscale application depends on the choice of HPC architectures to execute single scale sub-models [14]. This would involve considerations, such as employing data locality when using large data sets (cf. Subsection 1.2.3) and software licenses. To achieve better scalability of single scale sub-models, some reprogramming and even redesign of algorithms may be necessary. This also applies to the attempts to improve the resilience of single scale models, as well as utilize energy saving options if such are provided.

In terms of programming development issues, such as programmability (Subsection 1.3), and productivity (Subsection 1.3.3), the strategy of using well-established legacy codes, coupled together with general-purpose coupling frameworks [16], [38] and deployed flexibly on distributed HPC resources via the virtualisation tools, such as AHE [105] is currently considered to be state-of-the-art. However, significant programming effort may be needed in some cases to bring the single scale models to scale well on the emerging ultrascale architectures.

2.2. Scientific Computing applications

In the remaining part of this section we present seven applied problems, that are based on the solution of discrete systems of PDEs and lead to important classes of supercomputing applications. During the last few decades computational mathematics and numerical simulations have been steadily drawing much attention, enabling the development of advanced technologies and contributing to better understanding of numerous natural phenomena that are not tractable via classical theoretical research or lab or field experiments. Performing numerical simulation of very complex physical, biological or social systems enables the society to address important issues such as identifying environmental problems, improving technological processes, developing biomedical applications, new materials, etc. In addition, numerical simulations are sometimes the only viable option in studying large systems, for example when the experiments are too expensive, time consuming or unethical to perform.

A significant class of mathematical models involves partial differential equations (PDEs), which are converted into discrete models using some suitable discretization methods such as Finite Differences (FDM), Finite Elements (FEM), Finite Volume (FVM), Isogeometric Analysis (IgA), referred to as *local methods* or as Boundary Integral Methods (BEM), meshless methods or spectral methods, referred to as *global methods*. Some combinations of local and global methods

are also in use. We note that, in general, the local discretization methods give raise to very large linear algebraic systems of equations with *sparse* matrices, while the matrices arising from global methods are much smaller, but *dense*. The type of discretization method is tightly related to certain data structures that in turn have a significant impact on the potential performance of those problems, when implemented on HPC and parallel computer platforms.

The methods of choice for solving large sparse systems of linear algebraic equations are the iterative solution methods, in particular, the preconditioned Krylov solvers, see [87].

2.2.1. Finite element based supercomputer applications

Problem 1 [FEM]: Nowadays, FEM are considered as one of the leading computational technologies for continuum (macroscopic) modelling in science and engineering. The advanced FEM-based simulations are often inter-disciplinary and involve multiple spatio-temporal scales, leading to practically unlimited requirements for supercomputing resources. The FEM supercomputing applications are inherently computationally intensive. At the same time, the most frequently used algorithms and their parallel implementations, are strongly coupled, e.g., via scalar products that entail global reduction operations, causing specific requirements with respect to the balance between computations and communications. In this context, we focus on topics related to single process problems (scalar or vector) which could be stationary (e.g. elliptic PDEs) or time dependent (e.g. parabolic PDEs).

More than 70% of the entire computing time of FEM-based engineering simulations is spent in solving linear algebra problems. This can be verified by using popular libraries, such as Trilinos [8] and HYPRE [3]. The included efficient fast parallel preconditioned conjugate gradients type solvers are often of (nearly) optimal complexity, see e.g. the available implementations of algebraic multigrid (AMG) methods. However, the robustness of these solvers for some classes of problems is still a challenging problem. As an example, we mention models of strongly heterogeneous media and/or strong anisotropy, where the well-established solution techniques face difficulties. The same applies to singular perturbations of the elliptic PDEs, such as the convection-diffusion problem, which is a major issue in fluid mechanics and transport phenomena.

The efficient implementation of FEM models requires mesh generation and partitioning of the graph representing the sparsity pattern of the matrices in the resulting linear systems. The available mesh generators construct a coarse unstructured mesh (for example using Netgen), which is then refined uniformly in parallel. This is not necessarily a computationally optimal scenario and adaptive refinement may be a better option, but requires frequent grid repartitioning to preserve the load balance. One potential solution to this problem can be the low cost mechanism for particle distribution, described in Subsection 2.2.7. The complete parallel generation of conforming unstructured meshes is still a challenge. The next related problem concerns the mesh partitioning. To illustrate it, we could refer to the commonly used software packages ParMETIS and SCOTCH. These packages are based on recursive partitioning strategies balancing the measure of the sub graphs and minimizing at the same time the measure of the interfaces. In this respect, the quality of the results could be considered as acceptable. The problem is that the number of the neighbors is not properly controlled which leads to serious problems mapping the graph of the algorithm onto the graph of the parallel architecture.

The last related comment concerns the more general problem of balancing local and global communications. For parallel distributed systems with hundreds of thousands of pro-

cessors/cores, the global communications related, e.g., to dot product, Fast Fourier Transform (FFT), etc. have become one of the fundamental bottlenecks, indicating that some of the user communities will have to change their way of thinking. As a consequence, most probably some of the FFT-based codes will have to be modified to AMG solvers as a way to avoid the transposition step reducing also avoid the logarithmic factor in the almost optimal order of computational complexity. Similar to other approaches, AMG may have its own problems when mapped to an architecture with a large number of processors – recent works aimed at reducing the operator complexity, improving the quality of interpolation, reducing the communication patterns at coarse levels, exploring fine-grained parallelism, while retaining numerical robustness are definitely of interest.

Challenges towards ultrascale FEM applications: The energy efficiency of FEM applications on ultrascale systems is one of the key challenges. New proper metrics need to be developed and tuned to get a complex assessment of the related simulators. The fault-tolerance issues need to be addressed in a proper way at the method, algorithm and software implementation level. Iterative solvers and the time-stepping algorithms have inherently self-correcting mechanisms which should be developed further and tuned in the context of ultrascale computing. The development of specific algorithm-based fault tolerance mechanisms will become increasingly significant with the scale of the computer system. For example, faults that occur in the parallel geometric multigrid solver are studied in various model scenarios in [47].

The general conclusion is that systematic algorithmic adaptations will be required if anticipated ultrascale hardware is to fully utilise its potential for FEM applications. Such algorithms aim to minimize synchronizations, memory usage, and memory transfers, while extra flops on locally cached data are almost "free", see e.g. [55]. In multilevel/multigrid solvers this could be achieved by more aggressive coarsening. An important complementary approach is the hybridization of algorithms (including hybrid deterministic-stochastic solvers), aimed at better fit to the hybrid hardware architecture.

Current experience: The experience of the IICT-BAS team includes FEM supercomputing simulations of bio-medical, environmental and engineering problems. High parallel scalability (both, strong and weak) is obtained on heterogeneous Linux platforms, including IBM Blue Gene/P, HPC CPU clusters, and more recently hybrid CPU/GPU/MIC clusters. The used programming methods and environments include C, MPI, OpenMP, CUDA. Among more recently released libraries for platforms with accelerators, we could mention PARALUTION. More information can be found in [74, 75, 79].

2.2.2. Parallel preconditioning of multi-physics problems

Problem 2 [BlockPrec]: Recent advances in computational modelling techniques and the increasing computing power allow us to tackle complex multi-physics and engineering problems that involve several unknown physical quantities described by a system of PDEs, such as thermal convection, fluid-structure interaction, magnetohydrodynamics. Grouping the discrete unknowns of the same type imposes in a natural way a block structure on the coefficient matrix. In this context, block preconditioners are commonly deployed to accelerate the convergence of Krylov solvers. The block structure of the coefficient matrix enables the use of existing preconditioners for the constituent single-physics sub-problems and available software libraries (such as AMG implemented in Hypre [3] or Trilinos [8]) to solve approximately the scalar subproblems. Implementing the solution of subproblems using available highly tuned and computationally

efficient software toolboxes reflects the programmability and productivity issues, highlighted in Section 1. Block preconditioners also favour local interprocessor communications, as opposed to long range communications that are prevalent in global multigrid solvers for multi-physics problems.

Current experience: We have developed the block preconditioning framework (BPF) within OOMPH-LIB (see http://oomph-lib.org/), an object-oriented multi-physics finite element library [29, 73, 91]. The BPF facilitates rapid development of new preconditioners, while hiding the low-level implementation details (including parallelisation). However, the overall parallel performance of any multi-physics solver crucially depends on scalar solvers (usually library codes produced by third parties), such as algebraic multigrid (AMG). Standard AMG codes, such as BoomerAMG [3] (developed at LLNL) perform robustly and show good scalability over hundreds of processors. Novel coarsening techniques (PIMS, HIMS, non-Galerkin) and techniques for enhanced (long-distance) interpolation are designed to reduce communication and enhance scalability, while retaining robustness, especially for complex diffusion and convection-diffusion problems.

Challenges towards ultrascale FEM applications: In contrast to the extensive studies of the numerical properties of these methods, there is comparatively little work on resilience and energy efficiency of AMG solvers (some recent results are found in [47]). AMG and the more general multilevel and domain decomposition frameworks are among the most numerically efficient techniques for solving large scale linear systems with sparse matrices, arising from discrete PDE models. The numerical efficiency, usually measured in the number of iterations, required to obtain convergent solution, relates to the underlying hierarchical structure of these methods, that allows a fast transfer of error information through the utilization of *coarse* problem representations. The implementational counterpart of the coarse data structures is the long range communication, and this is expected to be a serious bottleneck for ultrascale systems. Local communications are very attractive for massively parallel implementations, however in general, these do not guarantee sufficient numerical efficiency. Therefore, upgrading the existing and developing new optimal linear solvers that perform efficiently on ultrascale computers while satisfying resilience and ennergy efficiency requirements is very relevant.

2.2.3. Numerical solutions of multi-physics problems using Meshless methods

Problem 3 [Meshless]: A common feature of the local discretization methods, such as DFM, FEM, FVM, IgA, BEM, is that they rely on some discretization mesh and this fact sometimes poses additional difficulties as resolving complex geometries, adaptive refinements that involve local mesh refinements and derefinements, large stencils, handling moving meshes to resolve dynamical interfaces, etc., see Subsection 2.2.1 and [73]. The latter requirements affect the parallelization and implementation of these methods on HPC platforms, namely, the load balancing, the amount of local communications, etc.

A promising alternative to the mesh-based methods is the class of the so-called Local Meshless Methods (LMM), based on scattered discretization points. Of particular interest are the methods that result in algebraic systems of equations with better conditioned matrices [63]. LMM allow for easy implementation of local refinements and derefinements [61], basis augmentation, increasing approximation quality, treating special features in the problem, such as sharp discontinues or other intricate situations, which might occur in complex simulations. These potential advantages can be usually accomplished by an increased number of discretization nodes

to preserve the desired accuracy with additional work in the identification of nearest meshless nodes that influence the solution [99]. The validation of potential benefits of LMM on ultrascale architectures remains a significant research challenge [32].

Current experience: A parallel computational framework for solving multi-physics problems based on LMM has been developed, (see http://www-e6.ijs.si/ParallelAndDistributedSystems/) providing the possibility to model and perform numerical simulations of problems originating from molecular dynamics, graph algorithms (clique) and discrete simulations (ECG simulator based on Action Potential in cells), multiple transport equations (heat, bio-heat, solute, radiation, etc.), multi-phase fluid flow (free fluid, porous media), phase change dynamics on micro-macro levels, drift-diffusion equations (semiconductor simulations) [62], r-adaptive dynamic nodal distributions, all on non-uniform domains. The code is parallelized and can be efficiently executed on multi-core computers and distributed systems [60]. The communication issues have been studied and tested on an in-house computing cluster. The solvers are coupled with an evolutionary multiobjective optimization package for automatic optimization of parameters. The local meshless code could be extended to exascale range and could be used for performance benchmarking on novel ultrascale hardware platforms.

2.2.4. Earth Sciences Applications

Problem 4a [GEO1]: Earth sciences include a number of scientific disciplines such as geology, geophysics, ecology, hydrology, oceanography, climatology etc., that are relevant for the living conditions of human society, the extraction of raw materials and circulation of wastes. Earth System Science mixes together physics, geology, geophysics, engineering, chemistry, mathematics and computations to study interconnected systems operating on extreme time and spacial scales where small-scale heterogeneities affect large-scale phenomena (see also the discussion in Section 2.1). The scientific research accommodating these scales pushes and challenges the frontiers of numerical and computational methods [31, 49, 103].

The complexity of physical, chemical and biological processes, as well as the volume of data structures makes the HPC an imperative for the analysis, modeling and simulation of the underlying processes. For such problems, where experiments are impossible, the extreme-scale computer simulations can enable the solution of high resolution models and the analysis of very large data sets, including: regional climate changes (sea level rise, drought and flooding, and severe weather patterns). Climate models, developed through decades, have over one million lines of code. At the same time the architectural changes of the computer platforms need more sophisticated algorithms and computer techniques [12, 51, 52], in order to utilize fully the computing power provided by the new hardware. Oceanographic, atmospheric and climate simulations are typical examples of applications that require HPC to process a huge volume of data (for example, the analysis of remote sensing data in space-time domains to evaluate environmental changes and predict their future evolution) [9, 68, 101]. Moreover, such an analysis must be coupled with the simulation of related environmental processes and interfaced with human activities.

A particular instance of Earth science applications is geophysical modelling and inversion. A related work within the FP7 project HP-SEE on "Geophysical Modeling and Inversion" is performed in the Center for Research and Development in IT of the Polytechnic University of Tirana in collaboration with specialists from the Faculty of Geology and Mining and of the Academy of Sciences in Albania. It illustrates the scalability of geoscience applications in HPC systems and the need for ultra-scale computing in order to cope with high resolution models

required for regional and local scientific and engineering studies. The project targets inverse problems, related to gravity anomalies using approximation based on relaxation methods with runtimes in the range of $O(N^8)$ where N is the spatial resolution in one dimension. Geosections are represented by 3D matrices, the structure and density of which changes during iterations. Due to 2D to 3D mapping, the problem is, in general, very ill-conditioned.

Current experience: The implementation has been developed in C with MPI and OpenMP. Tests have been performed on the HPCG Cluster at the Institute of Information and Communication Technologies, Bulgarian Academy of Sciences, and the SGE system of the NIIFI Supercomputing Center at University of Pécs, Hungary, with up to 1000 cores. It was possible to achieve in a reasonable runtime a resolution with $N = 80$ nodes corresponding to a spatial step of 50 meters. This may be sufficient when simulating gravity effects but the resolution should be much higher when studying other geophysical phenomena of interest, for instance, searching for 2D structures with thickness of the order of one meter in a complicated heterogeneous and partially anisotropic medium for applications in other areas, such as magnetism and electricity.

The implementation maintains a balance between energy consumption and network communications. To facilitate this, each process on the individual cores does some redundant computations, thus reducing the communication at the expense of the increased energy consumption.

Problem 4b [GEO2]: Another example of a geophysical application problem that is of significant importance and a potential impact for our society is the so-called glacial isostatic adjustment (GIA) that comprises the response of the solid Earth to redistribution of mass due to alternating glaciation and deglaciation periods. The processes that cause subsidence or uplift of the Earth surface are active today. To fully understand the interplay between the different processes, and, for example, be able to predict how coast lines will be affected and how glaciers and ice sheets will retreat, these have to be coupled also to recent global warming trend and melting of the current ice sheets and glaciers world wide.

Due to the extreme space and time scales involved (the simulations should be performed on the Earth globe for time periods of about 100000 years) the GIA processes can only be studied via computer simulations. A detailed model of the phenomena includes three-dimensional geometry, viscoelastic and inhomogeneous material behavior, self-gravitation effects, modelled via a coupled system of partial differential equations.

Current experience: Presently, at the Division of Scientific Computing, Uppsala University, a two-dimensional benchmark, often used by the geophysicists, has been studied from a point of view of accuracy as well as regarding numerical and computational efficiency. The problem is discretized using FEM and performance studies have been done on CPU and GPU platforms using OpenMP and MPI paradigms (cf. [27]). The long-term aim is to couple GIA modeling with other large scale models, such as Climate and Sea-level changes, Ice modeling etc.

Problem 4c [GEO3]: Massive paralleism can be achieved when simulating environmental systems, for instance studying surface water - groundwater interactions. The HydroGeoSphere (HGS) software package ([17]) is an advanced tool, allowing for modelling physics-based interactions and feedback mechanisms between the two compartments. HGS is a numerically demanding code implementing a 3D control-volume finite element hydrologic model with a fully integrated surface-subsurface water flow and solute, including thermal energy transport.

The model parameters employed in HGS need to be calibrated in order to adequately represent a given environmental system. So-called data assimilation systems provide an alternative to conventional model calibration systems: they allow sequential update of system states and

model parameters whenever new data becomes available, thus guaranteeing a continuous improvement of the predictions. The Ensemble Kalman Filter (EnKF) [33] allows to quantify the prediction uncertancies, thus providing an optimal data assimilation mechanism in conjuction with HGS and the environmental data. The prediction provided by an EnKF-based simulation is then represented by the statistical moments of the ensemble of realizations. Such modelling systems are ideal for parallel processing, due to the high number of required simulations, and the readjustment and the recalibration of the model parameters allowed by the EnKF data assimilation technologies.

Current experience: At the University of Neuchatel, a cloud-based environment (OpenStack, AWS S3 compliant object store) has been developed to provide *near-real-time* re-adjustment of system states and re-calibration of model parameters whenever new monitoring data becomes available [65, 67]. When simulating larger fine grained HGS models, parallelization is essential because tightly-coupled highly-nonlinear partial differential equations are solved. The target parallelization includes the assembly of the Jacobian matrix for the iterative linearization method and the iterative solution method, in this case the preconditioned BiCGSTAB method, [48]. Performance studies of linear solvers are currently undertaken on CPU and GPU using OpenMP and MPI paradigms.

2.2.5. OpenACC acceleration for Nek5000: spectral element CFD code

Problem 5 [CFD]: This application targets problems in Computational Fluid Dynamics (CFD), in particular, very large scale simulations of incompressible flow problems, efficient and robust solvers for the arising linear systems and achieving high performance efficiency on heterogeneous HPC platforms.

Current experience: At present, the numerical simulations are performed using Nek5000 – an open-source code for simulating incompressible flows and its discretization scheme is based on the spectral-element method [37, 70]. In this approach, the incompressible Navier-Stokes equations are discretized in space by using high-order, weighted residual techniques employing tensor-product polynomial bases.

The code is widely used in a broad range of applications and more than 200 users are using Nek5000 in the world. Within the EU project CRESTA (Collaborative Research into Exascale Systemware, Tools and Applications), PDC-HPC at KTH Royal Institute of Technology mainly focuses on software challenges using hybrid computer architectures with accelerators for ultrascale simulations in collaboration with KTH Mechanics, EPCC, Cray UK and Argonne National laboratory. We have ported the CFD code Nek5000 on massive parallel hybrid CPU/GPU systems and presented a case study of porting simplified version, NekBone, to a parallel GPU-accelerated system. We reached a parallel efficiency of 68.7% on 16,384 GPUs of the Titan XK7 supercomputer at the Oak Ridge National Laboratory. Currently the full Nek5000 code is ported and optimized to multi-GPU systems and can run on 1,024 GPUs. The application is written in mixed C/Fortran and requires a system with multi-GPUs.

As discussed in Section 1.3, a particular attention should be paid to portability and productivity of ultrascale applications on heterogeneous HPC architectures. Productivity will be decreased if codes are rewritten in a low-level language such as CUDA for GPU accelerator systems. With the OpenACC [77] compiler directives, to port Nek5000 to GPU systems only requires a few additional command lines of code [70].

Nek5000 employs a multigrid preconditioner that combines the Schwarz overlapping method with subdomain solvers based on the fast diagonalization method. A sophisticated multigrid preconditioner is expected to reduce the global solution time when the Nek5000 code is ported to multi-GPU systems. For further improvements we intend to investigate the efficient preconditioner discussed in Section 2.2.2.

2.2.6. The EULAG numerical model

Problem 6 [STENCIL]: The EULAG model [86] is an ideal tool for performing numerical experiments in a virtual laboratory using time-dependent 3D adaptive meshes and complex, time-dependent model geometries. The flexibility is due to the unique model design that combines the nonoscillatory forward-in-time (NFT) numerical algorithms and a robust elliptic solver with generalized coordinates. The code is written as a research tool with numerous options controlling the numerical accuracy, and allows for a wide range of numerical sensitivity tests. The computational core of EULAG consists of two main parts: MPDATA advective transport algorithm [94] and the Generalized Conjugate Residual (GCR) elliptic solver [18] with the Thomas preconditioner. A more sophisticated preconditioner could be used in this context, see Section 2.2.2.

When implementing the EULAG model, the most time-consuming parts correspond to stencil-based computations. The suitability of stencil-based computations for ultrascale systems is studied in [56] for the CFD simulations on clusters with GPU nodes and in [102] for mesoscale atmospheric modeling on the Tianhye-2 system with Intel Xeon Phi co-processors. In the multi-GPU implementation, the PCI Express bandwidth and synchronization overheads are among the main bottlenecks. An important observation for this rather complex code is that the operations performed at the subdomain boundaries are currently performed much slower than fast stencil operations in the subdomain interiors, which restricts the overall code performance. In line with this observations, the algorithm proposed for Tianhye-2 is simplified as much as possible, allowing to achieve weak scaling efficiency up to 6,144 nodes, with over 8% of the peak performance in double precision.

Current experience: The development of the model and the implementation are a collaborative effort between the Czestochowa University of Technology, Poznan Supercomputing and the Networking Center, Institute of Meteorology and Water Management in Warsaw. The code is ported to multi GPUs and multi Intel Xeon Phi platforms [86, 94]. The application is written in C++/Fortran using CUDA, OpenMP and MPI standards. It requires a cluster with nodes containing NVIDIA GPUs or Intel Xeon Phi co-processors. The application is optimized for Fermi and Kepler GPU architectures, as well as Intel MIC architecture. Memory requirements include about 20 GB of HDD (for input and output data), and about 16 GB of RAM per node and about 8 GB of inter co-processor memory.

The performance of the EULAG system within a single node of cluster is mostly limited by the low flop-per-byte ratio of computation - less than 1.7 for MPDATA, and even less than 0.2 for the GCR solver, while the minimum flop-per-byte ratio required e.q. by NVIDIA Tesla K40m to achieve the maximum performance is 5.2. The main constraint for providing scalability of the application across cluster nodes is the presence of global communications in the GCR elliptic solver. As already pointed out, this is the crucial bottleneck for all sparse matrix calculations – a low computation to fetch ratios, made even worse on multicore architectures where multiple cores have common fetch buses.

2.2.7. Particle–In-Cell method for particle distributions – Helsim

Problem 7 [SPACE]: Space weather refers to conditions on the Sun, in the interplanetary space and in the Earth space environment. These conditions can influence the performance and reliability of space-borne and ground-based technological systems, and can affect human life or health [54].

Astrophysicists researching space weather are interested in the behavior of plasma, a high energy and highly conductive gas, where the atoms have been broken into their nuclei and their freely moving electrons. To study the small-scale plasma behavior, necessary to understand its kinetic effects, the Vlasov equation, self consistently coupled with Maxwell's equations, needs to be solved [64]. This is commonly solved using the particle-in-cell (PIC) method [42]. In PIC, individual macro-particles in a Lagrangian frame, that mimic the behaviour of the distribution function, are tracked in a continuous phase space. Moments of the distribution function, such as charge densities for plasma physics simulations, are computed simultaneously on an Eulerian frame (fixed cells).

Current experience: The software package Helsim implements an explicit three-dimensional electro-magnetic PIC simulation. It was developed in the ExaScience Lab in Leuven, Belgium, with contributions from many partners in the project. A particular care of load balancing is taken in Helism, which allows the simulation of experimental configurations with highly non-uniform particle distributions. Moreover, Helsim includes interactive in-situ visualization capabilities.

Helsim uses the Shark Library [20] [15] to store all distributed data structures, including the particles and the various grids. Shark is a high-level library for programming distributed n-dimensional grids in a highly productive manner, aiming to improve programmability and productivity while remaining portable (see Section 1.3). Broadly speaking, Shark manages the bookkeeping and the distribution of grid data structures, and offers specific computation and communication operations to work with the grid data. It extensively uses C++ constructs, such as lambda-expressions, to make the work with distributed n-dimensional grids easy. The Shark runtime system manages parallelism on three levels, which are common in today's multicore cluster architectures: distributed memory parallelism using one-sided communication from MPI 2/3; shared memory parallelism using a thread scheduler such as OpenMP, direct Pthreads, Intel Threading Building Blocks (TBB), and others; and SIMD vector instructions using compiler auto-vectorization, assisted with pragmas. The work on Shark continues in the context of the FP7 project Exa2ct, where we are integrating with the GASPI/GPI (Global Address Space Programming Interface) [16].

Specific for Helsim, when compared to other PIC simulators, is that particles are evenly distributed over the cluster such that each core holds the same amount of particles, stored according to the cell they belong to. As particles move throughout space during the simulation a low-cost, lightweight mechanism is used to adjust the particle distributions. It was initially opted for this dedicated mechanism but it could be worthwhile to use (or at least get inspiration from) grid or mesh partitioning techniques as discussed in Section 2.2.1.

The 3D fields (electric field, magnetic field, etc.) are block-distributed over the cluster, completely decoupled from the particle data structures. When particle information is propagated to the fields (charge density and current interpolation), and vice versa (interpolating the electric and magnetic fields to particle positions) each core uses a local representation of the grid in

[15] Freely available on github: https://github.com/ExaScience/shark
[16] See http://www.gpi-site.com/gpi2/gaspi/

order to be able to work locally and overlap computation with communication (updating the actual distributed grid). When done, this local information is then propagated to (and merged with) the distributed grid.

The current version of Helsim uses a fairly simple CG solver but we are currently integrating the state-of-the-art pipelined CG solvers developed at the lab [36]. It may be worthwhile to explore whether preconditioning techniques might help even more, see Section 2.2.2.

Helsim also features in-situ visualisation that runs in parallel with the simulation, directly using the data from the simulation, using a custom distributed raycasting engine. Helsim was run on up to 32 thousands cores on the Curie T0 System in France, as well as on various smaller clusters. The primary goal is to increase *weak scaling* (see 1.2.1), which is important to be able to tackle ever larger simulations. Helsim was developed at the ExaScience Lab, a collaboration between Imec, Intel, and all five Flemish universities.

Conclusions

We have put forward a number of hardware, software and program execution issues which will, in our opinion, influence the development and upgrades of computing applications for future ultrascale architectures. In this context, a prototype multiscale application and seven specific simulation applications from science and engineering serve as a testbed for a discussion on their current state-of-the-art and their future development directions, based on current experience. Although each of the applications has specific demands for porting to ultrascale systems, a pattern favouring coupled applications of increasing complexity and size as candidates for ultrascale execution, is clearly emerging. This means that future computational power will be used for advanced applications simulating increasingly complex phemonena, while reusing the existing single scale/physics models. Thus, it seems important to invest an effort into support for programmability enabling the porting of legacy codes into a single application that couples well-established sub-models. A variety of general or domain-specific well-established coupling platforms already exist, and it is timely to provide the standardization, which would support productivity for the application programmer. Full and proper understanding the algorithmic or mathematical model behind the coupling is essential for the standardization process, which is currently restricted to single coupled simulation codes. Mathematical standardization of model coupling would help to create more accurate and computationally efficient ultrascale applications. In summary, there seems to be a high potential for future ultrascale applications from the simulation problem point-of-view. A challenging task is the integration of the cross-cutting concerns, fault tolerance and reduction of the energy consumption. An open research question is whether these issues should be included into the simulation algorithm or being solved separately.

The work presented in this paper has been partially supported by EU under the COST programme Action IC1305, "Network for Sustainable Ultrascale Computing (NESUS)" and is co-authored by members of the Working group 6 on Applications of this action.

References

1. Ansys simulation software, http://www.ansys.com/.

2. Comsol multiphysics, http://www.comsol.com/.

3. Hypre, http://acts.nersc.gov/hypre/.

4. Linear algebra package (LAPACK), http://www.netlib.org/lapack/.

5. OpenFOAM, http://www.openfoam.com/.

6. Portable, Extensible Toolkit for Scientific Computation (PETSC), http://www.mcs.anl.gov/petsc/.

7. SUite of Nonlinear and DIfferential/ALgebraic equation Solvers (SUNDIALS), http://acts.nersc.gov/sundials/.

8. Trilinos ML, http://trilinos.org/packages/ml/.

9. Modeling Atmospheric and Oceanic Flows: Insights from Laboratory Experiments and Numerical Simulations. American Geophysical Union, Wiley, 2014.

10. D. Agrawal, S. Das, and A. El Abbadi. Big Data and Cloud Computing: Current State and Future Opportunities. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT/ICDT '11, pages 530–533. ACM, 2011. DOI: 10.1145/1951365.1951432.

11. M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A View of Cloud Computing. *Commun. ACM*, 53(4):50–58, April 2010. DOI: 10.1145/1721654.1721672.

12. S. Ashby, P. Beckman, J. Chen, P. Colella, B. Collins, D. Crawford, J. Dongarra, D. Kothe, R. Lusk, P. Messina, T. Mezzacappa, P. Moin, M. Norman, R. Rosner, V. Sarkar, A. Siegel, F. Streitz, A. White, and M. Wright. Report on Exascale Computing. Technical report, ASCAC, 2010.

13. G.B. Berriman, G. Juve, J Vöckler, E. Deelman, and M. Rynge. The Application of Cloud Computing to Scientific Workflows: A Study of Cost and Performance. *Phil. Trans. R. Soc. A*, 371(1983), January 2013. Funding Acknowledgements: NSF OCI-0910812, NSF OCI-0943725, NASA NCC5-626, and Amazon Educational Grant.

14. J. Borgdorff, M. Ben Belgacem, C. Bona-Casas, L. Fazendiero, D. Groen, O. Hoenen, A. Mizeranschi, J.L. Suter, D. Coster, P.V. Coveney, W. Dubitzky, A.G. Hoekstra, P. Strand, and B. Chopard. Performance of distributed multiscale simulations. *Philospohical Transactions of the Royal Society*, A372:20130407, 2014.

15. J. Borgdorff, J.-L. Falcone, E. Lorenz, C. Bona-Casas, B. Chopard, and A.G. Hoekstra. Foundations of distributed multiscale computing: formalization, specification and analysis. *Journal of Parallel and Distributed Computing*, 73:465–483, 2013.

16. J. Borgdorff, M. Mamonski, B. Bosak, K. Kurowski, M. Ben Belgacem, B. Chopard, D. Groen, P.V. Coveney, and A.G. Hoekstra. Distributed multiscale coupling with MUSCLE2, the Multiscale Coupling Library and Environment. *Journal of Computational Science*, 5:719–731, 2014.

17. P. Brunner and C.T. Simmons. HydroGeoSphere: A Fully Integrated, Physically Based Hydrological Model. *Ground Water*, 50(2):170–176, 2012.

18. Eisenstat S. C., H. C. Elman, and M. H. Schultz. Variational iterative methods for nonsymmetric systems of linear equations. *SIAM J. Numer. Anal.*, 20:345–357, 1983.

19. F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. Toward Exascale Resilience: 2014 update. *Supercomputing Frontiers and Innovations*, 1:5–28, 2014.

20. I. Chakroun, T. Vander Aa, B. De Fraine, P. Costanza, T. Haber, R. Wuyts, and W. Demeuter. ExaShark: A Scalable Hybrid Array Kit for Exascale Simulation. In *Proceedings of 23rd High Performance Computing Symposium (HPC 2015)*, 2015.

21. F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D. A Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

22. J.C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J.J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, and et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

23. P.V. Coveney, G. Giupponi, S. Jha, S. Manos, J. MacLaren, S.M. Pickles, R.S. Saskena, T. Soddermann, J.L. Suter, M. Thyveetil, and S.J. Zasada. Large scale computational science on federated international grids: The role of switched optical networks. *Future Generation Computer Systems*, 26:99–110, 2010.

24. W. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2004.

25. J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

26. J. Dongarra and et al. The International Exascale Software Project Roadmap. *International Journal of High Performance Computing Applications*, 25(1):3–60, February 2011.

27. A. Dorostkar, D. Lukarski, B. Lund, M. Neytcheva, Y. Notay, and P. Schmidt. Parallel performance study of block-preconditioned iterative methods on multicore computer systems. In *Proceedings of the Europar 2014 conference*. Springer LNCS, 2014.

28. D. Drutskoy, E. Keller, and J. Rexford. Scalable network virtualization in software-defined networks. *IEEE Internet Computing*, 17:20–27, 2013.

29. H.C. Elman, M.D. Mihajlović, and D.J. Silvester. Fast iterative solvers for buoyancy driven flow problems. *Journal of Computational Physics*, 230(10):3900–3914, 2011.

30. V. Escuder, R. Duran, and R. Rico. Analysis of x86 ISA Condition Codes Influence on Superscalar Execution. In Aluru S., Parashar M., Badrinath R., and Prasanna V.K., editors, *High Performance Computing - HiPC 2007: 14th International Conference, Goa - India*, 2007.

31. A. Fichtner, D. Giardini, A. Jackson, T. Nissen-Meyer, D. Peter, J. Robertsson, P. Tackley, L. Dalguer, D. Roten, O. Schenk, and M. Grote. HPC Roadmap. White paper, Solid Earth Dynamics, 2013.

32. M.J. Flynn, O. Mencer, V. Milutinović, G. Rakocević, P. Stenstrom, R. Trobec, and M. Valero. Moving from Petaflops to Petadata. *Commun. ACM*, 56(5):39–42, 2013.

33. H.J. Hendricks Franssen and W. Kinzelbach. Ensemble Kalman filtering versus sequential self-calibration for inverse modelling of dynamic groundwater flow systems. *Journal of Hydrology*, 365(3-4):261–274, 2009.

34. B. Fryxell, K. Olson, P. Ricker, F.X. Timmes, M. Zingale, D.Q. Lamb, P. MacNeice, R. Rosner, J.W. Truran, and H. Tufo. FLASH: An Adaptive Mesh Hydrodynamics Code for Modeling Astrophysical Thermonuclear Flashes. *The Astrophysical Journal*, 131:273–334, 2000.

35. S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

36. P. Ghysels and W. Vanroose. Hiding global synchronization latency in the preconditioned Conjugate Gradient algorithm. *Parallel Computing*, 40(7):224–238, 2014.

37. J. Gong, S. Markidis, M. Schliephake, E. Laure, D.S. Henningson, P. Schlatter, A. Peplinski, A. Hart, J. Doleschal, D. Henty, and P. F. Fischer. Nek5000 with OpenACC. In S. Markidis and E. Laure, editors, *Solving Software Challenges for Exascale*, pages 57–68, Heidelberg Dordrecht London New York, 2015. Springer LNCS 8759.

38. D. Groen, S.J. Zasada, and P.V. Coveney. Survey of Multiscale and Multiphysics Applications and Communities. *Computing in Science and Engineering*, 16:34–43, 2014.

39. P.K. Gunda, L. Ravindranath, C.A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic Management of Data and Computation in Datacenters. In *OSDI*, volume 10, pages 1–8, 2010.

40. M. Heil, A.L. Hazel, and J. Boyle. Solvers for large-displacement fluid-structure interaction problems: Segregated vs. monolithic approaches. *Computational Mechanics*, 43:91–101, 2008.

41. D.J. Hill. Nuclear Energy for the Future. *Nature Materials*, 7:680–682, 2008.

42. R. Hockney and J. Eastwood. *Computer Simulation Using Particles*. Taylor & Francis, 1988.

43. A. Hoekstra, B. Chopard, and P. Coveney. Multiscale modelling and simulation: a position paper. *Philosophical Transactions of the Royal Society*, A372:20130377, 2014.

44. A.G. Hoekstra, E. Lorenz, J.-L. Falcone, and B. Chopard. Towards a complex automata formalism for multiscale modeling. *International Journal for Multiscale Computational Engineering*, 5:491–502, 2007.

45. Q. Huang, C. Yang, K. Liu, J. Xia, C. Xu, J. Li, Z. Gui, M. Sun, and Z. Li. Evaluating Open-source Cloud Computing Solutions for Geosciences. *Comput. Geosci.*, 59:41–52, September 2013. DOI: 10.1016/j.cageo.2013.05.001.

46. S. Huang, S. Xiao, and W. Feng. On the energy efficiency of graphics processing units for scientific computing. In *IPDPS 2009. IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 2009.

47. M. Huber, B. Gmeiner, U. Ruede, and B. Wohlmuth. Resilience for Exascale Enabled Multigrid Methods. *arXiv*, 1501.07400v1, 2015.

48. H.-T. Hwang. *Development of a parallel computational framework to solve flow and transport in integrated surface-subsurface hydrologic systems*. PhD thesis, University of Waterloo, Canada, 2012.

49. L. Hwang, T. Jordan, L. Kellogg, J. Tromp, and R. Willemann. Advancing Solid Earth System Science Through High-Performance Computing. Technical report, Lawrence Livermore National laboratory, University of California, 2014.

50. H. V. Jagadish, J. Gehrke, A. Labrinidis, Y. Papakonstantinou, J.M. Patel, R. Ramakrishnan, and C. Shahabi. Big Data and Its Technical Challenges. *Commun. ACM*, 57(7):86–94, July 2014.

51. H. Johansen, D. Bernholdt, B. Collins, M. Heroux, R. Jacob, P. Jones, L.C. McInnes, J.D. Moulton, T. Ndousse-Fetter, D. Post, and W. Tang. Extreme-Scale Scientific Application Software Productivity: Harnessing the Full Capability of Extreme-Scale Computing. White paper, ASCR, 2013.

52. H. Johansen, L.C. McInnes, D.E. Bernholdt, J. Carver, M. Heroux, R. Hornung, P. Jones, B. Lucas, and A. Siegel. Software Productivity for Extreme-Scale Science Workshop Report. White paper, Workshop on Software Productivity for Extreme-scale Science, January 13-14, 2014, Rockville, MD, 2014.

53. G. Juve, E. Deelman, B. Berriman, B. Berman, and P. Maechling. An Evaluation of the Cost and Performance of Scientific Workflows on Amazon EC2. *J. Grid Comput.*, 10(1):5–21, March 2012. DOI: 10.1007/s10723-012-9207-6.

54. J. Kappenman. Geomagnetic Storms and Their Impacts on the U.S. Power Grid. Technical Report Meta-R-319, Metatech Corporation, January 2010.

55. D. Keyes. Keynote Presentation: Adapting Upstream Applications to Extreme Scale. In *EAGE Workshop on High Performance Computing for Upstream*, 2014.

56. A. Khajeh-Saeed and J. Blair Perot. Computational Fluid Dynamics Simulations Using Many Graphics Processors. *Computing in Science & Engineering*, 14(3):10–19, 2012.

57. R. Klein, S. Vater, E. Paeschke, and D. Ruprecht. Multiple scales methods in meteorology. In *Asymptotic Methods in Fluid Mechanics: Survey and Recent Advances, CISM Courses and Lectures*, volume 523, pages 127–196. Springer, 2010.

58. M. Korch and T. Rauber. A Comparison of Task Pools for Dynamic Load Balancing of Irregular Algorithms. *Concurrency and Computation: Practice and Experience*, 16:1–47, 2004.

59. M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *7th Biennial Conference on Innovative Data Systems Research (CIDR'15)*. CIDR, 2015.

60. G. Kosec, M. Depolli, A. Rashkovska, and R. Trobec. Super linear speedup in a local parallel meshless solution of thermo-fluid problems. *Computers and Structures*, 133:30–38, 2014.

61. G. Kosec and B. Šarler. H-adaptive local radial basis function collocation meshless method. *CMC: Computers, Materials, and Continua*, 26(3):227–253, 2011.

62. G. Kosec and R Trobec. Simulation of semiconductor devices with a local numerical approach. *Engineering Analysis with Boundary Elements*, 50:69–75, 2015.

63. G. Kosec and P. Zinterhof. Local strong form meshless method on multiple Graphics Processing Units. *CMES: Computer Modeling in Engineering and Sciences*, 91(5):377–396, 2013.

64. N.A. Krall and A.W. Trivelpiece. *Principles of plasma physics*. International Series in Pure and Applied Physics. McGraw-Hill, 1973.

65. P. Kropf, E. Schiller, P. Brunner, O. Schilling, D. Hunkeler, and A. Lapin. Wireless Mesh Networks and Cloud Computing for Real Time Environmental Simulations. In *Recent Advances in Information and Communication Technology - Proceedings of the 10th International Conference on Computing and Information Technology, IC2IT 2014, Angsana Laguna, Phuket, Thailand, 8-9 May, 2014*, number 265 in Advances in Intelligent Systems and Computing, pages 1–11. Springer, 2014.

66. J. L. Lang and G. Rünger. An execution time and energy model for an energy-aware execution of a conjugate gradient method with CPU/GPU collaboration. *Journal of Parallel and Distributed Computing*, 74(9):2884–2897, 2014.

67. A. Lapin, E. Schiller, P. Kropf, O. Schilling, P. Brunner, A. Jamaković-Kapić, T. Braun, and S. Maffioletti. Real-time envoronmental monitoring for cloud-based hydrogeological modelling with Hydrogeosphere. In *High Performance Computing and Communications conference HPCC*, Paris, 2014. IEEE.

68. M. Guest (lead author). The Scientific Case for High Performance Computing in Europe 2012-2020. Technical report, FP7 Project PRACE RI-261557, 2014.

69. A. Lucia. Multi-Scale Methods and Complex Processes: A Survey and Look Ahead. *Computers and Chemical Engineering*, 34:1467–1475, 2010.

70. S. Markidis, J. Gong, M. Schliephake, E. Laure, A. Hart, D. Henty, K. Heisey, and P. F. Fischer. OpenACC Acceleration of Nek5000, Spectral Element Code. *International Journal of High Performance Computing Applications (accepted)*, 2015.

71. A. Melnyk and V. Melnyk. *Personal Supercomputers: Architecture, Design, Application.* Lviv Polytechnic National University Publishing, 2013.

72. M.T. Fisher M.L. Abbott. *Scalability Rules: 50 Principles for Scaling Web Sites.* Addison-Westey, 2011.

73. R.L. Muddle, M.D. Mihajlović, and M. Heil. An efficient Preconditioner for Monolithically-Coupled Large-Displacement Fluid-Structure Interaction Problems with Pseudo-Solid Mesh Updates. *Journal of Computational Physics*, 231(21):7315–7334, 2012.

74. S. Margenov N. Kosturski and Y. Vutov. Balancing the Communications and Computations in Parallel FEM Simulations on Unstructured Grids. In K. Karczewski R. Wyrzykowski, J. Dongara and J. Wasniewski, editors, *Parallel Processing and Applied Mathematics*, pages 211–220, Heidelberg Dordrecht London New York, 2012. Springer LNCS 7204.

75. S. Margenov N. Kosturski and Y. Vutov. Computer Simulation of RF Liver Ablation on an MRI Scan Data. In M.D. Todorov, editor, *Application of Mathematics in Technical and Natural Sciences*, pages 120–126, Melville, NY, USA, 2012. AIP Conf. Proc. 1487.

76. C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

77. OpenACC. `http://www.openacc.org`.

78. S. Owen, R. Anil, T. Dunning, and E. Friedman. *Mahout in action.* Manning, 2011.

79. S. Margenov P. Arbenz and Y. Vutov. Parallel MIC(0) preconditioning of 3D elliptic problems discretized by Rannacher-Turek finite elements. *Computers and Mathematics with Applications*, 55(10):2197–2211, 2008.

80. T. Rauber and G. Rünger. A Transformation Approach to Derive Efficient Parallel Implementations. *IEEE Transactions on Software Engineering*, 26(4):315–339, 2000.

81. T. Rauber and G. Rünger. Tlib - A Library to Support Programming with Hierarchical Multi-Processor Tasks. *Journal of Parallel and Distributed Computing*, 65(3):347–360, 2005.

82. T. Rauber and G. Rünger. *Parallel Programming for Multicore and Cluster Systems, Second edition.* Springer, 2013.

83. T. Rauber and G. Rünger. Modeling and Analyzing the Energy Consumption of Fork-Join-based Task Parallel Programs. *Concurrency and Computation: Practice and Experience*, 27(1):211–236, 2015.

84. T. Rauber, G. Rünger, M. Schwind, H. Xu, and S. Melzner. Energy Measurement, Modeling, and Prediction for Processors with Frequency Scaling. *The Journal of Supercomputing*, 70(3):1451–1476, 2014.

85. J. Rehr, F. Vila, J. Gardner, L. Svec, and M. Prange. Scientific Computing in the Cloud. *Computing in Science and Engineering*, pages 34–43, 2010.

86. K. Rojek, M. Ciznicki, B. Rosa, P. Kopta, M. Kulczewski, K. Kurowski, Z. Piotrowski, L. Szustak, D. Wojcik, and R. Wyrzykowski. Adaptation of fluid model EULAG to graphics processing unit architecture. *Concurrency and Computation: Practice and Experience*, 27(4):937–957, 2014. DOI: 10.1002/cpe.3417.

87. Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, Philadelphia, 2nd edition, 2003.

88. S. Schnell, R. Grima, and P.K. Maini. Multiscale Modeling in Biology. *American Scientist*, 95:134–142, 2007.

89. Y. Shimizu and M. Takashi. Effect of topology on parallel computing for optimizing large scale logistics through binary PSO. In Lockhart Bogle I.D. and Fairweather M., editors, *Computer Aided Chemical Engineering Volume 30, Pages 1-1435 (2012) - 22 European Symposium on Computer Aided Process Engineering*, 2012.

90. K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10, May 2010.

91. C.A. Smethurst, D.J. Silvester, and M.D. Mihajlović. Unstructured finite element method for the solution of the Boussinesq problem in three dimensions. *International Journal for Numerical Methods in Fluids*, 73(9):791–812, 2013.

92. Science Staff. Special Collection: Dealing with Data. *Science*, 331(6018), 2011.

93. J. Suter, D. Groen, L. Kabalana, and P.V. Coveney. Distributed Multiscale Simulations of Clay-Polymer Nanocomposites. In *MRS Proceedings*, volume 1470. Cambridge University Press, 2012.

94. L. Szustak, K. Rojek, T. Olas, L. Kuczynski, K. Halbiniak, and P. Gepner. Adaptation of MPDATA heterogeneous stencil computation to Intel Xeon Phi coprocessor. *Scientific Programming*, page 642705, 2015.

95. S. Tansley and K.M. Tolle. *The fourth paradigm: data-intensive scientific discovery*. Microsoft Research, 2009.

96. A. Thusoo, J.S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

97. Top500. *Top500 supercomputers site*. Available on: http://www.top500.org, (accessed August 2014).

98. R. Trobec. Two-dimensional Regular d-meshes. *Parallel Comput.*, 26(13-14):1945–1953, 2000.

99. R. Trobec. *Advances in the MLPG meshless methods*, chapter Experimental analysis of methods for moving least squares support determination, pages 307–358. Duluth: Tech Science Press, 2009.

100. C. Vecchiola, S. Pandey, and R. Buyya. High-Performance Cloud Computing: A View of Scientific Applications. In *Proc. of the 2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks*, ISPAN '09, pages 4–16. IEEE Computer Society, 2009. DOI: 10.1109/I-SPAN.2009.150.

101. W.M. Washington and C.L. Parkinson. *Introduction To Three-dimensional Climate Modeling*. University Science Books, California, 2005.

102. W. Xue, Ch. Yang, H. Fu, Y. Xu, J. Liao, L. Gan, Y. Lu, R. Ranjan, and L. Wang. Ultra-scalable CPU-MIC Acceleration of Mesoscale Atmospheric Modeling on Tianhe-2. *IEEE Transaction on Computers*, 8, 2014. DOI: 10.1109/TC.2014.2366754.

103. C. Yang, M. Goodchild, Q. Huang, D. Nebert, R. Raskin, Y. Xu, M. Bambacus, and D. Fay. Spatial cloud computing: how can the geospatial sciences use and help shape cloud computing? *International Journal of Digital Earth*, 4(4):305–329, July 2011.

104. M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

105. S.J. Zasada and P.V. Coveney. Virtualizing access to scientific applications with the Application Hosting Environment. *Computer Physics Communications*, 180:2513–2525, 2009.

Received February 27, 2015.

DOI: 10.14529/jsfi150103

Dense Matrix Computations on NUMA Architectures with Distance-Aware Work Stealing

Rabab Al-Omairy[1]***, Guillermo Miranda***[2]***, Hatem Ltaief***[1]***, Rosa M. Badia***[2,3]***, Xavier Martorell***[2,4]***, Jesus Labarta***[2,4]***, David Keyes***[1]

We employ the dynamic runtime system `OmpSs` to decrease the overhead of data motion in the now ubiquitous non-uniform memory access (NUMA) high concurrency environment of multicore processors. The dense numerical linear algebra algorithms of Cholesky factorization and symmetric matrix inversion are employed as representative benchmarks. Work stealing occurs within an innovative NUMA-aware scheduling policy to reduce data movement between NUMA nodes. The overall approach achieves separation of concerns by abstracting the complexity of the hardware from the end users so that high productivity can be achieved. Performance results on a large NUMA system outperform the state-of-the-art existing implementations up to a twofold speedup for the Cholesky factorization, as well as the symmetric matrix inversion, while the `OmpSs`-enabled code maintains strong similarity to its original sequential version.

Keywords: Dense Matrix Computations, Dynamic Runtime Systems, Software Productivity, Non-Uniform Memory Access, Data Locality, Work Stealing, High Performance Computing.

Introduction

Multicore Non-Uniform Memory Access (NUMA) machines are increasingly common in high performance computing, and a primary challenge of extreme computing today (see, e.g., the international exascale campaign [13]) is in expanding the number of cores per node in strong scaling. In contrast, expanding the number of nodes, which has already reached 10^5 for the BlueGene/Q system ranked #3 in the November 2014 TOP500 list, in weak scaling is well understood, at least for typical scientific SPMD codes based on load-balanced domain decomposition run on performance-reliable nodes [21]. In the last decade, dense linear algebra software underwent drastic algorithm and software stack redesigns, to maintain pace with the hardware evolution towards high concurrency. The Standard dense numerical algorithms and their state-of-the-art implementations in `LAPACK` [5] and `ScaLAPACK` [7] rely on the bulk synchronous parallel model for performance purposes. This model will display increasing vulnerability to synchronization going forward. Parallel programming models based on fine-granularity computations have shown promising results to weaken global synchronizations and to reduce data motion. In particular, task-based programming models have been successfully developed and integrated in several high performance dense linear algebra libraries (i.e., `PLASMA` [1] and `Libflame` [30]). Along with "taskifying" existing dense numerical algorithms, one of the main challenges is to deal with the actual scheduling of these tasks and the ever-changing hardware with its NUMA complexity.

While the potential of multicore NUMA architectures can be exploited with considerable ease for algorithms with good load balance at arbitrary concurrency, such as matrix multiplication, there are others in which load balance and data locality cannot be maintained simultaneously,

[1]Extreme Computing Research Center, King Abdullah University of Science and Technology, Thuwal, Saudi Arabia, email: Rabab.Alomary, David.Keyes, Hatem.Ltaief@kaust.edu.sa

[2]Barcelona Supercomputing Center, Centro Nacional de Supercomputación, Barcelona, Spain, email: Guillermo.Miranda, Rosa.M.Badia, Xavier.Martorell, Jesus.Labarta@bsc.es

[3]Artificial Intelligence Research Institute (IIIA) - Spanish National Research Council (CSIC)

[4]Universitat Politècnica de Catalunya

and the cost of accessing remote NUMA nodes comes with performance penalties. This is the case of the Cholesky factorization and the symmetric matrix inversion algorithms studied in this paper, which are representative benchmarks of dense factorizations and more advanced dense matrix operations, respectively. These compute-intensive operations are the basic blocks for many scientific applications (e.g., in statistics and machine learning), which require the explicit calculation of the inverse of large covariance matrices [2]. The trade-off between load balance and data locality is the key not only to the future of algorithms with time-varying work per unit memory, such as the Cholesky factorization and even more for the symmetric matrix inversion, but also to the future of hardware that is less performance-reliable due to compromises required to reduce the energy of computation. As future hardware is operated closer to unit signal to noise ratio in voltage level and as core clock rates are varied to maintain safe thermal dissipation levels, work stealing will be required to maintain load balance even for algorithms with regular work per unit memory ratios throughout their execution. Work stealing that does not cost more in performance than the imbalance it is designed to rectify is challenging to arrange.

The authors herein consider a limited type of "distance-aware" work stealing that respects the critical path of execution and the realities of NUMA. We have extended some of the functionalities of the `OmpSs` task-based programming model [6], [15] to efficiently handle NUMA architectures through an innovative distance-aware scheduling policy to reduce data movement between NUMA nodes, for instance, while performing work stealing. We have chosen `OmpSs` as the programming model because it provides a simple and non-intrusive interface (OpenMP-like) to program challenging hardware systems by abstracting the hardware complexity from end users, while keeping high productivity in mind. This new scheduling policy is aware of the NUMA architecture on which it is running, spreads work over the available cores, and implements stealing to prevent starvation.

On shared-memory systems composed of tens of NUMA nodes and for asymptotic matrix sizes, the Cholesky factorization and the symmetric matrix inversion achieve up to a twofold improvement in flop rate relative to less discriminating policies, while improving performance by a twofold speedup over the best existing implementations for both dense matrix operations on the same hardware overall.

This paper is structured as follows. We continue with the related work in Section 1. Section 2 describes the `OmpSs` framework and presents its different components. In Section 3, we briefly recall the Cholesky factorization and the symmetric matrix inversion. The implementation details of the scheduling policy are given in Section 4. In Section 5, we present the performance impact of the scheduling policy on various systems and compare against the state-of-the-art commercial and open-source high performance dense numerical libraries. Section 6 shows performance traces of both algorithms to support our performance results, and we conclude in Section 6.

1. Related Work

The volume of literature on NUMA-aware work stealing indicates the importance of adapting many classes of algorithms, linear algebra among them, to strong scaling within shared memory. Our review cannot be complete within page scope, but focuses on contributions that provide the context for ours.

Runtime frameworks for scheduling dense linear algebra algorithms have been well studied in the last few years [10, 20, 23, 24, 26]. The key idea is the redesign of the numerical algorithms so that more parallelism can be exposed and a runtime system is then employed to concurrently

schedule the computational tasks. This is the approach adopted by the PLASMA and FLAME high performance dense linear algebra libraries. The PLASMA library [1] provides a collection of dense linear algebra operations and is intended eventually to supersede LAPACK [5]. It can use internally a static (originally introduced in [22]) or a dynamic runtime system [29]. It has shown significant improvement compared to the existing approaches [4]. The FLAME library [30] provides similar functionalities and relies on the dynamic runtime system SuperMatrix [11] to execute the algorithms-by-blocks. These are high-productivity runtime systems in the sense that the user does not need to adapt to the architecture at the source-code level. However, although the scheduling frameworks of both libraries aforementioned provide features for data locality, work stealing and task priority on shared-memory systems, they do not offer scheduling policies to cope with challenging NUMA architecture. Moreover, Jeannot [19] has proposed a symbolic mapping with a static data allocation on NUMA machines, specifically for the Cholesky factorization. The main idea is to group threads by NUMA nodes to exploit the memory hierarchy. This static methodology precludes work stealing and may further impede performance for load imbalanced applications.

Recently, LAWS [12] proposes a runtime library for Divide and Conquer applications in NUMA systems. It features a work stealing algorithm designed for NUMA systems, very focused in reducing remote memory accesses and last-level cache pollution. However, it does not take into account that distances between nodes might differ across the whole system; the applications targeted are recursive, unlike Cholesky; their tasks use the same amount of data, which allows the auto tuning of the cut-off threshold.

Drebes [14] presents a scheduling and allocation algorithm for the OpenStream language. While similar to this paper in topic and features (detection of the node with the most data for a given task, locality aware stealing that takes into account distances between NUMA nodes), there are considerable differences: a task is assigned to a thread based on its input location only, experimental results are validated only up to 64 threads, a slab allocator is used that may give incorrect information about the node where a memory chunk is allocated which in turn, according to the paper, results in a speedup of 0.99 over their base line (random stealing).

There are also standalone dynamic runtime system libraries for general purpose. HPT [28] presents an abstraction for task parallelism and data movement. The memory hierarchy is represented as a tree where workers belong to leaf nodes. In this approach, a task assigned to a memory place (cache, NUMA node, etc.) will be executed only by workers below its assigned place. For instance, a task assigned to an L3 cache can run in any core below that cache, but not in the cores that share a different L3 cache. In the scheduling policy described in this paper, we further leverage this principle to tackle NUMA node locality. The Wool library [16] presents an efficient work stealing approach, but it does not take into account data locality. Wool requires the programmer to modify the source code, whereas we use OmpSs to annotate the source code, with very few modifications. Furthermore, Wool tasks must be independent and therefore, the scope of applications susceptible to adhere to this restriction is rather narrowed. Recently, Muddukrishna [25] proposes some ideas to preserve locality in an OpenMP runtime/compiler infrastructure, but the evaluation was performed in a 48-core machine, and using benchmarks (such as the matrix multiplication) that do not have dependencies between tasks as complex as the dense matrix operations experimented in this paper. Last but not least, providing locality hints has been proposed and studied in Broquedis et al. [8], where scheduling hints are used to choose the best thread and data distribution. Their approach was evaluated using a 16-core

machine, while our experimental platforms are additionally composed by a 8-node, 48-core and a 128-node, 1024-cores NUMA systems. Similarly, `ForestGOMP` runtime maps nested thread teams to the underlying hardware resources.

2. The `OmpSs` Framework

`OmpSs` is a high-level, task-based, parallel programming model supporting SMPs, heterogeneous systems (like GPGPU systems) and clusters. `OmpSs` consists of a language specification, a source-to-source compiler for C, C++ and Fortran; and a runtime. The `OmpSs` programming model is powered by the Nanos++ runtime, which provides services to support task parallelism using synchronizations based on data-dependencies. Data parallelism is also supported by means of services mapped on top of its task support.

2.1. The Nanos++ Dynamic Runtime System

Figure 1 depicts the infrastructure of the Nanos++ dynamic runtime system. Nanos++ interfaces the application with the underlying hardware architecture. A core component of the runtime is in charge of handling data movement and ensuring coherency, so that the programmer does not need to deal with memory allocation and movements. Another core component is the module of scheduling policies, which will ultimately integrate the distance-aware work stealing policy introduced in this paper. `OmpSs` also supports heterogeneous programming and that is reflected in the modular support for different architectures in Nanos++ (SMP, CUDA, OpenCL, simulators such as tasksim, etc.). Nanos++ provides also instrumentation tools to help identifying performance issues.

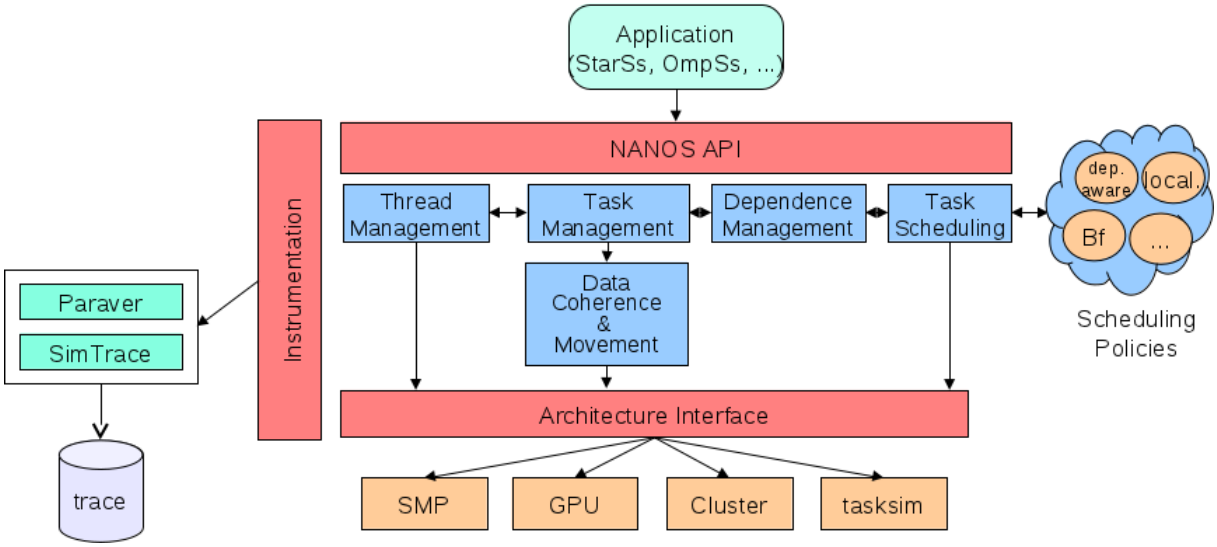


Figure 1. The Nanos++ infrastructure

2.2. Task-Based Programming Model in `OmpSs`

In `OmpSs`, data dependencies are contained in a directed acyclic graph (DAG). Tasks are blocked until all their dependencies are satisfied. Once a task is released, the scheduler is then in charge of taking the proper runtime decisions. Each scheduling policy defines a certain behavior. For instance, a simple policy might implement a global first-in, first-out (FIFO) queue, where

tasks are executed in the order of dependency release; or we could define a policy that holds one queue per worker thread, where tasks are sorted based on a priority value selected by the programmer. The thread management consists in a pool of worker threads that start to execute work once it becomes available. When a worker thread is aware of new available work, it will query the scheduling policy, which will provide a new task to the worker if it so decides. For instance, in a NUMA aware scheduling policy that ensures locality on top of every other aspect, if a worker thread is requesting new work and the scheduling policy only has work for threads of other NUMA nodes, it will not give a new task to the demanding worker thread, for the sake of enforcing locality. Regarding data dependencies, Nanos++ delegates that component to plugins, in a similar way to scheduling policies. As this runtime is designed to work with a wide range of applications, some may be simple (when it comes to specify dependencies) and do not need the added complexity inherent to the handling of more complex dependencies (such as non-contiguous memory regions) that other applications rely on.

2.3. `OmpSs` Tools for Performance Analysis

Nanos++ has an instrumentation plugin system used to obtain traces of the executions. In this paper, we have chosen the plugin that works with `Extrae`, the core instrumentation package developed by the Performance Tools group at Barcelona Supercomputing Center, and `Paraver`, a very flexible data browser developed by the same group. Together, they enable programmers to analyze the behavior of the applications, identify potential problems and understand how they can be solved. `Extrae` uses different interposition mechanisms to inject probes into target applications so as to gather information regarding the application performance. Nanos++ features an instrumentation module with support for `Extrae`, thus, obtaining traces for `OmpSs` applications is just a matter of running them with the instrumentation plugin enabled. As for `Paraver`, Nanos++ comes with a vast set of `Paraver` configuration files that convert tracing events to human-readable information that can be displayed as timelines (such as user functions duration), histograms (e.g., thread state to know how much the application was running, idling and the overhead introduced by the runtime) and three-dimensional histograms.

3. The Cholesky Factorization and the Symmetric Matrix Inversion Algorithm

This Section briefly recalls the standard block and the new tile variants of the Cholesky factorization and the symmetric matrix inversion. Further algorithmic details can be found in [3, 10, 11].

3.1. Block Algorithm Variant

Computing the Cholesky factorization is the first step toward solving dense systems of linear equations for symmetric positive-definite matrices, which arise in many scientific applications [17]. Based on the Cholesky factorization, the symmetric matrix inversion is also important for the computation of the variance-covariance matrix in statistics [18]. The state-of-the-art dense linear algebra library `LAPACK` [5] uses block algorithms. The computation is basically split into successive sequences composed by two phases: (1) the panel computation phase, mainly based on level 2 BLAS, in which the transformations are accumulated within a panel of the matrix

and (2) the update of the trailing submatrix, in which the transformations from the panel phase are applied at once to the trailing submatrix in terms of level 3 BLAS operations. One of the bottlenecks with such approach is the creation of unnecessary synchronization points between the phases. Moreover, `LAPACK` extracts its performance for the most part from the parallel multithreaded BLAS. The parallel paradigm behind it is very similar to the `OpenMP` fork-join model, which further exacerbates the issue related to artifactual synchronization points. The design of the block algorithms for calculating the Cholesky factorization and the symmetric matrix inversion also fall into this category. The original matrix is reduced using the Cholesky factorization $A = LL^T$ (using the *DPOTRF* routine), where L is a lower triangular square matrix with positive diagonal elements. Following the factorization, there are two additional dense operations for the symmetric matrix inversion: (1) triangular matrix inversion $A = L^{-1}$ (using the *DTRTRI* routine) and (2) triangular matrix product $A = L^{-T}L^{-1}$ (using the *DLAUUM* routine). The block variant of the Cholesky factorization and the symmetric matrix inversion are therefore very limited in terms of parallelism and cannot fully benefit from now commonly available highly-parallel processing units.

#pragma omp task inout([NB][NB]A) priority(HIGHEST)
void DPOTRF(double *A);
#pragma omp task inout([NB][NB]A)
void DTRTRI(double *A);
#pragma omp task inout([NB][NB]A)
void DLAUUM(double *A);
#pragma omp task input([NB][NB]A) inout([NB][NB]C) priority(NT - k)
void DSYRK(double *A, double *C);
#pragma omp task input([NB][NB]A) inout([NB][NB]C) priority(NT - k)
void DTRSM(double *A, double *C);
#pragma omp task input([NB][NB]A) inout([NB][NB]C) priority(NT - k)
void DTRMM(double *A, double *C);
#pragma omp task input([NB][NB]A, [NB][NB]B) inout([NB][NB]C)
void DGEMM(double *A, double *B, double *C);
for k = 0 to NT−1 **do**
  // Stage 1: Cholesky factorization $A = LL^T$
  DPOTRF($A_{k,k}$);
  for m = k+1 to NT−1 **do**
    DTRSM($A_{k,k}$, $A^T_{m,k}$);
  end for
  for m = k+1 to NT−1 **do**
    DSYRK($A_{m,k}$, $A_{k,k}$);
    for n = k+1 to m−1 **do**
      DGEMM($A_{m,k}$, $A^T_{n,k}$, $A_{m,n}$);
    end for
  end for
  // Stage 2: Calculate $A = L^{-1}$
  for m = k+1 to NT−1 **do**
    DTRSM($A_{k,k}$, $A_{m,k}$);
    for n = 0 to k−1 **do**

```
      DGEMM(A_{m,k}, A_{k,n}, A_{m,n});
    end for
  end for
  for m = k+1 to k−1 do
    DTRSM(A_{k,k}, A_{k,m});
  end for
  DTRTRI(A_{k,k});
  //Stage 3: Compute A^{-1} = L^{-T} × L^{-1}
  for n = 0 to k−1 do
    DSYRK(A^T_{k,n}, A_{n,n});
    for m = n+1 to k−1 do
      DGEMM(A_{k,m}, A^T_{k,n}, A_{m,n});
    end for
  end for
  for n = 0 to k−1 do
    DTRMM(A^T_{k,k}, A_{k,n});
  end for
  DLAUUM(A_{k,k});
end for
```

Algorithm 1: Tile `OmpSs`-enabled distance-aware Cholesky factorization and symmetric matrix inversion algorithms.

3.2. Tile Algorithm Variant

The idea behind tile algorithms is to transform the original matrix data with column-major data layout into tile data layout. The parallelism becomes then exposed to the user, thanks to the task fine granularity. Indeed, the matrix tiles can be seen as the fundamental unit of computations of the numerical algorithms. The rigid panel-update sequence, previously described in Section 3.1, is now replaced by an out-of-order task execution flow, where computational tasks operating on tiles from different loop iterations can concurrently run. The algorithmic complexity of the block variant of both dense matrix computation algorithms does not change with the tile variant and is equal to $1/3n^3$ and n^3 for the Cholesky factorization and the symmetric matrix inversion algorithm, respectively.

The sequential program can now be represented as a DAG, where nodes stand for tasks and edges correspond to data dependencies. The strong and artifactual synchronization points, seen in block algorithm variant, are considerably alleviated using tile algorithms. For instance, the next panel factorization can proceed while the updates of the previous panel have not finished yet, as long as data dependencies on the corresponding tiles are satisfied. Now, it is up to a runtime system to schedule all generated tasks and to enforce their inter-task data dependencies. In particular, the `OmpSs` programming model has the advantage of being non-intrusive and relies on simple pragmas, similar to OpenMP programming model syntax. In fact, OpenMP 3.0 onwards, the tasking concept has been integrated to further help supporting mainstream applications. Algorithm 1 shows the parallel `OmpSs`-enabled version of the tile sequential Cholesky factorization (stage 1 only) and the symmetric matrix inversion (all stages) for an $NT \times NT$ tile symmetric positive-definite matrix A with a tile size NB. It is still the user's duty to describe the data

directions (`input`, `output` and `inout`) for each computational task, through compiler directives (i.e., pragmas).

4. Distance-Aware Work Stealing Scheduling Policy

This section provides implementation details of the new distance-aware work stealing scheduling policy and its incorporation into Nanos++ runtime.

4.1. Task queues

In order to maintain data locality, we have a task queue per NUMA node. This queue is a linked list sorted by task priority: the programmer is able to specify the priority of each task as a way to outline the critical path. Sorting is performed on insertion, thus the queue is sorted at any given time (ensuring the execution of tasks based on priorities as much as possible).

Priority-sorted linked lists are not a burden, given the granularity of the tasks of the experiments we carried. To prove that point, we developed a synthetic benchmark creating 10000 tasks lasting one microsecond each. There was no statistically measurable difference between a vector-based queue and a linked list based queue in total execution time or in runtime overhead when analyzing execution traces.

Threads belonging to a NUMA node are only able to retrieve tasks from their node's queue. This is accomplished by obtaining the number of NUMA nodes and the node corresponding to each worker thread from the Portable Hardware Locality (*hwloc*) library [9], and assigning each thread the number of the queue it should query.

4.2. Data distribution and locality detection

The runtime is able to track the location of the data and schedule tasks in the node with the highest number of bytes. To track the locality, it assumes a first touch policy and looks for initialization tasks. The criteria to detect such tasks is plain and simple: tasks with an output dependency (at least one) where it is the first time that data will be written. In our Cholesky implementation those tasks are the ones that initialize a block of the matrix.

Initialization tasks are scheduled in round robin across the available NUMA nodes, enabling us to use a similar data distribution. When a task of that type is executed, the data it initializes will be marked as located in the NUMA node of the running thread.

Otherwise, when a non initialization task is submitted, the number of bytes accessed by each node will be computed, based on the dependency information provided by the programmer, and the task will be scheduled in the node with the largest amount of data.

Note that once work stealing is introduced, the locality information becomes a hint that the runtime will always follow unless there is starvation in the local node.

Kurzak [23] described an implementation of the Cholesky factorization using static scheduling where threads work only on a one-dimensional cyclic distribution in order to keep locality.

4.3. Distance-aware work stealing

We choose to steal from neighbor nodes following a round-robin approach, with each node having an independent node index for stealing: steal from nodes in a cyclic way. In a NUMA architecture where some nodes are further than others, the distance-aware work stealing schedul-

ing policy ensures a thread will only steal tasks that are as close as possible. Thankfully, this information is provided by the Linux kernel. The distance between nodes provided by the operating system is not an accurate measure of the practical latency. The OS first tries to get this information by reading the System Locality Information (SLIT) table in the BIOS and if it fails, it may generate its own table, even if the vendor has explicitly provided a SLIT table. For instance, assume a system with 8 nodes, with the distance between node 1 and node 3 is 21 and between node 1 and node 4 is 42. In fact, these distances are default values from the BIOS SLIT table and they do not necessarily mean that the latency of accessing memory of node 3 from node 1 is half of accessing node 4. We can only be sure that node 4 is further from 1 than 3, and this is the reason why we only consider valid steal targets adjacent nodes. If in the previous example there was a node 5 with distance 22, it could still have a latency higher enough than node 3 that would hamper performance and eliminate any benefit of work stealing. We observed when comparing executions with and without work stealing, where the execution time of the tasks was noticeably worse with work stealing. In the execution without work stealing we discovered small gaps between a finalized task and the next one that were not present in the work stealing trace. The reason for such gaps is that the coming tasks are not yet available in the corresponding ready queue. As worker threads are constantly looking for work, with work stealing is enabled we observed that delaying for a small number of iterations (of the `OmpSs` runtime loop that looks for work for a given thread before giving up) would prevent those situations, thus increasing locality at the cost of a lower balance factor. This is controlled via a user defined variable, which can be set to zero to disable waiting. Please note that if that value is set to a large number it could effectively disable work stealing: worker threads will have to wait so long that they might find work available in their local queues before allowed to steal. To sum up, stealing does not come for free: while it reduces load imbalance, it obviously reduces data locality for stolen tasks. In other words, some tasks will take longer to complete and that is the reason why work stealing cannot be allowed to be performed indiscriminately.

These mechanisms we have described are able to improve load balance, while minimizing an increase in task execution time, as highlighted in the next Section.

5. Experimental Results

This Section highlights the impact of the distance-aware with work stealing scheduling policy in terms of performance (Gflop/s) and compares the new `OmpSs`-enabled Cholesky factorization and symmetric matrix inversion against existing commercial and open-source high performance dense linear algebra libraries.

5.1. Environment Description

Experiments were performed on three NUMA systems. System (A) is a quad processor AMD Magny-Cours 6172 with four sockets, twelve cores each, running at 2.1GHz with *two* NUMA nodes per socket (there are two dies in each physical package), with four HyperTransport links by socket resulting in a maximum distance between NUMA nodes of two hops. System (B) is an AMD Istanbul 8439 SE Processor with eight sockets, six cores each, running at 2.8GHz with *one* NUMA node and three HyperTransport links per socket. Both systems have 128GB of memory. It is noteworthy to mention that the NUMA nodes of system (B) are unequally distant and further away from each other compared to the NUMA nodes of system (A), as this system (B)

is composed of two 4P boards connected via two HyperTransport connectors, for a maximum distance of three hops between nodes. The third system (C) is an SGI Altix 1000 UltraViolet shared-memory machine based on Intel Nehalem EX processors featuring 128 sockets, eight cores each, running at 2.0GHz with *one* NUMA node per socket. This system has 4 TB of global shared memory in a single system image. We compared the Cholesky factorization and the symmetric matrix inversion using `OmpSs` against `PLASMA` v2.4.5, `Libflame` v5.0, `LAPACK` v3.4.2, and Intel compiler/`MKL` v10.1.015 on systems (A) and (B) and v11.1.038 on system (C). The `PLASMA` (providing `QUARK` as well as a static runtime system) and `Libflame` (using `SuperMatrix`) libraries are compiled with the sequential `MKL` BLAS, while `LAPACK` uses the multithreaded `MKL` BLAS. `PLASMA` and `Libflame` have been tuned for the underlying hardware by selecting an optimal block size. The command `numactl --interleave=all` has been executed to ensure a fair comparison against `LAPACK` and `MKL` implementations, which uses static data distribution. Moreover, in order to prevent false sharing, memory is aligned to the page size using *posix_memalign*. Otherwise, the distance-aware scheduling policy would be working with invalid locality information. Last but not least, all performance graphs in Gflop/s report the theoretical peak performance of the different system. The idea is to provide a good (but not realistic) upper-bound on all performance curves.

5.2. Distance-Aware Scheduling Policy Optimization Analysis

One of the main critical tasks of the Cholesky factorization and the symmetric matrix inversion is the matrix multiplication kernel *DGEMM*. Based on its number of tasks (called in the inner loop of Algorithm 1) and its execution time, we observed that it was the utmost task to focus on, when it comes to increasing the overall performance.

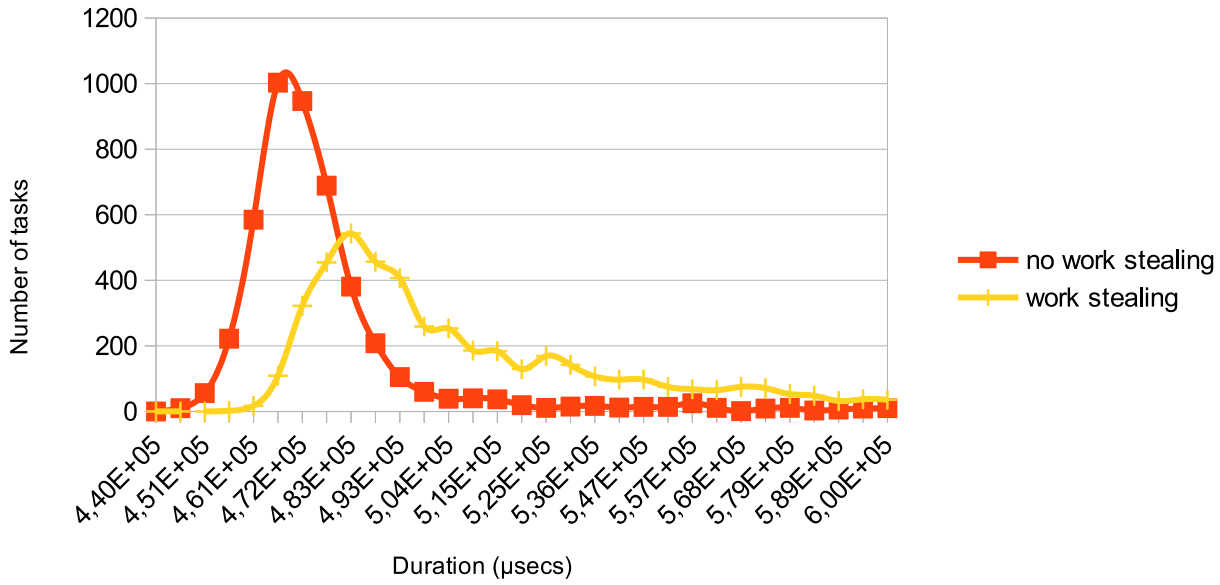


Figure 2. Task execution time histogram of *DGEMM*'s tasks on System (A) with the distance-aware scheduling policy

Figure 2 shows the task execution time histogram (horizontal axis) for *DGEMM*'s tasks, when distance-aware scheduling policy is turned on (work stealing mode) or not (no work stealing mode). A high value in the vertical axis represents a high concentration of *DGEMM*'s tasks for that particular time interval, and a low one means a small concentration. The other kernels involved in the dense matrix computation algorithms have been removed from the timing trace diagram for simplicity of presentation. Ideally, we should have an almost vertical line. This directly translates to very little variation (no scattered points across the horizontal and therefore, high concentration of tasks). In the case with no work stealing we have a situation very similar to the ideal one, where a very high data locality results in the system taking more or less the same time to execute tasks.

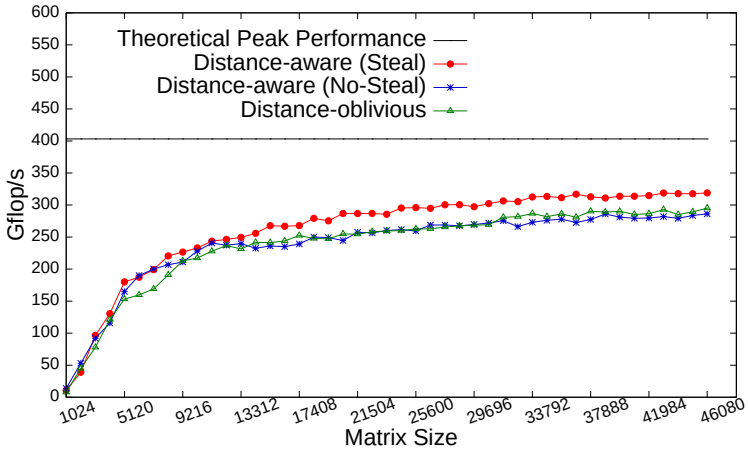
On the contrary, when work stealing takes effect, Figure 2 shows that most of *DGEMM*'s tasks have slightly shifted to the right of the histogram, because they take longer to terminate. There is also higher variability compared to the distance-aware without stealing policy. This indicates that threads have accessed data from remote NUMA nodes. Indeed, transferring data between farther NUMA nodes through the various hops creates a huge performance penalty because of the higher memory latency required when accessing distant nodes compared to local memory or adjacent NUMA nodes. Note that a perfect data distribution on large NUMA system is challenging due to the nature of the dense matrix computation algorithms and the complexity of the system studied here. Numerical algorithms using hierarchical data representation (e.g., fast multipole method [27]) or divide-and-conquer mechanism can better leverage such hardware architecture, hence the challenge.

These observations validate our initial concern that work stealing introduces a performance penalty and that we should only steal from adjacent nodes to mitigate its negative impact. The next Section demonstrates whether the `OmpSs` framework is still able to compensate the work stealing overhead by diminishing load imbalance and ultimately increasing the overall performance (Gflop/s).

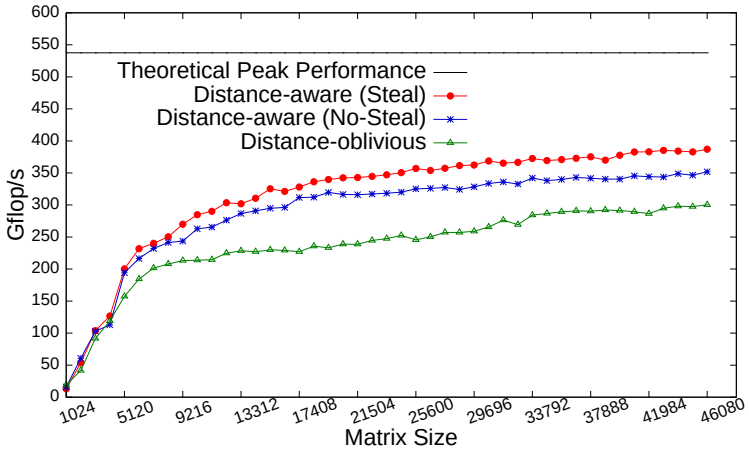
5.3. Performance Impact of Various Scheduling Policies

Figure 3 and 4 show the performance impact of various scheduling policies on the Cholesky factorization and the symmetric matrix inversion using systems (A) and (B), respectively, where the matrix size is increased in 4K increments along the horizontal axis until an asymptotic performance is reached. Due to the proximities of the NUMA nodes on system (A), Figure 3a does not show any difference whether we are running with or without the distance-aware policy. There is still a slight performance improvement, when work stealing is turned on (320 Gflop/s i.e., 80% of peak), thanks to a better data locality management. However, when NUMA nodes are farther, as in system (B), we can clearly distinguish the performance impact of scheduling policies. Figure 3b captures this discrepancy. The distance-aware with work stealing scheduling policy scores a 30% and 15% improvement in Gflop/s compared to the distance-aware without stealing and the distance-oblivious scheduling policies, respectively and reaches 390 Gflop/s i.e., 72% of peak. Although the symmetric matrix inversion presents more complex memory accesses due to two additional computational stages besides the Cholesky factorization, the distance-aware with work stealing scheduling policy is able to maintain as similar performance impact as the Cholesky factorization, on system (A) (Figure 4a) and system (B) (Figure 4b).

The performance impact is even further amplified with large number of NUMA nodes from system (C), as shown in Figures 5 and 6 for the Cholesky factorization and the symmetric matrix



a) System (A)



b) System (B)

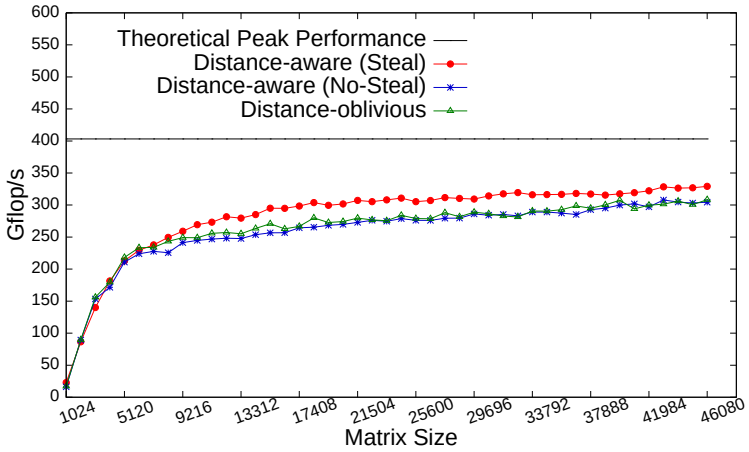
Figure 3. Performance impact of various scheduling policies on the Cholesky factorization

inversion, respectively. The distance-aware with work stealing scheduling policy become critical to sustain performance, as the number of NUMA nodes increases. On 32 sockets (256 threads), the distance-aware with work stealing scheduling policy achieves roughly fourfold and twofold performance improvement against the distance-aware without stealing and the distance-oblivious scheduling policies, respectively.

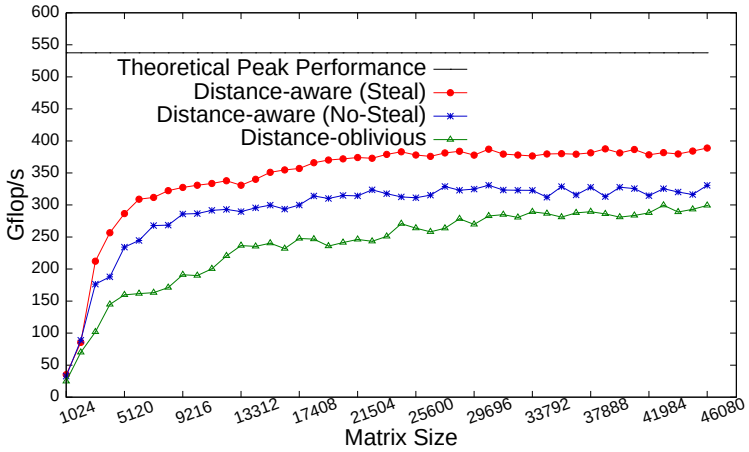
5.4. Performance Comparisons Against State-of-the-Art High Performance Dense Linear Algebra Libraries

Figures 7 and 8 show performance comparisons of the `OmpSs`-enabled Cholesky factorization and symmetric matrix inversion, respectively, with distance-aware with work stealing scheduling policy against existing high performance dense linear algebra implementations: `PLASMA` (providing two scheduler types: static and `QUARK`), `Libflame` (`SuperMatrix`), the commercial Intel `MKL` and the open-source `LAPACK` library. For the `PLASMA` library, only the scheduler type achieving the best performance is reported.

On system (A) (Figure 7a), the `LAPACK` implementation of the Cholesky factorization performs the worst due to the inefficient panel-update sequences, which generated lots of synchronizations, as previously mentioned in Section 3.1. The performance of the Intel `MKL` variant



a) System (A)



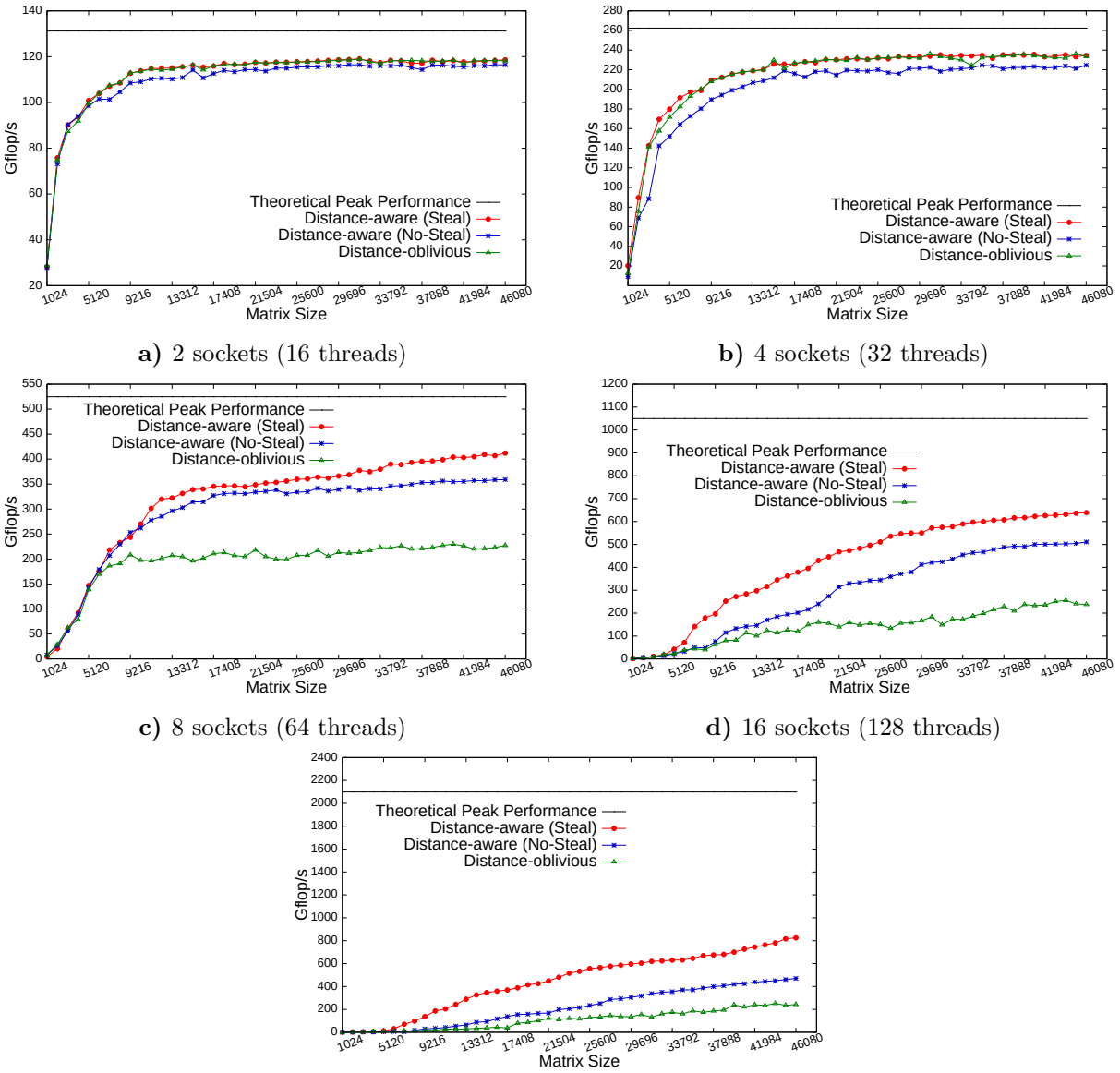
b) System (B)

Figure 4. Performance impact of various scheduling policies on the symmetric matrix inversion algorithm

increases substantially but still seems to suffer from memory accesses, as indicated by the curve's dips. The Cholesky implementations of `OmpSs`, `PLASMA` and `Libflame` have similar performance behavior on this system (A) with clustered NUMA nodes. On system (B) (Figure 7b), the OmpSs implementation scores up to 30% improvement against `PLASMA` with the dynamic scheduler `QUARK` (open-source package) for asymptotic sizes and more than twofold speedup against Intel `MKL` (commercial package).

Figures 8a and 8b show the performance of the symmetric matrix inversion on systems (A) and (B), respectively. The performance of the `LAPACK` symmetric matrix inversion is extremely low on both systems, again due to the lack of parallelism and data locality as well as artifactual barriers. The Cholesky factorization using the distance-aware with work stealing scheduling policy from `OmpSs` outperforms `Libflame` up to 50% across all matrix sizes and `PLASMA` by up to 50% and 25% for small and large matrix sizes, respectively.

The Cholesky factorization using static scheduler from `PLASMA` gives lower performance than `QUARK` and is not reported in Figures 7 and 8. The dynamic scheduler `QUARK` has thread binding, affinity mechanisms and is able to deal with load imbalance coming from thread starvation on these systems with rather small number of NUMA nodes. However, the overhead of the work stealing strategy as implemented in `QUARK` becomes significant in presence of large number



a) 2 sockets (16 threads)

b) 4 sockets (32 threads)

c) 8 sockets (64 threads)

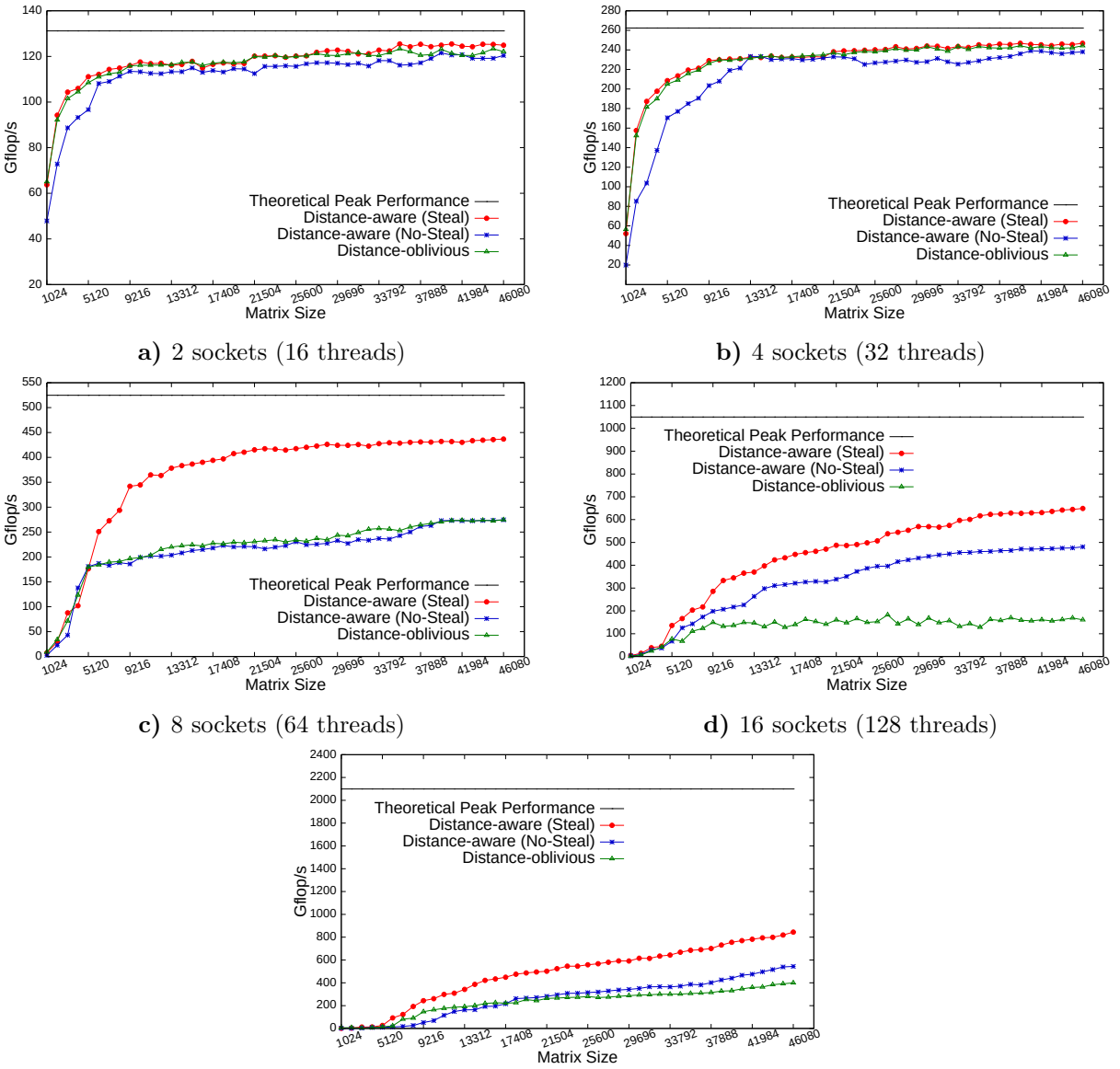
d) 16 sockets (128 threads)

e) 32 sockets (256 threads)

Figure 5. Performance impact of various scheduling policies on the Cholesky factorization on system (C)

of distant NUMA nodes and exceeds the overhead of load imbalance generated by the static scheduler due to idling of many worker threads. For the following subsequent graphs, we will therefore only refer to `PLASMA` static scheduler.

Figures 9 and 10 highlight the performance comparisons against existing Cholesky factorization and symmetric matrix inversion implementations on system (C). The `OmpSs`-enabled Cholesky factorization using the distance-aware with work stealing scheduling policy outperforms `PLASMA` (static scheduler) and `Libflame` implementations up to 65% on eight NUMA nodes and up to 200% on 32 NUMA nodes. `MKL` and `LAPACK` Cholesky (similarly for the symmetric matrix inversion) have not been run beyond eight sockets because performance would have been extremely low anyway. By the same token, the `OmpSs`-enabled symmetric matrix inversion using the distance-aware with work stealing scheduling policy is capable of sustaining the Cholesky factorization performance against the same other implementations i.e., twofold



a) 2 sockets (16 threads)

b) 4 sockets (32 threads)

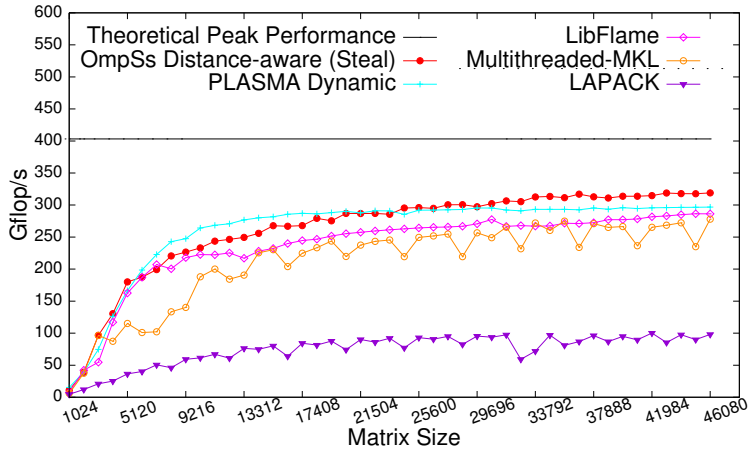
c) 8 sockets (64 threads)

d) 16 sockets (128 threads)

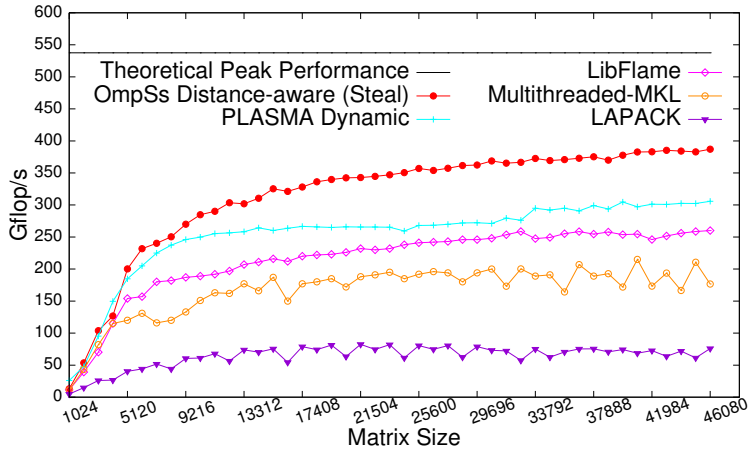
e) 32 sockets (256 threads)

Figure 6. Performance impact of various scheduling policies on symmetric matrix inversion using system (C)

performance improvement. Although the `OmpSs` dynamic runtime system associated with the new scheduling policy seems to decently exploit the underlying NUMA architecture, it is noteworthy to mention that the best performance achieved by the distance-aware with work stealing scheduling policy represents only 40% of the theoretical peak of system (C) on 32 sockets. It is well-known that a system's theoretical peak performance is a loose upper bound. We can still identify possible reasons for this low sustained peak performance. The shared-memory system is a shared resource and other users' applications running at the same time may engender memory bandwidth saturation causing further overheads. The overhead of the OS for ensuring cache coherency may also explain it, which is typical for such a the large memory system. There is also, of course, room for improvement at the runtime level and also further tuning of the tile size for large matrix sizes may further pay off.



a) System (A)



b) System (B)

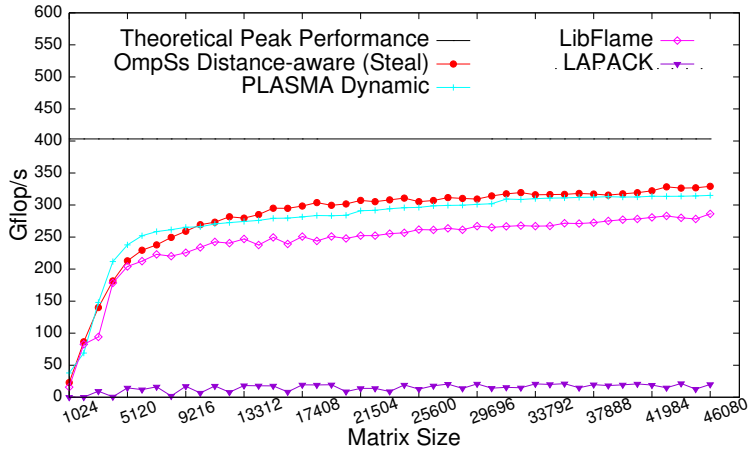
Figure 7. Performance comparisons against existing Cholesky factorization implementations

6. Performance Traces

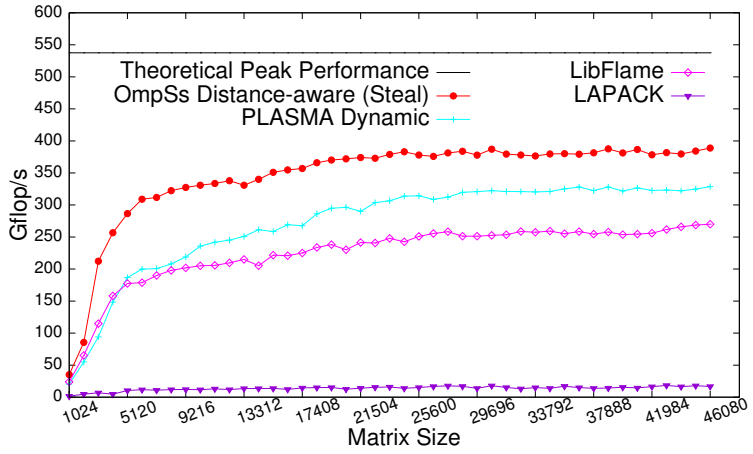
To further analyze the performance results shown in Section 5, the `OmpSs`-enabled Cholesky factorization and symmetric matrix inversion have been instrumented in order to generate traces using the `Extrae` tracing and the `Paraver` libraries.

Figures 11 and 12 show the execution traces of distance-oblivious and distance-aware (with and without work stealing) scheduling policies of the Cholesky factorization and the symmetric matrix inversion algorithm on system (B) using 48 cores, respectively, with a matrix size 30720×30720. The horizontal axis represents the timeline, the vertical axis the threads and the colors refer to different tasks. Figures 11a and 12a, representing traces for both dense matrix computation algorithms based on the distance-oblivious scheduling policy, show rather long but compact timelines.

Figures 11b and 12b represent traces for both dense matrix computation algorithms based on the distance-aware without stealing scheduling policy. These figures show shorter timelines but reveal severe idle time. This is mainly due to thread starvation, which engenders significant load imbalance between NUMA nodes. Targeting only data locality should not exclusively be the main concern for high performance applications. In fact, performance can be hindered by excessively hinting for data locality.



a) System (A)



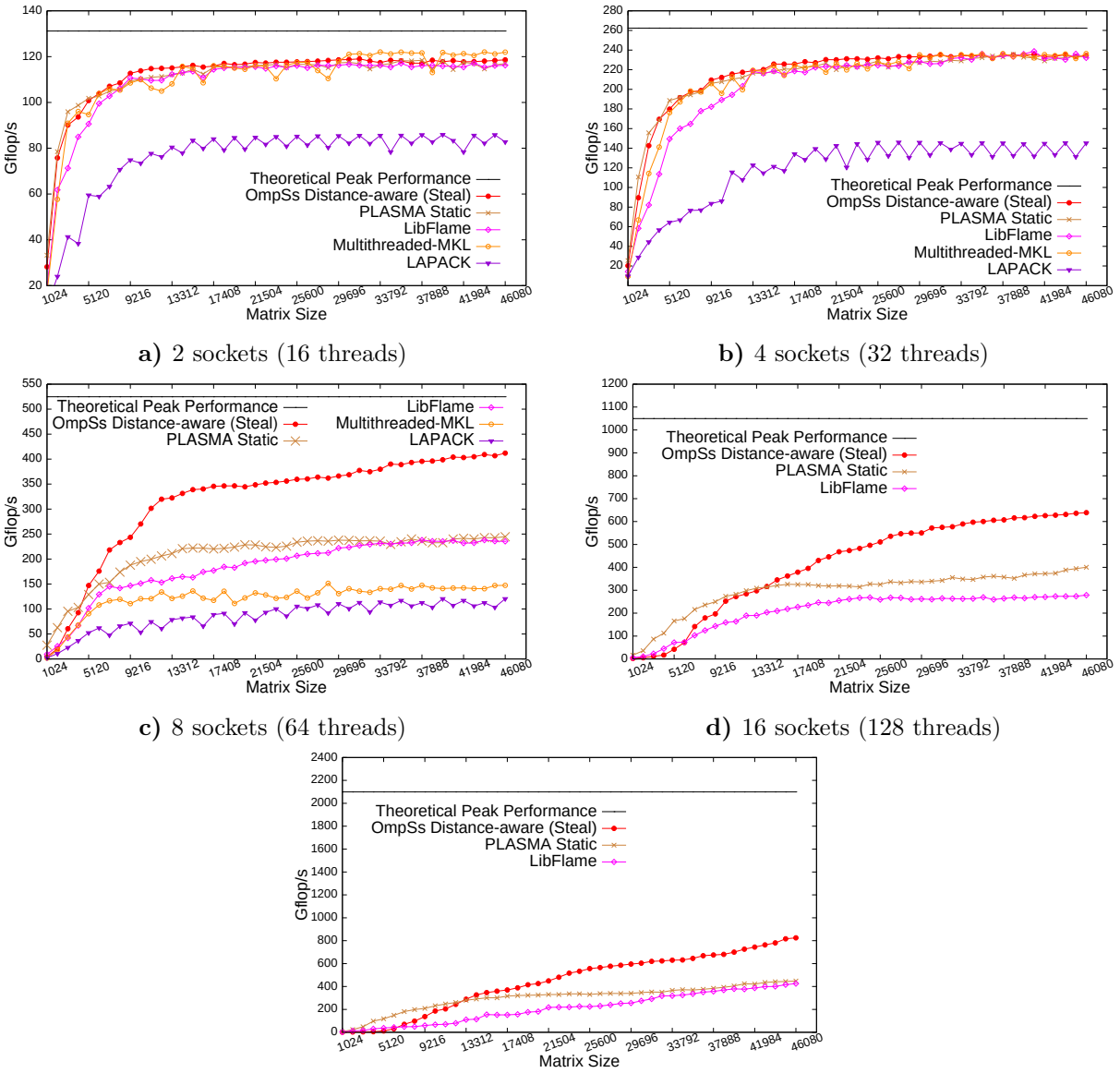
b) System (B)

Figure 8. Performance comparisons against existing symmetric matrix inversion implementations

Figures 11c and 12c highlight traces for both dense matrix computation algorithms based on the distance-aware with stealing scheduling policy. The timelines are now even shorter than distance-aware without stealing scheduling policy, despite a slightly longer elapsed time for *DGEMM* kernel, as detailed previously in Figure 2. The distance-aware with stealing scheduling policy is able to compensate the overhead of increased *DGEMM*'s elapsed time by removing most of stalls in the execution trace. One can now visualize the benefit of stealing from adjacent nodes, where tasks are continuously scheduled without gaps until the end of execution.

Conclusions and Future Work

We have demonstrated the role for the distance-aware scheduling policy, which increases data locality, to significantly improve the performance of two important dense linear algebra algorithms with time-varying work per unit memory at relatively high programmer productivity. We have also highlighted that work stealing in addition to distance-aware scheduling policy is paramount to attenuate load imbalance and to be ultimately effective on NUMA systems. Performance results on a large NUMA system outperform the best state-of-the-art existing implementations up to a twofold speedup for the Cholesky factorization as well as the symmetric



a) 2 sockets (16 threads)

b) 4 sockets (32 threads)

c) 8 sockets (64 threads)

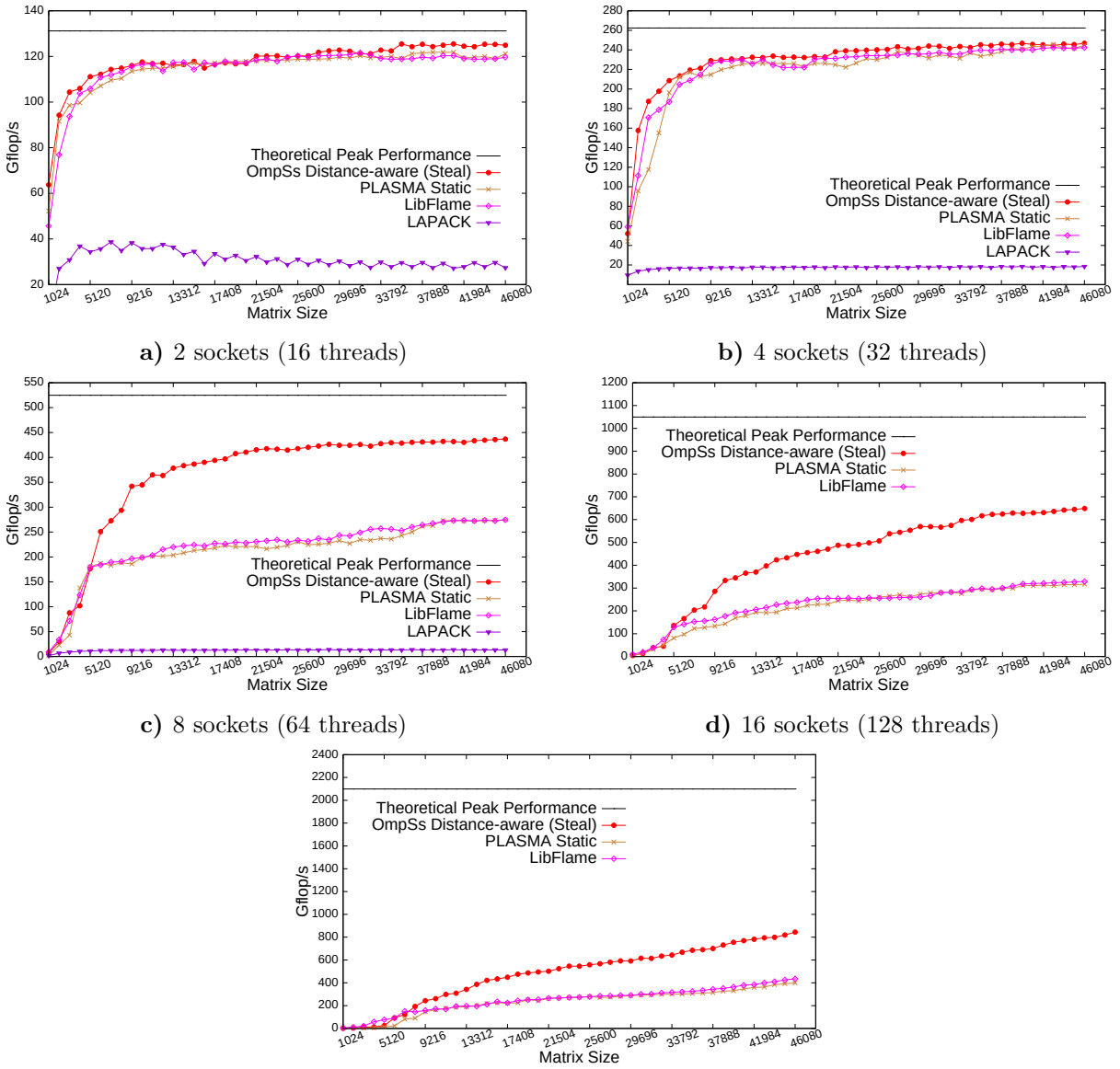
d) 16 sockets (128 threads)

e) 32 sockets (256 threads)

Figure 9. Performance comparisons against existing Cholesky factorization implementations on system (C)

matrix inversion algorithm. Developed in the context of the `OmpSs` framework, this new systematic scheduling policy approach allows the `OmpSs`-enabled code to maintain strong similarity to its original sequential version.

One of the challenges we faced is that the distance between NUMA nodes provided by the operating system does not proportionally translate into access times. A future refinement would be to compute an accurate distance matrix offline and provide that information to the runtime. The Portable Hardware Locality library has facilities to export the system topology information in an XML file, modify the topology information (in our case, the node distance), and use the modified XML file as the topology information. With this, we could relax the conditions for work stealing and allow stealing from non-adjacent nodes if the access time is feasible. Another way would be to modify the System Locality Information Table in the BIOS, but this is usually difficult in production supercomputers.



a) 2 sockets (16 threads)

b) 4 sockets (32 threads)

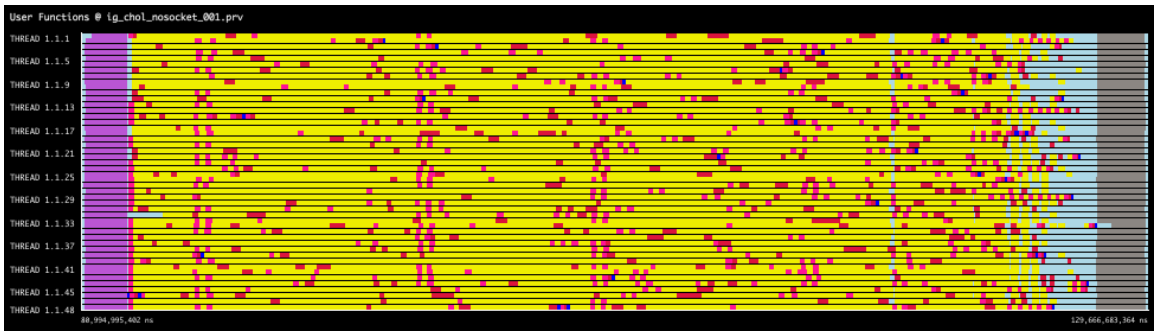
c) 8 sockets (64 threads)

d) 16 sockets (128 threads)

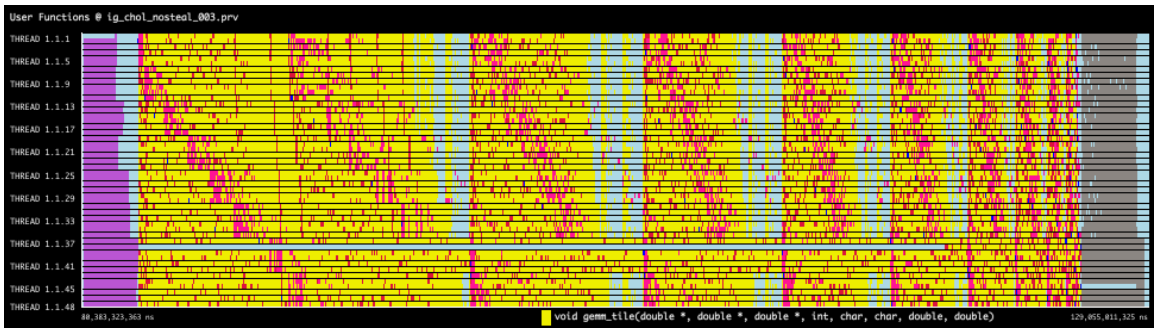
e) 32 sockets (256 threads)

Figure 10. Performance comparisons against existing symmetric matrix inversion implementations on system (C)

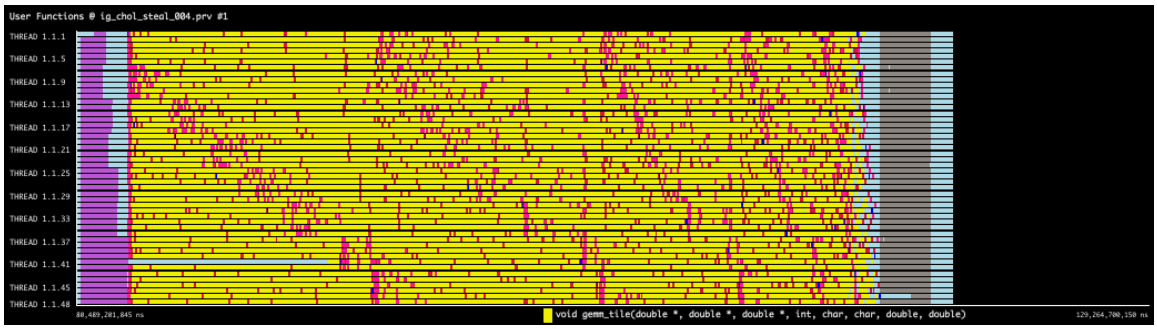
Other tasks in dense linear algebra are generally combinations of routines of similar time-varying load character and easier-to-handle regular workloads. The policy demonstrated herein should be applicable to such general tasks on multicore NUMA architectures, with benefits proportional to the fraction of dynamically varying workload. Many high profile computational tasks beyond dense linear algebra, such as sparse linear algebra, adaptive algorithms for partial differential equations, and complex physics/multiphysics simulations should also be amenable to performance improvements through the same philosophy, if not identical heuristics. Beyond multicore NUMA, multi-GPU systems and hybrid architectures introduce trade-offs between load balance and data locality that will require locality-aware work stealing. We believe that the mechanisms of `OmpSs` and similar programming models possess great potential in practically extending the performance portability of scientific simulation.



a) Distance-oblivious



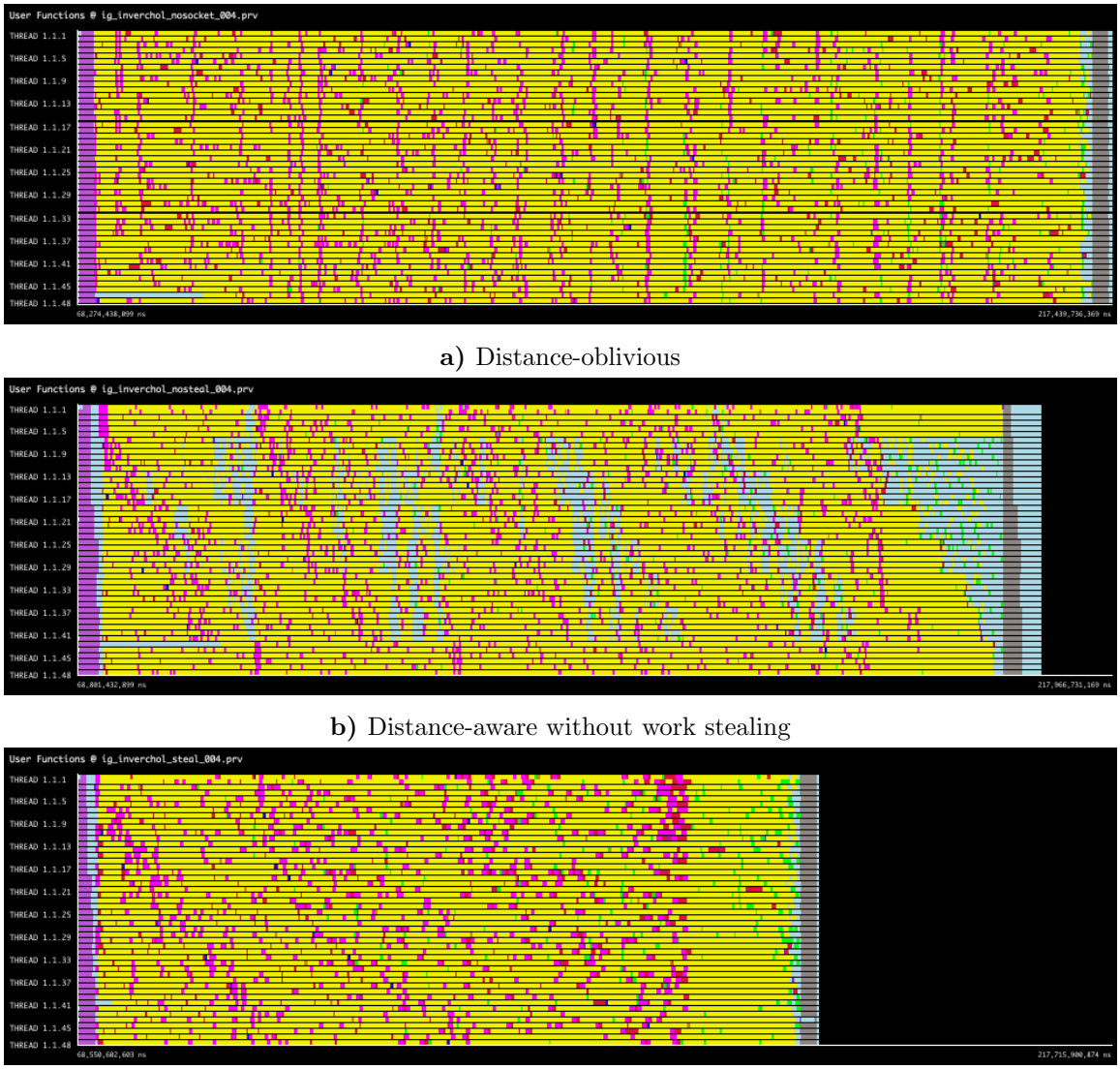
b) Distance-aware without work stealing



c) Distance-aware with work stealing

Figure 11. Traces of the Cholesky factorization using various scheduling policies on system (B)

The authors would like to thank the National Institute for Computational Sciences for granting us access on the Nautilus system. The KAUST authors acknowledge support of the Extreme Computing Research Center. The BSC-affiliated authors thankfully acknowledges the support of the European Commission through the HiPEAC-3 Network of Excellence (FP7-ICT 287759), Intel-BSC Exascale Lab and IBM/BSC Exascale Initiative collaboration, Spanish Ministry of Education (FPU), Computación de Altas Prestaciones VI (TIN2012-34557), Generalitat de Catalunya (2014-SGR-1051) and the grant SEV-2011-00067 of the Severo Ochoa Program.



a) Distance-oblivious

b) Distance-aware without work stealing

c) Distance-aware with work stealing

Figure 12. Traces of the symmetric matrix inversion algorithm using various scheduling policies on system (B)

References

1. *PLASMA Users' Guide, Parallel Linear Algebra Software for Multicore Architectures, Version 2.4.5*, November 2012.

2. A. Charara, H. Ltaief, D. Gratadour, D. Keyes, A. Sevin, A. Abdelfattah, E. Gendron, C. Morel and F. Vidal. Pipelining Computational Stages of the Tomographic Reconstructor for Multi-object Adaptive Optics on a Multi-GPU System. *SC '14: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 262–273, 2014.

3. E. Agullo, H. Bouwmeester, J. Dongarra, J. Kurzak, J. Langou, and L. Rosenberg. Towards an Efficient Tile Matrix Inversion of Symmetric Positive Definite Matrices on Multicore Architectures. *High Performance Computing for Computational Science VECPAR 2010*, 6449:129–138, 2011.

4. E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA Projects. *J. Phys.: Conf. Ser.*, 180(1), 2009.

5. E. Anderson, Z. Bai, C. Bischof, Suzan L. Blackford, James W. Demmel, Jack J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and Danny C. Sorensen. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, 3rd edition, 1999.

6. E. Ayguade, R. M. Badia, D. Cabrera, A. Duran, M. Gonzalez, F. Igual, D. Jimenez, J. Labarta, X. Martorell, R. Mayo, J. M. Perez, and E. S. Quintana-Ortí. A Proposal to Extend the OpenMP Tasking Model for Heterogeneous Architectures. In *Proceedings of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism*, IWOMP '09, pages 154–167, Berlin, Heidelberg, 2009. Springer-Verlag. DOI: 10.1007/978-3-642-02303-3_13.

7. L. Suzan Blackford, J. Choi, A. Cleary, E. F. D'Azevedo, J. W. Demmel, I. S. Dhillon, J. J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. W. Walker, and R. Clint Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, 1997.

8. F. Broquedis, O. Aumage, B. Goglin, S. Thibault, P. Wacrenier, and R. Namyst. Structuring the Execution of OpenMP Applications for Multicore Architectures. In *IEEE International Parallel and Distributed Processing Symposium*, pages 1–10, 2010. DOI: 10.1109/IPDPS.2010.5470442.

9. F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In IEEE, editor, *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, Pisa, Italie, February 2010. DOI: 10.1109/PDP.2010.67.

10. A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures. *Parallel Computing*, 35(1):38–53, 2009.

11. E. Chan, E. S. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn. Supermatrix Out-of-order Scheduling of Matrix Operations for SMP and Multi-core Architectures. In *SPAA '07: Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 116–125, New York, NY, USA, 2007. ACM. DOI: 10.1145/1248377.1248397.

12. Q. Chen, M. Guo, and H. Guan. LAWS: Locality-Aware Work-Stealing for Multi-Socket Multi-core Architectures. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, pages 3–12, New York, NY, USA, 2014. ACM. DOI: 10.1145/2597652.2597665.

13. J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig, F. Cappello, B. Chapman, C. Xuebin, A. Choudhary, S. Dosanjh, T. Dunning, S. Fiore, A. Geist, B. Gropp, R. Harrison, M. Hereld, M. Heroux, A. Hoisie, K. Hotta, J. Zhong, Y. Ishikawa, F. Johnson, S. Kale, R. Kenway, D. Keyes, B. Kramer, J. Labarta, A. Lichnewsky, T. Lippert, B. Lucas, B. Maccabe, S. Matsuoka,

P. Messina, P. Michielse, B. Mohr, M. Mueller, W. E. Nagel, H. Nakashima, M. E. Papka, D. Reed, M. Sato, E. Seidel, J. Shalf, D. Skinner, M. Snir, T. Sterling, R. Stevens, F. Streitz, B. Sugar, S. Sumimoto, W. Tang, J. Taylor, R. Thakur, A. Trefethen, M. Valero, A. Van Der Steen, J. Vetter, P. Williams, R. Wisniewski, and K. Yelick. The International Exascale Software Project Roadmap. *International Journal of High Performance Computer Applications*, 25(1):3–60, February 2011. DOI: 10.1177/1094342010391989.

14. A. Drebes, K. Heydemann, N. Drach, A. Pop, and A. Cohen. Topology-Aware and Dependence-Aware Scheduling and Memory Allocation for Task-Parallel Languages. *ACM Transactions on Architecture and Code Optimization*, 11(3):1–25, August 2014. DOI: 10.1145/2641764.

15. A. Duran, R. Ferrer, E. Ayguadé, R. M. Badia, and J. Labarta. A Proposal to Extend the OpenMP Tasking Model with Dependent Tasks. *International Journal of Parallel Programming*, 37(3):292–305, 2009. DOI: 10.1007/s10766-009-0101-1.

16. K. Faxén. Wool – A Work Stealing Library. *ACM SIGARCH Computer Architecture News*, 36(5):93–100, June 2009. DOI: 10.1007/10.1145/1556444.1556457.

17. G. H. Golub and C. F. Van Loan. *Matrix Computations*. John Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, Baltimore, Maryland, third edition, 1996.

18. N. Higham. *Accuracy and Stability of Numerical Algorithms, Second Edition*. SIAM, 2002.

19. E. Jeannot. Symbolic Mapping and Allocation for the Cholesky Factorization on NUMA Machines: Results and Optimizations. *IJHPCA*, 27(3):283–290, 2013. DOI: 10.1177/0734904113492410.

20. L. Karlsson and B. Kågström. 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.

21. D. E. Keyes. Exaflop/s: The Why and The How. *Comptes Rendus Mcanique*, 339(23):70 – 77, 2011. Le Calcul Intensif.

22. J. Kurzak, A. Buttari, and J. J. Dongarra. Solving Systems of Linear Equations on the CELL Processor Using Cholesky Factorization. *IEEE Transactions on Parallel and Distributed Systems*, 19(9):1–11, September 2008.

23. J. Kurzak, H. Ltaief, J. J. Dongarra, and R. M. Badia. Scheduling Dense Linear Algebra Operations on Multicore Processors. *Concurrency and Computation: Practice and Experience*, 21(1):15–44, 2009.

24. H. Ltaief, J. Kurzak, and J. Dongarra. Parallel Band Two-Sided Matrix Bidiagonalization for Multicore Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 21(4), April 2010.

25. A. Muddukrishna, P. A. Jonsson, V. Vlassov, and M. Brorsson. Locality-Aware Task Scheduling and Data Distribution on NUMA Systems. In *OpenMP in the Era of Low Power Devices and Accelerators*, pages 156–170. Springer, 2013.

26. G. Quintana-Ortí, E. S. Quintana-Ortí, E. Chan, R. A. van de Geijn, and F. G. Van Zee. Scheduling of QR Factorization Algorithms on SMP and Multi-Core Architectures. In *PDP*, pages 301–310. IEEE Computer Society, 2008. DOI: 10.1109/PDP.2008.37.

27. R. Yokota, G. Turkiyyah and D. Keyes. Communication Complexity of the Fast Multipole Method and its Algebraic Variants. *International Journal of Supercomputing frontiers and innovations*, 1(1), 2014.

28. Y. Yan, J. Zhao, Y. Guo, and V. Sarkar. Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement. In *Proceedings of the 22nd International Conference on Languages and Compilers for Parallel Computing*, LCPC'09, pages 172–187, Berlin, Heidelberg, 2010. Springer-Verlag. DOI: 10.1007/978-3-642-13374-9_12.

29. A. YarKhan, J. Kurzak, and J. Dongarra. QUARK Users' Guide: QUeueing And Runtime for Kernels. *University of Tennessee Innovative Computing Laboratory Technical Report ICL-UT-11-02*, 2011.

30. F. G. Van Zee, E. Chan, R. A. van de Geijn, E. S. Quintana-Orti, and G. Quintana-Orti. The `libflame` Library for Dense Matrix Computations. *Computing in Science and Engineering*, 11(6):56–63, November/December 2009. DOI: 10.1109/MCSE.2009.207.

Received March 16, 2015.

DOI: 10.14529/jsfi150104

Neo-hetergeneous Programming and Parallelized Optimization of a Human Genome Re-sequencing Analysis Software Pipeline on TH-2 Supercomputer

Xiangke Liao[1], *Shaoliang Peng*[1], *Yutong Lu*[1], *Chengkun Wu*[1], *Yingbo Cui*[1], *Heng Wang*[1], *Jiajun Wen*[1]

The growing velocity of biological big data is the way beyond Moore's Law of compute power growth. The amount of genomic data has been explosively accumulating, which calls for an enormous amount of computing power, while current computation methods cannot scale out with the data explosion. In this paper, we try to utilize huge computing resources to solve the big dataproblems of genome processing on TH-2 supercomputer. TH-2 supercomputer adopts neo-heterogeneous architecture and owns 16,000 compute nodes: 32000 Intel Xeon CPUs + 48000 Xeon Phi MICs. The heterogeneity, scalability, and parallel efficiency pose great challenges for the deployment of the genome analysis software pipeline on TH-2. Runtime profiling shows that SOAP3-dp and SOAPsnp are the most time-consuming parts (up to 70% of total runtime) in the whole pipeline, which need parallelized optimization deeply and large-scale deployment. To address this issue, we first design a series of new parallel algorithms for SOAP3-dp and SOAPsnp, respectively, to eliminatethe spatial-temporal redundancy. Then we propose a CPU/MIC collaborated parallel computing method in one node to fully fill the CPU/MIC time slots. We also propose a series of scalable parallel algorithms and large scale programming methods to reduce the amount of communications between different nodes. Moreover, we deploy and evaluate our works on the TH-2 supercomputer in different scales. At the most large scale, the whole process takes 8.37 hours using 8192 nodes to finish the analysis of a 300TB dataset of whole genome sequences from 2,000 human beings, which can take as long as 8 months on a commodity server. The speedup is about 700x.

Keywords: biological big data; parallelized optimization; TH-2; sequence alignment; SNP detection; whole genome re-sequencing.

Introduction

Whole genome re-sequencing refers to the procedure of genome sequencing of different individuals of species with a reference genome and the execution of divergence analysis on individual or colony. Using whole genome re-sequencing, researchers can obtain plentiful information of variations like single nucleotide polymorphisms (SNP), copy number variation (CNV), insertion/deletion (INDEL) and structure variation (SV). Its application areas include clinical pharmaceuticals (cancer, Ebola, AIDS, leukemia etc.), population genetics, correlation analysis, evolution analysis and so on. With the development of personalized medicine and sequencing technologies, the cost of sequencing has been decreasing (currently the sequencing of one human genome costs less than 1000$). Consequently, the scale of sequences is growing rapidly. However, if the accumulated data could not be analyzed efficiently, a large amount of useful information would be left unused or discarded.

BGI, one of the top three gene sequencing institutions in the world, produces 6TBs sequences every day. Their current analysis pipeline and solutions need two months to accomplish the comparison and mutation detection of those data with one single server. The DNA sequences of 2000 people constitute a dataset as large as 300TB. The ongoing million people genome project

[1]National University of Defense Technology, Changsha, China

needs to analyze 500PB of DNA sequences. Given the current analyzing efficiency, it would be a mission impossible.

Our analysis shows that sequencing alignment and mutation detection tools occupy 70% of total time. These two applications and the output data are essential to the whole pipeline; therefore, it would be beneficial for the whole pipeline if we can explore the parallelism and improve the computing scale and parallel efficiency of those two components using supercomputer.

The TH-2 supercomputer, developed by the National University of Defense Technology(NUDT), is the outstanding achievement of the National 863 Program. Now the TH-2 is installed in the Guangzhou Supercomputing Center. TH-2 is co-designed with international collaboration and adopts a new neo-heterogeneous architecture. The compute array has 16,000 compute nodes. Each node contains and 2 multi-core Xeon CPUs and 3 many-core Xeon Phi MICs.It is an innovative architecture which significantly expands the application domains of the TH-2. It is a big challenge to solve the problems of genome big data processing and high throughput biological applications on TH-2.

In this paper, we try to utilize huge computing resources to solve the big dataproblems of genome processing on TH-2 supercomputer. We aim to address the above biological big data problem by carrying out parallelization and scalability on the aforementioned key components and implementing the optimized pipeline on the TH-2. To address these issues of genome big data alignment and SNP detection, we propose a set of algorithms and parallel strategies of intra-node and inter-node.

Intra-node: We partition genome analysis tasks and data into each node of TH-2 evenly and fully. A series of methods are proposed in order to fully use 3 MICs and 2 CPUs on one node, such as three-channel IO latency hiding, elimination of computation redundancy, spatial-temporal compression, vectorization of 512 Bit wide SIMD command, CPU/MIC collaborated parallel computing method, and so on. Our programming and parallelized optimization also aim at one computation node on TH-2 in order to make our algorithms scalable and extendible when the data scale up.

Inter-node: A series ofscalable parallel algorithms and large scale programming methods are also proposed to reduce the amount of communications between different nodes according to the genome sequence data suffix and characteristics, such as fast windows Iteration and scalable multilevel parallelism. Moreover, experiments and evaluations in different scale from 512 to 8192 nodes are analyzed and shown on TH-2. And some inspiring results are generated: we can use 8192 nodes to finish the analysis a 300TB dataset of whole genome sequences from 2,000 human beings within 8.37 hours, which can take as long as 8 months on a commodity server.

1. Background

1.1. Pipeline of Human Genome Re-sequencing Analysis

The pipeline of human genome re-sequencing is composed of several components. The first step is to perform quality control of original reads in order to reduce noise. Sequence alignment and single nucleotide polymorphisms (SNP) detection are then conducted consecutively. Sequence alignment is the procedure of comparing screened sequences to the reference genome, confirming its occurring frequency and appearing location in the reference genome. The output will be stored in either SAM or BAM format. SNP calling is the procedure of detecting bases deletion, insertion, transition and perversion in DNA sequences. It takes alignment results, the

reference sequence, and curated SNP databases as input and detects SNPs. These two steps are essential for consequent analysis including correlation analysis, functional annotation, and population genomics analysis [6]. We performed runtime profiling and found that short-read alignment and variation detection are the most time-consuming parts (up to 70% of total runtime) in the whole pipeline. So we are devoted to accelerating the most worldwide used software for each part, namely SOAP3-dp and SOAPsnp correspondingly. We also ported other parts of the whole pipeline to TH-2 in order to fully utilize the computational resources of different nodes on TH-2.

1.2. Neo-heterogeneous Architecture of TH-2 supercomputer

Capable of a peak performance of 54.9PFlops, the TH-2 achieves a sustained performance of 33.9PFlops on Linpack with a performance-per-watt of 1.9GFlops/W. The compute array has 16,000 compute nodes: 32000 Intel Xeon CPUs + 48000 Xeon Phi MICs. Each node contains and two multi-core Xeon CPUs and three many-core Xeon Phi MICs. It is an innovative architecture, which significantly expands the application domains of the TH-2. We named it: Neo-Heterogeneous Architecture Compute Array, which significantly improves the compatibility, the flexibility and the usability of the applications. Here the phrase neo-heterogeneous refers to the fact that compared with CPU+GPU architecture, CPU+MIC is less heterogeneous as MIC is x-86 compatible.

Intel Many Integrated Core (MIC) architecture is also known as Intel Xeon Phi coprocessor. A MIC card is equipped with 50+ cores clocked at 1 GHz and 6 GB or more on-card memory; each core supports 4 hardware threads and owns a 512-bit wide SIMD vector process unit (VPU). Each MIC card can achieve a double precision arithmetic performance of 1TFlops per second, which makes it a competitive type of accelerator. MIC architecture is x86-compatible, which alleviates the efforts required to port applications to Xeon Phi compared to its counterpart GPUs. Some simple applications can even run directly on Xeon Phi simply after re-compilation. There are two major modes to employ Xeon Phi in an application:

1) *native* mode, where Xeon Phi has a whole copy of the application and runs the applications, just like a normal compute node.

2) *offload* mode, where the application runs as a master thread on CPU and offloads some selected part to Xeon Phi, which treats Xeon Phi as a coprocessor.

2. Related Work

Short Oligonucleotide Analysis Package (SOAP [3]) is an integrated software package includes a lot of software such as SOAPsnp, SOAP3-dp, SOAPfusion and so on. SOAP is dedicated to next generation sequencing data analysis.

SOAP3-dp is known as a fast, accurate, sensitive short read aligner [2]. It is designed by BGI and belongs to SOAP (Short Oligonucleotide Analysis Package) series analysis tools. It comes from SOAP3, a GPU-based software for aligning sort reads to reference sequence[2]. So as to adapt to the increasing throughput of next-generation sequencing, SOAP3-dp was proposed to increase the performance of SOAP3 [5] in terms of speed and sensitivity. For instance, it takes less than 10 seconds for SOAP3-dp to align length-100 single-end reads with the human genome per million reads, while SOAP3, tens of seconds. The main improvement is that SOAP3-dp can tackle lots of reads in parallel by means of exploiting compressed indexing and memory-

optimizing dynamic programming on GPU. As a result, the gapped alignments can be examined extensively and compared to other tools; SOAP3-dp has the advantages of high speed, accuracy and sensitivity [3]. To our best of knowledge, there is no MIC version for SOAP3-dp so it cannot utilize the compute power provided by MIC cards equipped on TH-2.

SOAPsnp is a popular SNP detection tool developed by BGI as a member of its SOAP (Short Oligonucleotide Analysis Package) series analysis tools. The software adopts the Bayesian model to call consensus genotype by carefully considering the data quality, alignment and recurring experimental errors. All these information is integrated into a single quality score for each base to measure the calling accuracy. SOAPsnp usually costs several days or even more to analyze a human genome, which accounts for more than 30% time of normal genome analysis pipeline. The low efficiency calls for a performance boost by advanced computing technologies [6].

3. Programming and Optimization on TH-2 supercomputer

3.1. MICA: A MIC version of SOAP3-dp

To utilize the compute power provided by TH-2, we developed MICA, a tool that can make full use of both CPU and MIC to execute alignment of short reads from high throughput sequencing. It took less than 1 hour to complete comparison of 17T short sequence data using 932 compute nodes when MICA was tested on TH-2, which would take about 3 months on a commodity server. The MICA is designed and developed by National University of Defense Technology (NUDT) and BGI, and it is open online to all biologists [7].

3.1.1. Three-channel IO Latency Hiding

MICA is optimized for the configuration of both software and hardware on Tian-he2. Considering that each compute node has three MICs, we allocate three input buffers and three output buffers to hide the delay produced by data transfer, as depicted in fig. 1.

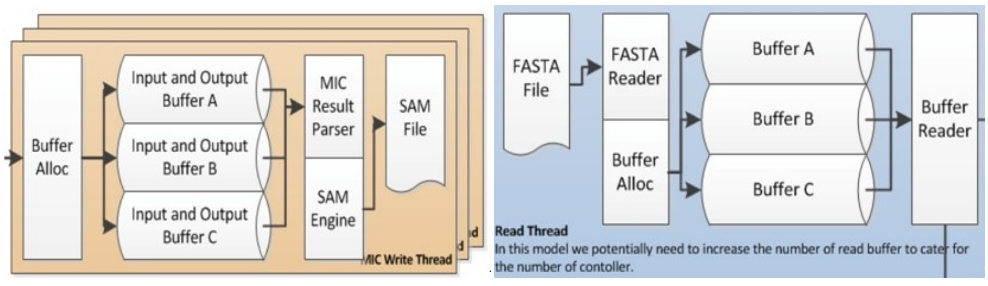


Figure 1. Three Channel IO Design

3.1.2. Vectorization of 512 Bit Wide SIMD Command

Each MIC core owns one 512-bit wide VPU, so we implement the core functions 512 Bit Wide SIMD Command to make full use of vector units of MIC. Core code can be seen as below (fig. 2).

```
mA = _mm512_add_epi32(
    rA,
    _mm512_mask_mov_epi32(dpw->rMismatch, _mm512_cmpeq_epi32_mask(_mm512_srli_epi32(rcodeB,4),rcodeC), dpw->rMa
);
mB = _mm512_add_epi32(
    rB,
    _mm512_mask_mov_epi32(dpw->rGapOpen,_mm512_cmpeq_epi32_mask(rtraceB, dpw->rMaskTraceB),dpw->rGapExtend)
);
mC = _mm512_add_epi32(
    rC,
    _mm512_mask_mov_epi32(dpw->rGapOpen,_mm512_cmpeq_epi32_mask(rtraceC, dpw->rMaskTraceC),dpw->rGapExtend)
);
// r1 stores the max score among A, B, C
r1 = _mm512_mask_mov_epi32(mA,_mm512_cmpge_epi32_mask(mB,mA),mB);
r1 = _mm512_mask_mov_epi32(r1,_mm512_cmpge_epi32_mask(mC,r1),mC);
r2 = _mm512_setzero_epi32();
r2 = _mm512_mask_add_epi32(r2, _mm512_cmpeq_epi32_mask(r1,mA), r2, _mm512_set1_epi32(0x00000001));
r2 = _mm512_mask_add_epi32(r2, _mm512_cmpeq_epi32_mask(r1,mB), r2, dpw->rMaskTraceB);
r2 = _mm512_mask_add_epi32(r2, _mm512_cmpeq_epi32_mask(r1,mC), r2, dpw->rMaskTraceC);
r1 = _mm512_and_epi32(r1,_mm512_set1_epi32(0xffff0000));
r1 = _mm512_add_epi32(r1,rcodeB);
r1 = _mm512_add_epi32(r1,rcodeC);
r1 = _mm512_add_epi32(r1,r2);
_mm512_store_epi32(dpw->matrix+coord(i,j),r1);
```

Figure 2. 512 Bit Wide Vector SIMD Code

3.1.3. Parallelized Construction and Vectorization of Smith-Waterman Dynamic Matrix

DNA sequence imprecise alignment adopts Smith-Waterman Dynamic Programming algorithm. Procedure of filling Smith-Waterman Dynamic Matrix is one of the hotspots of the whole software. The Smith-Waterman Dynamic Score Matrix is filled as below:

$$H(i,0) = 0,\ 0 \le i \le m$$
$$H(0,j) = 0,\ 0 \le j \le n$$
$$H(i,j) = \max \left\{ \begin{array}{ll} 0 & \\ H(i-1,j-1) + s(a_i,b_j) & \text{Match/Mismatch} \\ \max_{k \ge 1}\{H(i-k,j) + W_k\} & \text{Deletion} \\ \max_{l \ge 1}\{H(i,j-l) + W_l\} & \text{Insertion} \end{array} \right\},\ 1 \le i \le m, 1 \le j \le n$$

Here we calculate the values in the matrix diagonal wise, and use 512 bit wide vectorization to rewrite the original function. The new order of our filling matrix algorithm is from left to right and from top to bottom, and the Smith-Waterman Matrix can be parallelized built along diagonal, as is shown in fig. 3:

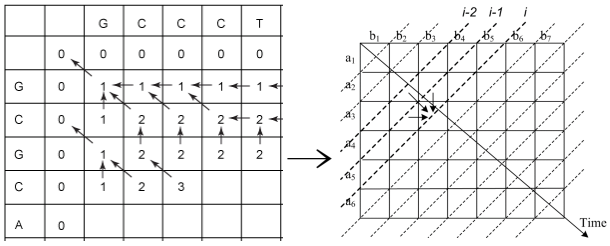


Figure 3. Parallel Constructing Smith-Waterman Matrixalong Diagonal

3.1.4. Using Inline Function to Decrease Overhead of Function Call

When it comes to the processing of large sequencing dataset, the overhead of function call cannot be ignored. For the processing of 17T sequence data, there will be 1.2P times calling of the function. So we set the core function to be inline function or macro to increase the efficiency of function execution.

3.1.5. Perfecting of Index Data

The BWT (BurrowsWheeler Transform) index array is a core data structure in SOAPsnp. It will be frequently visited by a number of threads. To avoid memory overhead, increase efficiency of CPU, we prefetching the data for next loop.

3.2. Optimization of SOAPsnp

Hotspots of SOAPsnp are two modules: likelihood and recycle, as shown in the tab. 1.

Table 1. Time breakdown of SOAPsnp

Module	*likelihood*	*recycle*	**Other modules**	**Total**
Time/s	1478.36	752.98	40.59	2552.18
Percentage	57.93%	29.50%	12.57%	1

We analyzed the characteristics of the original implementation of SOAPsnp and developed high performance version - mSOAPsnp. We devised a number of optimization methods to achieve an optimized performance assisted by MIC.

3.2.1. Compression of 4-D Sparse Matrix

The main function of the likelihood module is to traverse the base_info matrix, which is a 4-D sparse matrix with 0.08% non-zero elements. The matrix saves the information of short sequence, and occupies 1MB space. Frequent access to the base_info matrix leads to low efficiency due to unnecessary memory accesses. So we compressed the 4-D sparse matrix, only saving non-zero elements of the matrix; we also modified the original algorithm from 4 levels of loops to only one loop, which reduces the memory overhead down to 5%. By this improvement, the program gets a speedup of 2.3x.

3.2.2. Fast Windows Iteration

The recycle module handles sequences that are located at overlapped area of two windows in the phase of windows iteration, copying SNP information (100 SNPs) from one window to the next one. Therefore, the recycle module has a large amount of duplicated variables and memory copying operations. We found that copying the base_info matrix introduces the most overhead. The size of base_info matrix is 1MB, every SNP has one base_info matrix, so in each iteration of shifting the window, we need to copy 100 base_info matrixes, which is 100MB in total. The whole program needs to iterate over 50000 windows, which results in a total amount of 5TBs memory copying. Such a big overhead is not acceptable for efficient processing.

With sparse matrix compression, we decrease the memory usage down to 800B, every windows changing-over just needs 80KB, it is 875.6x faster than the original way.

3.2.3. Elimination of Redundancy via Building a Fast Table

When calculating the SNP probability, the likelihood module reads revising probability from p_matrix and stores the results to type_likely after calculation. The results has 262144 probable values, while there are 500 trillion calculation in one pass execution of the program. So that

most of the calculations are duplicated and can be avoided. We built a fast access table for computed values to eliminate redundant computation. We pre-calculate all probable results and save them into the table. Although it introduces additional 64MBs memory overhead, repeated calculation is prevented. For results storing, we save the data based on spatial locality of SNP probability calculation, so that we can increase the cache-hit rate. The likelihood module is 5x faster when employing this method.

3.2.4. Consistency Sort of the Gradient

Bases need to be sorted in likelihood the module to ensure the access order is in accordance with the calculation model. The original sorting algorithm consists of three rounds of multidimensional mixed gradient algorithm. For improvement, we carefully analyzed the characteristics of base data, and adjusted the original data storage pattern, making the original three rounds of hybrid gradient consistency gradient sorting sorted into one round.

3.2.5. Scalable Multilevel Parallelism

The original single CPU program is weak in scalability. For better performance, we exploit multilevel parallelism, which employs multi-threading for sites within the window and MPI based parallel processing across different compute nodes. A nearly linear scalability was achieved in our test.

3.2.6. Optimization of Utilizing MIC

After all above improvements, we found that the overall performance of the program is still not as good as expected and we discovered that the program has a low utilization rate of CPU and MIC wide vector unit. After a thorough analysis, we found that the original program is exhibiting poor locality of data accesses in the core algorithm.

To address this problem, we used loop expansion and space padding to improve spatial locality of data and the amount of calculation in loops, as well as making full use of CPU's 256-bit VPU and MIC's 512-bit VPU. Our test demonstrated that the utilization rate of CPU and MIC VPUs are up by more than 30%.

3.2.7. CPU/MIC Collaborated Parallel Computing Method

We used the offload mode to utilize the computing power of MIC. However, in a "naive" offload mode, CPU will be in an idle state after launching the offload part, and will only continue to work after the computation on MIC side completes and return the results back to the host CPU. Consequently, many CPU cycles are wasted.

To fully utilize all CPU cycles, we design the CPU/MIC neo-heterogeneous parallel computing framework, in which the CPU can do computation currently with the offload part running on MIC. This can be achieved by using one thread on CPU to communicate with MIC and using other threads for computation.

4. Evaluations on TH-2 supercomputer

4.1. Fast Windows Iteration

We evaluated the performance of MICA and compared it with SOAP3-dp and BWA in the following configuration: MICA: Compiled with Intel C/C++ Compiler version 13.1; one to three 57-core MIC cards (8G, ECC enabled). SOAP3-dp: version r176. Compiled with GCC 4.7.2 and CUDA SDK 5.5; one nVidia GeForce GTX680 (4G); 4 CPU cores; serial BAM output. BWA-MEM: version 0.7.5a. Compiled with GCC 4.7.2; 6 CPU cores; SAM output. Host machine: Intel i7-3730k, 6-core @3.2GHz; 64G memory. The performance comparison is listed in tab. 2.

Table 2. Performance of MICA

Data set		Yh150
MICA	1 card	22751(6.32h)
	2 cards	11479(3.19h)
	2 cards scale-up	1.98X
	3 cards	7728(2.15h)
	3 cards scale-up	2.94X
	Properly paired	95.48%
SOAP3-dp	1 card	22751(6.32h)
	Properly paired	95.01%
BWA-MEM	6 cores	101466(28.19h)
	Properly paired	92.32%

To note, there is no dependence among the sequences, so the sequence comparison has inherent parallelism, the parallel efficiency will not decrease when the number of nodes increases, as illustrated in tab. 3. The code of MICA is open-source online and evaluations in detail can be seen in [7].

Table 3. Speedup of MICA

Threads	Time elapsed	Speedup	Efficiency
56	58.3137	1.0000	1.0000
112	37.7260%	1.5457%	0.7729
224	21.8856%	1.6645%	0.6661

4.2. Evaluations of SOAPsnp

4.2.1. Evaluation Environment

Program optimization effects on a single compute node were tested in the Lv-Liang Supercomputing Center of China; the large-scale test was performed in TH-2 located in the National Supercomputing Center of Guangzhou. Compute nodes in two supercomputer centers share the same configuration, as shown in tab. 4.

Table 4. Data size and computation scale

Short sequence	Reference sequence	SNP database	Output results	Computation scale
2.2 GB	49 MB	9.7 MB	3.1 GB	50million sites

4.2.2. Effects of Algorithm Optimization

We have adopted a series of strategies to optimize our algorithms. Our strategies reduce the complexity of the algorithm, improve the efficiency of the algorithm of time and space, and further improve the program inherent parallelism, which has laid a solid foundation for heterogeneous parallel optimization. The effects of our optimization are illustrated in fig. 7.

Fig. 4 shows the acceleration effect of algorithm redesigning, "o0" represents the original algorithm, "o1" represents acceleration effect of 4-d matrix dimension reduction, it gets a speedup of around 2.3X, "o1+o2" represents acceleration effect using elimination of redundancy and building fast table on the foundation of "o1", the speedup is around 1.7X, "o1+o2+o3" represents acceleration effect of consistent gradient sorting on the foundation of "o1+o2", the speed up is about 1.6X.

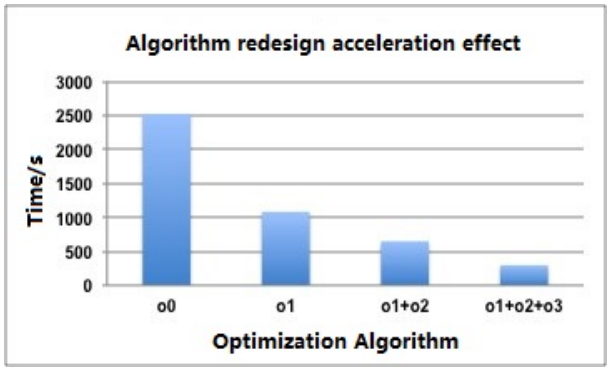


Figure 4. Algorithm Acceleration Effect of Mixed Methods

4.2.3. Overall Performance Boost via Parallelization on a Single Node

We utilized a CPU-MIC collaborated way to exploit the maximum of parallelism provided by each compute node. Fig. 5 demonstrates the speedups test on a single compute node of TH-2, using different number of MICs. It can be seen that one MIC card can achieve a performance equivalent to the CPU performance.

4.2.4. Hotspot Optimization Effect

Running time of the optimized program module is shown in Table V. Compared with the numbers before optimization (tab. 5), two hot module likelihood and recycle achieves a speedup of 85.4x and 875.6x respectively.

4.2.5. Overall Performance Boost via Parallelization and Scalability

The parallelism on a single node is mainly implemented via OpenMP while the parallelism across different compute nodes is achieve via MPI. We tested the scalability of our program on

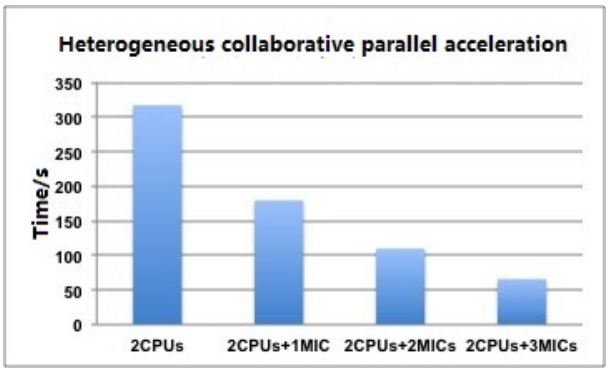


Figure 5. Heterogeneous Collaborative Parallel Accerleration

Table 5. Data size and computation scale

Module	likelyhood	recycle	Other modules	Total
Time/s	17.32	0.86	278.56	296.74
Percentage	5.84%	0.29%	93.87%	1

the TH-2 using at maximum 512 compute nodes. The scalability is depicted in fig. 6. Good scalability can be gained when we do large scale test using more human genome data (2000 human beings) on TH-2 supercomputer.

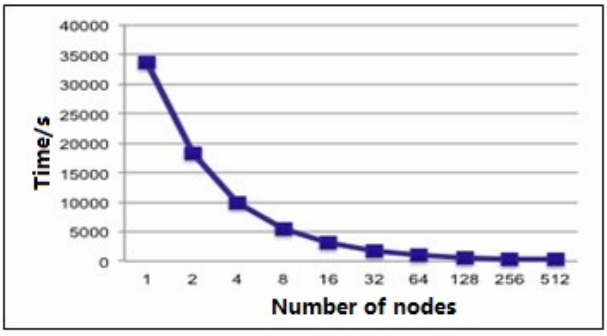


Figure 6. Scalability of the program

Conclusions

We analyzed the human genome resequencing software pipeline, and conducted in-depth parallel optimizations of two core components: SOAP3-dp and SOAP-snp. For SOAP3-dp, we improve the algorithm from a number of aspects, three-channel IO Latency hiding, vectorization of 512 Bit wide SIMD command, and parallelized construction of Smith-Waterman Dynamic matrix Etc. to accelerate the alignment procedure. We used 932 nodes on TH-2 to process 17.4TB whole human genome sequencing data, finished the work in one hour, getting a speedup of 2160x. For SOAPsnp, we propose 4-D matrix compression, fast windows iteration, scalable multilevel parallelism, CPU/MIC collaborated parallelism, and vectorization Etc. methods. We optimized the SOAPsnp in both algorithm and parallelism. The resulting program can achieve a 50x speedup on one single compute node, with a parallel efficiency over 73.6%. Tests on 512 nodes on TH-2 showed a speedup of 483.6x. It can finish the processing within one hour that would

take one week originally. At last, we analyze 300TB (2000 human beings whole genome) big data set using different scale nodes from 512 to 8192 nodes to analyze the scalability. At most, 8192 nodes on TH-2 are used to speed up the human genome resequencing analysis software pipeline deeply, which has generated some inspiring results. Using the improved pipeline, we performed an analysis of the 2000 peoples sequencing data and finished it in 8.37 hours, which would take 8 months using the original pipeline. In general, our optimized version is 700x faster. For future work, we are aiming for large-scale and long term deployment of our optimized pipeline on TH-2 supercomputer and we will perform analysis of genomics data from a much larger population of people, say 1 million human genome data.

We would like to thank Dr. Bingqiang Wang from BGI for providing the source code of SOAPsnp and Dr. Ruibang Luo and Lin Fang from BGI for providing related large scale test data. This work is supported by NSFC Grant 61272056, U1435222, 61133005, 61120106005, 91430218, and 61303191.

References

1. Marx V. Biology: The big challenges of big data[J]. Nature, 2013, 498(7453): 255-260.

2. Luo R, Wong T, Zhu J, et al. SOAP3-dp: fast, accurate and sensitive GPU-based short read aligner[J]. PloS one, 2013, 8(5): e65632. DOI: 10.1371/journal.pone.0065632.

3. Li R, Li Y, Kristiansen K, et al. SOAP: short oligonucleotide alignment program[J]. Bioinformatics, 2008, 24(5): 713-714. DOI: 10.1093/bioinformatics/btn025.

4. Li R, Yu C, Li Y, et al. SOAP2: an improved ultrafast tool for short read alignment[J]. Bioinformatics, 2009, 25(15): 1966-1967. DOI: 10.1093/bioinformatics/btp336.

5. Liu C M, Wong T, Wu E, et al. SOAP3: ultra-fast GPU-based parallel alignment tool for short reads[J]. Bioinformatics, 2012, 28(6): 878-879. DOI: 10.1093/bioinformatics/bts061.

6. Li R, Li Y, Fang X, et al. SNP detection for massively parallel whole-genome resequencing[J]. Genome research, 2009, 19(6): 1124-1132. DOI: 10.1101/gr.088013.108.

7. Chan S H, Cheung J, Wu E, et al. MICA: A fast short-read aligner that takes full advantage of Intel Many Integrated Core Architecture (MIC)[J]. arXiv preprint arXiv:1402.4876, 2014.

8. Luo R, Liu B, Xie Y, et al. SOAPdenovo2: an empirically improved memory-efficient short-read de novo assembler[J]. Gigascience, 2012, 1(1): 18.

9. Li H. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM[J]. arXiv preprint arXiv:1303.3997, 2013. DOI: 10.1186/2047-217x-1-18.

Received February 15, 2015.