A Dependency-Aware Parallel Programming Model

Josep M. Perez
josep.m.perez@bsc.es / perez@ac.upc.edu

Thesis submitted to
Universitat Politècnica de Catalunya

in partial fulfillment for the award of the degree of

DOCTOR OF PHILOSOPHY
in Architecture and Technology of Computers

Advisors:
Rosa M. Badia (rosa.m.badia@bsc.es)
Jesus Labarta (jesus.labarta@bsc.es)

Department d'Arquitectura de Computadors
Universitat Politècnica de Catalunya

Barcelona, October 2014

Acknowledgments

This thesis has been partially supported by the *Comisión Interministerial de Ciencia y Tecnología* (CICYT) and the *Spanish Ministry of Science and Innovation* under contracts TIN2004-00739-C02-01, TIN2007-60625, TIN2012-34557 and CAC2007-00052; the *Spanish Government* under grant SEV-2011-00067 of the *Severo Ochoa Program*; the *Generalitat de Catalunya* under contract 2009-SGR-980; the *BSC-IBM MareIncognito research agreement*; the *US Air Force* under grant FA8655-09-1-3075; and the *European Community* through the *Seventh Framework Programme* [FP7-INFRASTRUCTURES-2010-2] under the *TEXT Project*, with grant 261580.

This thesis has also benefited from the help of the anonymous individuals that have reviewed the publications generated during the course of this thesis, and from the improvements suggested by Eduard Ayguadé to early versions of those. This text has also benefited from the insightful comments from Xavier Martorell, Daniel Jimenez and Vicenç Beltran, who reviewed this text and evaluated an early rehearsal of the defense act.

Finally, this work could not have been possible without the participation of Rosa M. Badia and Jesus Labarta, who are the directors of this work and have contributed in every aspect.

Abstract

Designing parallel codes is hard. One of the most important roadblocks to parallel programming is the presence of data dependencies. These restrict parallelism and, in general, to work them around requires complex analysis and leads to convoluted solutions that decrease the quality of the code.

This thesis proposes a solution to parallel programming that incorporates data dependencies into the model. The programming model is able to handle that information and to dynamically find parallelism that otherwise would be hard to find. This approach improves both programmability and parallelism, and thus performance.

While at the time of this publication this problem has already been solved in OpenMP 4, this research begun before the problem was even being considered for OpenMP 3. In fact some of the contributions of this thesis have had an influence on the approach taken in OpenMP 4. However, the contributions go beyond that and cover aspects that have not been considered yet in OpenMP 4.

The approach we propose is based on function-level dependencies across disjoint blocks of contiguous memory. While finding dependencies under those constraints is simple, it is much harder to do so over strided and possibly partially overlapping sets of data. This thesis also proposes a solution to this problem. By doing so we increase the range of applicability of the original solution and increase the span of applicability of the programming model. This aspect is not currently covered in OpenMP 4.

Finally, we present a solution to take advantage of the performance characteristics of Non-Uniform Memory Access architectures. Our proposal is at the programming model level and does not require changes in the code. It automatically distributes the data and does not rely on data migration nor replication. Instead it is based exclusively on scheduling the computations. While this process is automatic, it can be tuned through minor changes in the code that do not require any change in the programming model.

Throughout the thesis we demonstrate the effectiveness of the proposal through benchmarks that are either hard to program using other paradigms or that have different solutions. In most cases our solutions perform either on par or better than already existing solutions. This includes the implementations available in well known high performance parallel libraries.

Contents

List of Figures

List of Tables

List of Source Codes

Chapter 1

Introduction

The computing landscape in the last recent years has changed radically. In the past multiprocessors were a specialty reserved only for high performance computing facilities. Today multicores have become mainstream and are present even on mobile devices. While this could be considered a hardware revolution, it will not be fully effective until it is accompanied by a software revolution that allows to exploit it to its full potential.

To make multicore programming widely accepted, parallel programming must become simple and effective. Data dependencies are one of the aspects that complicate the development of parallel applications. Most parallel programming models are not aware of data dependencies. Instead, the coder must perform the analysis of those and must adapt the code to them according to the desired performance goals. This task is complex, tedious, prone to errors, and those errors in turn are hard to diagnose.

Since most parallel programming models do not handle data dependencies, programmers usually have to structure their code in parts that launch parallel work and must isolate them by placing barriers between them. In addition to the effort required to do that, the final result can be suboptimal since barriers restrict parallelism.

The programming model that we present in this thesis has been designed to simplify parallel programming. It approaches this objective in two directions. First, through minimalism in the language and the number of parallelism constructs. And second, by moving complexity to the runtime. More specifically, the runtime handles the data dependencies, manages the distribution of data, and schedules the computations. The programmer only has to define the computation units that will run in parallel and has to synchronize those with the rest of the code.

1.1 Contributions

The main contributions of this thesis are organized in three chapters numbered 3, 4 and 5. The first presents the general programming model. The second extends it to support strided and overlapping data accesses. And finally the third optimizes it to exploit the performance characteristics of NUMA architectures.

1.1.1 General Programming Model

Chapter 3 presents the SMP superscalar (SMPSs) parallel programming model. At the language level it consists of a set of annotations added to regular C that make it capable of supporting parallelism. The key aspect that differentiates it over other models is that it relies on a powerful runtime that handles data dependencies, and thus removes that burden from the programmer. The main contributions of the chapter are the following:

A dependency-aware parallel programming model We propose a parallel programming model that is dependency-aware and that takes advantage of this knowledge to simplify the development of parallel applications and that has the potential to perform better than dependency-unaware programming models due to its ability to find more parallelism. Its main characteristics are the following:

Simple and task based While many parallel programming models provide several means to parallelize an application, the solution that we propose only relies on *tasks*, that we define as regular functions with an additional annotation to extend the information about their parameters. By using only one parallelism construct we strive to make the model simple through minimalism.

Address space independent The design of the language is such that tasks must specify all the information about the data that they access. This allows the runtime to abstract the address space, which allows the model to be applied to environments with separated address spaces. While this thesis only covers shared memory, the model has been successfully applied to distributed environments.

Unconstrained parallelism versus potential parallelism While traditional parallel programming models are based on specifying the computations that must be able to run in parallel, the model that we present allows to specify the computations that have the potential to run in parallel. This difference is crucial. The actual parallelism is determined by the data dependencies between the tasks, and by the order in which they are satisfied.

Long and dynamic parallel span Dependency-unaware programming models usually cope with dependencies by using barriers. These limit the potential window in which a computation must be executed. By being dependency-aware, the programming model we propose removes the limitations imposed by barriers over the parallel span of the tasks. Instead tasks can execute as soon as their dependencies have been fulfilled, and can be delayed as long as other tasks are also available. This enables the model to exploit distant parallelism that would otherwise not be available. In addition longer parallel spans increase the amount of parallelism that is available, and thus potentially reduce starvation.

A dependency-aware scheduler We propose a scheduling algorithm that follows data dependencies to favor the reuse of data already in the cache of the processors. This improves the performance of applications in which the cost of the task first-access data cache misses is significant.

A set of benchmarks parallelized with the programming model We describe a set of benchmarks that consist of a few embarrassingly parallel benchmarks and others that have data dependencies and that are either hard to implement or do not generate enough parallelism in other programming models.

An evaluation of the performance and programmability The benchmarks are compared with implementations in other programming models and with the implementations within high performance libraries. The evaluation covers both the performance and how the problem is solved in each model.

1.1.2 Extensions to Support Strided and Overlapping Data Accesses

Chapter 4 discusses extensions to the programming model to add support for finding dependencies between strided and possibly overlapping data accesses. Its main contributions are the following:

Extensions to the programming syntax To support the specification of the data accesses of a task, we present an extension to the annotations that allows to specify accesses over multidimensional rectilinear subsets of the data. It allows the programmer to cope with non-blocked data layouts, and dependencies that span a dynamic number of subsets of the data. Thus, it makes the model suitable to a wider set of applications.

A compact representation for sets of addresses To perform the data dependency analysis that the extended syntax enables, we propose an internal compact representation for a set of addresses, and an efficient algorithm to construct it from its representation in the language. The compact representation allows to calculate the intersection between two sets with a complexity equal to the number of bits of an address.

A data structure to perform data dependency analysis efficiently Data dependency analysis consists in checking the relations between accesses to the same data. To perform this efficiently, we design a data structure based on the compact representation to hold the state required to calculate data dependencies, that allows to perform intersection look-ups and replacements efficiently.

An evaluation of the performance and programmability We extend the evaluation started in the previous chapter with versions of the benchmarks that use flat multidimensional arrays and additional benchmarks that are not possible without the extensions. Where possible we compare them to the blocked implementations, alternative implementations with other programming models and highly tuned parallel library implementations.

1.1.3 Extensions to Exploit NUMA

Finally, chapter 5 proposes extensions to exploit the characteristics of NUMA systems within the programming model. The main contributions in this area are the following:

A model of data distribution We demonstrate that the elements of the syntax of the SMPSs programming model with region support are enough to define data distributions. The proposal differs from previous work in that the user only needs to define the *units of distribution* through the use of *data initialization tasks* and does not require to specify their placement nor where to run the computations. This is instead determined by the runtime.

A runtime policy for data placement We propose a simple data placement policy to allow the automatic distribution of data within the runtime. The policy is dynamic and balances the amount of data placed in each memory. The final data distribution is determined by this policy in conjunction with the units of distribution defined by the data initialization tasks in the program code.

A NUMA-aware scheduling policy We design a scheduler that exploits NUMA affinity by favoring the execution of tasks in the cores that are local to the memories that contain most of its data. As a side effect the scheduler also balances the homogeneous usage of the memory bandwidth. This part also demonstrates that the data structure proposed in the previous chapter to calculate data dependencies over strided data accesses can be reused for other purposes.

Analysis of the effects of data distribution and NUMA-aware scheduling over performance We expand the evaluation of the previous chapter by analyzing the effects that NUMA aspects have over performance. First we analyze the effects of data distribution with several *distribution shapes*. Second, we analyze the effects of NUMA affinity by using several policies and evaluate the effectiveness of the NUMA-aware scheduling policy over alternative schedulers. Finally we compare against NUMA-unaware schedulers and competing implementations.

1.2 Thesis Organization

This thesis is organized in five main chapters and six appendixes. Their respective contents are the following:

Background Chapter 2 describes the state of the art at the beginning of this thesis and previous work that is related to some of the solutions that we propose.

A dependency-aware task-based programming model Chapter 3 presents the general programming model, its implementation and evaluates its performance.

Strided and overlapping data accesses Chapter 4 extends the programming model to support data dependency analysis between tasks that access strided an possibly overlapping data.

Exploiting Non-Uniform Memory Access Chapter 5 is centered on improving performance of the programming model with the extensions of the previous chapter on NUMA architectures.

Conclusions, impact and future work Chapter 6 highlights the conclusions that derive from the previous chapters, presents the impact of the contributions of this thesis and sets up future directions that complement the work presented in this text.

SMPSs extensions to the C99 grammar Appendix A summarizes the complete set of grammar rules that extend C99 with the annotations that define the language including the extensions to support strided and overlapping data accesses.

Compiler and runtime integration Appendix B discusses the interface used by the runtime of the programming model without any extension and overviews the compilation steps to support the language and runtime.

Additional benchmarks for the block-based programming model Appendix C extends the evaluation of the programming model of chapter 3 with additional benchmarks.

Compiler and runtime interface extensions for region support Appendix D presents the extensions to the runtime interface and compiler to support strided and overlapping data accesses.

Additional benchmarks for the region-based programming model Appendix E extends the evaluation of the programming model presented in chapter 4 with additional benchmarks and presents versions of the ones in chapter 3 and appendix C using flat arrays.

Additional benchmarks for the NUMA-aware runtime Finally, appendix F presents the performance evaluation of regions-based benchmarks under the NUMA-aware runtime that do not appear in chapter 5. These benchmarks did not improve their performance significantly or their improvement diminished as the problem size was bigger.

Chapter 2

Background

This chapter overviews the state of the art on the areas covered by the contributions of this thesis. These have been classified into three main topics: parallel programming models for shared memory, techniques to exploit unstructured parallelism, and techniques to exploit non-uniform memory access (NUMA) architectures. The analysis is centered on the programming models from the point of view of the programmer and explores the memory model, the control flow model and the elements of the language.

Note that since this thesis started in 2005, the research presented in this document has already had an impact on current programming languages, and thus while some contributions may be taken today for granted, they were at most at their early planning stages when this research was conducted.

2.1 Parallel programming models for shared memory

For our discussion of parallel programming we are interested on one hand on control flow and address space, and on the other on data distribution and data location awareness. These aspects are considered throughout this thesis and we consider that they play an important role in the programmability of parallel systems.

Regarding the aspects that define how the programmer sees the control flow and the address space, we are interested in the differences between *local-view* and *global-view* as presented by [Chamberlain et al., 2007], how the user sees the address space, and how it can be accessed. Local-view corresponds to models in which the code only has direct access to the address space of the current thread or process. Global-view programming models have a unique and complete view of the address space. All these aspects have a direct impact on the simplicity of the programming model and are related to the programming model presented in chapter 3.

Regarding data distribution and data-location awareness the most relevant topics are how the data is placed, which data distribution shapes are possible, and how the placement of the units of work are related to the location of the data they access. These aspects are related to the contents of chapters 4 and 5, and have a direct relation to the performance that can be achieved and the effort necessary to achieve it.

While parallel programming models can be specifically aimed at either distributed processing and shared memory, in many cases distributed models are also used in shared memory systems. Of special notice is the use of MPI on NUMA systems to make good use of memory affinity.

2.1.1 MPI

The Message Passing Interface (MPI), standarized by the [Message Passing Interface Forum, 2009], is a library-based interface used for programming parallel applications using message passing.

This paradigm is typically used in distributed systems. However, it is also used in NUMA systems since it moves computations close to the memory of the data they access and thus can be beneficial for performance.

MPI is one of the most used message passing standards for High Performance Computing on distributed systems.

Control Flow and Address Space

While MPI allows many forms of parallelism, in most cases it is used for SPMD (Single Program Multiple Data) programming. That is, a single program is launched at several *locations* in parallel, and each instance, the *MPI process*, processes part of the data and communicates with the rest of the MPI processes using the facilities provided by the model, in this case the MPI API.

In MPI, processes can only access their local data. In general, to access remote data, MPI processes must explicitly perform data transfers between local and remote address spaces. However, MPI also provides limited support for global operations in the form of reductions, and synchronization primitives like barriers.

To parallelize an application using the SPMD paradigm, the global data must be manually divided in parts that are local to each process, and each process must perform the calculations that correspond to its local data, and communicate with other processes. Communication can be either one-sided or two-sided. In the first case, processes initiate data copies from one address space to another, one of them being their own address space. Two-sided communication requires the processes of each address space to submit in a coordinated way matching pairs of send and receive operations according to the direction of the data copy.

The combination of an SPMD model with an explicit communication philosophy makes it a pure local-view model, and makes programs that use MPI hard to understand and hard to debug. Moreover, the order of the messages is a critical aspect when programming in MPI, since a bad ordering may lead to dead-locks.

Data Distribution and Location-Awareness

MPI does not provide any high-level primitives to distribute the data nor any means to inquire about its location. Instead, the data must be manually placed by the programmer, and the accesses must be always performed locally. Data placement and computation placement must be controlled manually by the programmer.

Orthogonality

While MPI is a complete programming model, due to its distributed nature, it can be combined with shared memory programming models. For instance, [Rabenseifner, 2003], [Lusk and Chan, 2008], and many others propose parallelizing the code that runs on each node by using OpenMP, instead of running as many MPI tasks as cores within a node.

Similarly, MPI can also be combined with the programming model that this thesis presents. While OpenMP is helpful in reducing the unnecessary data copies between MPI processes of the same node, [Marjanović et al., 2010] demonstrates that SMPSs further simplifies overlapping computations with communications and can help to accelerate the critical path.

2.1.2 Partitioned Global Address Space models

Partitioned Global Address Space models, abbreviated as PGAS, replace the distributed memory model used in message passing by a global address space. However, while the address space is unique, it is divided in parts that are distributed across the computing nodes. Hence, while computations that only involve data that is local can be executed as-is, computations involving remote memory require communication. An overview of the status of the PGAS landscape has been recently presented by [El-Ghazawi and Smith, 2013].

Execution Model

The most common execution model used in the PGAS models is the SPMD. Unified Parallel C (UPC) by [UPC Consortium, 2005], Co-Array Fortran (CAF) by [Numrich and Reid, 1998], X10 by [Charles et al., 2005] and Titanium by [Yelick et al., 1998] use SPMD as their base execution model. X10 additionally supports submitting asynchronous units of work to remote nodes.

Chapel by [Chamberlain et al., 2007], similarly to OpenMP 3.0 by [OpenMP Architecture Review Board, 2008], combines the fork-join model with asynchronous units of work.

Address Space

While all PGAS models have a partitioned and unique address space, they differ on how it can be accessed. The distinction between local and remote accesses can be exposed in the language syntax or hidden under a homogeneous syntax. In UPC, X10, Chapel and Titanium, the syntax does not differentiate between local and remote data. However, while UPC, X10 and Chapel allow to directly express references for remote data, Titanium requires to get references to remote data through communications. In addition X10 does not allow to access remote data directly, instead it forces to send the computation to the location of the data. In CAF, data declarations and data accesses explicitly contain information about their location.

For performance reasons, UPC, CAF and Titanium also have a specific syntax for data that is local.

Data Distribution

Data in UPC and CAF is distributed in rectilinear subsets of fixed dimensions. That is, arrays are declared as being partitioned in equally sized rectilinear chunks. The X10 and Titanium languages, since they follow the SPMD paradigm, distribute the data implicitly like MPI. Chapel allows more flexibility by allowing arbitrary distributions that can be defined programmatically.

Data and Computation Co-Location

The SPMD model, as used by UPC, CAF, X10 and Titanium, lets the user determine where computations take place. In that sense, the user is responsible for placing computation and distributing data in a way that is suitable for the desired performance goals. While UPC, CAF and Titanium allow to freely access remote data, X10 is more constrained. In X10, the user code must send computations to the location where the data that is written is stored. Remote data that is read is sent as a copy when the computation is sent to its destination.

In Chapel, computations can be sent manually to specific nodes, and are sent automatically in parallel global operations. Parallel zippered iterations are also sent automatically to the corresponding location as defined by the work of [Chamberlain et al., 2011].

Orthogonality

All the PGAS programming models mentioned in this text have parallel programming primitives similar to the tasks of SMPSs. However, they do not have the information to calculate the dependencies nor the dependency semantics of SMPSs. In this sense, the contributions of chapters 3 and 4 are orthogonal and thus their task-like constructs could be extended to incorporate those contributions.

2.1.3 Cilk

Cilk by [Frigo et al., 1998] is a minimal extension of the C programming language that adds nested task parallelism.

Execution Model

Cilk has a sequential main control flow. *Tasks* are asynchronous computation units that are instantiated either in the main control flow or within the control flow of other tasks. They execute asynchronously in the different execution locations.

To allow using data generated by tasks, Cilk includes a synchronization primitive that guarantees that all tasks instantiated in the context before the synchronization point have been executed after the synchronization point has been crossed. This allows Cilk programs to have a certain amount of dependency control between tasks.

Cilk supports spawning tasks within tasks. This is called task nesting and allows it to express in a simple manner parallelism in recursive algorithms. While expressing a recursive algorithm as a decomposition up to the most elemental operation may be desirable in terms of elegance, the cost of spawning tasks below a certain threshold can be prohibitive. For this reason, the Cilk programming model incorporates the notion of *fast task*, that corresponds to the tasks generated at a certain level of the recursion, that are not further decomposed into smaller tasks. Instead, those tasks handle task spawns as regular function calls, and ignore further synchronization primitives.

This model achieves parallelism in two ways. First, by decomposing problems recursively, it effectively parallelizes the generation of work. And second, the work is parallelized at the granularity of the fast tasks and at the granularity of the processing performed by their parent tasks.

Address Space, Data Distribution and Computation Co-Location

Cilk is based on a shared memory model with a single address space. Some works like the ones of [Peng et al., 2000] and [Blumofe et al., 1996] analyze the model on top of Distributed Shared Memory (DSM). [Amos, 2006] discusses the model when applied to Cache Coherent NUMA (ccNUMA). While it presents mechanisms for data distribution and computation co-location, its results sections does not show significant performance improvement over NUMA-unaware data and computation placement.

Orthogonality

Cilk tasks are similar to SMPSs tasks, but allowing nesting. However, they do not have the information required to calculate dependencies nor the dependency semantics of SMPSs. While nesting is a form of dependency, it is orthogonal to the data dependencies that this thesis proposes. Therefore both can types of dependencies can coexist. In this sense, the contributions of chapters 3 and 4 are orthogonal and thus Cilk tasks could be extended to incorporate those contributions.

2.1.4 OpenMP 3.0

OpenMP 3.0 by the [OpenMP Architecture Review Board, 2008] is a standardized programming model for shared memory multiprocessors. When this thesis started, the standard was in version 2.5 and did not contain task-based parallelism. It only had primitives for fork-join style parallelism. In version 3.0 it added primitives for Cilk-style tasks. Most recently, the [OpenMP Architecture Review Board, 2013] published version 4.0 of the standard, which incorporates some of the ideas contributed in this thesis in chapter 3. For this reason, in this thesis when comparing to OpenMP we only do so against version 3.0 instead of the most recent one.

Execution Model

OpenMP has a fork-join model combined with asynchronous units of computation.

Address Space, Data Distribution and Computation Co-Location

OpenMP is a shared memory model and thus has only one address space. While [Bircsak et al., 2000] present extensions to the language to add NUMA support, these have not been considered in further versions of the standard. [Nikolopoulos et al., 2000] present a transparent mechanism that does not involve any change in the syntax and that is based on page migration. However, to this author's knowledge, such mechanism has yet to have an impact on commercial OpenMP frameworks.

Orthogonality

The OpenMP 3.0 tasks are very similar to the Cilk tasks. Therefore the applicability of the contributions of this thesis are the same. While nesting is a form of dependency, it is orthogonal to same level data dependencies. Therefore, the contributions of chapters 3 and 4 could be incorporated into OpenMP 3.0 tasks. In fact, OpenMP 4 tasks already does incorporate the contributions of chapter 3, and in part an alternative to the syntax proposed in chapter 4. However, it still does not include support for strided and partially overlapping data dependencies.

2.1.5 Sequoia

Sequoia is a parallel language based on asynchronous computation decomposition that is aware of the memory hierarchy.

Execution Model

Similarly to Cilk, it has a main execution control control flow that spawns asynchronous units of computation with recursion.

Address Space

Sequoia has a single global address space.

Data Distribution and Computation Co-Location

In Sequoia data moves transparently across the different levels of the memory hierarchy. While to this author's knowledge, it is NUMA-unaware, many of the ideas and techniques used in its implementation for the Cell/B.E. also apply to NUMA architectures.

Orthogonality

Sequoia tasks are very similar to the Cilk tasks. Therefore the applicability of the contributions of this thesis is the same. While nesting is a form of dependency, it is orthogonal to same level data dependencies. Consequently the contributions of chapters 3 and 4 could be incorporated into Sequoia.

2.2 Unstructured Parallelism

Unstructured parallelism is naturally present in algorithms that deal with highly dynamic data structures. For instance, algorithms that operate over sparse matrices, graphs or trees. Usually, the units of parallel computation are discovered as these data structures are traversed, which makes them difficult to parallelize using constructs designed for structured parallelism.

2.2.1 Tasking and Task Nesting

Unstructured parallelism can be exploited by decoupling the execution of actual computation from the place it is instantiated. In this thesis we refer to the spawned unit of computation as *task*. By decoupling instantiation from execution, the main control flow can instantiate tasks dynamically without waiting for their execution, thus generating parallelism.

To preserve the correctness of the code against reordering between task code and out-of-task code, task-based programming models provide synchronization mechanisms. The *barrier* is the most commonly implemented synchronization primitive. It guarantees that after crossing it, all previously instantiated tasks within its scope will have already been finished.

Task nesting allows tasks to be created within the scope of the execution of other tasks. Tasks instantiated within the scope of a parent task may or may not need to be executed within that scope depending on the programming model. Some programming models add implicit barriers to the end of each task to guarantee nested execution, and others detach children tasks and require manual barriers to force it. This model is the basic parallel paradigm used by [Frigo et al., 1998] in Cilk and was added to OpenMP in version 3.0 [OpenMP Architecture Review Board, 2008].

2.2.2 Futures

Futures have been proposed by [Baker and Hewitt, 1977; Chatterjee, 1989; Friedman and Wise, 1978], among others, as a way to parallelize programs composed of functions without side-effects. Futures decouple the point in which a function is invoked from the time it is executed. The outputs of such functions are populated with *promises* for their future values which can be passed on to other functions.

Functions instantiated using futures, also known as *called-by-future*, may be executed in parallel at any time and may also be delayed. [Friedman and Wise, 1978] use the term *suspension* to describe the act of delaying a task, and *coercion* to the act of starting it. Whenever a value is needed, if the value has not been calculated already, it is *coerced* by forcing the execution of the function that calculates it. This action may result in further coercion of other values.

Future-based models have similarities to task nesting models. Function calls in future-based models are similar to tasks since they can be delayed, and can call other functions, hence leading to semantics similar to task nesting. Coercion, which can be transparent, is similar to partial barriers in task-based models. However, due to the recursive nature of coercion, future-based models have similar capabilities to dependency-aware models. When used exclusively with functions free of side effects, futures allow exploiting as much parallelism as the model proposed in this thesis plus task nesting. However, when implemented on Object Oriented languages, since method calls may alter the state of the target object, they are not free of side effects. This creates a series of shortcomings:

Less potential parallelism or convoluted mutual exclusion Objects can have several method calls outstanding. They can be either served sequentially or in parallel. Serving them sequentially reduces the potential amount of parallelism that can be extracted, and blocks the generation of further work. Serving them in parallel might require using mutual exclusions (to avoid concurrent modification, inconsistent state, ...), which makes it complex since the user then has to deal with potential distributed dead-locks. However, as noted by [Friedman and Wise, 1978], these problems can be eliminated by changing the semantics of assignment to make it non-destructive.

More opportunities for deadlocks [Ábrahám et al., 2009] describe how futures, when used with side effects, can lead to deadlocks.

Additionally, futures do not preserve the semantics of sequential execution, which can lead to inconsistent results between executions. Our model does preserve sequential execution semantics in the general case, and only violates it in reductions.

2.2.3 Dependency Analysis

Dependence analysis has been a hot topic for many years in the field of automatic parallelization. Initially [Banerjee, 1976, 1988; Wolfe, 1993] used it at compile time for finding loop-carried dependencies between array accesses to parallelize loops. Their effectiveness has been analyzed by [Blume, 1992; Eigenmann et al., 1998].

Dependence analysis evolved to cover parallelism beyond loop iterations. [Balasundaram and Kennedy, 1989] used a summarized representation for data accesses called the *Data Access Descriptor*. By using summaries, the authors were able to extend data dependence analysis across loop nests and subroutines. This technique was also extended by [Girkar and Polychronopoulos, 1992] to find task-level parallelism using the *Hierachical Task Graph*.

2.2.4 Dependency-Aware Programming Models

Several authors have proposed adding extensions to handle dependencies to OpenMP. [Sinnen et al., 2008] proposed adding a pair of new constructs called *tasks* and *task* similar to the *sections* and *section* constructs. However the new constructs had a name and a clause to specify dependencies by name. The scheduling they proposed is static and is decided at compile time. However, the semantics of these directives do not prevent scheduling the tasks at run time.

A similar model has been proposed by [Gonzalez et al., 2000, 2003]. However, this model does not limit dependencies to section-like constructs. Their syntax allows to specify dependencies between work-sharing constructs, and even refer to particular iterations. They also present an implementation that schedules at run time.

While the proposals of both [Sinnen et al., 2008] and [Gonzalez et al., 2000] allow to *program with dependencies*, they leave the burden of finding out the dependencies to the programmer, who must specify them explicitly. This is a major difference with the model of this thesis.

[Abdelrahman and Huynh, 1996; Huynh, 1996] presented a task-based programming model with nesting. The model is dependency-aware thanks to a combination of compiler analysis and run time constraint checks. The compiler recognizes accesses to arrays and includes code in each task that verifies that an access does not violate the semantics of the sequential execution. Otherwise, it suspends the task until that access can be performed safely. In that sense, it is similar to future-based models. However it allows functions with side-effects without indeterminism by forcing sequential execution semantics.

While this thesis is centered on the design of the programming model and its applicability to shared memory multiprocessors, the model is also applicable to distributed memory systems. In particular [Bellens et al., 2006; Perez et al., 2007] have applied it to the Cell Broadband Engine processor in the form of *Cell superscalar (CellSs)*, and [Perez et al., 2006] have applied it to a Grid environment. Selected contributions from this thesis have also been proposed as an extension to OpenMP by [Duran et al., 2008, 2009] which further guided the changes in OpenMP 4 to include tasks with dependencies.

Chapter 3

A Dependency-Aware Task-Based Programming Model

3.1 Introduction

Parallelizing applications, beyond those that are embarrassingly parallel, is a complex endeavor. The presence of dependencies imposes limitations and is a great contributor to the complexity of the analysis and the solutions.

Parallelizing codes, in many cases requires a certain level of code restructuring. Often, the code has to be reordered in a form that is suitable to be parallelized. In some others, restructuring is needed to achieve good performance.

When parallelizing in the traditional forms, the programmer must think about *the order of the execution*, and *the interactions between the code that runs in parallel*. These aspects tend to make parallel programming complex and hard to debug. Moreover, the solutions to these problems usually add elements that are extraneous to the underlying algorithm.

These sort of shortcomings can be solved in some cases by using already parallel domain-specific frameworks. However, these exist only for certain domains, and only cover a limited subset of problems. An thus, do not solve the need for a general solution.

Parallelizing compilers have the potential to provide a general solution. However, to this date they have not been widely accepted. Some of the issues that limit their effectiveness are due to the design of the target programming languages. The most common programming languages used in parallelizing compilers make it difficult to determine pointer aliasing and the range of data that is accessed at compile time. Moreover, they lack the information to aid the compiler in deciding whether to parallelize a section of code and which granularity to use.

Traditional parallel programming models rely on the programmer to specify the parts of the code that *must* or *can always* run in parallel to other parallel parts of the code. Throughout this document we call their instances *parallel execution units*. These are defined with precision and typically amount to small spans of the code. For instance, the OpenMP standard provides several parallel constructs that are added to

C and Fortran that determine the parallel execution units. They can be either single statements, blocks of code or groups of loop iterations. OpenMP 3 also provides some dynamism through the instantiation of tasks, these are restricted to run in parallel to other tasks and parallel execution units. In this document we refer to the *execution span* as the span during which a parallel execution unit must start and finish. The OpenMP 3 tasks, although they are dynamically created, have a typically small execution span.

Partitioned Global Address Space programming models (PGAS) like Co-Array Fortran by [Numrich and Reid, 1998], Chapel by [Chamberlain et al., 2007], Titanium by [Yelick et al., 1998] or X10 by [Charles et al., 2005] also rely on the programmer to decide which parts of the code *must* or *can always* run in parallel, and these typically have a small execution span too.

The programming model we present in this chapter, SMP superscalar (SMPSs), relies on the programmer to specify the parts of the code that *could* run in parallel to the rest of the code. This contrasts with most parallel programming models which specify the parts of the code that *must* run in parallel to a small and concrete part of the code. Unlike other programming models, the execution span in SMPSs is only bound by explicit barriers, and those are infrequent. While other programming models use barriers to guarantee correct executions, SMPSs uses a much finer approach. Instead, it uses data dependencies between its parallel execution units.

To make the programming model as simple as possible, it only has the *task* as parallel construction. Although the foundations of the model do not require it, in SMPSs tasks are always functions. They can be invoked at any time in the code, and thus they are instantiated dynamically.

In OpenMP 3, tasks within a parallel span *can always* execute in parallel. This forces the programmer to determine data dependencies and to isolate parts that can run in parallel from parts that cannot. SMPSs does not impose that restriction, and thus reduces the effort needed to produce a correct parallel program. Instead, the programmer indicates that tasks are the units that *could* run in parallel and specifies enough information for the SMPSs runtime to detect the dependencies. With that information, the runtime produces valid parallel executions.

By basing the execution span on the data dependencies, it can become larger than that of a model that constrains it with barriers. As a consequence, parallel execution units that would otherwise have disjoint execution spans, can enlarge it and thus increase the potential parallelism of the application. This effect is described in more detail in the evaluation section.

The programming model consists of an execution model, that is discussed in section 3.2, and a set of annotations that extend standard sequential programming languages with the capabilities of the execution model. This is presented in section 3.3. Due to the dynamism of the model, the role of the compiler is very limited. Instead, all the features of the model are implemented in the runtime, which is described in section 3.4. Section 3.5 presents the features of the model and the runtime that help in debugging and profiling applications. And finally section 3.6 presents and measures the performance of a set of applications under the programming model and compares it to that of other programming models and highly tuned parallel libraries. The evaluation is further extended in appendix C with additional benchmarks that have been left out of this chapter for brevity.

3.2 Execution Model

The SMPSs execution model is based on the execution in parallel of the parallel execution units while preserving the semantics of the unannotated sequential code. During the program initialization, the model spawns a set of worker threads that it uses to execute the tasks. The main thread runs the main code as if it was a sequential program. As it finds task calls, it instantiates them, calculates their dependencies and continues running the main code. As the dependencies are satisfied, the task instances get scheduled to the worker threads. These are kept alive until the program finishes.

Parallel Execution Unit

In SMPSs, unlike other programming models, there is only one parallel execution unit: the *task*. Tasks are functions that have been added an annotation that indicates that it is a task. The annotation also includes information that allows to find their data dependencies at run time.

Dynamic Creation of Work

Like in OpenMP, SMPSs tasks are instantiated dynamically within the program. Since their parameters cannot be determined during compile time, the dependencies are determined at run time. This flexibility allows the programming model to handle the parallelization of codes whose units of computation heavily depend on their input data. For instance, sparse algorithms typically have unstructured parallelism that depends on the sparseness of the data they handle.

Control Flow and Asynchronism

Task invocations have asynchronous semantics. After their instantiation, the main program control flow can continue running, and the actual task invocation may be delayed. Task invocations can run in parallel to other tasks and to the main program control flow.

By calculating the task dependencies through their parameters, the runtime guarantees safe data accesses between tasks of only the data passed by parameters. That is, it does not guarantee concurrent access safety between the tasks and the main control flow nor over data other than the one passed by parameter to the tasks.

Thus, to achieve inter-task data concurrent access safety, all the data must be passed by parameters to the tasks. However, to achieve data concurrency safety between the tasks and the main control flow, the programming model provides two synchronization primitives.

Parallel Execution

By executing tasks only when their dependencies have been satisfied, the runtime can guarantee that the execution generates the same result as the original sequential code. Moreover, since more than one task may have its input dependencies satisfied at a given time, there is potential to run them in parallel.

Out-of-Order Execution

Data dependencies determine a partial order of execution that must be followed to produce valid results. However, in many of the valid schedules are not just parallel, but have tasks executed in a different order than that of their instantiation.

Parallel Span

Since some parts of the main program may not be executed in parallel to some tasks or have ordering issues, the programming model provides two synchronization primitives to limit the parallel span of tasks. One is a global barrier which forces all previously instantiated tasks to be executed. The other is a partial synchronization point, which is finer grained. This one forces all tasks that access a given set of data to be executed.

In other parallel programming models that use barriers to isolate the parallel execution units that can coexist, the parallel execution units that are instantiated between each pair of barriers typically have parallel spans that essentially cover the span between the two barriers.

In our programming model, since dependencies determine when task instances can be started, the parallel spans are not structured in barrier generations, and in fact they are typically very different from task instance to task instance. Tasks can start to execute from the time that their input dependencies have been satisfied, which could be as soon as when they are instantiated, and must have finished before crossing a full barrier, or a partial synchronization point over any of the data accessed by the given task.

Address Space Independence

Task declarations include additional information that allows the runtime to calculate dependencies. This information is in essence the directionality of the task parameters. That is, whether a parameter is read, written or read and written. Tasks must only access the data passed to them as parameters to prevent accesses that violate the semantics of the sequential execution.

By following this approach, it is possible to run the tasks over data allocated on different addresses than those of their instantiation, as long as the actual data definitions are transferred from one memory location to the other. This allows the model to work in a manner that is independent of the address space.

While SMPSs is a programming model for shared memory multiprocessors, it can be used on distributed memory systems too due to these properties. This has been demonstrated by [Perez et al., 2006].

Renaming

Address space independence enables more aggressive techniques to improve parallelism. Renaming is a technique that removes so called *false dependencies*. It has been successfully used in out-of-order superscalar processors, as decribed by [Smith and Sohi, 1995], and optimizing compilers, as described by [Kuck et al., 1981]. False dependencies occur when some form of storage is first read and then written, or when it is written and then written a second time. In both cases, to preserve the correct semantics, the operations must be performed sequentially. However, in each case renaming moves the second operation to a different storage location, and thus it removes the dependency.

In processors, *register renaming* allows instructions that would be delayed due to false dependencies to be executed earlier by using a different register than the one that produces the false dependency. In optimizing compilers, *variable renaming* allows to reorder statements that share a variable that produces a false dependency.

3.3 Language Syntax and Semantics

SMPSs is an extension to existing sequential programming languages that adds parallel capabilities. Similarly to OpenMP by the [OpenMP Architecture Review Board, 2008], it consists of a set of annotations that are added on top of sequential programming languages that add information related to the parallelization. This approach simplifies the reuse of already existing highly tuned kernel libraries, and the parallelization of already existing codes. This thesis only covers the syntax for C99, although a similar syntax has also been produced for Fortran.

The grammar rules of this thesis are written using the Backus-Naur Form (BNF). Terminals appear in bold face, and non-terminals appear between angle brackets. Each rule is identified by a number in parenthesis followed by the symbol it defines. Rules may have one or more subrules that satisfy it. The symbol of each rule is followed by an arrow (→), and one line for each possible subrule. Subrules begin with a group of numbers in parenthesis that correspond to the rule number followed by a period and the subrule number. The first subrule then continues with the sequence of terminals and non-terminals that satisfies it. The rest continue first with a vertical bar that signifies that it is an alternative and then the sequence that satisfies it.

3.3.1 Initialization and Finalization

The execution model of SMPSs consists of a main control flow that traverses the main code and instantiates the tasks, and a set of worker threads that execute them. The *start* and *finish* directives determine the points where the worker threads are forked and joined respectively. Their grammar rules are the following:

(1) ⟨start-directive⟩ →
(1.1) **#pragma css start**

(2) ⟨finish-directive⟩ →
(2.1) **#pragma css finish**

These directives may appear anywhere that a C99 statement may appear.

3.3.2 Tasks

The main parallelism construct in SMPSs is the *task*. A task is a function that has been annotated with information about how it accesses its parameters. The grammar of the task construct is the following:

(3) ⟨task-construct⟩ →
(3.1) ⟨task-declaration⟩
(3.2) | ⟨task-definition⟩

(4) ⟨task-declaration⟩ →
(4.1) ⟨task-pragma⟩ ⟨function-declaration⟩

(5) ⟨task-definition⟩ →
(5.1) ⟨task-pragma⟩ ⟨function-definition⟩

(6) ⟨task-pragma⟩ →
(6.1) **#pragma css task** ⟨opt-task-clauses⟩ ⟨new-line⟩

(7) ⟨opt-task-clauses⟩ →
(7.1) ⟨task-clauses⟩
(7.2) |

(8) ⟨task-clauses⟩ →
(8.1) ⟨task-clauses⟩ ⟨task-clause⟩
(8.2) | ⟨task-clause⟩

(9) ⟨task-clause⟩ →
(9.1) **input (** ⟨task-parameter-list⟩ **)**
(9.2) | **output (** ⟨task-parameter-list⟩ **)**
(9.3) | **inout (** ⟨task-parameter-list⟩ **)**
(9.4) | **reduction (** ⟨task-parameter-list⟩ **)**
(9.5) | **highpriority**

(10) ⟨task-parameter-list⟩ →
(10.1) ⟨task-parameter⟩
(10.2) | ⟨task-parameter⟩ **,** ⟨task-parameter-list⟩

(11) ⟨task-parameter⟩ →
(11.1) ⟨identifier⟩ ⟨opt-task-parameter-dimensions⟩

(12) ⟨opt-task-parameter-dimensions⟩ →
(12.1) ⟨task-parameter-dimensions⟩
(12.2) |

(13) ⟨task-parameter-dimensions⟩ →
(13.1) ⟨task-parameter-dimensions⟩ ⟨task-parameter-dimension⟩
(13.2) | ⟨task-parameter-dimension⟩

```
#pragma css task input(N, a) output(b)
void add_one(int N, int const a[N], int b[N]) {
   for (int i = 0; i < N; i++)
      b[i] = a[i] + 1;
}
```

Listing 3.1: Task defintion example.

(14) ⟨task-parameter-dimension⟩ →
(14.1) [⟨expression⟩]

A C99 translation unit can contain calls to (3) tasks defined in the same translation unit and tasks defined externally. External tasks are declared by placing the task pragma before their function declaration (4). Local tasks are defined by placing the task pragma before their function definition (5).

The task pragma (6) consists in the **#pragma css task** text, a space separated list of clauses (8), and an end of line delimiter. The **input** (9.1), **output** (9.2), and **inout** (9.3) clauses define the kind of accesses of a task over each of its parameters. They indicate the parameters that are only read; only written; and read and written, respectively. Each parameter must appear exactly one time in one of these clauses.

The parameters in those clauses are separated by commas (10) and consist of (11) the name of the parameter and optionally (12) the dimensions of the parameter (13), which have the same syntax as in C99 (14). The optional dimensions allow to specify the dimensions of array parameters that are passed as pointers.

The **highpriority** clause (9.5) indicates that during scheduling, the task must have higher priority than other tasks without the clause.

Due to the asynchronous nature of tasks, and for simplicity, the return type of a task must be **void**. Additional parameters passed as **output** pointers can be used for the same purpose.

Listing 3.1 shows a task definition that takes an array a of N elements and sets an array b of the same size to the same values as a plus 1. Notice that since N and a are only read by the task, they appear in the **input** clause. Since b is writen and its initial value is discarded, it appears in the **output** clause.

3.3.3 Reductions

SMPSs supports task-based reductions. Task instances that perform an **inout** access over a parameter may be run in parallel if instead of including the parameter on an **inout** clause, they do it on the **reduction** clause (9.4). The syntax of the parameters of this clause is identical to the one of the directionality clauses (10). All consecutive task instances that perform a **reduction** update over the same data are not serialized due to their dependencies over that data. Instead, the run-time allows them to run in parallel.

While the compiler could be used to automatically protect the accesses to the reduction parameter, as an initial implementation, any mutual exclusion to these memory locations must be explicitly programmed by using either compiler intrinsics, external libraries or the mutual exclusion primitives provided to that effect.

```
1  #pragma css task input(BS, a, b) reduction(result)
2  void dot_product_task(int BS, double a[BS], double b[BS], double *result) {
3      double partial_value = 0.0;
4      for (int i = 0; i < N; i++)
5          partial_value += a[i] * b[i];
6
7      #pragma css lock(result)
8      *result += partial_value;
9      #pragma css unlock(result)
10 }
11
12 void dot_product(int N, double a[N], double b[N], double *result) {
13     *result = 0.0;
14     for (int j = 0; j < N; j += BS)
15         dot_product_task(BS, &a[j], &b[j], result)
16 }
```

Listing 3.2: Dot product implemented using a reduction.

The syntax of the mutual exclusion primitives is the following:

(18) ⟨lock-directive⟩ →
(18.1) **#pragma css mutex lock (** ⟨expression⟩ **)**

(19) ⟨unlock-directive⟩ →
(19.1) **#pragma css mutex unlock (** ⟨expression⟩ **)**

The first primitive (18) starts a mutual exclusion and the second one (19) terminates it. Both use a C99 integer expression to discriminate between mutual exclusions or exclusions to different data. The value of the expression can be arbitrary but must be consistent between the lock and unlock primitives and between exclusions to the same data. Both primitives may appear wherever a C99 statement may appear.

Listing 3.2 shows an implementation of the dot product using a reduction. The code calculates $result = \sum_{i=1}^{N} a_i \cdot b_i$ in tasks that perform partial dot products of BS elements at a time. That is, the algorithm is implemented as $result = \sum_{j=1}^{N/BS} \sum_{i=1}^{BS} a_{i+j \cdot BS} \cdot b_{i+j \cdot BS}$. The outer sum corresponds to the loop that starts at line 14 and is implemented as a reduction in lines 7–9. The inner partial sum is performed over a local variable and corresponds to the loop that starts at line 4. In this case the reduction part is implemented using the mutual exclusion primitives. To identify uniquely the mutual exclusion we use the address of the result.

3.3.4 Synchronization

Tasks are asynchronous with respect to the main control flow. Data read by already instantiated tasks cannot be reliably modified in the main control flow until those task instances that access that data have finished. Similarly, data written by instantiated tasks cannot be reliably read in the main control flow until those task instances have finished.

Synchronization primitives provide means to suspend the execution of the main control flow until all or some tasks have finished. Their grammar is the following:

(20)	⟨barrier-directive⟩ →
(20.1)	**#pragma css barrier**
(21)	⟨waiton-directive⟩ →
(21.1)	**#pragma css wait on (** ⟨waiton-expression-list⟩ **)**
(22)	⟨waiton-expression-list⟩ →
(22.1)	⟨conditional-expression⟩
(22.2)	| ⟨conditional-expression⟩ , ⟨waiton-expression-list⟩

The **barrier** primitive (20) stops the main control flow until all previously instantiated tasks have finished. This primitive has been used in our evaluation to reliably measure the time that it takes to instantiate and execute the tasks of the computations and to discard the time taken by the data initialization.

The **wait on** primitive (21) stops the main control flow until all task instances that access a specific set of data have finished. The data is specified as a comma separated list of addresses that point to the beginning of each target data (22). The ⟨conditional-expression⟩ used in production (22) corresponds to the same non-terminal symbol defined in the C99 standard. This primitive is useful for accessing data generated by previous tasks from within the main code.

Listing 3.3 shows the skeleton of a minimization algorithm. The algorithm initializes its data using tasks in the first loop. In this case, the implementation also measures the time that it takes to execute the minimization. The barrier in line 6 guarantees that the timer is started in line 7 after the initialization tasks have finished. Similarly the **barrier** in line 19 guarantees that the timer is stoped in line 20 after all the previous tasks have finished.

The code skeleton assumes that target_variable stores the result of the minimization. The minimize function is an arbitrarily complex function that uses tasks to perform the minimization. These update the values stored in target_variable to reflect the minimization status at that point. To check if the minimization has finished, the code must access the contents of target_variable, and since the main control flow depends on its contents, the access must be performed after the variable has received its last value. The **wait on** primitive in line 13 guarantees that the following line can access the variable in isolation and that it will contain its latest updated value.

3.3.5 Compiler and Runtime Integration

To implement the language, the programming environment has a compiler and a runtime library. The role of the compiler is to transform the program code into calls to the runtime, which in turn is responsible of handling all of the languages features. Appendix B.2 that starts in page 195 contains a detailed description of the internal programming interface of the runtime, the correspondence between that and the language and the functioning of the compiler.

```
void minimization_algorithm(int N, int BS, ...) {
   for (int i = 0; i < N; i += BS) {
      initialization_task(BS, ...)
   }

   #pragma css barrier
   start_timer();

   bool finised = false;
   while (!finished) {
      minimize(..., target_variable);

      #pragma wait on(target_variable)
      if (have_finished(target_variable)) {
         finished = true;
      }
   }

   #pragma css barrier
   end_timer();
}
```

Listing 3.3: Minimization algorithm skeleton.

3.4 Runtime

The runtime is where most of the programming model functionality resides. The transformed code calls its functions to perform the task instantiation and synchronization, and the runtime calls the task code to perform the actual task execution.

The runtime is responsible of the task instantiation, the dependency calculation, the scheduling, the actual task execution and the synchronization.

3.4.1 Dependency Analysis

The task instantiation order and their accesses define a partial order that is determined by the creation and consumption of data definitions.

A task instance that writes to a memory location creates a new definition. A following task instance in sequential order that reads that memory location consumes that definition. This relation is a data dependency relationship. A task instance that reads a memory location followed by another that writes to that memory location also establishes a dependency relationship, since the read must be performed before the write to allow it to access the correct definition. Two tasks instances that write to the same memory location also have a dependency relationship, since to preserve sequential semantics, further task instances that read the memory location need to read the correct definition. These dependency relationships are called Read-after-Write (RaW), Write-after-Read (WaR) and Write-after-Write (WaW) respectively.

To detect dependencies, the runtime keeps a map of which task instance produces the last definition of a memory position at a given time and which task instances consume it. For simplicity, it is assumed that the program only accesses contiguous and non-overlapping segments of memory. These restrictions are relaxed in chapter 4. Under those conditions, the runtime can keep track of the producer and consumers of the last definition of each segment of memory by indexing it by its base address. This data structure can be implemented using any indexing strategy. For instance, it could be implemented using a search tree indexed by the base address of the memory segment and containing the producer and consumers of the last definition of the segment.

Given a task instance that reads a segment of memory with the above restrictions, a RaW dependency can be detected by finding the last writer to that memory segment. A further task instance that writes to that segment will have a WaR dependency over it. To enable that check in the future, the runtime adds this task instance to the list of readers of the last definition of that memory segment.

Given a task instance that writes to a memory segment with the above restrictions, WaR dependencies can be detected by finding the readers of the last definition of that memory segment, and a WaW dependency can be detected by finding the writer of the last definition of that memory segment. To enable detecting further dependencies against this task, and since the task creates a new definition, the memory segment has its last writer set to this task, and the list of readers cleared.

Combined read and write accesses can be handled similarly to a read access followed by a write access. The only difference lies in avoiding adding the task as a reader of the memory segment, to prevent creating a self-dependency.

Limitations

This dependency analysis strategy has two main limitations. First, it is restricted to contiguous and non-overlapping memory segments. And second, it has unnecessary dependencies. This thesis covers the first limitation in chapter 4, and the second in the following section.

3.4.2 Data Renaming

RaW dependencies are also called *true dependencies*, because they cannot be avoided without resorting to speculation. In contrast, WaW and WaR dependencies are called *false dependencies* because they can be removed without resorting to speculation.

False dependencies occur between accesses to different definitions of a common memory location. By moving these definitions to different memory locations, these dependencies disappear. A write followed by another write to the same memory location does not produce a dependency when the second write is moved to a different location. Similarly, a read followed by a write to the same memory location does not produce a dependency when the write is moved to a different location.

To implement renaming, the runtime must be made aware of data definitions. These may either reside in their original memory segment or may need to be allocated in a different memory location.

To execute a task, the runtime must be able to map the definitions of its parameters to their actual memory addresses. Parameters that are only written, may have their definitions stored in their original locations if these do not already hold another live definition. Otherwise, the runtime needs to allocate new space for the definition.

Parameters that are only read must point to the memory used by the task that created their definition. Parameters that are read and written have two definitions: a definition they consume, and a definition they create. However, since the task implementation expects a parameter to only have one base address, the runtime must use a single base memory location. If the task is the only reader of the read definition, its memory can be reused for the write definition. Otherwise, a new location must be used for the write definition. In the latter case the runtime has to copy the read definition data over to the write definition location and use it as the storage of the parameter.

Definitions become dead when they no longer have any pending accesses and they correspond to memory segments that have younger definitions. As they become dead, the runtime deallocates their memory.

Synchronization primitives force the last definition of a memory segment to be copied back to its original memory location. This effectively makes the original memory segment the storage of the last definition. While barriers affect all the last definitions, the “wait on” primitives affect only the last definition of the memory segments that appear in their “on” clause.

3.4.3 Reductions

The scope of reductions is determined by the accesses to the target memory segment of the reduction. A reduction is started by a task instance that performs a reduction access over a memory segment. It spans all following task instances that perform the same access and is finished by a synchronization or by the first task instance that performs a different access over the same memory segment. As defined in section 3.3.3, task instances that participate in the same reduction do not have dependencies between them due to the reduction access.

Whenever a reduction is initiated, the runtime performs the same actions as in the inout case. However, instead of setting the task instance as the writer of the new definition, it adds the task instance as a “parallel updater” and sets a mark in the definition that indicates that a reduction has been started. Further task instances that perform a reduction over the same memory segment, recover the read definition for finding their RaW dependencies, and add themselves to the list of parallel updaters of the last definition.

The locking primitives accept arbitrary numbers to identify memory segments. Internally they map the identifiers to mutexes. These can be implemented using standard locking libraries, platform intrinsics, or atomic operation compiler extensions. The identifier mapping can be performed using hashing to a preallocated set of mutexes, or using a dynamic data structure to allow allocating and deallocating them on demand.

This thesis evaluates an implementation that uses a static hash table with preallocated mutexes.

3.4.4 Synchronization

Synchronization primitives perform two actions: they wait until a set of tasks finishes and they copy back the last definition of a set of memory segments to its original location. Barriers wait all tasks in the system and copy back all the last definitions of all memory segments that have been created.

Partial barriers wait for the task instances that access the last definition of each memory segment specified in the **on** clause, and the task instances that access their original memory location. Then they copy back the data from the last definition, to their original memory location.

3.4.5 Scheduling

One of the main goals when scheduling under SMPSs is to exploit data locality. In that regard the scheduler takes advantage of the graph information to schedule dependent tasks sequentially to the same core so that output data is reused immediately.

The scheduler has two main ready lists, one for high priority tasks and one for normal priority tasks. The tasks of the high priority list get scheduled by the worker threads as soon as they become idle, and independently of any locality consideration. The tasks of the normal priority list are scheduled only when the high priority queue is empty. Whenever the main thread instantiates a task without any input dependency, it moves it into the main ready list or the high priority list where it can be scheduled by the worker threads.

Each worker thread has its own ready list that contains tasks whose last input dependency has been removed by that thread. Whenever a thread has finished running a task, it removes it from the graph and moves all tasks that have become ready to its ready list.

Threads look for ready tasks first in the high priority list. If it is empty, then they look for them in their own ready list. If they do not succeed, they proceed to check out the main normal priority list. In case of failure, they proceed to steal work from other threads in creation order starting from the next one.

Threads consume tasks from their own list in LIFO order, they get tasks from the main list in FIFO order, and they steal from other threads in FIFO order. This policy allows them to consume the graph in a pseudo-depth-first order as long as they can can get ready tasks, and to perform task stealing in a pseudo-breadth-first order. The Cilk scheduler by [Frigo et al., 1998] has a very similar policy. To preserve some locality of reference in the presence of task stealing, whenever a thread finishes running a task, it protects against stealing one of the tasks that it liberates.

Since renaming removes all the false dependencies and only leaves true dependencies, all the predecessors of a task are always the generators of its input data. By executing tasks in depth-first order, the scheduling algorithm favors running tasks in the threads that have just generated one of its input parameters.

This policy also favors keeping each thread on a different region of the graph and thus to keep them accessing the same data and consequently has the potential to produce lower cache coherency overhead. As long as a thread can find ready tasks in the zone that it is exploring (thread ready list), or there are unexplored zones in the graph (main ready list), it will not steal tasks and thus it will keep the working-sets independent. Work-stealing in FIFO order tries to minimize the effect on the cache of the victim thread by choosing the task that has spent most time on the queue and thus that has more probability of having most of its input data already evicted from cache.

The main ready list is a point of distribution of tasks in areas of the graph that are not being explored. Tasks that are free when instantiated are inserted into one of the main ready lists according to their priority.

The main thread also contributes to run tasks. Whenever it reaches a blocking condition (a synchronization point, a memory limit, or a graph size limit), it behaves as a worker thread until an unblocking condition is reached.

3.5 Debugability

The ability to inspect the behavior of the code of an application is an important aspect towards productivity. In this sense, the model facilitates debugging through the design of the language and through runtime features.

3.5.1 Language Transparency

The primitives that compose the language are based on C pragmas. Due to the design of the language, a code with the pragmas removed is a valid sequential version of the code. Since in C, compilers that do not implement some pragmas are allowed to ignore them, it is possible to obtain a regular sequential executable by compiling the code with a regular C compiler. Sequential versions can then be used with conventional debugging tools to resolve potential issues not related to the parallelization.

This approach is also used in OpenMP, although in OpenMP it is slightly violated by some runtime functions that are available directly to the user.

3.5.2 Integrated Tracing

The runtime implemented in this thesis has integrated tracing capabilities. For performance reasons, the runtime has a version without tracing, one with tracing and another with tracing and the possibility of gathering hardware counters. The version used by an application is specified by a flag when invoking the linker.

Tracing records events during the execution of an application and generates a file with those events, their time of occurrence, and the thread in which they occurred. This file is in Paraver format and can be analyzed using the tool with the same name by [Labarta et al., 1996].

Traces contain information about internal aspects of the runtime, and aspects directly related to the application. Some of them are:

- Execution of a task instance
  - Time interval of the execution
  - Type of task
  - Task instantiation order
- Intervals of idleness
- Intervals of runtime execution

With the aid of the Paraver tool, this information allows to measure and visually inspect among others the following metrics:

- Task granularity
- Runtime overhead
- Unbalance
- Task instance reordering

- Amount of parallelism

These metrics help in diagnosing performance problems due to the parallelization. For instance they allow detecting lack of parallelism and suboptimal task granularity.

Additionally, traces may also contain hardware counter information. Some of the most useful hardware counters may measure:

- Instructions Per Cycle (IPC)
- Cache access information
- Memory access information
- Number of integer instructions of a task instance
- Number of floating point operations of a task instance
- Instruction mix of a task instance
- Other information related to the functional units (e.i. branch information)
- Actual cycles spent running a task instance

Hardware counter metrics are useful for analyzing task performance. They can be inspected visually along the time line. They can also be compared between different task types. Moreover, Paraver allows to find correlations between the metrics.

The integrated tracing system has been used to obtain detailed measurements in this thesis. These help understand the performance of the applications, their characteristics, and the effects of the scheduling policies.

3.6 Evaluation

This section evaluates the programming model and its implementation in SMP superscalar. The evaluation is centered on the main aspects: programmability and performance. It consists of a selection of 5 algorithms that have been implemented in SMP superscalar, other programming models and, in some cases, preexisting high performance parallel libraries. In addition to an evaluation of the programming model, we also evaluate the scheduling policy by comparing it to three other policies.

Hardware Configuration

The performance measurements have been obtained on an SGI Altix 4700 computer with 128 cores and 512 GB of memory. It is composed of 32 NUMA nodes, each with 2 dual core 1.6 GHz Itanium2 processors and local memory. The nodes are interconnected through a ccNUMA link. The measurements have been performed inside a cpuset of 32 cores on 8 nodes with all memory pages bound to those nodes.

Figure 3.1 shows the scheme of a NUMA node. It consists of 2 dual core processors that share a 128-bit wide front side bus (FSB) that connects them to an SGI Super-Hub ASIC (SHub). The FSB operates at 533 MHz and is capable of transferring 8.5 GB/s. The SHub acts as a crossbar that connects the FSB the local memory and to the other NUMA nodes through 2 NUMAlink 4 interfaces. The local memory bus has 4 channels and is capable of transferring 17 GB/s. Each NUMAlink 4 port interface is capable of transferring 3.2 GB/s in each direction. The nodes are interconnected by those interfaces by a network of router ASICs.

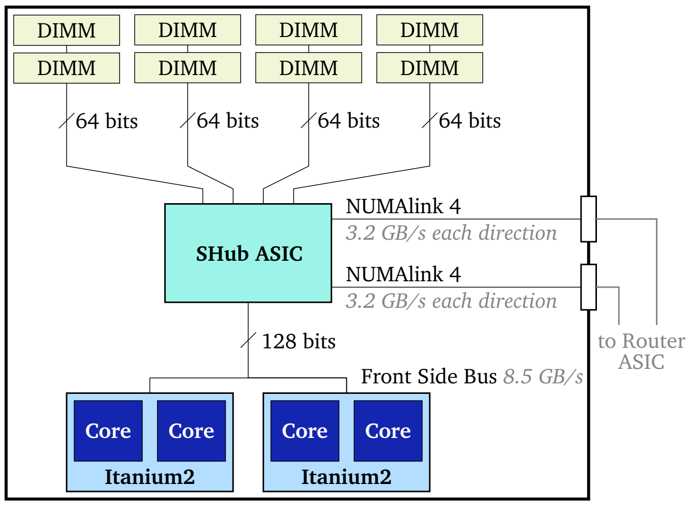


Figure 3.1: Hardware configuration of a node of the experimental platform.

Compilers and Libraries

The codes have been compiled using GCC 4.6.1 as the native compiler for SMP superscalar and for the OpenMP versions. In some specific cases where GCC did not produce efficient code we have used ICC 11.0 instead. The implementations of the linear algebra algorithms use BLAS by [Dongarra et al., 1990]. The SMPSs and the OpenMP versions use it in sequential mode for small calculations performed in the tasks, and the parallel BLAS implementations use it in parallel mode to solve whole algorithms. In all cases we have used MKL BLAS version 10.1.1.019.

For the OpenMP versions, either with the GCC or the ICC compiler, we always use the Intel ICC OpenMP runtime, which implements both its own OpenMP runtime API and the GCC OpenMP runtime API.

Scheduling Policies

To verify the effectiveness of the scheduling policy, we compare it against three other policies. The first one randomizes the order of task execution. Instead of having each thread free and consume tasks in LIFO order, the random policy places each liberated task in a random thread queue at a random position. Task consumption remains identical to the original policy. In the figures of this section we label it as *random*. By comparing to this policy we can verify if the standard policy is better than the mean produced by random orders.

In addition to the random policy, we have included a variant that also uses the task protection mechanism of the original scheduler that is described in section 3.4.5 (page 27). Whenever a thread liberates tasks as a result of having executed their last predecessor, this scheduler keeps one of them protected. Instead of adding it to a random thread queue, the thread keeps it for itself and executes it next. In the figures we label it as *random + PT* (random with Protected Task). By comparing it to the standard policy we can verify if the task protection mechanism is enough to generate schedules that perform better than random schedules.

Finally we include a FIFO policy. Similarly to the standard policy, it also uses one queue per thread. However, instead of consuming the last queued task, threads always consume the first queued task of their own queue, the global queue and other thread queues. This policy allows threads to keep running tasks of a certain part of the graph similarly to the standard policy, but may be less effective at exploiting locality.

Measurement Methodology

Performance in SMPSs is essentially determined by individual task performance, parallelism, and runtime overhead. Since all these aspects depend on the problem size, the number of threads and the task granularity, we have measured the benchmarks with several configurations of these parameters.

Each configuration has been executed 30 times, and the values in the summary tables indicate their means. To illustrate the variability of the metrics, we also show violin plots of the metrics. They show the probability density of the metric at different values.

Since the hardware has Non-Uniform Memory Access (NUMA) characteristics, data placement has an important effect on the performance that we can achieve. In chapter 5 we analyze those subjects. However, in this chapter we try to homogenize NUMA effects by allocating memory pages in round robin order across the local memories of the cores used in each experiment. That is, CPUs are assigned sequentially to the NUMA nodes one at a time until filled, and the experiments have their data page-wise interleaved across the memories of the nodes on which the CPUs have been assigned.

The main performance metrics have been obtained by running each benchmark without instrumentation. However, we also analyze the finer grained information to help us explain the reasons that lead to such performance. These aspects have been obtained by running the applications a second time with instrumentation. On SMPSs, we obtain these metrics by using the tracing mechanisms integrated in the framework. On OpenMP programs and libraries, we have obtained them by writing a preloaded library that intercepts the ICC and GNU internal OpenMP runtime API. The library keeps track of the state of the threads and measures performance metrics by reading hardware counters when needed.

We have tried to keep the comparison between programming models as fair as possible. For this reason all codes are written in C and share a common implementation of the computational kernel that has been compiled using the same compiler. In cases where the kernel was external, we have used either the same exact library in all cases, or a sequential variant and parallelized variant of the same library and version number.

The purpose of this approach is to isolate the experiments from the effects of compiler maturity and the ability of the programming language to be optimized. Instead we try to measure only the effects that each programming model have over the parallelization. However, it is not always possible to isolate the measurements from the actual performance of the kernels. In fact in some cases it not. In particular in some codes SMPSs achieves more parallelism that the rest. This allows us to decompose the problems into bigger tasks, which potentially perform better due to better cache utilization.

```
1 #pragma css task input(size, b, c, alpha) output(a)
2 void triad(long size, double a[size], double b[size], double c[size], double
     alpha) {
3    for (long i = 0; i < size; i++)
4        a[i] = b[i] + alpha*c[i];
5 }
```

Listing 3.4: Triad task code.

Set of Benchmarks

In this chapter we have selected a set of 5 benchmarks to demonstrate the effectiveness of the programming model. The first two are the STREAM triad and the matrix multiplication. While those algorithms do not exploit de potential of the programming model, they serve us as simple examples to show the syntax. However, for brevity we describe their implementation in SMPSs here and leave the evaluation of their performance to appendix C.

The following subsections describe each benchmark, how it has been implemented, how it performs, and compares it to other implementations. In addition we demonstrate that the overhead of data dependency analysis in those cases does not incur in significant overhead compared to other programming models.

3.6.1 Triad

The STREAM benchmark by [McCalpin, 1995] is a synthetic benchmark that measures sustainable memory bandwidth and the computation rate for simple vector kernels. In particular, the triad kernel has been chosen by [Luszczek et al., 2006] to measure the memory bandwidth of supercomputers in the HPC Challenge benchmark suite.

Algorithm

This benchmark measures the memory bandwidth and double precision floating point computation rate for the following operation over a constant α and three arrays named a, b and c:

$$a \leftarrow b + \alpha c$$

Parallelization with SMPSs

Since the kernel is embarrassingly parallel, to parallelize it we can tile the operation over N elements into chunks of BS elements and divide the data according to those. Listings 3.5 and 3.4 show the main code and the triad task respectively.

The main code in figure 3.5 performs and measures the whole vector operation ten times and reports the memory bandwidth of the fastest repetition. To include the cost of the runtime in the measurement, each repetition is isolated by barriers and the time measurement spans from the start of task creation to the end of the barrier at the end of the repetition.

Line 1 of figure 3.4 indicates that triad is a task; that it receives as inputs, the size of the subvectors (size), the subvectors b and c, and the constant alpha; and that it writes to subvector a. Note that we specify the size of the subvectors in their declaration in line 2.

```
for (int rep = 0; rep < NTIMES; rep++) {
    START_TIME();
    for (long i = 0; i < N; i += BS)
        triad(BS, &a[i], &b[i], &c[i], alpha);
    #pragma css barrier
    STOP_TIME();
    total_time[rep] = GET_TIME();
}
find_best_bandwidth(total_time, NTIMES);
```

Listing 3.5: Triad main code.

Appendix C shows a detailed analysis of the performance of this algorithm under the SMPSs model and alternative implementations.

3.6.2 Matrix multiplication

Algorithm

The matrix multiplication algorithm we evaluate is the standard matrix multiplication for double precision floating point numbers that given two matrices A and B of $M \times K$ and $K \times N$ elements respectively, calculates a third matrix $C = A \times B$ of $M \times N$ elements. Each element C_{ij} of the result can be calculated as follows:

$$C_{ij} = \sum_{k=1}^{K} A_{ik} B_{kj} \tag{3.1}$$

To make the code more reusable, the implementation calculates the generalized matrix-matrix multiplication as specified by the dgemm function from BLAS. This form calculates $C' = \alpha A \times B + \beta C$, where α and β are floating point numbers. The operation stores the result C' in the original location of C.

Parallelization with SMPSs

The standard matrix multiplication can be parallelized by dividing the matrices logically and by decomposing the operation into smaller operations over the resulting submatrices. The following equations show two possible decompositions:

$$\begin{aligned} C &= \left[\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right] \\ &= \left[\begin{array}{c|c} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ \hline A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{array}\right] \\ &= \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right] \times \left[\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right] = A \times B \end{aligned} \tag{3.2}$$

$$\begin{aligned}
C &= \left[\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right] \\
&= \left[\begin{array}{c|c} A_{1*}B_{*1} & A_{1*}B_{*2} \\ \hline A_{2*}B_{*1} & A_{2*}B_{*2} \end{array}\right] \\
&= \left[\begin{array}{c} A_{1*} \\ \hline A_{2*} \end{array}\right] \times \left[\begin{array}{c|c} B_{*1} & B_{*2} \end{array}\right] = A \times B
\end{aligned} \tag{3.3}$$

The first is a decomposition by blocks, and the second by panels, where A_{i*} is the i-th horizontal panel of A, and B_{*j} is the j-th vertical panel of B.

We have implemented the SMPSs version by decomposing the multiplication by blocks of a fixed number of elements per dimension (NBS, MBS and KBS for the N, M and K dimensions respectively). Since the programming model only supports whole arrays of non-overlapping contiguous subarrays, the matrices are stored in memory in blocks that correspond to each submatrix. Further discussion about how to remove this limitation is presented in the chapter 4.

Since we implement the generalized form of the algorithm, our decomposition differs slightly from equation 3.2. If we apply the decomposition to the generalized form we obtain:

$$\begin{aligned}
C' &= \alpha A \times B + \beta C \\
&= \alpha \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right] \times \left[\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right] + \beta \left[\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right] \\
&= \left[\begin{array}{c|c} \alpha A_{11}B_{11} + \alpha A_{12}B_{21} + \beta C_{11} & \alpha A_{11}B_{12} + \alpha A_{12}B_{22} + \beta C_{12} \\ \hline \alpha A_{21}B_{11} + \alpha A_{22}B_{21} + \beta C_{21} & \alpha A_{21}B_{12} + \alpha A_{22}B_{22} + \beta C_{22} \end{array}\right] \\
&= \left[\begin{array}{c|c} ((\beta C_{11}) + \alpha A_{11}B_{11}) + \alpha A_{12}B_{21} & ((\beta C_{12}) + \alpha A_{11}B_{12}) + \alpha A_{12}B_{22} \\ \hline ((\beta C_{21}) + \alpha A_{21}B_{11}) + \alpha A_{22}B_{21} & ((\beta C_{22}) + \alpha A_{21}B_{12}) + \alpha A_{22}B_{22} \end{array}\right]
\end{aligned} \tag{3.4}$$

Thus, each submatix C'_{ij} can be calculated as follows:

$$\begin{aligned}
C'_{ij} &= \beta C_{ij} + \alpha \sum_{k=1}^{K} A_{ik}B_{kj} \\
&= \beta C_{ij} + \alpha A_{i1}B_{1j} + \alpha A_{i2}B_{2j} + \ldots + \alpha A_{iK}B_{Kj} \\
&= 1(\ldots 1(\beta(C_{ij}) + \alpha A_{i1}B_{1j}) + \alpha A_{i2}B_{2j} \ldots) + \alpha A_{iK}B_{Kj}
\end{aligned} \tag{3.5}$$

which gives us a common operation (kernel) that is applied recursively and has the following form:

$$c'_{ijk} = \mu_k c'_{ij(k-1)} + \alpha A_{ik}B_{kj} \tag{3.6}$$

where

$$\begin{aligned}
\mu_k &= \begin{cases} \beta & \text{if } k = 1 \\ 1 & \text{otherwise} \end{cases} \\
c'_{ij0} &= C_{ij} \\
c'_{ijK} &= C'_{ij}
\end{aligned} \tag{3.7}$$

The SMPSs generalized matrix multiplication is implemented by calculating each submatrix of C' using the kernel from equation 3.6. Listing 3.6 shows the main code.

```
void tiled_dgemm(
   int MBS, int NBS, int KBS,
   int M, int N, int K,
   double ALPHA,
   double const A[M/MBS][K/KBS][MBS][KBS],
   double const B[K/KBS][N/NBS][KBS][NBS],
   double BETA, double C[M/MBS][N/NBS][MBS][NBS])
{
   for (int i=0; i<M/MBS; i ++)
      for (int j=0; j<N/NBS; j++) {
         if (BETA == 0.0)
            dgemm_nobeta_tile(MBS, NBS, KBS, ALPHA, A[i][0], B[0][j],
              C[i][j]);
         else
            dgemm_tile(MBS, NBS, KBS, ALPHA, A[i][0], B[0][j], BETA,
              C[i][j]);
      }
   for (int k=1; k<K/KBS; k++)
      for (int i=0; i<M/MBS; i ++)
         for (int j=0; j<N/NBS; j++)
            dgemm_tile(MBS, NBS, KBS, ALPHA, A[i][k], B[k][j], 1.0, C[i][j]);
}

#pragma css task input(MBS, NBS, KBS, ALPHA, A, B) output(C)
void dgemm_nobeta_tile(int MBS, int NBS, int KBS,
   double ALPHA, double const A[MBS][KBS], double const B[KBS][NBS],
   double C[MBS][NBS])
{
   static const double dzero = 0.0;
   dgemm_("N", "N", &NBS, &MBS, &KBS, &ALPHA, B, &NBS, A, &KBS,
     &dzero, C, &NBS);
}

#pragma css task input(MBS, NBS, KBS, ALPHA, BETA, A, B) inout(C)
void dgemm_tile(int MBS, int NBS, int KBS,
   double ALPHA, double const A[MBS][KBS], double const B[KBS][NBS],
   double BETA, double C[MBS][NBS])
{
   dgemm_("N", "N", &NBS, &MBS, &KBS, &ALPHA, B, &NBS, A, &KBS,
     &BETA, C, &NBS);
}
```

Listing 3.6: Double precision generalized matrix-matrix multiplication in SMPSs.

```
for (int iter=0; iter < L; iter++)
  for (long i=1; i <= N; i++)
    for (long j=1; j <= N; j++)
      A[i][j] = 0.2 * (A[i][j] + A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1]);
```

Listing 3.7: Sequential implementation of the Gauss-Seidel algorithm for the 2D heat transfer problem.

The tiled_dgemm function implements the main algorithm. The two nested loops in lines 9 and 10 traverse dimensions M and N in steps of MBS and NBS elements respectively and use the kernel to compute the values of c'_{ij1}. The three nested loops in lines 16, 17 and 18 calculate $c'_{ij2}, \ldots, C'_{ij}$. The intermediate values of c'_{ijk} and the result C'_{ij} are stored in the same location as C_{ij}.

The kernel is implemented in tasks dgemm_nobeta_tile and dgemm_tile. They take as **input** the dimensions of each submatrix, the α and β parameters, and the submatrix of A and B. Since the value of C_{ij}, the intermediate values, and the result share the same memory location, the directionality of the C parameter for $k = 1$ is **inout** when $\beta \neq 0$ and **output** when $\beta = 0$. The dgemm_tile task implements the first case and the dgemm_nobeta_tile task implements the second.

Appendix C shows a detailed analysis of the performance of this algorithm under the SMPSs model and alternative implementations.

3.6.3 Gauss-Seidel 2D Heat Transfer

Algorithm

Stencil algorithms are a class of iterative algorithms that consist in updating the elements of an array according to a pattern. They perform a sequence of sweeps through the data, which is an array of a certain number of dimensions that represents a regular grid. The stencil determines for each element, the surrounding elements that are used to update its value.

The Gauss-Seidel method is an algorithm for determining the solutions of a system of linear equations with largest absolute values in each row and column dominated by the diagonal element. When the method is applied to the following 2D heat-transfer problem

$$\nabla^2 T(x, y) = 0, x \in [0, 1], y \in [0, 1] \tag{3.8}$$

the system of equations can be solved using a 5 point 2D stencil algorithm. Listing 3.7 shows a sequential implementation of the algorithm.

The Jacobi method is a similar algorithm that serves the same purpose, however it converges more slowly. One notable difference between them is that while at each iteration the Gauss-Seidel method uses the results of the current and the previous iteration, the Jacobi method only uses the results of the last iteration. This difference makes the Jacobi method easier to parallelize, since the calculations within an iteration are independent between themselves. In contrast, the Gauss-Seidel method has dependencies within one iteration. For instance, the accesses to A[i−1][j] and A[i][j−1] for $i \neq 1$ and $j \neq 1$ correspond to values updated during the same iteration of the outermost loop.

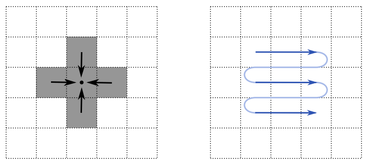


Figure 3.2: Shape of the stencil and order of the updates.

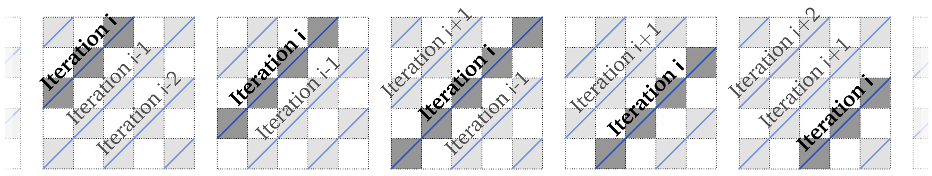


Figure 3.3: Relation between the iteration numbers and the wavefronts that can be updated in parallel.

Figure 3.2 shows the layout of the stencil on the left and the update order on the right. The order of the updates and the shape of the stencil allow parallelism in the form of wavefronts. Figure 3.3 shows the progression of the wavefronts over a small matrix. The elements of a each diagonal are independent and thus can be updated in parallel. Moreover, the iterations of the outer loop can be pipelined in the form shown in the figure to allow several wavefront updates in parallel.

Wavefront parallelism is a recurrent topic in the research community. Some of the most frequently covered aspects include the scheduling of the wavefronts and the effective use of the memory hierarchy (see [Hoisie et al., Winter 2000] and [Kerbyson et al., 2011]).

Parallelization with SMPSs

The Gauss-Seidel algorithm, and in particular the case exposed in listing 3.7, is defined in terms of element-wise operations. Since the sequential code consists of a series of loops that traverse the data in a regular manner, we can tile these accesses and derive a blocked version from it. Listing 3.8 shows a tiled version with tiles of size BS×BS elements. Notice that the loop that iterates over ii has been moved inside to the loop that iterates over j.

Transforming the tiled version into a parallel version is almost straightforward.

```
for (int iter=0; iter < L; iter++)
   for (long i=1; i <= N; i+=BS)
      for (long j=1; j <= N; j+=BS)
         for (long ii=0; ii < BS; ii++)
            for (long jj=0; jj < BS; jj++)
               A[i+ii][j+jj] = 0.2 * (A[i+ii][j+jj] + A[i+ii-1][j+jj] +
                 A[i+ii+1][j+jj] + A[i+ii][j+jj-1] + A[i+ii][j+jj+1]);
```

Listing 3.8: Tiled version of the sequential implementation of the Gauss-Seidel algorithm for the 2D heat transfer problem.

```
#pragma css task input(N, BS, top, left, bottom, right) inout(A)
void gs_tile(long N, long BS, double A[BS][BS], double top[BS][BS], double
  left[BS][BS], double bottom[BS][BS], double right[BS][BS]) {
  for (long i=0; i < BS; i++)
    for (long j=0; j < BS; j++)
      A[i][j] = 0.2 * ( A[i][j]
        + (i > 0L ? A[i-1][j] : top[BS-1][j])
        + (i < BS-1L ? A[i+1][j] : bottom[0][j])
        + (j > 0L ? A[i][j-1] : left[i][BS-1])
        + (j < BS-1L ? A[i][j+1] : right[i][0]) );
}

void gauss_seidel(long N, long BS, double A[N/BS][N/BS][BS][BS]) {
  for (int iters=0; iters<L; iters++)
    for (long i=1; i < N/BS-1; i++)
      for (long j=1; j < N/BS-1; j++)
        gs_tile( N, BS, A[i][j], A[i-1][j], A[i][j-1], A[i+1][j], A[i][j+1] );
}
```

Listing 3.9: SMPSs version of the Gauss-Seidel algorithm for the 2D heat transfer problem.

First, we transform the matrix into a blocked matrix or a hypermatrix. Then we move the loops that traverse the tile into its own function, and finally we convert that function into a task. Listing 3.9 shows the resulting code. Notice that due to the conversion of the data layout from a flat matrix into a matrix by blocks, the call to the outlined function and the expression that calculates the updated value are more complex. These limitations are eliminated in chapter 4.

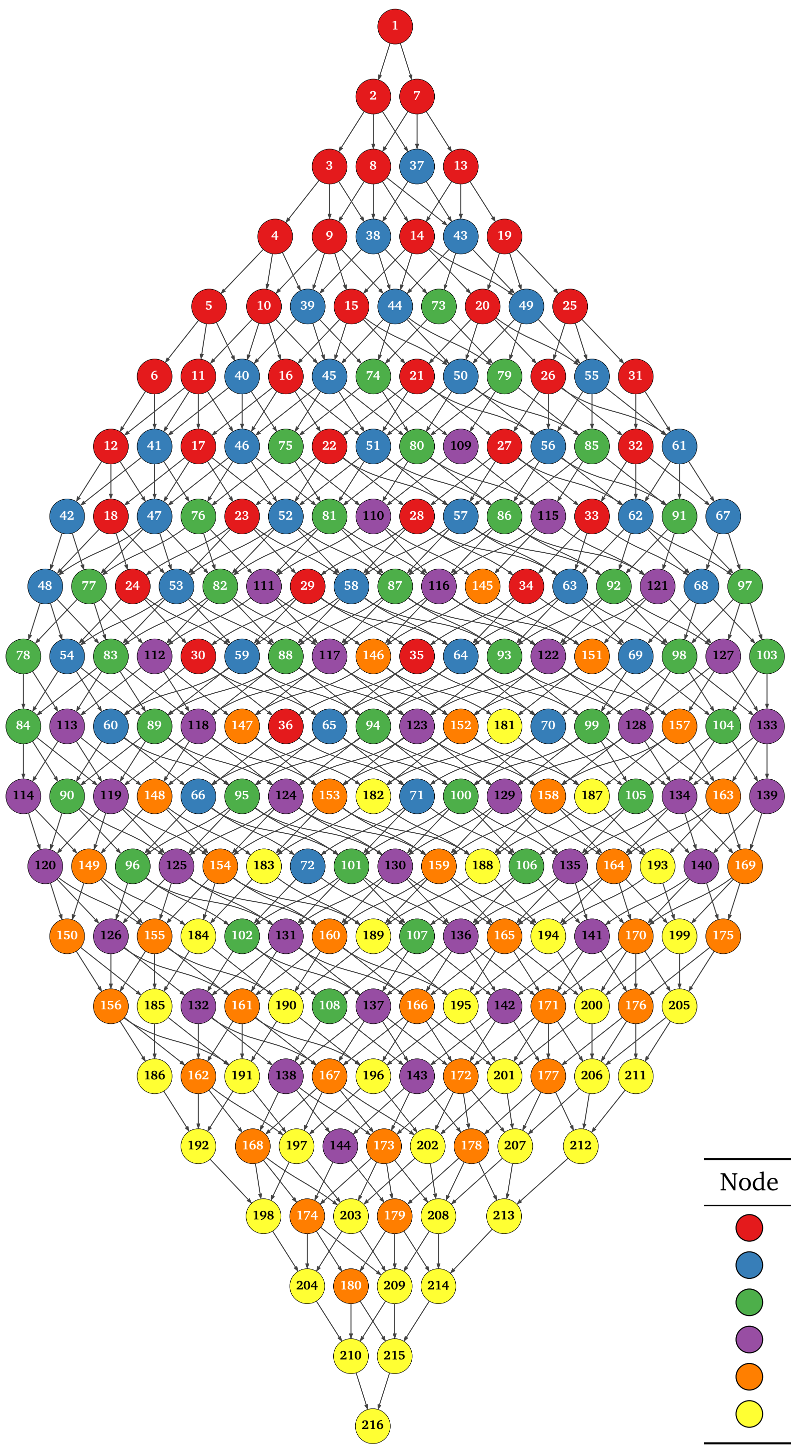
The SMPSs programming model is able to exploit wavefront parallelism in a generic manner, while the solutions found in the literature are specifically tailored for that kind of parallelism. Figure 3.4 shows the task graph that our implementation generates when the data is divided in 6 by 6 blocks (BS = N/6) and we iterate over it 6 times (L=6). Each node represents a task instance and has number that corresponds to its instantiation order. To illustrate the ability to automatically exploit wavefront parallelism, the node colors represent the outer loop iteration of each task instance. Notice that the central row shows that for this problem size, there is parallelism between tasks of each iteration of the outer loop.

In our tests we have found that the Intel compiler (ICC) optimizes the gs_tile task much better than GCC. For this reason we have used ICC in all the implementations of this algorithm. Unless stated otherwise, the measurements have been performed with 16 outer loop iterations.

Parallelization with OpenMP

To compare the performance of the algorithm in SMPSs to other approaches we have made an implementation in OpenMP 3 with tasks. Since OpenMP 3 does not provide means to parallelize with dependencies, the OpenMP version is written to follow the parallelism of a single wavefront. Listing 3.10 contains the code.

Notice that to achieve parallelism the matrix traversal has had to be split into two



Node	Iteration
🔴	1
🔵	2
🟢	3
🟣	4
🟠	5
🟡	6

Figure 3.4: Graph of 6 iterations of then SMPSs Gauss-Seidel with data divided in 6 by 6 blocks showing the relation between the task instantiation order, the outer loop iteration and the parallelism.

```
void gs_tile(long N, long BS, double A[BS][BS], double top[BS][BS], double
  left[BS][BS], double bottom[BS][BS], double right[BS][BS]) {
  for (long i=0; i < BS; i++)
    for (long j=0; j < BS; j++)
      A[i][j] = 0.2 * ( A[i][j]
        + (i > 0L ? A[i−1][j] : top[BS−1][j])
        + (i < BS−1L ? A[i+1][j] : bottom[0][j])
        + (j > 0L ? A[i][j−1] : left[i][BS−1])
        + (j < BS−1L ? A[i][j+1] : right[i][0]) );
}

void gauss_seidel(long N, long BS, double A[N/BS][N/BS][BS][BS]) {
  for (int iters=0; iters<L; iters++) {
    for (long i=1; i < N/BS−1; i++)
      #pragma omp parallel for
      for (long j=1; j <= i; j++)
        #pragma omp task untied
        gs_tile( N, BS, A[i][j], A[i−1][j], A[i][j−1], A[i+1][j], A[i][j+1] );

    for (long i=1; i < N/BS−1; i++)
      #pragma omp parallel for
      for (long j=1+1; j < N/BS−i; j++)
        #pragma omp task untied
        gs_tile( N, BS, A[i][j], A[i−1][j], A[i][j−1], A[i+1][j], A[i][j+1] );
  }
}
```

Listing 3.10: Tiled OpenMP version of the Gauss-Seidel algorithm for the 2D heat transfer problem.

doubly nested loops that traverse the matrix in inverse diagonal direction. These kind of transformations are not necessary in the SMPSs version. Moreover, the OpenMP version does not exploit the existing parallelism between wavefronts. To do so, the code would need further complex changes. On the other hand, it can generate the tasks in parallel, whereas the SMPSs version cannot.

While we can construct an OpenMP version using a flat data layout format, the performance of the physically blocked layout is significantly better. To make the comparison more reliable we have used the blocked data layout.

Determining the Tile Dimensions

The tile size of this algorithm size determines the computational cost of each task, the amount of data that the task traverses, and the potential number of concurrent waveforms. Figure 3.5 shows the performance of the SMPSs implementation with several problem sizes and decomposition granularities. Each panel corresponds to a set of measurements with a fixed number of cores. The vertical axis determines the matrix side size, and the horizontal axis is the block side size BS. The figure shows the mean performance of 30 executions of each configuration measured as the number of updates per second per core.

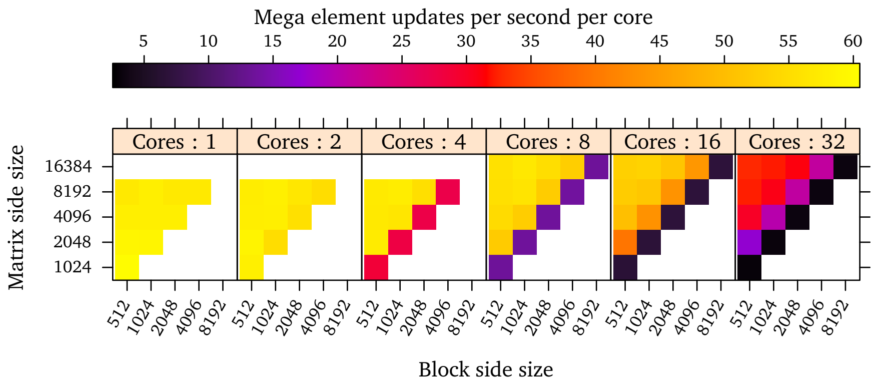


Figure 3.5: Performance of the SMPSs Gauss-Seidel implementation with several matrix and blocking sizes.

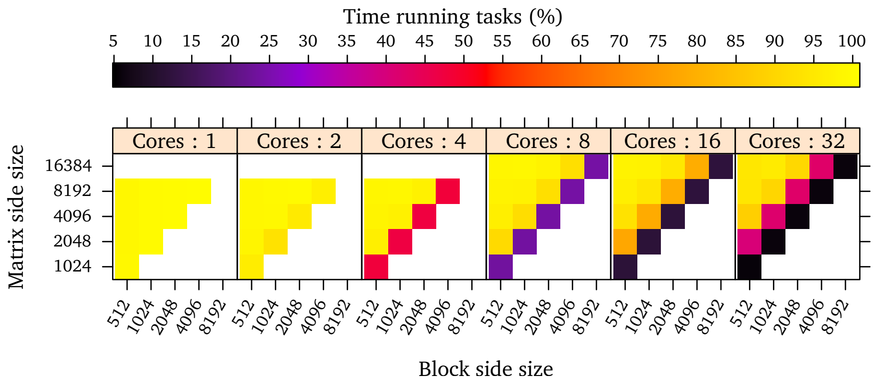


Figure 3.6: Mean time that each thread is kept busy running tasks in the SMPSs Gauss-Seidel implementation with several matrix and blocking sizes.

The measurements indicate that scalability is poor with more than 4 cores, which is the number of cores per NUMA node. This is due to memory placement issues that this chapter does not cover. The diagonal configurations with 4 or more cores also exhibit bad performance. Figure 3.6 shows that those configurations are limited by the amount of parallelism that is available.

For each problem size and number of cores we have selected the tile size with greatest mean performance without tracing. Table 3.1 summarizes the performance metrics of these configurations. Notice that with one thread, despite the runtime overhead, the problem still benefits from tiling.

Scheduling

Figure 3.7 shows the mean floating point performance of the SMPSs implementation under several strong scalability scenarios with the four schedulers. Each panel corresponds to a matrix side size N. The horizontal axis indicates the number of cores and the vertical axis the performance in mega element updates per second with a logarithmic scale. The right side of the figure shows the performance of the configurations with 32 cores using a linear scale. Each point represents the mean

Cores	N[a]	BS[b]	Tasks	MU/s[c]	IPC[d]	FPC[e]	Ovh.[f] (%)	Eff.[g] (%)	Idle (%)	Tr.O.[h] (%)
1	1024	512	64	59.9	1.84	0.19	0.93	99	0	0
1	2048	512	576	59.0	1.81	0.19	1.06	98	0	0
1	4096	512	3136	58.0	1.77	0.18	1.08	98	0	1
1	8192	1024	3136	58.0	1.77	0.18	0.44	99	0	0
2	1024	512	64	116.1	1.83	0.19	1.26	95	3	0
2	2048	512	576	117.3	1.80	0.19	1.51	98	0	0
2	4096	1024	576	115.9	1.77	0.18	0.55	99	0	0
2	8192	1024	3136	116.2	1.77	0.18	0.58	99	0	0
4	1024	512	64	115.3	1.82	0.19	1.15	47	51	0
4	2048	512	576	229.1	1.79	0.19	2.53	96	2	0
4	4096	512	3136	227.6	1.76	0.18	2.49	97	0	0
4	8192	1024	3136	230.3	1.76	0.18	0.89	98	0	0
8	1024	512	64	107.9	1.74	0.18	1.06	23	76	0
8	2048	512	576	416.5	1.72	0.18	4.57	90	7	0
8	4096	512	3136	437.0	1.70	0.18	4.70	96	2	0
8	8192	1024	3136	448.3	1.74	0.18	1.57	97	2	0
8	16384	1024	14400	453.7	1.74	0.18	1.59	98	0	0
16	1024	512	64	104.5	1.69	0.18	1.01	11	88	0
16	2048	512	576	633.2	1.55	0.16	7.12	77	21	0
16	4096	512	3136	802.6	1.63	0.17	8.68	93	5	0
16	8192	512	14400	834.8	1.63	0.17	9.00	96	1	0
16	16384	1024	14400	854.0	1.65	0.17	2.83	97	1	0
32	1024	512	64	100.2	1.64	0.17	1.02	5	94	0
32	2048	512	576	542.6	1.26	0.13	6.85	40	59	1
32	4096	512	3136	956.5	1.03	0.11	13.85	87	10	0
32	8192	512	14400	1033.0	1.04	0.11	15.07	94	4	0
32	16384	512	61504	1048.2	1.04	0.11	14.74	94	4	0

[a] Matrix side size..
[b] Submatrix side size.
[c] Mega element updates per second.
[d] Mean instructions per cycle while running tasks.
[e] Mean floating point operations per cycle while running tasks.
[f] Time that the main thread spends generating tasks and idle.
[g] Mean time that threads spend running tasks.
[h] Increment of the execution time when enabling tracing.

Table 3.1: Best submatrix side sizes for the SMPSs Gauss-Seidel implementation and their performance characteristics.

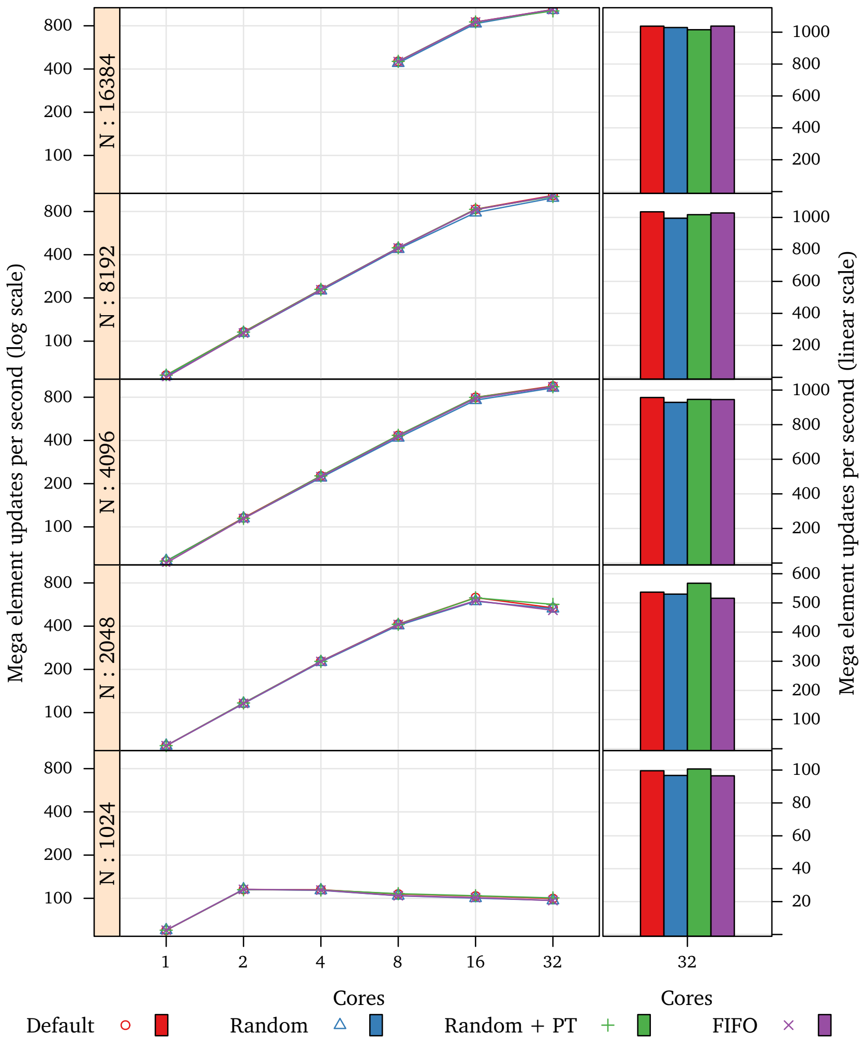


Figure 3.7: Strong scalability of the SMPSs Gauss-Seidel implementation with several matrix sizes under each scheduling policy and performance with 32 cores.

performance of the experiments with the block sizes that produced the best mean performance for each case with the default scheduler. Due to the time that it takes to generate these measurements, the biggest problem size has only been executed with more than 4 cores.

The figure shows that the smallest problem size does not scale well beyond 2 cores, the following one only scales up to 16 cores, and the rest do scale up to 32 cores.

It also indicates that all schedulers have similar performance characteristics. The reason is that none of the schedulers is able to exploit data reuse effectively. The numbers that appear in the graph shown previously in figure 3.4 indicate the task instantiation order. Since they correspond to a matrix divided in 6×6 blocks ($N/BS = 6$), each task instantiation number has also a numeric correspondence to the block that it updates. That is, two task instances t_i and t_j update the same block if and only if $i \equiv j \pmod{(N/BS)^2}$. However, the paths that traverse the tasks that update one block, for instance the first block (tasks 1, 37, 73, 109, 145 and 181), are multiple and contain tasks that update other blocks. Moreover, the amount data read from the surrounding blocks is very small. These conditions preclude the scheduler from following one of the successors in a greedy manner.

Figure 3.8 shows that the level-3 cache miss ratio is in most configurations around 50% which confirms that behavior. The figure is a violin plot of the mean level-3 data cache miss ratio while running tasks. Each row of panels corresponds to a matrix size. Each column of panels corresponds to a number of cores, and the horizontal axis groups the measurements with each scheduler. The violins shown in each panel are an estimation of the density of the metric, in this case the cache miss ratio. Wide sections correspond to miss ratios with high frequency, and narrow sections correspond to miss ratios with low frequency. Horizontal lines indicate that the given configuration has very low variability.

In all cases performance decreases with the number of cores. Since this is a memory intensive application and NUMA aspects are not being considered yet, as we increase the number of cores, we reduce the mean memory affinity and thus the memory latency increases.

Table 3.2 summarizes the mean values of the main performance metrics of the Gauss-Seidel decomposition with each scheduler.

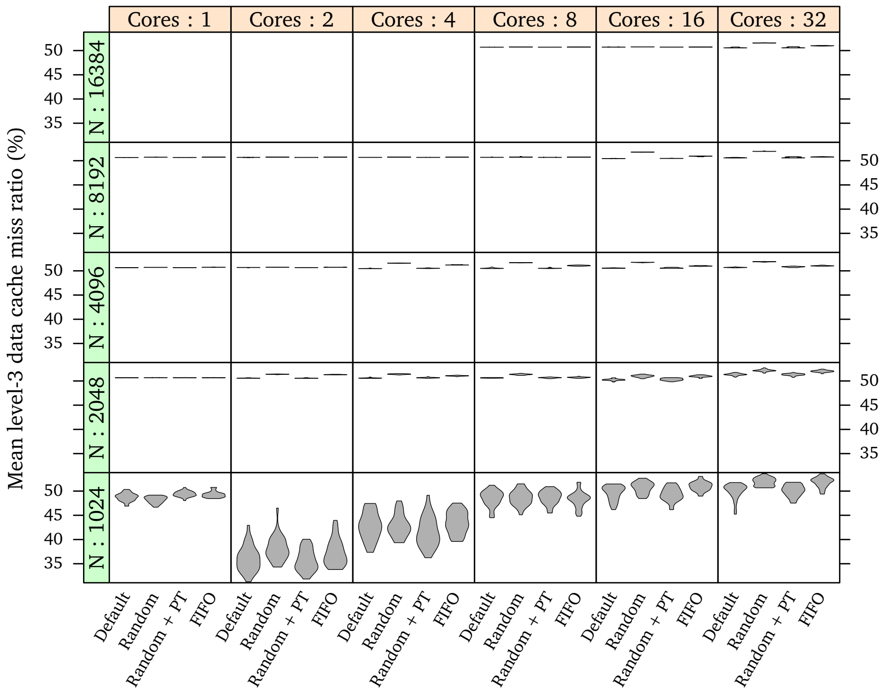


Figure 3.8: Average task level-3 cache miss ratio of the SMPSs Gauss-Seidel implementation with several matrix sizes under each scheduling policy.

Cores	N[a]	MU/s[b]	MU/s[b]	MU/s[b]	MU/s[b]	IPC[c]	IPC[c]	IPC[c]	IPC[c]	Eff.[d]	Eff.[d]	Eff.[d]	Eff.[d]
										(%)	(%)	(%)	(%)
1	1024	59	59	59	59	1.8	1.8	1.8	1.8	99	98	98	99
1	2048	58	58	58	58	1.8	1.8	1.8	1.8	99	99	99	99
1	4096	57	57	57	56	1.8	1.8	1.8	1.7	99	99	99	99
1	8192	58	57	58	56	1.8	1.8	1.8	1.7	99	99	99	99
2	1024	115	115	114	115	1.8	1.8	1.8	1.8	95	95	95	95
2	2048	116	115	116	115	1.8	1.8	1.8	1.8	98	98	98	98
2	4096	115	114	114	114	1.8	1.8	1.8	1.7	99	99	98	99
2	8192	116	113	115	114	1.8	1.7	1.8	1.7	99	99	99	99
4	1024	114	113	114	114	1.8	1.8	1.8	1.8	47	47	47	47
4	2048	227	225	227	227	1.8	1.8	1.8	1.8	96	96	96	96
4	4096	226	220	226	223	1.8	1.7	1.8	1.7	97	98	97	98
4	8192	229	224	229	228	1.8	1.7	1.8	1.7	98	99	98	99
8	1024	107	104	107	104	1.7	1.7	1.7	1.7	23	23	23	23
8	2048	412	404	412	410	1.7	1.7	1.7	1.7	90	91	90	91
8	4096	433	417	434	428	1.7	1.6	1.7	1.7	96	97	96	97
8	8192	446	438	446	445	1.7	1.7	1.7	1.7	97	98	97	98
8	16384	452	439	452	450	1.7	1.7	1.7	1.7	98	99	98	99
16	1024	103	100	104	101	1.7	1.6	1.7	1.6	11	11	11	11
16	2048	632	596	630	602	1.5	1.5	1.5	1.5	77	75	77	76
16	4096	798	763	797	786	1.6	1.5	1.6	1.6	93	93	93	93
16	8192	830	786	828	822	1.6	1.5	1.6	1.6	96	96	96	97
16	16384	850	828	850	849	1.7	1.6	1.6	1.6	97	97	97	98
32	1024	99	96	100	96	1.6	1.6	1.7	1.6	5	5	5	5
32	2048	536	530	567	515	1.3	1.2	1.3	1.2	40	40	40	40
32	4096	956	929	946	945	1.0	1.0	1.0	1.0	87	88	88	88
32	8192	1035	995	1017	1028	1.0	1.0	1.0	1.0	94	94	93	93
32	16384	1037	1029	1015	1038	1.0	1.0	1.0	1.0	94	94	93	95

Scheduler:	Default	Random	Random + PT	FIFO

[a] Matrix side size.
[b] Mega element updates per second.
[c] Mean instructions per cycle while running tasks.
[d] Mean time that threads spend running tasks.

Table 3.2: Performance summary of the scheduler on the SMPSs Gauss-Seidel decomposition.

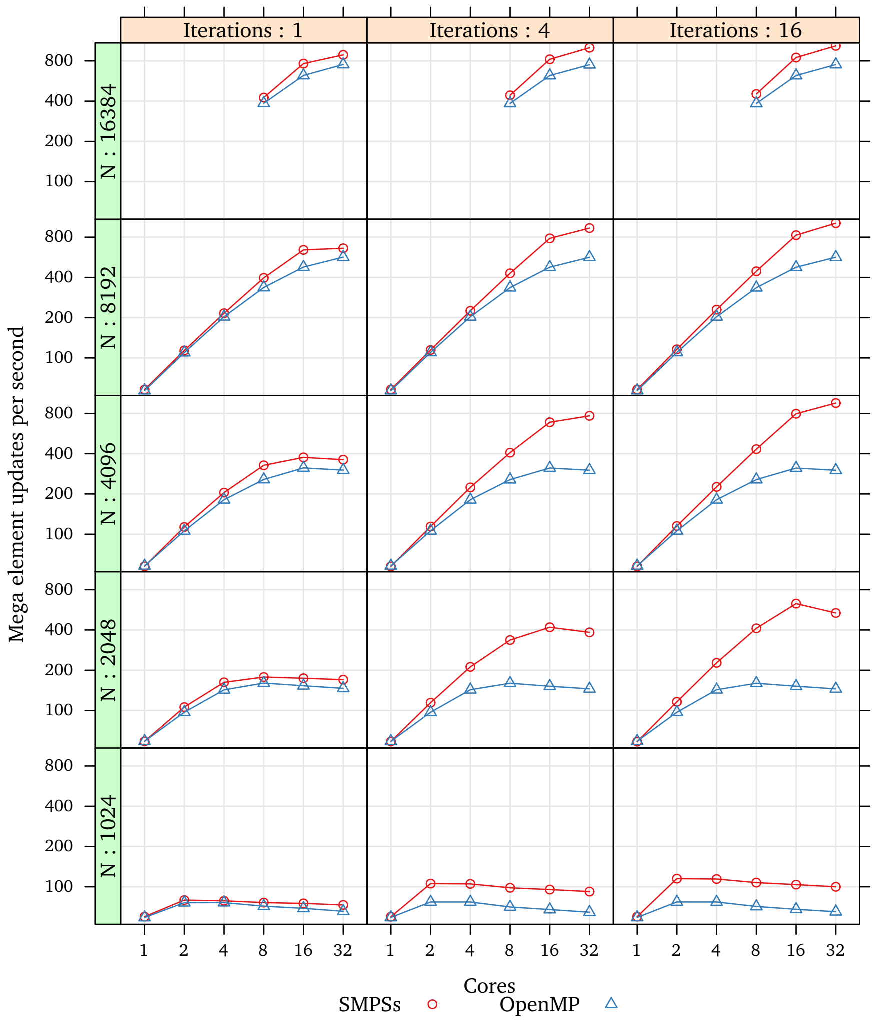


Figure 3.9: Strong scalability of each parallel implementation of Gauss-Seidel with several matrix sizes and 1, 4 and 16 outer loop iterations.

Performance of the Implementations

Figure 3.9 shows the mean element update rate of the SMPSs and the OpenMP implementations under several strong scalability scenarios and with 1, 4 and 16 outer loop iterations. Each column of plots corresponds to executions with a fixed number of outer loop iterations, and each row of plots corresponds to a problem size. In all cases the SMPSs version scales better than the OpenMP version.

With just one outer loop iteration, the SMPSs version is able to absorb part of the unbalance that is present in the OpenMP version due to the barrier between wavefronts. By taking into account dependencies, the SMPSs version can start to execute tasks of a wavefront before the previous one has finished. As we increase the number of outer loop iterations, the SMPSs version is able to exploit the parallelism that lies between the tasks of those iterations.

Figure 3.10 shows the performance improvement of the SMPSs implementation

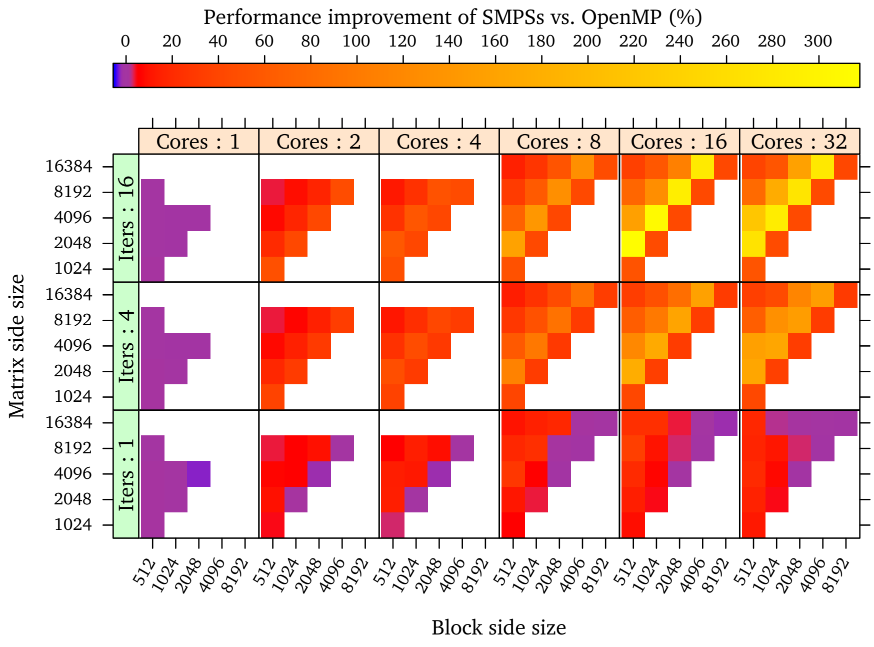


Figure 3.10: Performance gained by the SMPSs implementation of the Gauss-Seidel algorithm compared to the OpenMP implementation.

with respect to the OpenMP implementation. A 0% improvement indicates that both implementations perform equally, and a 100% improvement indicates that the SMPSs version performs two times as fast. Row of panels contains plots for a fixed number of outer loop iterations, and each column corresponds to a fixed number of cores. Each plot shows the mean performance for several problem sizes and block sizes for a fixed number of cores and outer loop iterations. Notice that the SMPSs version performs up to 4 times as fast with 16 and 32 cores when running 16 outer loop iterations. This improvement is the result of the increased parallelism. Figure 3.11 shows that the SMPSs executions are able fill up the cores with work up to 4 times as much as the OpenMP version.

Table 3.3 summarizes the mean values of the main performance metrics of each Gauss-Seidel implementation with 16 outer loop iterations.

Cores	N[a]	BS[b]	BS[b]	MU/s[c]	MU/s[c]	FPC[d]	FPC[d]	Eff.[e] (%)	Eff.[e] (%)
1	1024	512	512	59.9	59.0	0.19	0.19	99	98
1	2048	1024	1024	58.8	58.9	0.19	0.19	99	99
1	4096	1024	1024	57.7	58.0	0.18	0.18	99	99
1	8192	1024	1024	58.0	58.0	0.18	0.18	99	99
2	1024	512	256	115.3	90.0	0.19	0.18	95	80
2	2048	512	256	116.2	101.1	0.19	0.18	98	89
2	4096	1024	256	115.7	107.0	0.18	0.18	99	93
2	8192	1024	512	116.0	110.0	0.18	0.18	99	95
4	1024	512	256	114.8	128.5	0.19	0.17	47	60
4	2048	512	256	227.9	168.6	0.19	0.18	96	76
4	4096	512	256	226.6	193.0	0.18	0.18	97	86
4	8192	1024	512	229.6	203.0	0.18	0.18	98	89
8	1024	512	256	107.1	127.8	0.18	0.15	23	35
8	2048	512	256	412.9	210.8	0.18	0.15	90	56
8	4096	512	256	433.4	287.7	0.18	0.16	96	71
8	8192	1024	256	446.4	343.4	0.18	0.17	97	82
8	16384	1024	512	452.1	384.3	0.18	0.17	98	87
16	1024	512	256	103.7	116.0	0.18	0.13	11	18
16	2048	512	256	632.2	242.9	0.16	0.13	77	37
16	4096	512	256	798.4	375.0	0.17	0.14	93	55
16	8192	512	256	830.2	505.2	0.17	0.14	96	70
16	16384	1024	256	850.8	622.1	0.17	0.15	97	81
32	1024	512	256	99.6	107.1	0.17	0.12	5	9
32	2048	512	256	536.8	224.4	0.13	0.12	40	19
32	4096	512	256	957.0	429.9	0.11	0.11	87	38
32	8192	512	256	1035.3	595.4	0.11	0.10	94	59
32	16384	512	256	1037.8	753.8	0.11	0.10	94	74

Implementation: SMPSs OpenMP

[a] Matrix side size.
[b] Block side size
[c] Mega element updates per second.
[d] Mean floating point operations per cycle while running tasks.
[e] Mean time that threads spend running tasks.

Table 3.3: Performance summary of the Gauss-Seidel implementations.

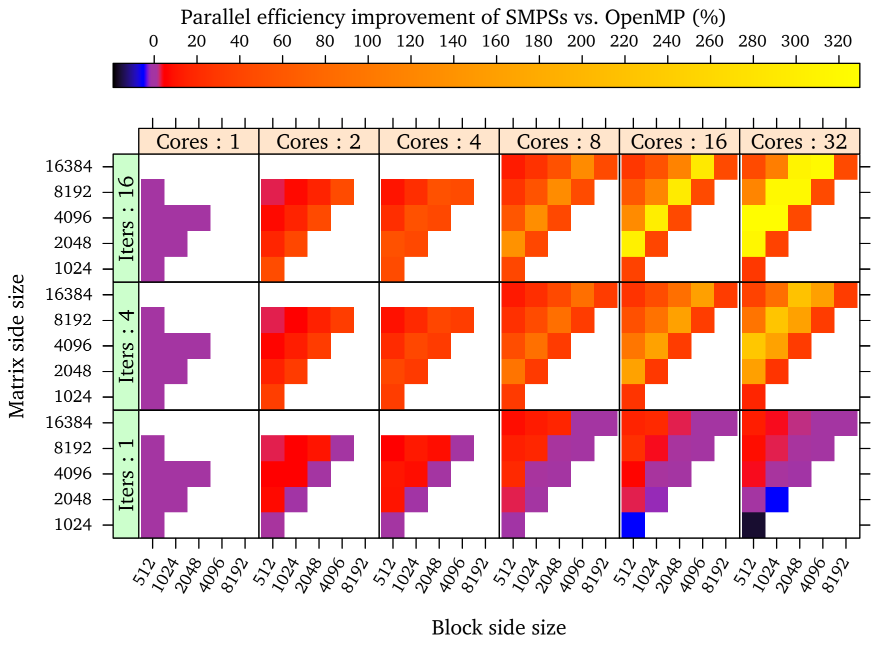


Figure 3.11: Parallel efficiency improvement by the SMPSs implementation of the Gauss-Seidel algorithm compared to the OpenMP implementation.

3.6.4 Cholesky

Algorithm

The Cholesky algorithm decomposes a real symmetric positive definite matrix A into a product of a lower triangular matrix L by its transposed L^T. The algorithm takes a matrix A as input and generates a lower triangular matrix L such that:

$$A = LL^T \tag{3.9}$$

Given the definition of the matrix multiplication, the individual elements of A and L are such that:

$$A_{ij} = \sum_{k=1}^{j} L_{ik} L_{jk}^T = \sum_{k=1}^{j} L_{ik} L_{kj} = \sum_{k=1}^{j-1} L_{ik} L_{kj} + L_{ij} L_{jj} \tag{3.10}$$

from which we can isolate the values of the diagonal of L:

$$L_{jj} = \sqrt{A_{jj} - \sum_{k=1}^{j-1} L_{jk} L_{kj}} \tag{3.11}$$

and the values below the diagonal:

$$L_{ij} = \frac{A_{jj} - \sum_{k=1}^{j-1} L_{ik} L_{kj}}{L_{jj}} \tag{3.12}$$

Parallelization with SMPSs

The Cholesky decomposition can be implemented in-place. That is, the result L can be stored at the same memory location as A. Most implementations of the algorithm are in-place, since the algorithm is typically used to solve linear systems of equations and after the decomposition, the original values of A are no longer needed.

The algorithm has several variants that differ in the order in which the result is calculated. [Dongarra and Walker, 1993] propose a methodology to derive blocked algorithms from their element-wise versions. One of the possible blockings of the Cholesky algorithm is presented by [Andersen et al., 2005], and is the one that we parallelize in this section.

In our implementation we use matrices of N by N elements and we operate over it in submatrices of BS by BS elements. Similarly to the matrix multiplication code, we have laid it out in memory in blocks that match the sub-matrix dimensions. In listing 3.11 we show the main code and its tasks. It has four tasks, and all are implemented as simple calls to the LAPACK and BLAS libraries.

While previous algorithms generated regular graphs, the Cholesky parallelization generates a complex graph. Figure 3.12 shows the tasks of the Cholesky algorithm when the matrix is divided in 8 by 8 blocks ($BS = N/8$). Each node represents a task instance and has a color that corresponds to its type. The node numbers indicate their instantiation order.

Figure 3.13 shows an identical graph that has each node colored according to its instantiation order. The right side of the legend indicates the outer loop iteration that creates each task. The figure illustrates the presence of parallelism between distant parts of the code. Notice that the task instances in the third, fourth and fifth rows are from distant outer iterations. For instance, the fifth row shows that task 17, that is the first task of iteration j=1, can run in parallel with task 113, which belongs to iteration j=6.

While this algorithm consists of one main function that creates all the tasks, neither the programming model nor the runtime do limit the ability to detect distant parallelism to a single function. A more complex program could make use the result of the Cholesky decomposition as part of its calculation. In such scenario, the tasks of the Cholesky decomposition would also run in parallel to the following computations as the blocks of the result are calculated.

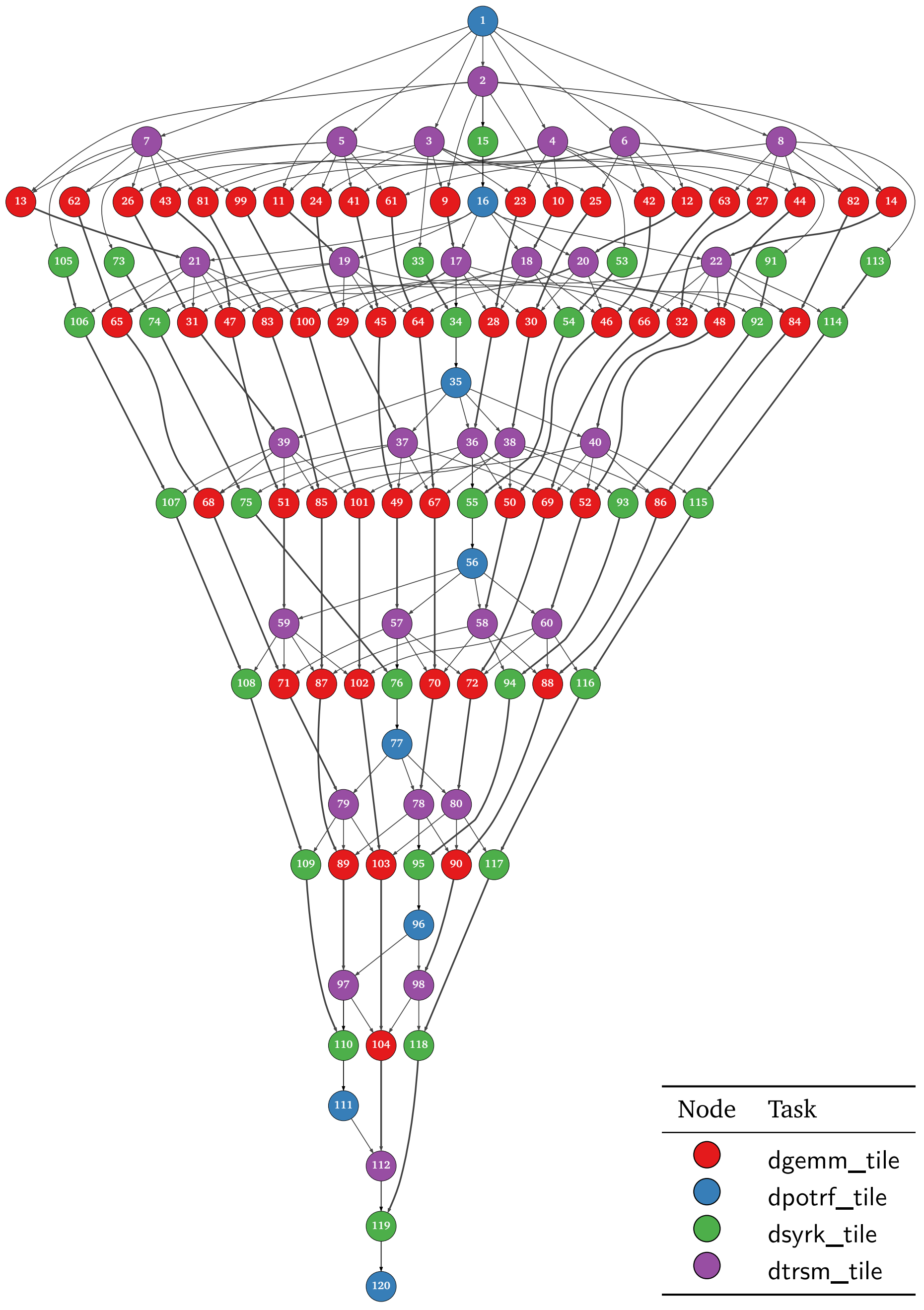
Parallelization with OpenMP

To evaluate the performance of the SMPSs implementation, we have also implemented a similar version using OpenMP 3 tasks. Listing 3.12 shows the main code and its tasks. This version, as opposed to the SMPSs version, generates tasks in parallel. However, since OpenMP 3 does not track dependencies, the tasks of the outer loop that could have interdependencies have been isolated by the implicit barriers of the parallel loops. Moreover, the OpenMP version uses flat matrix instead of the blocked layout of the SMPSs version.

Both the SMPSs version and the OpenMP version use the BLAS and LAPACK functions from the MKL library in sequential mode.

```
void cholesky(int N, int BS, double A[N/BS][N/BS][BS][BS]) {
    for (int j = 0; j < N/BS; j++) {
        for (int k = 0; k < j; k++)
            for (int i = j+1; i < N/BS; i++)
                dgemm_tile(BS, A[k][i], A[k][j], A[j][i]);
        for (int i = 0; i < j; i++)
            dsyrk_tile(BS, A[i][j], A[j][j]);
        dpotrf_tile(BS, A[j][j]);
        for (int i = j+1; i < N/BS; i++)
            dtrsm_tile(BS, A[j][j], A[j][i]);
    }
}

#pragma css task input(BS) inout(A) highpriority
void dpotrf_tile(int BS, double A[BS][BS]) {
    int info;
    dpotrf_("L", &BS, A, &BS, &info);
}

#pragma css task input(A, B, BS) inout(C)
void dgemm_tile(int BS,
    double const A[BS][BS], double const B[BS][BS], double C[BS][BS])
{
    double const done=1.0, dmone=-1.0;
    dgemm_("N", "T", &BS, &BS, &BS, &dmone, A, &BS, B, &BS, &done, C,
      &BS);
}

#pragma css task input(T, BS) inout(B)
void dtrsm_tile(int BS, double const T[BS][BS], double B[BS][BS]) {
    double const done=1.0;
    dtrsm_("R", "L", "T", "N", &BS, &BS, &done, T, &BS, B, &BS);
}

#pragma css task input(A, BS) inout(C)
void dsyrk_tile(int BS, double const A[BS][BS], double C[BS][BS]) {
    double const done=1.0, dmone=-1.0;
    dsyrk_("L", "N", &BS, &BS, &dmone, A, &BS, &done, C, &BS);
}
```

Listing 3.11: Double precision blocked Cholesky decomposition in SMPSs.



Node	Task
🟥	dgemm_tile
🟦	dpotrf_tile
🟩	dsyrk_tile
🟪	dtrsm_tile

Figure 3.12: Graph of the Cholesky algorithm when executed over a matrix composed of 8 by 8 blocks ($BS = N/8$).

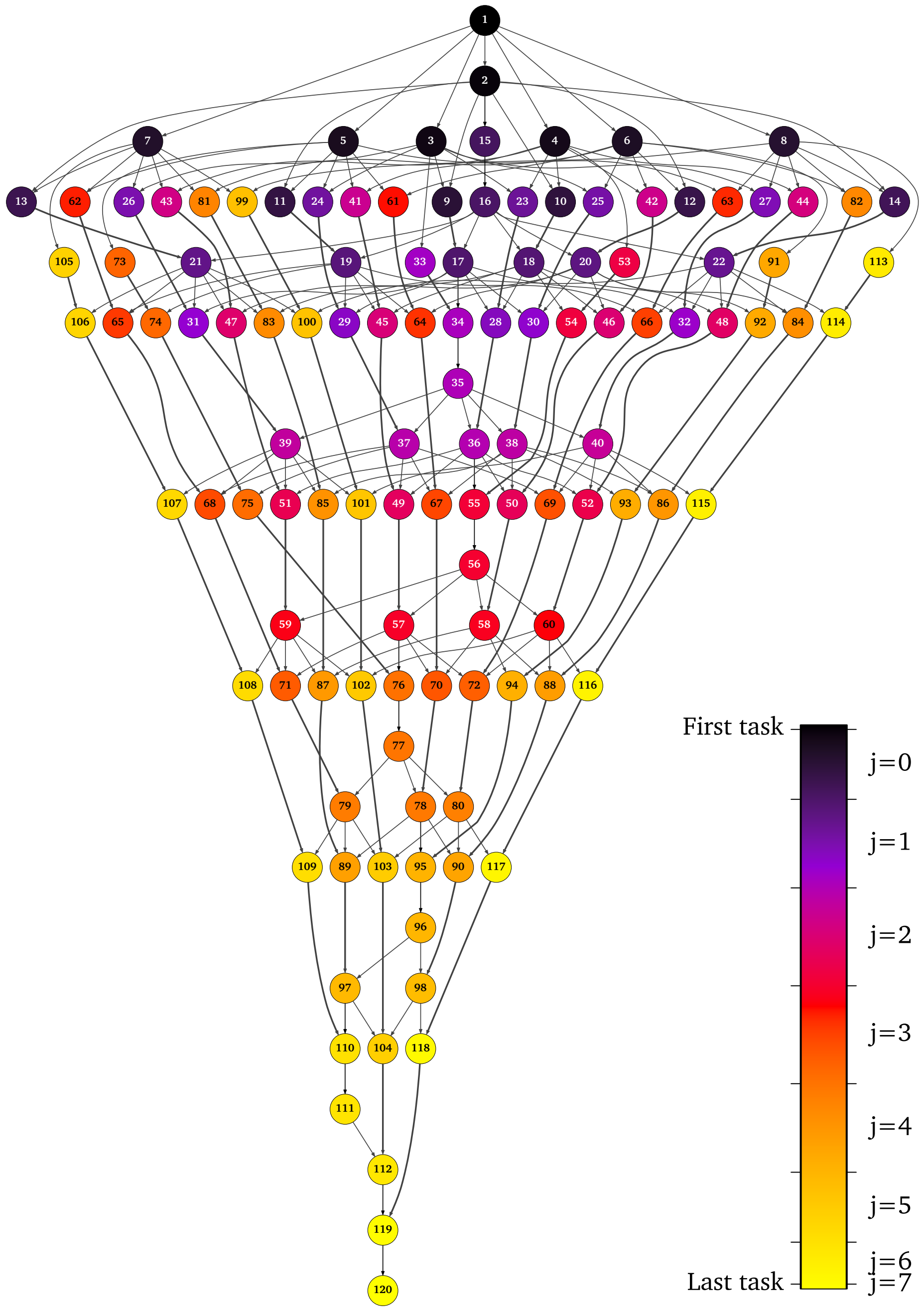


Figure 3.13: Graph of the Cholesky algorithm when executed over a matrix composed of 8 by 8 blocks ($BS = N/8$) showing the relation between the task parallelism and the instantiation order.

```
void cholesky(int N, int BS, double A[N][N]) {
  for (int j = 0; j < N; j+=BS) {
    #pragma omp parallel for
    for (int k= 0; k< j; k+=BS)
      for (int i = j+BS; i < N; i+=BS)
        #pragma omp task untied
        dgemm_tile(&A[k][i], &A[k][j], &A[j][i], BS, N);
    #pragma omp parallel for
    for (int i = 0; i < j; i+=BS)
      #pragma omp task untied
      dsyrk_tile(&A[i][j], &A[j][j], BS, N);
    dpotrf_tile(&A[j][j], BS, N);
    #pragma omp parallel for
    for (int i = j+BS; i < N; i+=BS)
      #pragma omp task untied
      dtrsm_tile(&A[j][j], &A[j][i], BS, N);
  }
}

void dpotrf_tile(double *A, int BS, int N) {
  int info;
  dpotrf_("L", &BS, A, &N, &info);
}

void dgemm_tile(double *A, double *B, double *C, int BS, int N) {
  double const done=1.0, dmone=-1.0;
  dgemm_("N", "T", &BS, &BS, &BS, &dmone, A, &N, B, &N, &done, C,
    &N);
}

void dtrsm_tile(double *T, double *B, int BS, int N) {
  double const done=1.0;
  dtrsm_("R", "L", "T", "N", &BS, &BS, &done, T, &N, B, &N);
}

void dsyrk_tile( double *A, double *C, int BS, int N) {
  double const done=1.0, dmone=-1.0;
  dsyrk_("L", "N", &BS, &BS, &dmone, A, &N, &done, C, &N);
}
```

Listing 3.12: Double precision Cholesky decomposition in OpenMP with tasks.

Parallel Library Version

The Cholesky algorithm is implemented in the LAPACK API as the potrf family of functions. To compare with a highly optimized version, we have chosen to use the implementation provided by the MKL library in parallel mode. The code consists of a single call to dpotrf over a flat matrix. By using the same library as in the other versions we guarantee that the underlying kernel computations are the same and thus we are evaluating the programming models and parallelization instead of the quality of the kernels.

Determining the Submatrix Dimensions

Like the case of the Gauss-Seidel algorithm, the value of the submatrix dimensions BS also determines the number of tasks of the problem decomposition and the computational weight of each task. Big submatrices generate few tasks with long computations, while small submatrices generate more and shorter tasks. These parameters have an impact on the parallelism, the efficiency of the tasks and the runtime overhead.

Figure 3.14 shows the floating point performance of our implementation. Each panel corresponds to a set of measurements with a fixed number of cores. The vertical axis determines the matrix side size (N), and the horizontal axis is the submatrix side size (BS). The figure shows the mean performance of 30 executions of each configuration with respect to the theoretical hardware peak measured in floating point operations per second.

Compared to the previous algorithms, Cholesky has a much more complex dependency structure and its graph is much more narrow with respect to the blocking size. As a result, its amount of parallelism is much more constrained by the blocking size as demonstrated by the diagonal gradients present in figure 3.14 which show low performance. This effect is confirmed by the average idle time graph in figure 3.15, which shows high idleness in those cases.

The blocking size has also an effect on the overhead of the main thread. Figure 3.16 shows that metric. Notice that with submatrices of 128 by 128 elements it is always higher than with other configurations, and that it grows with the number of cores and starts to be noticeable with 8. Despite the increased overhead, figure 3.15 shows that it is not enough to cause starvation, and is not the reason for the low parallelism in the diagonal configurations.

The mean task floating point operations per cycle are shown in figure 3.17. The executions with 128 by 128 element submatrices have also lower task floating point performance than the rest and this difference also grows with the number of cores. This explains the lower total performance shown in figure 3.14 with 16 and 32 cores for those configurations.

Table 3.4 summarizes the performance of the best submatrix configurations. Notice that with one thread some of the matrix sizes have better performance when decomposed into more than one task ($\mathsf{BS} < \mathsf{N}$), than when solved only by the MKL library ($\mathsf{BS} = \mathsf{N}$). The tracing overhead remains low in all cases, and thus all columns after the Gigaflops per second can be considered reliable.

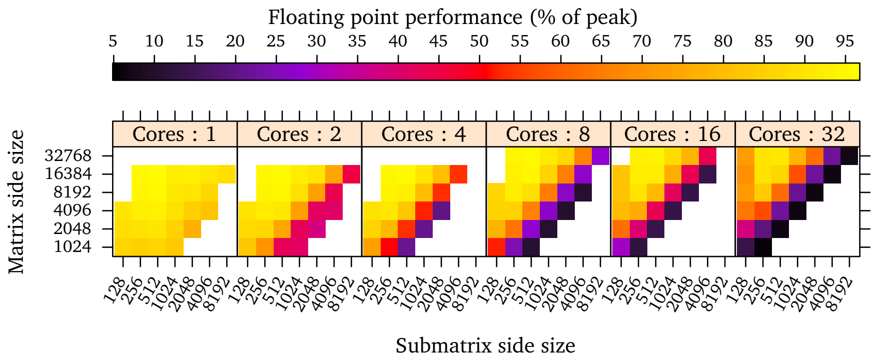


Figure 3.14: Performance of the SMPSs Cholesky implementation with several matrix and blocking sizes.

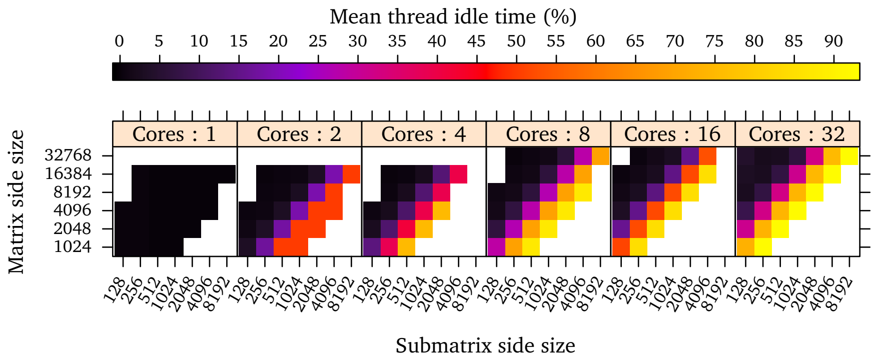


Figure 3.15: Average time taken by each thread idling in the SMPSs Cholesky implementation with several matrix and blocking sizes.

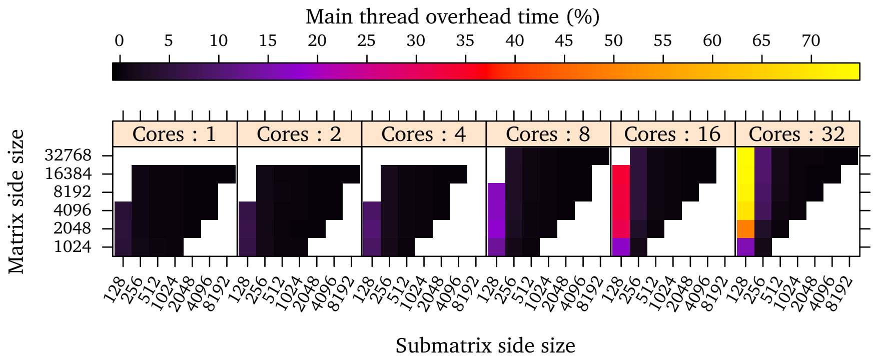


Figure 3.16: Percentage of time time that the main thread spends managing tasks when running the SMPSs Cholesky implementation with several matrix and blocking sizes.

Cores	N[a]	BS[b]	Tasks	GF[c]	IPC[d]	FPC[e]	Ovh.[f] (%)	Eff.[g] (%)	Idle (%)	Tr.O.[h] (%)
1	1024	512	4	5.5	5.55	3.54	0.26	99	0	0
1	2048	512	20	5.8	5.69	3.69	0.18	99	0	0
1	4096	512	120	6.0	5.79	3.80	0.16	99	0	0
1	8192	512	816	6.1	5.86	3.87	0.16	99	0	0
1	16384	512	5984	6.1	5.90	3.90	0.16	99	0	0
2	1024	128	120	10.6	5.61	3.54	5.93	92	3	3
2	2048	128	816	11.2	5.71	3.68	6.40	94	0	3
2	4096	256	816	11.8	5.79	3.78	1.00	98	0	0
2	8192	512	816	12.1	5.86	3.87	0.20	99	0	0
2	16384	512	5984	12.2	5.90	3.90	0.20	99	0	0
4	1024	128	120	18.6	5.49	3.46	8.52	81	13	5
4	2048	128	816	21.9	5.70	3.67	10.06	92	2	3
4	4096	256	816	23.2	5.78	3.78	1.52	97	1	0
4	8192	256	5984	24.0	5.84	3.84	1.44	98	0	0
4	16384	512	5984	24.4	5.89	3.90	0.28	99	0	0
8	1024	128	120	26.6	4.96	3.13	13.41	66	28	3
8	2048	128	816	39.8	5.45	3.50	17.80	87	6	3
8	4096	128	5984	43.7	5.57	3.62	16.69	93	1	3
8	8192	256	5984	47.1	5.80	3.81	2.56	97	1	0
8	16384	512	5984	48.3	5.86	3.88	0.44	98	1	0
8	32768	512	45760	48.8	5.88	3.90	0.44	99	0	0
16	1024	128	120	30.5	4.13	2.61	17.30	45	51	4
16	2048	128	816	63.8	4.93	3.17	31.35	78	15	3
16	4096	128	5984	81.0	5.32	3.45	33.02	90	3	3
16	8192	256	5984	91.9	5.73	3.77	4.73	96	2	0
16	16384	512	5984	94.4	5.82	3.85	0.74	97	2	0
16	32768	512	45760	96.3	5.83	3.87	0.73	98	1	0
32	1024	128	120	27.4	3.39	2.14	15.84	25	72	2
32	2048	128	816	77.6	3.81	2.45	49.38	62	31	1
32	4096	128	5984	131.0	4.63	3.00	69.31	84	8	3
32	8192	256	5984	171.2	5.61	3.69	9.06	92	6	0
32	16384	256	45760	183.0	5.68	3.74	9.30	96	2	0
32	32768	512	45760	186.2	5.72	3.80	1.33	97	2	0

[a] Matrix side size..
[b] Submatrix side size.
[c] Gigaflops per second.
[d] Mean instructions per cycle while running tasks.
[e] Mean floating point operations per cycle while running tasks.
[f] Time that the main thread spends generating tasks and idle.
[g] Mean time that threads spend running tasks.
[h] Increment of the execution time when enabling tracing.

Table 3.4: Best submatrix side sizes for the SMPSs Cholesky implementation and their performance characteristics.

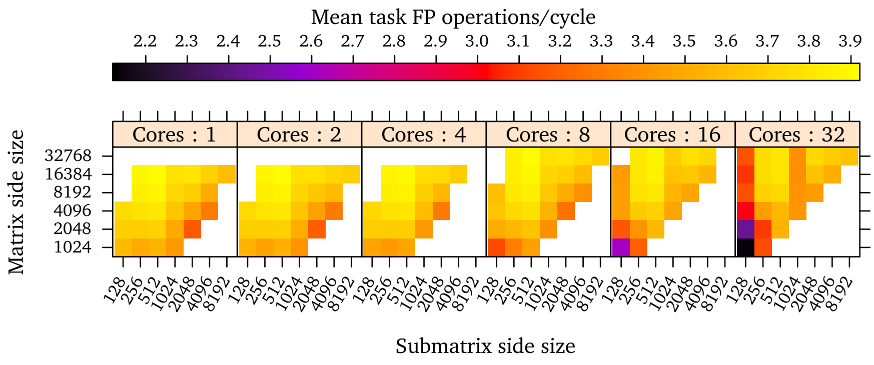


Figure 3.17: Mean floating point operations per cycle while running each task of the SMPSs Cholesky implementation with several matrix and blocking sizes.

Scheduling

Figure 3.18 shows the mean floating point performance of the Cholesky under several strong scalability scenarios with each scheduler. The measurements have been performed with the submatrix sizes that produced the best mean performance. Horizontal lines on the left panels correspond to the theoretical peak for 1, 2, 4, 8 and 16 cores respectively. The right side of the figure shows the mean performance of the configurations with 32 cores with a linear scale.

The difference between the schedulers grows with the number of cores and the problem size. In all cases the random scheduler has the worst performance and the rest have similar performance. However, the difference decreases as we increase the problem size until disappearing at the biggest one. The reason is that as we increase the problem size, the task size can also increase, and thus reduce the impact of the bad reuse of the cache between tasks that the random scheduler produces. The default scheduler and the random scheduler with the protected task mechanism perform almost identically. This result suggests that the protection mechanism is the factor that produces those results.

Since the Cholesky graph is more narrow than the previous ones, more parallelism requires greater reductions in block size than in the previous cases. Therefore, as we increase the number of cores, we must reduce granularity and as a consequence task IPC decreases and so does the total performance relative to the hardware peak. Figure 3.19 shows the average task floating point operations per cycle. Notice that the protected task mechanism is enough to reach the best performance.

Table 3.5 summarizes the mean values of the main performance metrics of the Cholesky decomposition with each scheduler.

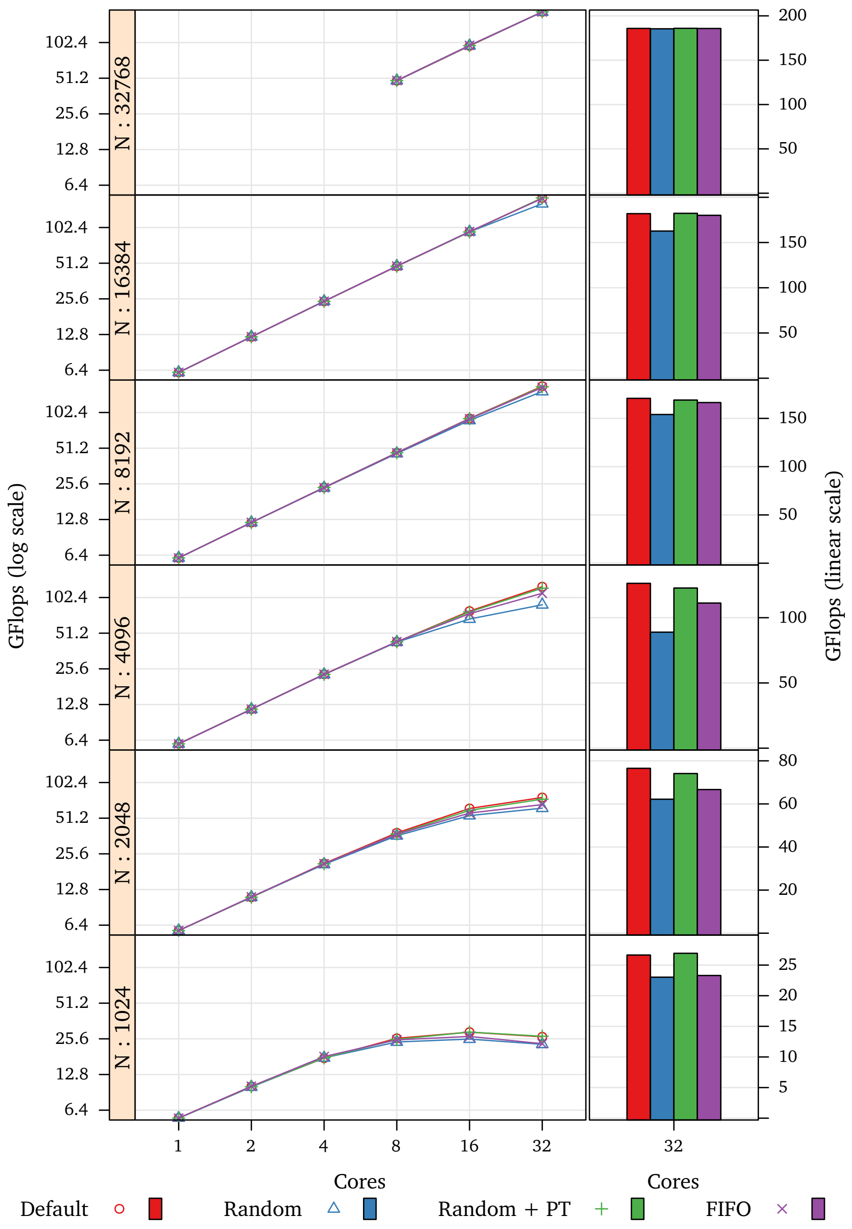


Figure 3.18: Strong scalability of the SMPSs Cholesky implementation with several matrix sizes under each scheduling policy and performance with 32 cores.

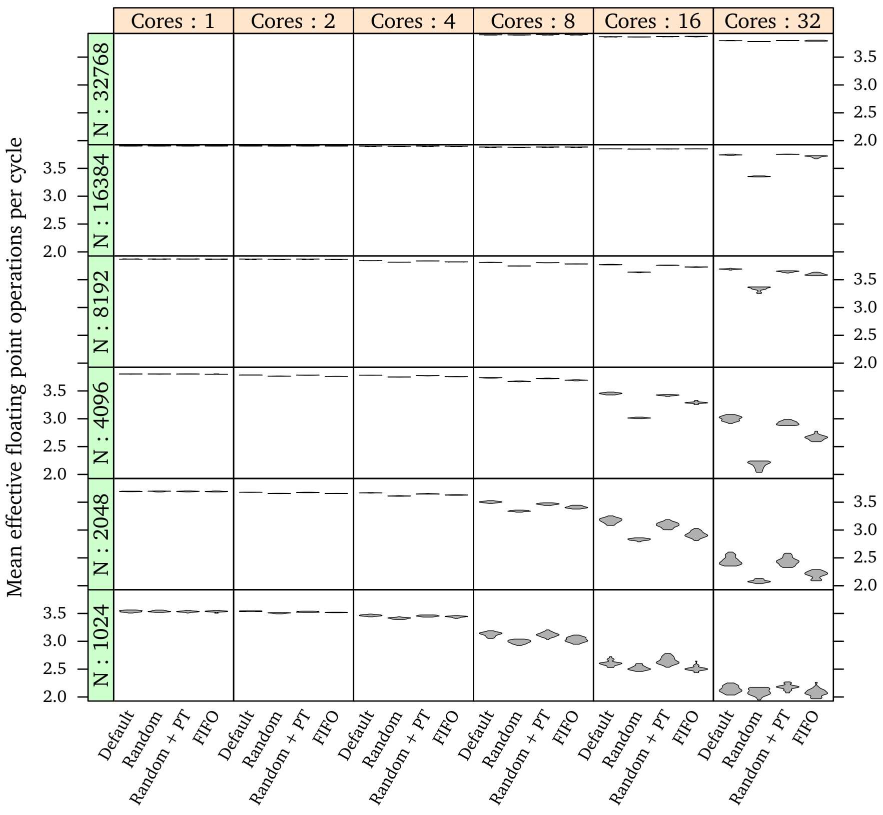


Figure 3.19: Average task floating point operations per cycle of the SMPSs Cholesky implementation with several matrix sizes under each scheduling policy.

Cores	N[a]	GF[b]	GF[b]	GF[b]	GF[b]	FPC[c]	FPC[c]	FPC[c]	FPC[c]	Eff.[d] (%)	Eff.[d] (%)	Eff.[d] (%)	Eff.[d] (%)
1	1024	5.5	5.5	5.5	5.5	3.5	3.5	3.5	3.5	99	99	99	99
1	2048	5.8	5.8	5.8	5.8	3.7	3.7	3.7	3.7	99	99	99	99
1	4096	6.0	6.0	6.0	5.9	3.8	3.8	3.8	3.8	99	99	99	99
1	8192	6.1	6.1	6.1	6.1	3.9	3.9	3.9	3.9	99	99	99	99
1	16384	6.1	6.1	6.1	6.1	3.9	3.9	3.9	3.9	99	99	99	99
2	1024	10	10	10	10	3.5	3.5	3.5	3.5	92	92	91	92
2	2048	11	11	11	11	3.7	3.7	3.7	3.7	95	96	95	97
2	4096	11	11	11	11	3.8	3.8	3.8	3.8	98	98	98	98
2	8192	12	12	12	12	3.9	3.9	3.9	3.9	99	99	99	99
2	16384	12	12	12	12	3.9	3.9	3.9	3.9	99	99	99	99
4	1024	17	17	17	18	3.5	3.4	3.5	3.4	81	83	82	84
4	2048	21	20	21	21	3.7	3.6	3.7	3.6	92	92	92	93
4	4096	23	22	22	22	3.8	3.7	3.8	3.8	97	97	97	97
4	8192	23	23	23	23	3.8	3.8	3.8	3.8	98	98	98	99
4	16384	24	24	24	24	3.9	3.9	3.9	3.9	99	99	99	99
8	1024	25	24	25	25	3.1	3.0	3.1	3.0	66	65	65	67
8	2048	38	36	37	37	3.5	3.3	3.5	3.4	87	86	86	87
8	4096	43	43	43	43	3.7	3.7	3.7	3.7	92	93	92	93
8	8192	46	46	46	46	3.8	3.7	3.8	3.8	97	98	98	98
8	16384	48	48	48	48	3.9	3.9	3.9	3.9	98	99	98	99
8	32768	48	48	48	48	3.9	3.9	3.9	3.9	99	99	99	99
16	1024	29	25	29	26	2.6	2.5	2.7	2.5	45	41	44	43
16	2048	61	53	59	56	3.2	2.8	3.1	2.9	78	76	77	77
16	4096	78	67	77	74	3.5	3.0	3.4	3.3	90	89	90	90
16	8192	91	87	90	90	3.8	3.6	3.8	3.7	96	96	96	96
16	16384	94	94	93	94	3.9	3.8	3.9	3.9	97	97	97	97
16	32768	95	96	96	96	3.9	3.9	3.9	3.9	98	99	99	99
32	1024	26	23	26	23	2.1	2.1	2.2	2.1	25	22	25	22
32	2048	76	62	74	66	2.5	2.1	2.4	2.2	62	60	61	61
32	4096	126	88	122	111	3.0	2.2	2.9	2.7	84	81	83	83
32	8192	170	154	168	166	3.7	3.3	3.6	3.6	92	91	92	92
32	16384	181	162	182	180	3.7	3.4	3.8	3.7	96	96	96	96
32	32768	185	185	185	185	3.8	3.8	3.8	3.8	97	97	97	97

Scheduler: Default Random Random + PT FIFO

[a] Matrix side size.
[b] Gigaflops per second.
[c] Mean floating point operations per cycle while running tasks.
[d] Mean time that threads spend running tasks.

Table 3.5: Performance summary of the scheduler on the SMPSs Cholesky decomposition.

Performance of the Implementations

Figure 3.20 shows the mean floating point performance of the three implementations under several strong scalability scenarios. The measurements have been selected from the ones with submatrix sizes that produced the best mean performance, except for the parallel MKL version, which has its own blocking controlled by the implementation itself. In all cases the SMPSs implementation has better absolute mean performance and better strong scalability. Notice that in most cases the MKL parallel version does not scale from 16 to 32 cores.

In all cases performance decreases with the number of cores, since the amount of work per core is smaller and so is the potential IPC. Figure 3.21 shows the average floating point operations of the useful sections of time (that is, not idling nor executing the runtime) and figure 3.22 shows the mean fraction of time that threads spend running the algorithm. Despite the fact that in many cases the MKL version has the best parallel efficiency, its number of floating point operations per cycle is lower and thus its total performance is worse. Moreover, it is the implementation with least variability. This might be a result of using a static scheduling policy as opposed to the SMPSs and the taskified OpenMP implementations.

While the SMPSs version is capable of finding distant parallelism, the MKL does not, and thus to increase the amount of parallelism it must decompose the problem into smaller parts. This explains the useful time ratio of the MKL version and its lower floating point operations per cycle.

Another important aspect is that for a given block size, the SMPSs implementation can find more parallelism than the OpenMP version. Figure 3.23 shows the average difference in parallel efficiency between the two versions at each problem size and decomposition granularity. Notice that in most configurations, the SMPSs version has at least 40% more parallel efficiency than the OpenMP version.

Table 3.6 summarizes the mean values of the main performance metrics of each Cholesky implementation.

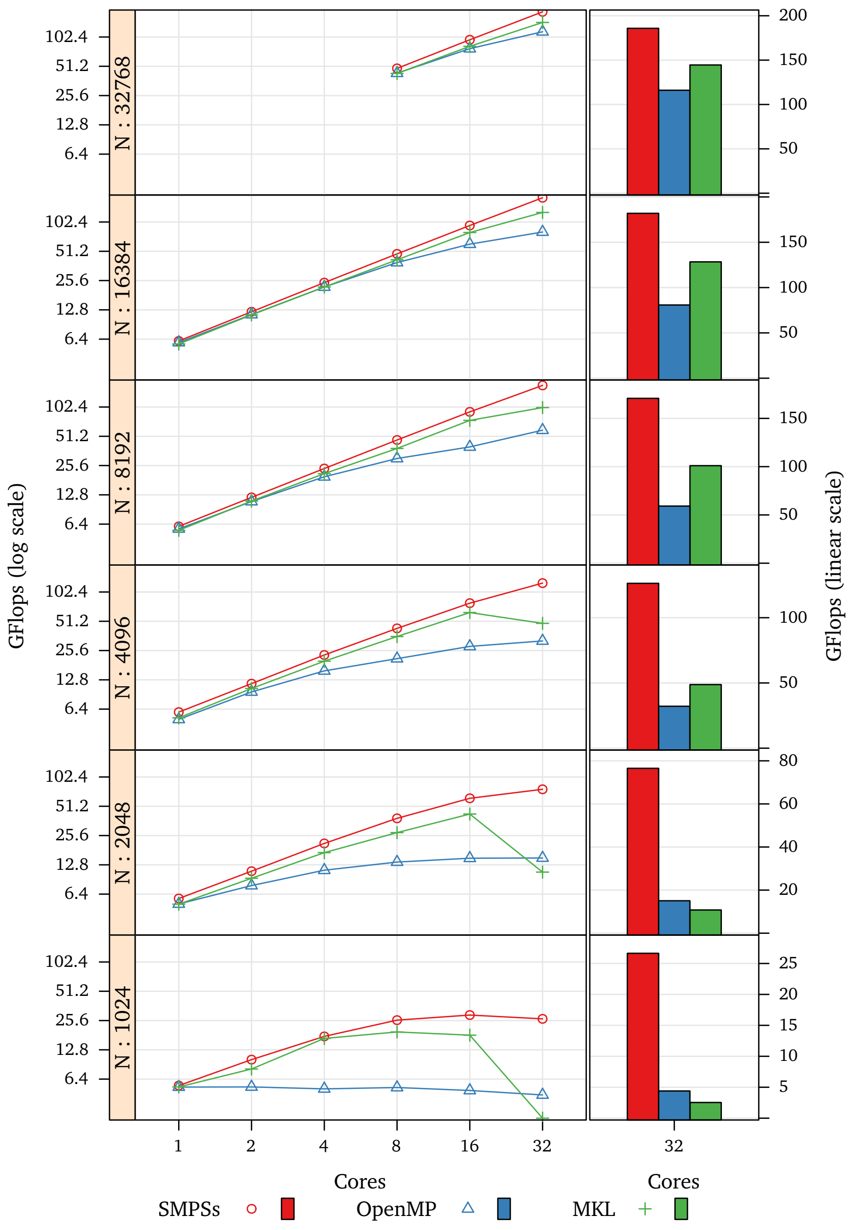


Figure 3.20: Strong scalability of each parallel implementation of Cholesky with several matrix sizes and performance with 32 cores.

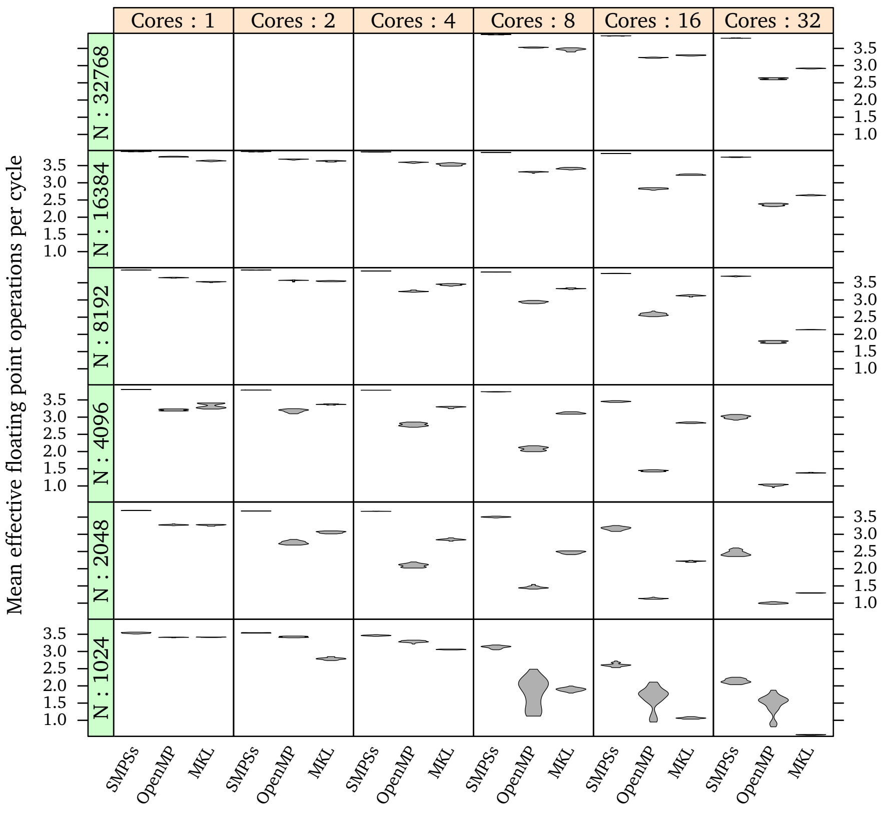


Figure 3.21: Average floating point operations per cycle of each implementation of the Cholesky while running effective work with several matrix sizes.

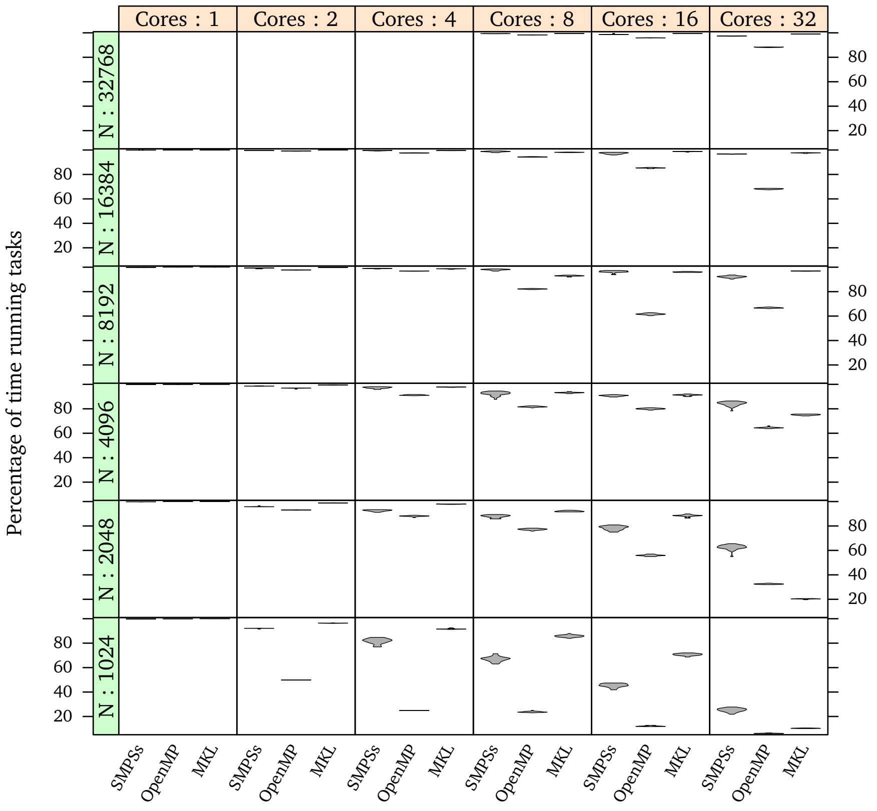


Figure 3.22: Fraction of time that threads spend running the code of the Cholesky with several matrix sizes.

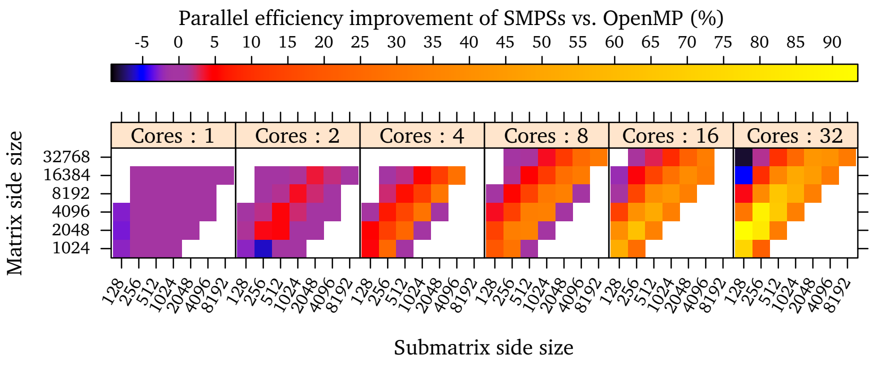


Figure 3.23: Improvement in parallel efficiency between the SMPSs version of Cholesky and the OpenMP version with several matrix sizes at several decomposition granularities.

Cores	N[a]	BS[b]	BS[b]	GF[c]	GF[c]	GF[c]	FPC[d]	FPC[d]	FPC[d]	Eff.[e] (%)	Eff.[e] (%)	Eff.[e] (%)
1	1024	512	1024	5.5	5.3	5.3	3.5	3.4	3.4	99	99	99
1	2048	512	1024	5.8	5.1	5.0	3.7	3.3	3.3	99	99	99
1	4096	512	1024	6.0	5.0	5.2	3.8	3.2	3.3	99	99	99
1	8192	512	2048	6.1	5.7	5.5	3.9	3.6	3.5	99	99	99
1	16384	512	2048	6.1	5.9	5.7	3.9	3.8	3.6	99	99	99
2	1024	128	1024	10	5.3	8.1	3.5	3.4	2.8	92	49	96
2	2048	256	128	11	7.8	9.3	3.7	2.8	3.1	95	93	98
2	4096	256	256	11	9.6	10	3.8	3.2	3.4	98	96	99
2	8192	512	512	12	10	11	3.9	3.6	3.5	99	97	99
2	16384	512	512	12	11	11	3.9	3.7	3.6	99	99	99
4	1024	128	512	17	5.1	16	3.5	3.3	3.1	81	24	91
4	2048	128	128	21	11	17	3.7	2.1	2.8	92	88	97
4	4096	256	256	23	15	19	3.8	2.8	3.3	97	91	97
4	8192	256	256	23	19	21	3.8	3.3	3.4	98	96	98
4	16384	512	512	24	21	21	3.9	3.6	3.5	99	97	99
8	1024	128	256	25	5.2	19	3.1	1.8	1.9	66	23	85
8	2048	128	128	38	13	27	3.5	1.5	2.5	87	77	92
8	4096	256	256	43	21	35	3.7	2.1	3.1	92	81	93
8	8192	256	512	46	30	38	3.8	2.9	3.3	97	82	92
8	16384	512	512	48	39	41	3.9	3.3	3.4	98	94	98
8	32768	512	512	48	43	43	3.9	3.5	3.5	99	98	99
16	1024	128	256	29	4.9	18	2.6	1.7	1.1	45	12	70
16	2048	128	128	61	14	42	3.2	1.1	2.2	78	55	88
16	4096	128	128	78	28	62	3.5	1.4	2.8	90	80	91
16	8192	256	512	91	39	74	3.8	2.6	3.1	96	61	96
16	16384	512	512	94	60	79	3.9	2.8	3.2	97	85	98
16	32768	512	512	95	77	82	3.9	3.2	3.3	98	95	99
32	1024	128	256	26	4.4	2.5	2.1	1.5	0.6	25	6	10
32	2048	128	128	76	15	10	2.5	1.0	1.3	62	32	20
32	4096	128	128	126	32	48	3.0	1.0	1.4	84	64	75
32	8192	256	256	170	59	101	3.7	1.8	2.1	92	66	96
32	16384	256	512	181	80	128	3.7	2.4	2.6	96	68	97
32	32768	512	512	185	116	144	3.8	2.6	2.9	97	88	99

Implementation: SMPSs OpenMP MKL

[a] Matrix side size.
[b] Submatrix side size.
[c] Gigaflops per second.
[d] Mean floating point operations per cycle while running tasks.
[e] Mean time that threads spend running tasks.

Table 3.6: Performance summary of the Cholesky implementations.

3.6.5 Strassen-Winograd

Algorithm

The Strassen algorithm by [Strassen, 1969] performs a matrix multiplication with $O(N^{2.807})$ complexity, where N is equal to the matrix side size in the case of square matrices. The Strassen-Winograd algorithm by [Winograd, 1971] is an improved version of the original algorithm and is the one we evaluate.

Given two matrices A and B of sizes $M \times K$ and $K \times N$ respectively, the algorithm calculates their product C by dividing them by submatrices as follows:

$$\left[\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right] = \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right] \times \left[\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right] \tag{3.13}$$

where the submatrices of A have $(M/2) \times (K/2)$ elements and the submatrices of B have $(K/2) \times (N/2)$ elements. The algorithm calculates the result through the following submatrix operations:

$$\begin{array}{lll} S_1 = A_{21} + A_{22} & & \\ S_2 = S_1 - A_{11} & P_1 = A_{11}B_{11} & U_1 = P_1 + P_2 = C_{11} \\ S_3 = A_{11} - A_{21} & P_2 = A_{12}B_{21} & U_2 = P_1 + P_6 \\ S_4 = A_{12} - S_2 & P_3 = S_4B_{22} & U_3 = U_2 + P_7 \\ T_1 = B_{12} - B_{11} & P_4 = A_{22}T_4 & U_4 = U_2 + P_5 \\ T_2 = B_{22} - T_1 & P_5 = S_1T_1 & U_5 = U_4 + P_3 = C_{12} \\ T_3 = B_{22} - B_{12} & P_6 = S_2T_2 & U_6 = U_3 - P_4 = C_{21} \\ T_4 = T_2 - B_{21} & P_7 = S_3T_3 & U_7 = U_3 + P_5 = C_{22} \end{array} \tag{3.14}$$

The submatrix products $P_1, \ldots, P_7$ can be calculated by applying the algorithm recursively.

While the algorithm uses a total of 22 temporary submatrices, the additional space can be reduced dramatically. [Boyer et al., 2009] show several schedules of the temporary space for the matrix multiplication and the generalized form shown previously in section 3.6.2 ($C' = \alpha A \times B + \beta C$). However, those versions reuse the temporary storage and the storage of the result several times. Since these changes generate false dependencies, if we used one of them, the runtime would automatically rename the reuses and as a result keep using the same space as the original version.

Nevertheless, the algorithm allows some reuse of the temporary storage and the result storage without introducing false dependencies. The algorithm we evaluate is a variant that uses 12 temporary submatrices and also reuses the storage of the result. Figure 3.24 shows a graph of its operations and operands. The nodes correspond to the operations and have their inputs and outputs as input and output edges respectively. The edges are labeled with their storage location.

To simplify the evaluation, we have restricted it to the cases in which the matrices are square ($M = N = K$) and as a consequence also the submatrices ($BSM = BSN = BSK$).

Parallelization with SMPSs

Similarly to the matrix multiplication and Cholesky, the Strassen-Winograd implementation operates over blocked matrices. The intermediate matrix multiplications

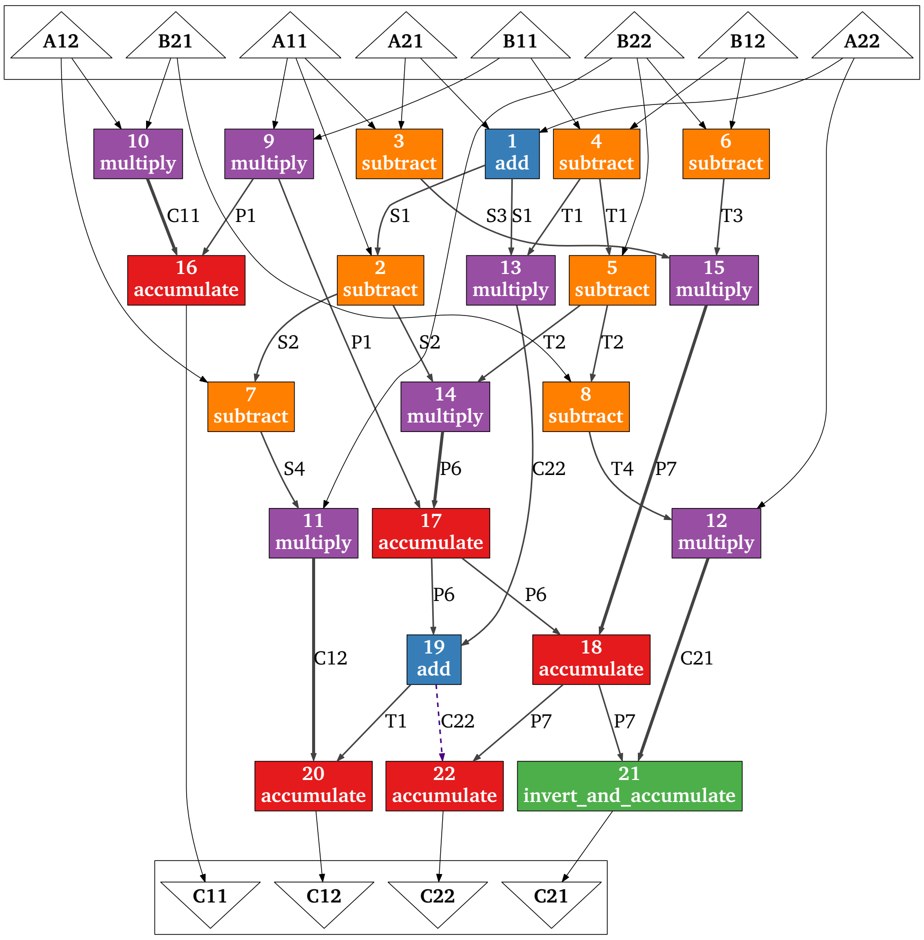


Figure 3.24: Operations, data flow and storage of the Strassen-Winograd variant with 12 temporary submatrices.

```
1  add_iterative(..., a_21, a_22, s1);                 // S1 := A21 + A22
2  subtract_iterative(..., s1, a_11, s2);              // S2 := S1 - A11
3  subtract_iterative(..., a_11, a_21, s3);            // S3 := A11 - A21
4  subtract_iterative(..., b_12, b_11, t1);            // T1 := B12 - B11
5  subtract_iterative(..., b_22, t1, t2);              // T2 := B22 - T1
6  subtract_iterative(..., b_22, b_12, t3);            // T3 := B22 - B12
7  subtract_iterative(..., a_12, s2, s4);              // S4 := A12 - S2
8  subtract_iterative(..., t2, b_21, t4);              // T4 := T2 - B21
9  strassen_multiply(..., a_11, b_11, p1);             // P1 := A11 * B11
10 strassen_multiply(..., a_12, b_21, c_11);           // P2 := A12 * B21 [-> C11]
11 strassen_multiply(..., s4, b_22, c_12);             // P3 := S4 * B22 [-> C12]
12 strassen_multiply(..., a_22, t4, c_21);             // P4 := A22 * T4 [-> C21]
13 strassen_multiply(..., s1, t1, c_22);               // P5 := S1 * T1 [-> C22]
14 strassen_multiply(..., s2, t2, p6);                 // P6 := S2 * T2
15 strassen_multiply(..., s3, t3, p7);                 // P7 := S3 * T3
16 accumulate_iterative(..., p1, c_11);                // C11 := P1 + P2[C11]
17 accumulate_iterative(..., p1, p6);                  // U2 := P1 + P6 [-> P6]
18 accumulate_iterative(..., p6, p7);                  // U3 := U2[P6] + P7 [-> P7]
19 add_iterative(..., p6, c_22, u4);                   // U4 := U2[P6] + P5[C22]
20 accumulate_iterative(..., u4, c_12);                // C12 := U4 + P3[C12]
21 invert_and_accumulate_iterative(..., p7, c_21);  // C21 := U3[P7] - P4[C21]
22 accumulate_iterative(..., p7, c_22);                // C22 := U3[P7] + P5[C22]
```

Listing 3.13: Recursive branch of the Strassen-Winograd implementation in SMPSs.

are solved recursively until the multiplication involves one block per matrix, which is solved as a standard matrix multiplication. All other operations are decomposed into block operations in an iterative manner.

Listing 3.13 shows the recursive branch of the Strassen-Winograd implementation. Line numbers correspond to the numbers of the nodes from figure 3.24. Comments on the right indicate the operation and show the actual storage in square brackets if different to the actual mathematical definition in equations 3.14.

The recursion is controlled by the matrix size and the block size. When the strassen_multiply function is called with a matrix that has the same size as the block size, then it is implemented as a normal matrix multiplication task instantiation, otherwise it decomposes it using the code shown in listing 3.13. All calculations that are not multiplications are implemented by functions that decompose those operations by blocks.

Parallelization with OpenMP

To compare the performance of the algorithm in SMPSs to other approaches we have made a second implementation in OpenMP using tasks and nesting. Listing 3.14 shows the code of its recursive branch. Note that since this is a Strassen-Winograd implementation, there is some memory reuse, and thus, there are dependencies between the calculations of the decomposition that are not present in the original Strassen algorithm. Hence, the implementation contains 6 barriers, whereas the original Strassen would have required only 3. The tasks and the barriers match the rows of the graph in figure 3.24. For each multiply operation in the graph, the

```
#pragma omp task untied
add_iterative(..., a_21, a_22, s1);
#pragma omp task untied
subtract_iterative(..., a_11, a_21, s3);
#pragma omp task untied
subtract_iterative(..., b_12, b_11, t1);
#pragma omp task untied
subtract_iterative(..., b_22, b_12, t3);
#pragma omp task untied
strassen_multiply(..., a_11, b_11, p1);
#pragma omp task untied
strassen_multiply(..., a_12, b_21, c_11);

#pragma omp taskwait

#pragma omp task untied
subtract_iterative(..., s1, a_11, s2);
#pragma omp task untied
subtract_iterative(..., b_22, t1, t2);
#pragma omp task untied
strassen_multiply(..., s1, t1, c_22);
#pragma omp task untied
strassen_multiply(..., s3, t3, p7);
#pragma omp task untied
accumulate_iterative(..., p1, c_11);

#pragma omp taskwait

#pragma omp task untied
subtract_iterative(..., a_12, s2, s4);
#pragma omp task untied
subtract_iterative(..., t2, b_21, t4);
#pragma omp task untied
strassen_multiply(..., s2, t2, p6);

#pragma omp taskwait

#pragma omp task untied
strassen_multiply(..., s4, b_22, c_12);
#pragma omp task untied
strassen_multiply(..., a_22, t4, c_21);
#pragma omp task untied
accumulate_iterative(..., p1, p6);

#pragma omp taskwait

#pragma omp task untied
accumulate_iterative(..., p6, p7);
#pragma omp task untied
add_iterative(..., p6, c_22, u4);

#pragma omp taskwait

#pragma omp task untied
accumulate_iterative(..., u4, c_12);
#pragma omp task untied
invert_and_accumulate_iterative(..., p7,
   c_21);
#pragma omp task untied
accumulate_iterative(..., p7, c_22);

#pragma omp taskwait
```

Listing 3.14: Recursive branch of the Strassen-Winograd implementation in OpenMP.

OpenMP code has a task that may be a recursive Strassen-Winograd task or a simple block multiplication task. For each of the others, it has a task that implements the operation iteratively with tasks over the blocks. Dependencies are respected by placing an OpenMP **taskwait** directive between the rows of the graph.

The recursion is controlled by the same parameters as in the SMPSs implementation. However, the OpenMP version takes advantage of task nesting by using tasks to launch the strassen_multiply function. Moreover, the tasks that decompose the operations that are not matrix multiplications are generated in parallel, similarly to the previous algorithms. Finally, since OpenMP does not require to physically block the matrices, they are stored using a standard flat layout in memory.

Both versions use MKL in sequential mode for the submatrix operations.

Determining the Submatrix Dimensions

Like in previous algorithms, the value of the submatrix side size BS also determines the number of tasks of the problem decomposition and the computational weight of each task. Figure 3.25 shows the floating point performance of the SMPSs implementation in terms of the standard matrix multiplication. That is, the values are

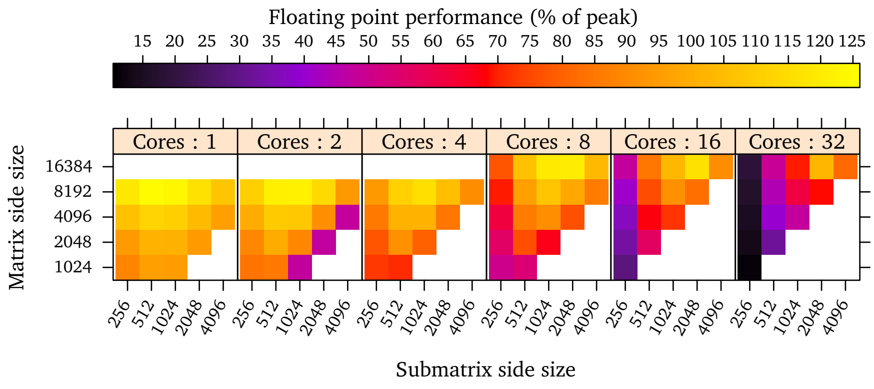


Figure 3.25: Performance of the SMPSs Strassen implementation with several matrix and blocking sizes.

calculated as if the algorithm performed $2N^3$ floating point operations. Hence the value can be above the hardware peak. To explain the performance displayed in this figure we analyze several aspects and metrics that affect the final performance.

The graph of the Strassen algorithm is complex like the Cholesky one, but it widens faster when we decrease the blocking size. For instance, the graph with just one recursion level is identical to the one in figure 3.24. However, when we go through two recursion levels, we obtain the graph in figure 3.26, which is so wide that it has been rotated to make it fit in the page (dependencies go from left to right).

Figure 3.27 indicates that several configurations cause starvation. The diagonal configurations correspond to executions with just one matrix multiplication for the cases with 1 and 2 cores, with just one recursion level for 4 and 8 cores, and for two levels of recursion with 16 and 32 cores. In those cases, the amount on parallelism is too small. Figure 3.28 shows that the rest of the configurations with poor performance and high idle time occur when the main thread has a high amount of overhead.

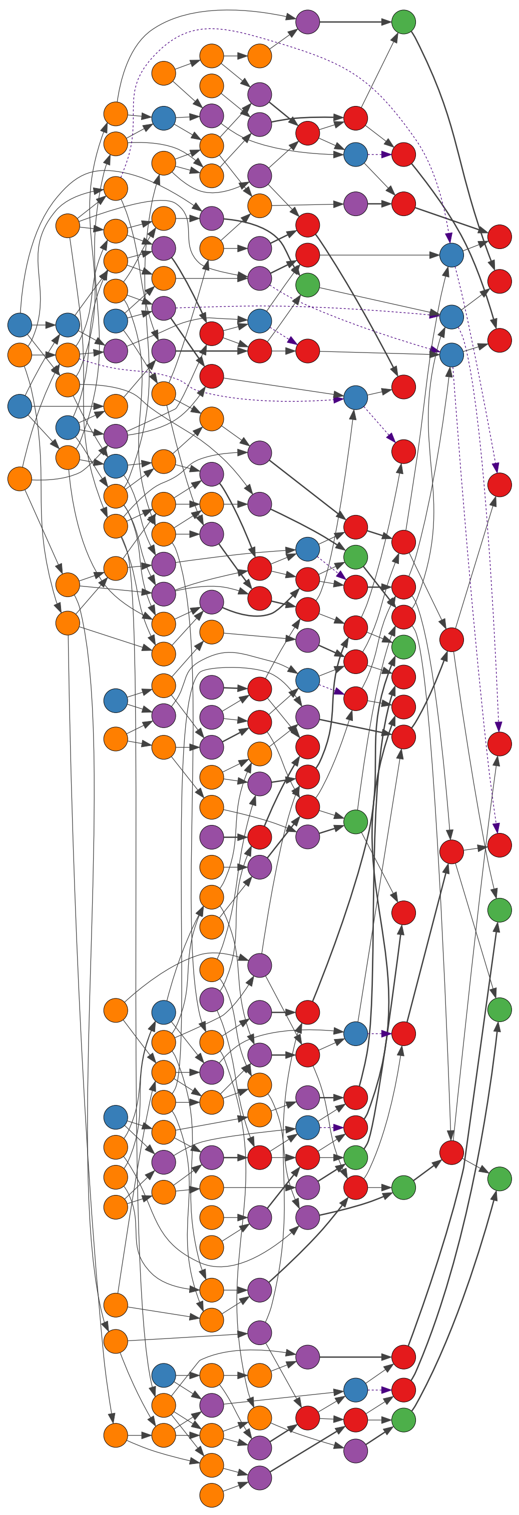
The mean task floating point operations per cycle are shown in figure 3.29. As opposed to the previous experiments, the Strassen's task floating point performance improves steadily when we increase the submatrix size.

Table 3.7 summarizes the performance of the best submatrix sizes. Notice that with one thread the best submatrix size for matrices is always smaller than the whole matrix size. Thus, the Strassen algorithm is faster than the regular matrix multiplication for every problem size that we tried.

Scheduling

Figure 3.30 shows the mean floating point performance of the application under several strong scalability scenarios with each scheduler. Each point corresponds to the submatrix size that produced the best mean performance. None of the scheduling policies scales well for the smallest matrix size, and in general they perform similarly. These results indicate that either the schedulers are too simple to exploit data reuse effectively, or that the dependencies do not allow it.

Table 3.8 summarizes the mean values of the main performance metrics of the Strassen algorithm with each scheduler.



Node	Task
🔴	accumulate
🔵	add
🟢	invert_and_accumulate
🟣	multiply
🟠	subtract

Figure 3.26: Task graph of the SMPSs Strassen implementation with matrices divided into 4 × 4 submatrices, laid out from left to right.

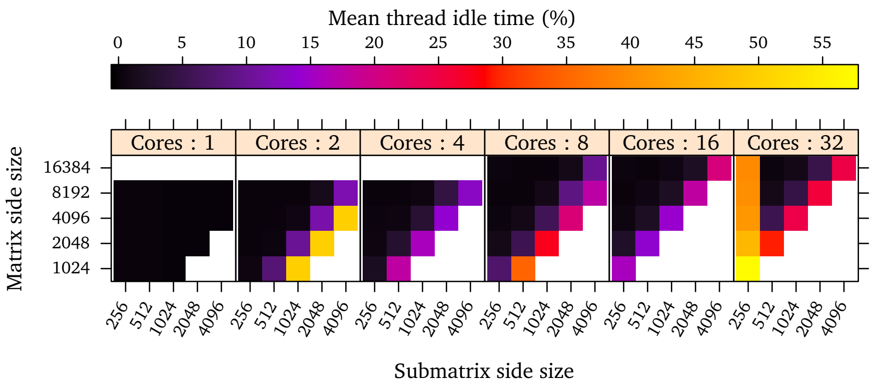


Figure 3.27: Average time taken by each thread idling in the SMPSs Strassen implementation with several matrix and blocking sizes.

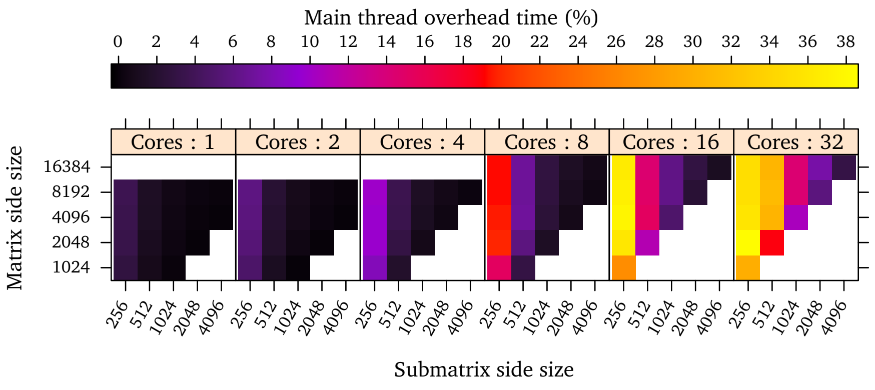


Figure 3.28: Percentage of time time that the main thread spends managing tasks when running the SMPSs Strassen implementation with several matrix and blocking sizes.

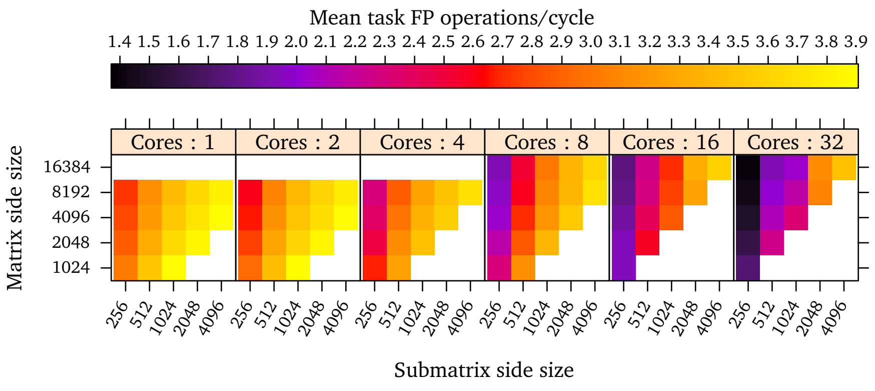


Figure 3.29: Mean floating point operations per cycle while running each task of the SMPSs Strassen implementation with several matrix and blocking sizes.

Cores	N[a]	BS[b]	Tasks	GF[c]	IPC[d]	FPC[e]	Ovh.[f] (%)	Eff.[g] (%)	Idle (%)	Tr.O.[h] (%)
1	1024	512	22	6.1	5.32	3.51	0.75	99	0	0
1	2048	512	214	6.5	5.04	3.32	1.02	98	0	0
1	4096	512	1738	7.2	4.88	3.21	1.19	98	0	0
1	8192	512	13126	8.0	4.77	3.13	1.30	98	0	0
2	1024	512	22	10.9	5.22	3.45	1.12	91	7	0
2	2048	512	214	12.7	4.97	3.28	1.67	98	0	0
2	4096	512	1738	14.0	4.80	3.15	1.93	98	0	0
2	8192	1024	1738	15.6	5.14	3.40	0.71	99	0	0
4	1024	256	214	18.5	4.11	2.67	8.35	94	1	0
4	2048	512	214	23.3	4.74	3.12	2.94	96	2	0
4	4096	1024	214	26.0	5.10	3.37	1.11	96	3	0
4	8192	1024	1738	29.6	4.93	3.26	1.34	99	0	0
8	1024	512	22	27.5	4.78	3.15	2.95	64	34	0
8	2048	512	214	39.0	4.32	2.84	5.81	92	5	0
8	4096	1024	214	46.7	4.78	3.16	2.35	93	5	0
8	8192	1024	1738	54.8	4.68	3.09	2.70	98	0	0
8	16384	2048	1738	61.6	5.09	3.38	1.28	98	0	0
16	1024	256	214	29.2	2.99	1.94	26.66	79	15	0
16	2048	512	214	58.2	3.91	2.57	11.08	84	13	1
16	4096	1024	214	73.8	4.32	2.86	4.86	84	14	0
16	8192	1024	1738	93.7	4.17	2.75	6.18	97	1	0
16	16384	2048	1738	119.1	5.00	3.32	2.87	97	1	0
32	1024	256	214	23.6	2.67	1.74	29.80	39	57	-1
32	2048	512	214	65.8	3.41	2.24	18.79	68	29	1
32	4096	1024	214	97.9	3.52	2.33	10.46	74	24	1
32	8192	2048	214	140.4	4.61	3.06	5.73	74	25	2
32	16384	2048	1738	209.8	4.69	3.11	7.53	94	4	1

[a] Matrix side size..
[b] Maximum submatrix side size for addition and subtraction tasks
[c] GFlops.
[d] Mean instructions per cycle while running tasks.
[e] Mean floating point operations per cycle while running tasks.
[f] Time that the main thread spends generating tasks and idle.
[g] Mean time that threads spend running tasks.
[h] Increment of the execution time when enabling tracing.

Table 3.7: Best submatrix side sizes for the SMPSs Strassen implementation and their performance characteristics.

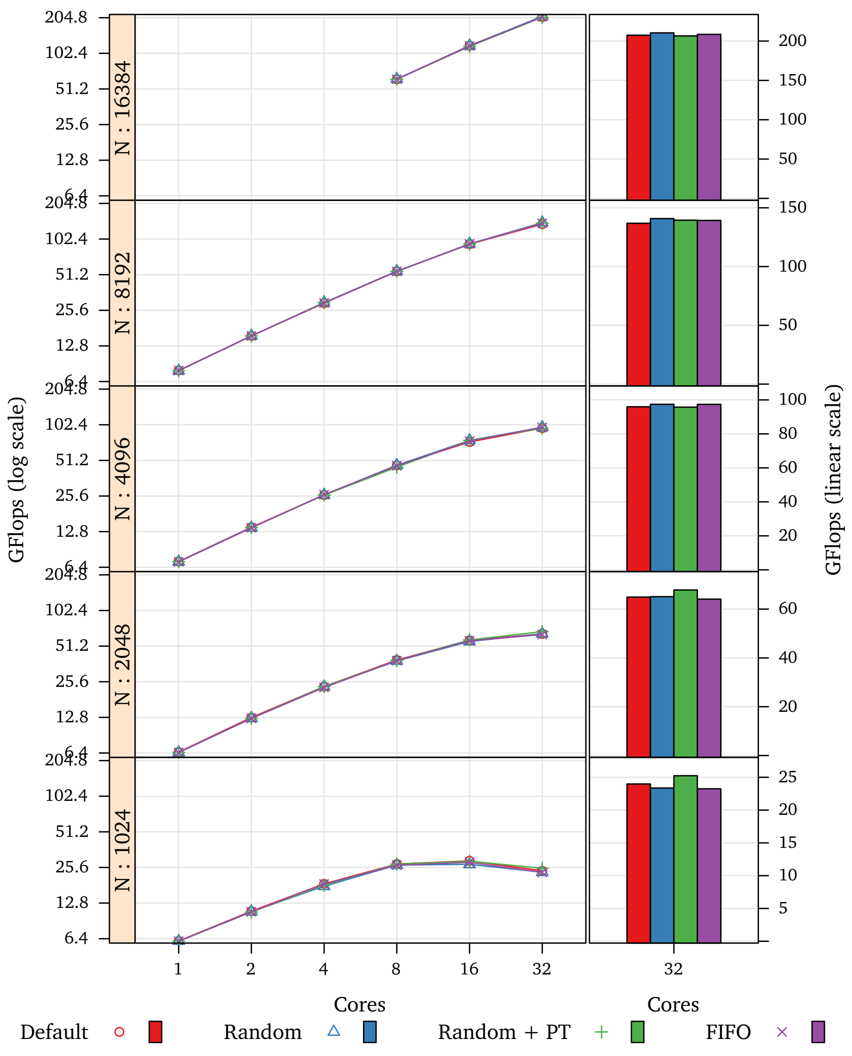


Figure 3.30: Strong scalability of the SMPSs Strassen implementation with several matrix sizes under each scheduling policy and performance with 32 cores.

Cores	N[a]	GF[b]	GF[b]	GF[b]	GF[b]	FPC[c]	FPC[c]	FPC[c]	FPC[c]	Eff.[d] (%)	Eff.[d] (%)	Eff.[d] (%)	Eff.[d] (%)
1	1024	6.1	6.1	6.1	6.1	3.5	3.5	3.5	3.5	99	98	98	98
1	2048	6.5	6.5	6.5	6.5	3.3	3.3	3.3	3.3	98	98	98	98
1	4096	7.2	7.1	7.1	7.1	3.2	3.2	3.2	3.2	98	98	98	98
1	8192	7.9	7.9	7.9	7.9	3.1	3.1	3.1	3.1	98	98	98	98
2	1024	10	10	10	10	3.4	3.5	3.5	3.4	91	89	89	90
2	2048	12	12	12	12	3.3	3.2	3.3	3.2	98	98	97	98
2	4096	13	13	13	13	3.2	3.1	3.1	3.1	98	98	98	98
2	8192	15	15	15	15	3.4	3.4	3.4	3.4	99	99	99	99
4	1024	18	17	18	18	2.7	2.6	2.6	2.7	94	93	92	93
4	2048	23	23	23	22	3.1	3.1	3.1	3.1	96	96	96	95
4	4096	26	26	26	26	3.4	3.4	3.4	3.4	96	97	97	97
4	8192	29	29	29	29	3.3	3.3	3.3	3.3	99	99	99	99
8	1024	27	26	27	26	3.2	3.1	3.2	3.1	64	64	64	64
8	2048	39	38	38	38	2.8	2.8	2.8	2.8	92	91	92	92
8	4096	46	46	44	46	3.2	3.2	3.2	3.2	93	93	92	92
8	8192	54	54	54	54	3.1	3.1	3.1	3.1	98	98	98	98
8	16384	61	62	61	61	3.4	3.4	3.4	3.4	98	98	99	98
16	1024	29	27	28	28	1.9	1.8	2.0	1.9	79	75	73	75
16	2048	57	56	57	56	2.6	2.5	2.6	2.5	84	83	84	84
16	4096	73	75	75	74	2.9	2.9	2.9	2.9	84	86	86	85
16	8192	92	93	92	93	2.8	2.8	2.7	2.8	97	97	97	97
16	16384	118	118	116	118	3.3	3.3	3.3	3.3	97	97	97	97
32	1024	23	23	25	23	1.7	1.7	1.8	1.6	39	37	37	38
32	2048	64	65	67	64	2.2	2.2	2.3	2.2	68	68	68	67
32	4096	95	97	95	97	2.3	2.4	2.4	2.3	74	74	72	74
32	8192	136	140	139	139	3.1	3.1	3.1	3.1	74	74	74	74
32	16384	207	210	206	208	3.1	3.2	3.1	3.1	94	94	94	94

Scheduler: Default Random Random + PT FIFO

[a] Matrix side size.
[b] GFlops.
[c] Mean floating point operations per cycle while running tasks.
[d] Mean time that threads spend running tasks.

Table 3.8: Performance summary of the scheduler on the SMPSs Strassen implementation.

Performance of the Implementations

Figure 3.32 shows the mean floating point performance of both implementations under several strong scalability scenarios. The measurements have been selected from the ones with submatrix sizes that produced the best mean performance. In all cases the SMPSs implementation scales better than the OpenMP version. By taking advantage of dependencies, SMPSs is able to find parallelism that cannot be exploited through task nesting.

Performance decreases with the number of cores with both implementations, since the amount of work per core is smaller and so is the potential IPC. Figure 3.33 shows the average task floating point operations per cycle and figure 3.34 shows the mean fraction of time that threads spend running tasks. Both figures show that parallelism and floating point operation per cycle decrease as the number of cores increases. However, since the SMPSs implementation finds more parallelism, it can keep threads busy during more time and thus it can choose bigger block sizes when the problem is big enough, and improve its task performance as a result. Figure 3.31 shows in detail for each problem and blocking size the effective parallelism gained by using dependencies in SMPSs instead of task nesting with OpenMP.

Table 3.9 summarizes the means of the main performance metrics of each implementation. Notice that the SMPSs version when running with more than one core has more parallel efficiency in all cases, and for the biggest problems uses bigger submatrices than the OpenMP version. These results show that task dependencies are able to extract more parallelism than task nesting and can perform better as a result, despite the additional overhead of calculating them.

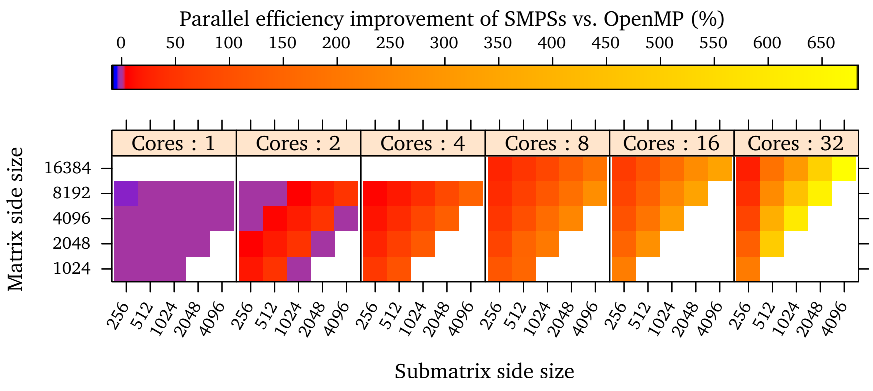


Figure 3.31: Effective parallelism improvement of the Strassen algorithm in SMPSs with dependencies compared to OpenMP with task nesting.

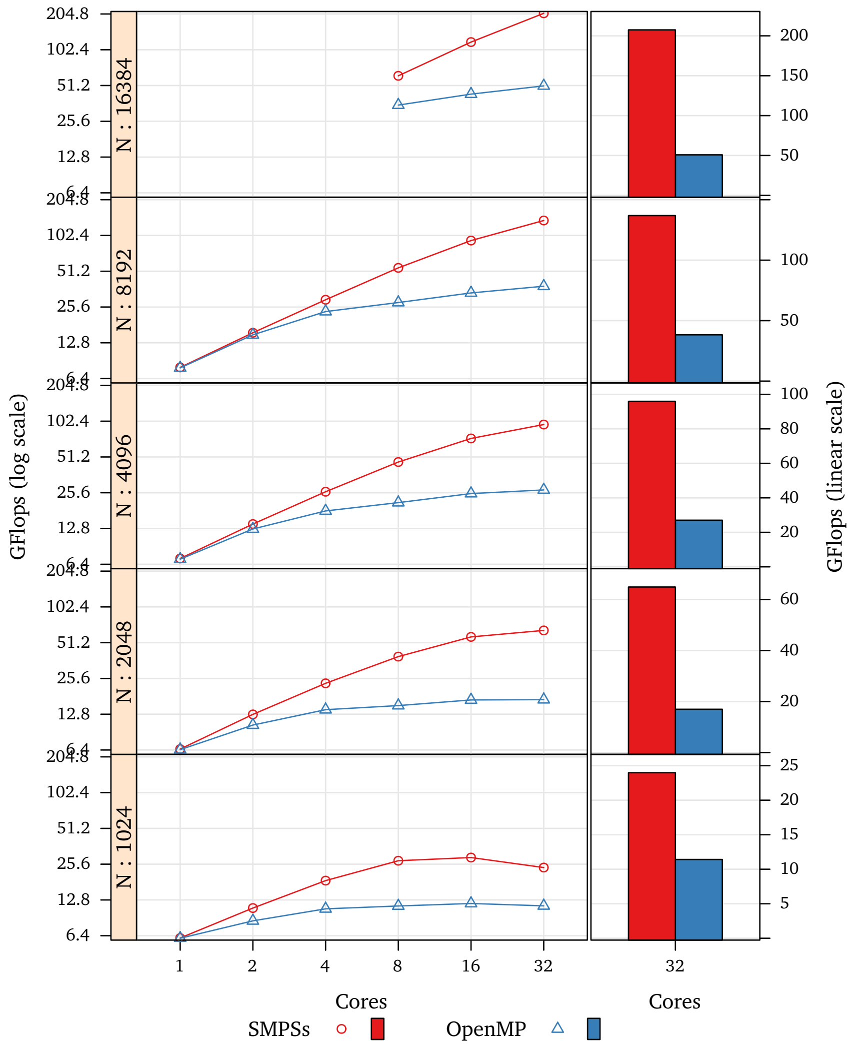


Figure 3.32: Strong scalability of each implementation of Strassen with several matrix sizes and performance with 32 cores.

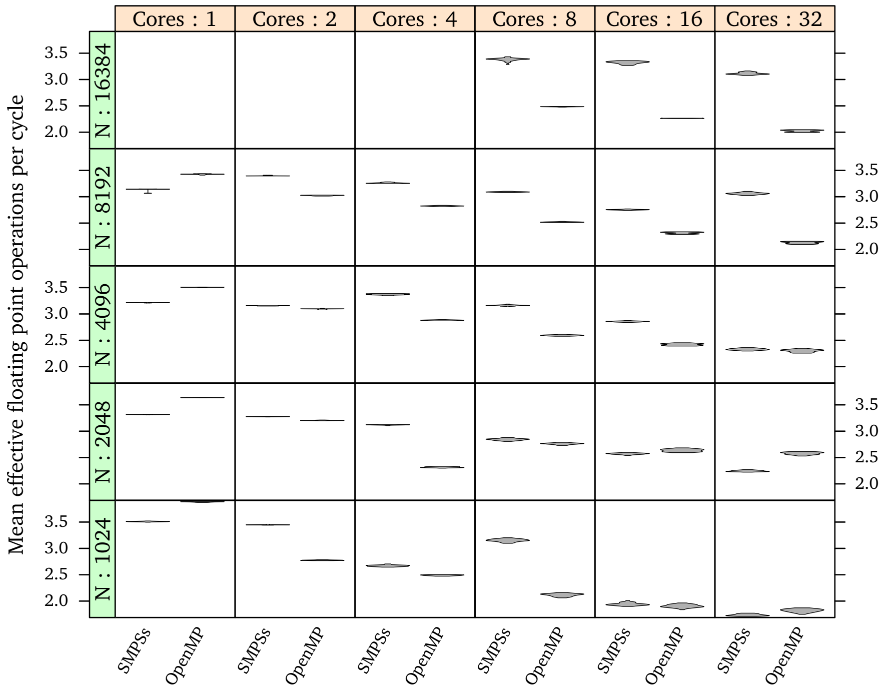


Figure 3.33: Average floating point operations per cycle of each implementation of Strassen while running effective work with several matrix sizes.

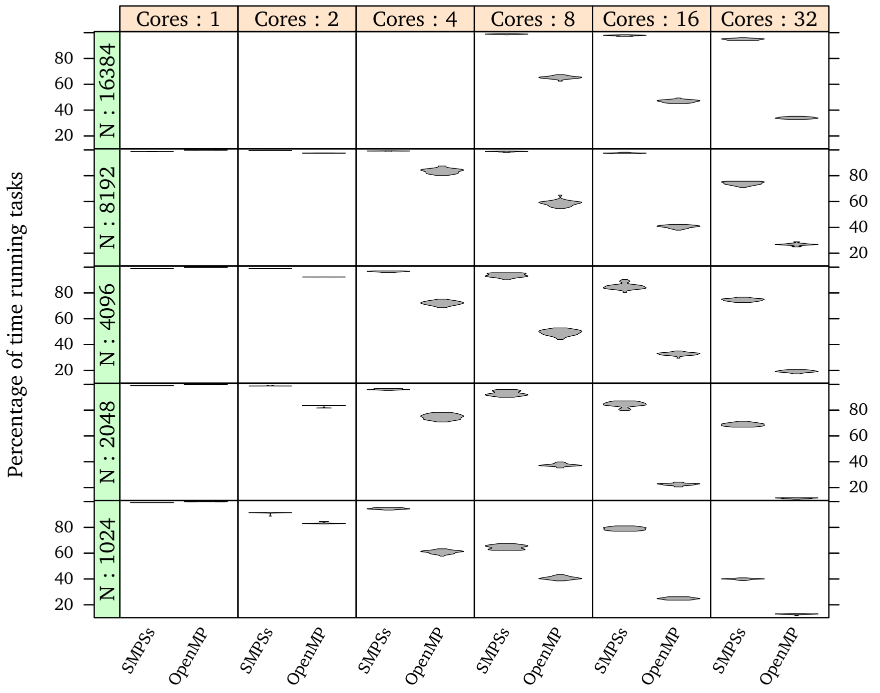


Figure 3.34: Fraction of time that threads spend running the Strassen code on each implementation with several matrix sizes.

Cores	N[a]	BS[b]	BS[b]	GF[c]	GF[c]	FPC[d]	FPC[d]	Eff.[e] (%)	Eff.[e] (%)
1	1024	512	1024	6.1	6.1	3.51	3.88	99	99
1	2048	512	1024	6.5	6.4	3.32	3.64	98	99
1	4096	512	1024	7.2	7.1	3.21	3.51	98	99
1	8192	512	1024	7.9	7.9	3.13	3.43	98	99
2	1024	512	256	10.9	8.5	3.45	2.77	91	83
2	2048	512	512	12.8	10.4	3.28	3.20	98	83
2	4096	512	512	14.0	12.7	3.15	3.10	98	92
2	8192	1024	512	15.5	14.9	3.40	3.02	99	97
4	1024	256	256	18.6	10.8	2.67	2.49	94	60
4	2048	512	256	23.3	14.0	3.12	2.32	96	74
4	4096	1024	512	26.1	18.0	3.37	2.88	96	72
4	8192	1024	512	29.5	23.4	3.26	2.82	99	83
8	1024	512	256	27.4	11.4	3.15	2.12	64	40
8	2048	512	512	39.1	15.1	2.84	2.76	92	37
8	4096	1024	512	46.4	21.1	3.16	2.59	93	49
8	8192	1024	512	54.7	27.9	3.09	2.52	98	58
8	16384	2048	512	61.8	35.0	3.38	2.48	98	65
16	1024	256	256	29.1	12.0	1.94	1.91	79	24
16	2048	512	512	57.3	16.9	2.57	2.64	84	22
16	4096	1024	512	73.3	25.2	2.86	2.42	84	32
16	8192	1024	512	92.8	33.6	2.75	2.31	97	40
16	16384	2048	512	118.8	43.3	3.32	2.26	97	46
32	1024	256	256	24.0	11.4	1.74	1.82	39	12
32	2048	512	512	64.9	17.0	2.24	2.58	68	11
32	4096	1024	512	96.0	27.0	2.33	2.30	74	19
32	8192	2048	512	136.9	38.3	3.06	2.13	74	26
32	16384	2048	512	207.5	50.8	3.11	2.02	94	33

Implementation: SMPSs OpenMP

[a] Matrix side size.
[b] Submatrix side size
[c] GFlops.
[d] Mean floating point operations per cycle while running tasks.
[e] Mean time that threads spend running tasks.

Table 3.9: Performance summary of the Strassen implementations.

Chapter 4

Strided and Overlapping Data Accesses

4.1 Introduction

Productivity is an important aspect of High Performance Computing. It is usually defined as a relation between the time and effort that it takes to develop a code, and the performance of the solution. The SMPSs programming model has been designed to find a balance between those two factors. On one hand, it reduces the development effort by providing just one parallel construct, by following the sequential execution semantics, by basing the parallelization in code annotations and by moving the effort analyzing data dependencies form the programmer to the programming model. On the other hand, it strives to obtain more performance by trading the overhead of data dependency analysis in exchange of potentially more parallelism. All these aspects have already been covered in the previous chapter.

Despite these advantages, the model has tight requirements on data layout and data access that limit its performance and applicability. First, the data that the tasks access must be contiguous in memory. And second, those accesses must be performed over segments of contiguous memory that do not overlap.

These restrictions affect the data layout of applications. For instance, the programmer may need to physically block the data. This is the case of several of the test cases of the previous chapter, that use hyper-matrices instead of plain matrices.

The limitations also affect how the code accesses the data. For instance, the Gauss-Seidel task from the previous chapter in listing 3.8 (page 37) has many parameters and its code is more complex than its sequential version due to the data blocking. The need to make data layout changes and the complexity that it introduces could be avoided by providing means to handle flat data structures.

Using the same blocking size for all tasks has also an impact on performance. This is usually the case of applications that have tasks with very different computational costs. On one hand, raising the blocking size may improve the performance of fast tasks, but it may make the slower tasks too small and too few, which may cause imbalance. On the other hand, a blocking size tuned to the parallelism of slow tasks, may produce too many of the faster tasks that may be too fast and lead to too much runtime overhead. This case affects the HPL benchmark that we evaluate in this chapter.

Allowing different tasks to use different granularities even if they operate over the same data would compensate the computational complexity mismatch and its effects. In practice this requires non fixed blocking sizes and overlapping accesses.

Finally, some algorithms may be hard to express using a fixed and contiguous memory layout. This sort of algorithms includes cases with several phases that accesses the data in a different ways. For instance a matrix may be accessed by rows, by columns and by tiles at different stages of the algorithm. This is the case of the Fast Fourier Transform implementation we present in the evaluation section using the Matrix Fourier Algorithm decomposition.

The rest of this chapter is structured as follows. Section 4.2 discusses how to extend the syntax to allow expressing strided and overlapping data accesses. Section 4.3 explores how to represent such information within the runtime. Section 4.4 explores how to use that representation to organize the information needed to calculate data dependencies efficiently. And section 4.5 evaluates the benchmarks of the previous chapter under the new conditions and extends the analysis with algorithms that could not be expressed or that had compromised solutions. In addition, appendix D shows the changes needed in the runtime API and the compiler to support regions respectively. For brevity most of the benchmarks of the previous chapter have been moved to appendix E.

Thus, the contributions of this chapter are (1) the SMPSs language extensions, (2) a compact representation of a set of memory addresses, (3) an algorithm to check for intersections between two sets, (4) a searchable and updatable data structure to store information about memory data sets with an efficient intersection search operation, and (5) an evaluation of the performance obtained on a set of algorithms under SMPSs with such extensions.

4.2 Language-Level Array Regions

Some programming languages include the means to specify subsets of arrays. However, their purpose and semantics are different from one to the other. In Fortran, programmers can pass a subset of an array to a function, which in turn can use a different indexing to access it. While the syntax is fully specified, the semantics are implementation-specific. In many implementations, passing an array subset has the semantics of a partial copy to a temporary array, and thus changes to that data get discarded when the function call finishes. In others it does not have copy semantics, and thus the changes are reflected on the original data.

Partitioned Global Address Space (PGAS) languages also allow the programmer to specify subsets of an array, but they play a more central role. Their purpose is to define how arrays are distributed between the different memories; to define iteration spaces; and to establish the breadth of global array operations.

In this section we refer to array subset as *array regions*. In the Fortran literature they are referred to as *array sections*. In the PGAS area, the Chapel language by [Chamberlain et al., 2007] and Titanium by [Yelick et al., 1998] call them *domains*. The X10 language by [Charles et al., 2005] calls them *regions*.

In SMPSs, array regions are useful for defining partial accesses to arrays. For instance, a task may update a limited number of columns of a matrix each time. Without array regions, the programmer cannot specify that kind of access and is forced to either declare the matrix as a flat matrix and to declare the accesses as full matrix accesses, or to resort to using barriers. Both cases limit the ability of the model to extract parallelism.

PGAS languages define array regions as language-level classes that must conform to a standard interface that defines the operations that the region must have. These interfaces include set-like operations like the intersection. This approach is very flexible since it allows the scope and behavior to be unrestricted. However, it comes at the expense of too much overhead for our purposes. For instance, most operations related to array regions consist in traversing them. Therefore, the cost of an intersection operation between arbitrary regions, which in X10 consists in generating all the indexes of one array region, checking their inclusion in the other, and adding them as a list to the result, is usually in line with the cost of their traversal.

In SMPSs, array regions are used for determining dependencies, and thus their main operation is the intersection. Since a solution similar to the one used by PGAS languages incurs in too much overhead, the solution must constrain the expressiveness to allow the intersection operation to be implemented more efficiently.

To this degree, an SMPSs array region is defined as one range of values for each dimension, that determines the indexes of the elements contained in the region.

To support the specification of array accesses, the grammar has been extended as follows:

(11) ⟨task-parameter⟩ →
(11.1) ⟨identifier⟩ ⟨opt-task-parameter-dimensions⟩ ⟨opt-region⟩

(12) ⟨opt-task-parameter-dimensions⟩ →
(12.1) ⟨task-parameter-dimensions⟩
(12.2) |

(13) ⟨task-parameter-dimensions⟩ →
(13.1) ⟨task-parameter-dimensions⟩ ⟨task-parameter-dimension⟩
(13.2) | ⟨task-parameter-dimension⟩

(14) ⟨task-parameter-dimension⟩ →
(14.1) **[** ⟨expression⟩ **]**

(15) ⟨opt-region⟩ →
(15.1) ⟨region-specifiers⟩
(15.2) |

(16) ⟨region-specifiers⟩ →
(16.1) ⟨region-specifiers⟩ ⟨region-specifier⟩
(16.2) | ⟨region-specifier⟩

(17)	⟨region-specifier⟩ →
(17.1)	**{ }**
(17.2)	| **{** ⟨expression⟩ **}**
(17.3)	| **{** ⟨expression⟩ **..** ⟨expression⟩ **}**
(17.4)	| **{** ⟨expression⟩ **:** ⟨expression⟩ **}**

In the new grammar, parameters may have region specifiers (15) in addition to the optional dimension specifiers (11). A region specifier is a list of dimension range specifiers (16), one for each dimension and in the same order as the dimension specifiers.

Dimension range specifiers indicate the indexes accessed in the given dimension. They appear in the same order as the dimensions, and have four different forms. The first form (17.1) indicates that the dimension is fully accessed. A single element can be specified (17.2). A subset of the dimension can be specified as a pair of first element and last inclusive element (17.3) or as a pair of first element and length (17.4).

Since array regions allow to specify partial array accesses, as opposed to full-array accesses, a task instance may access several parts of an array with several types of access. Each appearance of a parameter in a directionality clause declares a *region access*.

Listing 4.1 contains a matrix multiplication similar to the one evaluated in the previous chapter, but performed over a flat matrix. Figure 4.1 contains a representation of its data accesses. A, B and C (line 9) are matrices of $N \times N$ doubles. The code operates in regions of $L \times L$ elements inside the matrices. These are analogous to the blocks in the previous chapter. The *matmul* task receives pointers to the beginning of each region (line 14). The directionality clauses in line 1 indicate that the task reads a region of $L \times L$ elements of a, starting from *a[0][0]*, an identical one from b, and that it reads and writes another one for c. The only data that produces dependencies is matrix C. The innermost loop in line 13 will generate chains of tasks since each iteration updates the same part of C.

While previous versions of SMPSs would require using hyper-matrices, by enabling regions this is no longer necessary. Moreover, by applying to regions the same principles as the Variable Length Arrays described by the [International Organization for Standardization and International Electrotechnical Commission, 1999], the region size (and thus the size of the task) can be determined at run time or even dynamically. This can be done by passing L as a parameter to the task and it does not require any change to the data layout.

Tasks may have more than one region access per parameter. Listing 4.2 shows a version of the Gauss-Seidel benchmark of the previous chapter but this time using a flat matrix and regions instead of a blocked matrix. The *gauss_seidel* task has 5 different regions that are also represented in figure 4.2(a). The regions that appear in the *input* clause correspond to the top row, the bottom row, the left column, and the right column of the stencil respectively. The *inout* clause contains the region of the inner square.

Figure 4.2(b) represents the regions accessed by a task over the whole array at some point during the execution. Careful inspection of the figure and the code reveals that each task call has at least two of the outer regions overlapping the inner square of its neighbor tasks (the task immediately to the north, south, east or west), and its inner square overlapping an outer region of those tasks. Those overlaps determine the dependencies between the task invocations.

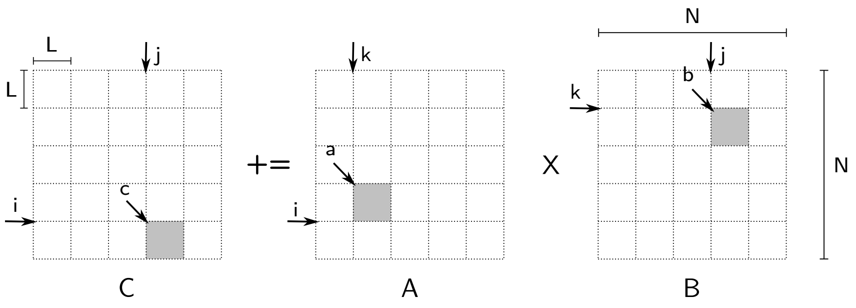


Figure 4.1: Structure and regions used in the matmul_task.

```
#pragma css task input(a{0:L}{0:L}, b{0:L}{0:L}) inout(c{0:L}{0:L})
void matmul(double a[N][N], double b[N][N], double c[N][N]) {
for (int i=0; i < L; i++)
    for (int j=0; j < L; j++)
        for (int k=0; k < L; k++)
            c[i][j] += a[i][k] + b[k][j];
}

double A[N][N], B[N][N], C[N][N];
...
for (int i=0; i < N; i+=L)
    for (int j=0; j < N; j+=L)
        for (int k=0; k < N; k+=L)
            matmul(&A[i][k], &B[k][j], &C[i][j]);
```

Listing 4.1: A matrix multiplication example that uses regions.

```
#pragma css task \
    input(a{0}{1:L}, a{L+1}{1:L}, a{1:L}{0}, a{1:L}{L+1}) \
    inout(a{1:L}{1:L})
void gauss_seidel(double a[N][N]) {
    for (int i=1; i<=L; i++)
        for (int j=1; j<=L; j++)
            a[i][j] = 0.2 * (a[i][j] + a[i-1][j] + a[i+1][j] + a[i][j-1] + a[i][j+1]);
}

double data[N][N];
...
for (int it=0; it < NITERS; it++)
    for (int i=0; i<N-2; i+=L)
        for (int j=0; j<N-2; j+=L)
            gauss_seidel(&data[i][j]);
```

Listing 4.2: An example showing code that uses regions to specify accesses over a stencil.

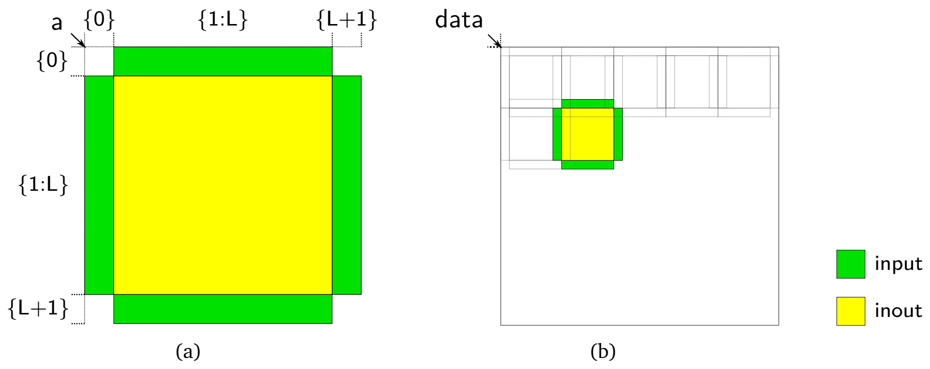


Figure 4.2: Regions used in the gauss_seidel task: (a) from the point of view of the task; and, (b) from the point of view of the caller.

Notice that in both examples we pass pointers to a starting position within the array and we are not restricted to always pass the base address. In a similar way, the language does not impose other restrictions on other techniques used in plain C like pointer arithmetic or altering the array dimensions.

While the declaration of the Gauss-Seidel task is more complex when using regions than when using the blocked layout (see page 38), it improves the programmability of the code of the task. More specifically where the blocked code had a conditional for every access to an element of the halo that surrounds the element that is updated, the version using region does not, and thus it is simpler and closer to the original sequential code.

4.3 Data Structures to Describe and Operate over Array Regions

The C language, due to its reliance on pointers, has some features that are typically not found in other languages. For instance, it allows pointer arithmetic, decaying arrays into pointers, passing pointers to positions inside an array, or even addressing parts of an array by dropping some of its most significant dimensions. While these characteristics may be seen as violating good coding rules, they are used in practice. For instance, a multidimensional Fast Fourier Transform (FFT) implementation in C may perform unidimensional FFTs followed by transpositions that operate on the same data but in two dimensions.

The need to handle those cases in SMPSs complicates data dependency analysis, since the addresses of the arrays passed to the tasks may actually point to a location different to the beginning of the array, and the dimensions of the array may differ from one task call to the other. This complexity originates from trying to make the internal representation a direct projection of language, that is, a representation based on a base addresses and the dimension sizes and indexes.

Chapel, Titanium and X10 only support "good coding rules" in relation to arrays. Therefore, their array indexing has a unique and direct correspondence to their elements. Because of this, the correspondence between their array region specification and the actual elements is also unique, which allows them to have an internal representation that is also based in the index domain, and thus closely resembles that of their language-level syntax. In contrast, since we are willing to support the whole range of C operations over arrays, the correspondence between indexes and elements is not unique, and that forces our internal representation to depart from the index domain representations and thus to differ significantly from its language-level representation.

4.3.1 Compact Array Region Representations

Compact array region representations have been previously proposed in the area of autoparallelizing compilers. However, like the representations of Chapel, Titanium and X10, they also rely on knowing the base address of the arrays.

The Triplet representation described by [Havlak and Kennedy, 1991] is similar to the SMPSs language-level representation. The Linear Memory Access Descriptor (LMAD) by [Paek et al., 2002] improves the Triplet representation by converting it into a linearized form. This way, it can detect dependencies even when the array dimensions change.

Both representations are used at compile time to model the accesses performed by statements across an iteration space and to determine the possible independence between them. In some cases the analysis can also span several function invocations. Being compile time techniques, they require knowing which specific array is being accessed in every case. This makes handling pointer arithmetic difficult, and for non-linearized representations, it makes array reshaping difficult too.

Hybrid methods like the extended LMAD presented by [Rus et al., 2002] combine symbolic analysis at compile time with symbolic substitution and run time combination of regions. However, they are used for fork-join parallelism instead of more unstructured parallelism models.

The work of [Paek et al., 1998] overviews some of the most used array region representations in the area of autoparallelizing compilers, including the Data Access Descriptor published by [Balasundaram and Kennedy, 1989], which has similar capabilities to the Triplet representation.

While in many cases the information provided by the code annotations can be generated automatically through compiler analysis, our proposal differs from the representations used in autoparallelizing compilers in that (1) it is used exclusively at run time, (2) it is used for finding the actual dependencies instead of just testing for independence, (3) it does not rely on knowledge about the specific array being accessed, which allows it to accept pointer arithmetic and array reshaping, and (4) it is used over a bigger search space (an average SMPSs program may have thousands of in-flight tasks at any given time).

The SMPSs array region language syntax restricts the possible shapes to just multidimensional rectangles (also called hyper-rectangles). Therefore, data structures that support range-like operations could be candidates for their internal representation. For instance, since the region is always rectilinear, the lowest and highest indexes of each dimension are enough to represent the region inside the array. This representation is compact and can be organized into a Space Partitioning Tree (SPT). For instance, the kd-tree that [Bentley, 1975] proposes would allow to check efficiently for overlaps. However, such representations cannot handle pointer arithmetic, arrays that degenerate into pointers, and arrays that change their dimensions. In addition, using the lowest and highest indexes of a region as a representation binds the representation to the base address of the array and its dimensions, since each array would define a different origin and coordinate system and would thus require its own SPT.

The Cache Snoop Filters presented by [Salapura et al., 2007] are used in processors to reduce the effect in performance and power of the cache coherence signaling. Their purpose it to detect whether an address is within a certain set of addresses. The solution they propose is to represent these sets using pairs of registers that store a base address and a bit mask. The low-level array region representation proposed in this thesis is based on that representation.

4.3.2 The SMPSs Array Region Representation

This thesis proposes a linearized representation based on the actual addresses of the elements that it covers. Linearization relative to the beginning of the array was first proposed by [Burke and Cytron, 2004]. Our proposal differs in that it is absolute and therefore does not depend on the base address of the array. This approach unties the representation from the base address of the array and its dimensions.

The low-level representation is very similar to the stream registers used in the cache snoop filters described by [Salapura et al., 2007]. We define $\mathfrak{R}(N)$ as the set of all possible regions of width N, where N indicates the number of binary digits of the representable addresses. A region $r \in \mathfrak{R}(N)$ is defined by an ordered sequence of digits $r = \langle r_N, r_{N-1}, \dots, r_1 \rangle$, such that the value of each digit can be either 0, 1, or X. The digits are ordered from most significant digit r_N to least significant digit r_1.

Regions are a generalization of numbers written in binary form (in this case memory addresses), that allow representing sets of values. Any address can be represented as a region by simply expressing it in binary form and interpreting it as a region description. The X digit value has a similar meaning to the one used logic synthesis of digital circuits: the value of the digit can be either 0 or 1.

Given a certain N and two regions $a, b \in \mathfrak{R}(N)$ we define several properties. For simplicity, we use subindexes to refer to the digits of a region by position. First, a region can only be expressed in a unique way as shown in equation 4.1. Second, a region can be a subset or a superset of another. Equation 4.2 shows the necessary conditions. Since an address is also a region, equation 4.2 also defines whether an address belongs to a region. Equation 4.3 defines the existence of intersection. In that case, equation 4.4 defines how to calculate it. Equations 4.3 and 4.4 can be deduced from 4.1 and 4.2.

$$a = b \iff \forall i\ a_i = b_i \tag{4.1}$$

$$a \supseteq b \iff \forall i\ a_i = b_i \lor a_i = X \tag{4.2}$$

$$a \cap b \neq \emptyset \iff \forall i\ a_i = X \lor b_i = X \lor a_i = b_i \tag{4.3}$$

$$a \cap b \neq \emptyset \implies \forall i\ (a \cap b)_i = \begin{cases} 0 & \text{if } a_i = b_i = 0 \\ 1 & \text{if } a_i = b_i = 1 \\ a_i & \text{if } b_i = X \\ b_i & \text{if } a_i = X \end{cases} \tag{4.4}$$

Figure 4.3 depicts a bidimensional array with a region. The grid represents the elements of the array and the gray positions indicate the elements specified by the range. Figure 4.3(a) contains the language representation and figure 4.3(b) the low-level representation, considering an 8 bit wide address space, that the base address of a is 0, and that the array is stored in row-major order.

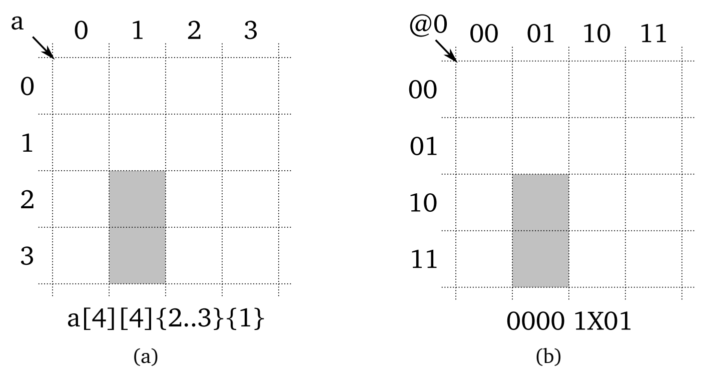


Figure 4.3: A region: (a) high-level representation; and, (b) low-level representation considering that the address of a is 0 and the elements are 1 bit wide.

4.3.3 Converting a Language-Level Region into the Internal Representation

In order to use the low-level representation, there must be a conversion process from the language representation. Let M be the number of dimensions, 1 being the contiguous dimension and M the most strided one. For simplicity we assume that all dimension ranges are zero based, that is, that each has a size d and its range is of the form $\{0 : l\}$. We also consider that b is the base address and that the size d and range length l of the contiguous dimension are expressed in terms of bytes (as opposed to elements).

By definition a digit with value X can have values 0 and 1 in the set of addresses of the region. Any other non-X digit position must have the same value in all addresses. A naïve approach would generate all possible addresses and check for digits that change their value.

A more efficient approach is to check digit positions of the base address to which we must add 1 to calculate an address of the set and to consider their propagation when they generate carries through the more significant digits. For instance, for a unidimensional array with base $b = 0010$ and range length $l = 0011$, digits b_1 and l_1 generate an X. Digits b_2 and l_2 generate an X but also propagate to the following digit, thus generating another X. The result of this algorithm is 0XXX.

Listing 4.3 shows pseudocode to calculate the set of Xs of a low-level representation. Each digit of the variable *mask* set to 0 corresponds to an X in the low-level representation. All other bits of the low-level representation must match the base address. This implementation iterates through all the dimensions. Each time, the variable *stride* contains the stride of the given dimension. The algorithm calculates the effect of each bit set to 1 in *stride* when generating the addresses as it jumps from digit to digit as determined by l_i.

Although this algorithm is faster than the naïve solution, it has a time complexity that is linear to the number of bits of an address and to the number of bits set to 1 in the dimension lengths and range lengths.

We can further reduce the cost of generating a low-level representation by using an approximate solution. Listing 4.4 calculates an approximate mask by determining the first X and the last X corresponding to each dimension and filling all digits in between with Xs. The result may have digits set to X that are not set when using the

```
mask = {1, 1, ..., 1}
current_address = base_address
stride = 1
**for each** dimension $i$ **starting from** 1
    **for each bit** $j$ **of** stride **equal to** 1
        **for each bit** $k$ **of** $l_i$ **equal to** 1
            **for** k_current **from** 0 **to** k−1
                {mask, current_address} = set_and_propagate(k_current+j,
                  current_address, mask)
    stride = stride $*$ $d_i$

**function** set_and_propagate(i, address, mask)
    **if bit** i **of** address **is equal to** 1
        $\text{address}_i = 0$
        **return** set_and_propagate(i+1, address, mask);
    **else**
        $\text{mask}_i = 0$
        $\text{address}_i = 1$
        **return** {mask, address}
```

Listing 4.3: Pseudocode that given a region in language representation calculates the mask that corresponds to the X values of its low-level representation.

previous algorithm. For instance, for a dimension with stride 1001, length 10 and start address 00000, the exact algorithm would return XX0XX, while the approximate algorithm would return XXXXX.

Approximating the low-level representation in this way does not lead to incorrect executions, since it does not exclude data that is accessed by the tasks. However, it can include non accessed data and thus include nonexistent dependencies that may reduce parallelism.

4.3.4 Aliasing and Alignment Restrictions

The low-level representation, independently of the algorithm used to construct it, can only represent with total precision certain sets of addresses. The rest can be represented approximately as a superset of the intended set. When converting a language level representation into a low-level representation, this kind of approximation can occur.

For instance a 1D region spanning from 0011 (3) to 0110 (6) would be represented by 0XXX, which is actually a superset that contains addresses from 0000 (0) to 0111 (7). We call this effect *aliasing*.

Given a dimension, we can fix the indexes for all other dimensions and calculate the addresses covered by the range of indexes of this dimension. Under those conditions and given a base address b, a stride s, and a dimension length l we can calculate:

$$\text{addresses}(b,s,l) = \bigcup_{0 \le i < l} \{b + s \times i\} \tag{4.5}$$

The displacement of the addresses with respect to the base address can be obtained with addresses$(0,s,l)$. We define changed_bits(b,s,l) as the set of indexes of the

```
mask = {1, 1, ..., 1}
current_address = base_address
stride = 1
**for each** dimension $i$ **starting from** 1
    **find maximum** lowest_digit **such that bit** lowest_digit **of** (stride−1) **is
      equal to** 1
    **find maximum** highest_digit **such that bit** highest_digit **of**
      (current_address+(length−1)∗stride) **is equal to** 1
    **for each** j **from** highest_digit **to** lowest_digit
        $mask_j = 0$
    current_address = current_address + (length−1)∗base
    stride = stride ∗ $d_i$
```

Listing 4.4: Pseudocode that given a region in language representation calculates the mask that corresponds to a superset of the X values of its low-level representation.

digits that change in addresses(b, s, l), that is:

$$\forall i \quad i \in \text{changed\_bits}(b,s,l) \iff \exists x, y \in \text{addresses}(b,s,l) \land x_i \neq y_i \qquad (4.6)$$

Aliasing at this level can happen for two different reasons. First, aliasing can happen if either the stride or the length are not powers of 2. In that case, the number of covered addresses is less than the number of combinations introduced by the changed bits.

The other source of aliasing is the base address. If a digit i that changes in a range is set to 1 in the base address b, then that bit position generates a carried 1 that is propagated to the next greater significant digit up to a digit that does not change in addresses$(0, s, l)$ and that it is not 1 in the base. This is the case of the example shown previously.

A carried 1 due to any of the two types of aliasings can propagate up to the bit positions corresponding to the following dimension, leading to interdimensional aliasing. For instance on a bidimensional matrix, interdimensional aliasing can generate aliasing dependencies between regions without any row and column in common.

Aliasing can be eliminated or reduced by splitting the language-level region into several that can be represented with more precision. For instance, in the previous example, the region could be split into three parts: 0011, 010X, and 0110. This solution, though, increases the number of low-level regions that must be handled in the run-time, which can have a big performance impact.

Aliasing can also be avoided entirely by always providing language level regions that can be represented with total precision. A number of conditions must be met: first, the base address must be aligned in such a way that all its possibly changing bits are zero; second, each dimension in the array must be a power of two; and finally, for each dimension, the length of the accessed indexes must be a power of 2 and the initial index must be a multiple of it.

Aliasing generates additional unnecessary dependencies that we call *aliasing dependencies*. They potentially limit the amount of parallelism that can be achieved. Although it may have a negative impact on performance, aliasing does not eliminate any dependency and thus preserves the correctness. While the previous conditions must be met to avoid aliasing, they are not absolutely necessary to obtain good

a_i	a_i^v	a_i^m
0	0	1
1	1	1
X	0	0

Table 4.1: Encoding of digit values of the compact representation as pairs of bit values.

a_i	b_i	a_i^v	a_i^m	b_i^v	b_i^m	$a \supseteq b$
X	X	0	0	0	0	1
X	0	0	0	0	1	1
X	1	0	0	1	1	1
0	X	0	1	0	0	0
0	0	0	1	0	1	1
0	1	0	1	1	1	0
1	X	1	1	0	0	0
1	0	1	1	0	1	0
1	1	1	1	1	1	1

Table 4.2: Truth table of the subset relation.

performance. For instance, in many cases, just eliminating interdimensional aliasing can provide enough parallelism.

4.3.5 Implementing Basic Operations

Our compact representation can be implemented as two bit fields a^v and a^m that encode the digit values of a region a. A first field a^v called *value* contains the values 0 and 1 for the digits with those values, and 0 for digits with value X. A second field a^m called *mask* contains value 0 for digits with value X in the region and value 1 for digits with values 0 and 1. Table 4.1 shows the equivalence between digit values and their encoding using separate bits for the value and the mask.

Equality of two regions a and b as defined in equation 4.1 can be implemented by checking that both pairs of bit fields are equal:

$$a = b \iff a^v = b^v \wedge a^m = b^m \tag{4.7}$$

The subset relation is defined in equation 4.2. The truth table for a single digit is shown in table 4.2. The relation is true when the function in the table is true for all digits. If we minimize the truth table and we apply to all digits, we can calculate the relation using bit-wise logical operations as follows:

$$a \supseteq b \iff (a^v \veebar b^v) \wedge a^m = \langle 0, 0, \ldots, 0 \rangle \tag{4.8}$$

The existence of an intersection has been defined in equation 4.3. Its truth table for one digit when representing the regions using pairs of bit fields is shown in table 4.3. The intersection exists when the function in the table is 1 (true) for all digits. If

a_i	b_i	a_i^v	a_i^m	b_i^v	b_i^m	$a \cap b \neq \varnothing$
X	X	0	0	0	0	1
X	0	0	0	0	1	1
X	1	0	0	1	1	1
0	X	0	1	0	0	1
0	0	0	1	0	1	1
0	1	0	1	1	1	0
1	X	1	1	0	0	1
1	0	1	1	0	1	0
1	1	1	1	1	1	1

Table 4.3: Truth table of the intersection existence.

a_i	b_i	a_i^v	a_i^m	b_i^v	b_i^m	$(a \cap b)_i^v$	$(a \cap b)_i^m$	$(a \cap b)_i$
X	X	0	0	0	0	0	0	X
X	0	0	0	0	1	0	1	0
X	1	0	0	1	1	1	1	1
0	X	0	1	0	0	0	1	0
0	0	0	1	0	1	0	1	0
0	1	0	1	1	1	–	–	$\varnothing$
1	X	1	1	0	0	1	1	1
1	0	1	1	0	1	–	–	$\varnothing$
1	1	1	1	1	1	1	1	1

Table 4.4: Truth table of the value and mask of the intersection.

we minimize the truth table and we apply to all digits, we can calculate the relation using bit-wise logical operations as follows:

$$a \cap b \neq \varnothing \iff a^m \wedge b^m \wedge (a^v \veebar b^v) = \langle 0, 0, \dots, 0 \rangle \tag{4.9}$$

Equation 4.4 defines the intersection of two regions when it is not empty. When representing regions using pairs of bit fields we can calculate the value and the mask of the intersection independently. Its truth table for one digit when representing the regions using pairs of bit fields is shown in table 4.4. When we minimize both functions and we apply them to all digits, we can calculate the intersection using bit-wise logical operations as follows:

$$(a \cap b)^v = a^v \vee b^v \tag{4.10}$$

$$(a \cap b)^m = a^m \vee b^m \tag{4.11}$$

4.4 Organizing Regions for Fast Insertion, Deletion and Matching

In the previous chapter we detected data dependencies by identifying each block of memory by its starting address. However, that characterization is a simplification that is only valid for disjoint blocks. In general, two tasks have a dependency if they access the same memory address and the type of access of at least one of them is a write or an update. The combination of the different kinds of accesses determines the type of dependency (Read after Write, Write after Read, or Write after Write). This can be generalized for regions. Two tasks have a dependency if they access memory regions that intersect and at least one of them is a write or an update.

To calculate them, each region access must be checked against those of previous tasks. Hence we need a that structure that allows fast insertion, deletion and lookup of regions. We call this data structure the *region tree*.

While the history of all previous task accesses is sufficient to calculate dependencies, it is not necessary to keep it fully. Instead only the last writer and the list of readers of the last definition are needed. All other dependencies are redundant by transitivity.

4.4.1 A Data Structure for Region Insertion, Removal, and Matching

The region tree indexes that information using the low-level representation as key. Each region stored in the tree is represented as a path from the root node to a leaf, and each leaf and its corresponding path from the root corresponds to a unique region. The sequence of edge values from the root node to each leaf determines the key. That is, a region $a \in \mathfrak{R}(N)$ is represented in the region tree as a path from the root node to a leaf such that the sequence of traversed edges are labeled with values $a_N, a_{N-1}, \ldots, a_1$.

Given a region $a \in \mathfrak{R}(N)$, to look up all intersecting regions, the tree must be traversed following the edges that could lead to intersecting regions as defined in equation 4.3. That is, for every digit a_i if its value is X, then the search must descend through all its children, otherwise, it must descend through the edge labeled with the same value as a_i and through the edge labeled with value X. In case of a non-match, it should abandon that path since it cannot lead to an intersection, as defined in equation 4.3.

Insertion and removal are very similar to searching. During insertion, the tree is traversed through the edges that match exactly the digits of the low-level region. Whenever an edge is missing, it is added. During removal, the corresponding node is searched, and it is removed together with its predecessors up to a node that has more than one child.

Traversal speed is dependent on the number of nodes traversed. In the worst case, the region intersects every region in the tree, for instance $\langle X, X, \ldots, X \rangle$, and the traversal time is in the order of the number of nodes in the tree. In the best case, the most significant digit does not match and the traversal finishes with no match below the root of the tree.

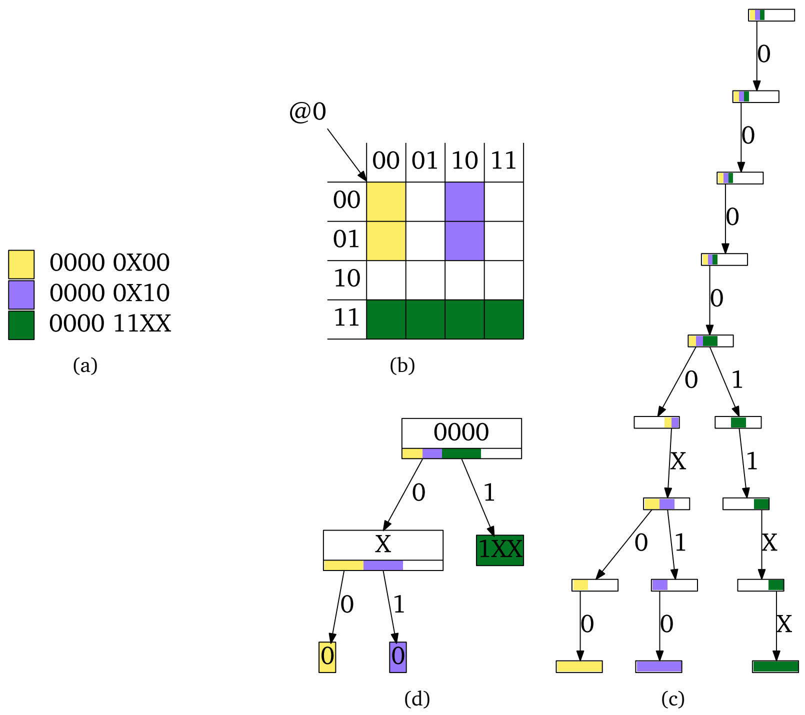


Figure 4.4: A region tree and its contents: (a) low-level representation; (b) graphical representation; (c) region tree; and, (d) compressed tree.

4.4.2 Improvements

Since the time required for traversal is dependent on the number of nodes in the tree, we can improve its performance by reducing the number of nodes. We accomplish this by compressing into one node each subpath containing only nodes with one child. This approach is similar to the one proposed by [Morrison, 1968] to compress common parts of strings in radix trees. In this case, since the “alphabet” of the index is limited to only 3 symbols and a fixed length equal to the address width, we can represent the common parts in the nodes and use the edges to indicate the discriminating bits. The implementation that this chapter evaluates uses this technique.

Figures 4.4(a), 4.4(b) and 4.4(d) show an example of such an arrangement. Figure 4.4(a) shows three different regions based on an 8 bit-wide address space. A graphical representation is shown in figure 4.4(b). It shows the regions as if they were part of an array of 4 by 4 bytes. Figure 4.4(c) shows a region tree with those regions before applying compression, and figure 4.4(d) shows the tree when we apply compression. Each leaf contains a small square with the same color as in figure 4.4(a), indicating that the path from the root node to itself represents that region. The digits of the low-level representation can be obtained by concatenating the digits in each node and the edges along the path from the root to the leaf.

4.4.3 Handling Overlaps

By using regions, programs are not restricted to a fixed block size. In fact programs can determine dynamically the regions and thus their size. Hence, any region access may fully or partially overlaps previous ones. Since the runtime does not need to keep the full history of data accesses, overlaps present an additional challenge when deciding which information can be safely be removed.

The run-time handles new region accesses in two phases. First, it looks up a new region access a in the region tree T to find possible overlaps (or matches) with already existing region accesses. This information is used for calculating the data dependencies. In the second phase, the run-time updates the region tree to reflect the effects of the new access. By construction, the region tree can hold several overlapping regions at any given instance. Thus, if the tree already contains the exact region, it is updated to reflect the new access. Otherwise, the region is added with such information independently of any overlap.

Regions in the tree that overlap have both current and old information in their intersection. The old information may produce dependencies that are redundant by transitivity, however they do not limit parallelism in any way. As tasks are executed, their information is removed from the tree, reducing those dependencies. To reduce the amount of unnecessary matches, when a new task writes or updates a region, any region in the tree fully contained within the new region is removed.

An alternative policy would be to keep only the necessary data to calculate dependencies. However, to do so would require to handle partial overlaps in a different way. First, it would require to find the intersection of the accesses, and then to fragment those in the tree to update only the part that is affected, and thus to represent in the tree the exact state. This option has not been considered further, since it would slow task creation in two ways. First to calculate the fragments and to perform the actual fragmentation. And second, due to the potentially slower lower lookup due to the increased number of matching region in the tree per lookup.

4.5 Evaluation

This section examines the programmability of the model and its performance. It extends the evaluation of the previous chapter with region-based solutions and additional algorithms that are hard to implement without regions.

The hardware, software, configuration and measurement methodology are the same as in the previous chapter and have been described in section 3.6 that starts on page 29. The alternative implementations in OpenMP, MPI and the parallel libraries are also the same as in the previous chapter unless otherwise stated.

The benchmarks that already appeared in the previous chapter have been adapted to use regions, but for brevity the evaluations of most of them have been moved to appendix E. The Triad benchmark does not use regions, however in the appendix we evaluate the impact that supporting regions in the runtime has over the overhead. The changes in the code of the matrix multiplication and the Gauss-Seidel codes have already been presented in figures 4.1 and 4.2 in pages 87 and 87. The Cholesky algorithm has similar changes to the matrix multiplication. Thus, all these benchmarks in their region-based form are evaluated in the appendix.

The Strassen-Winograd code has also similar changes, but in addition to the advantages to programmability, it can benefit from having different data sizes for each type of task. For this reason, this one is evaluated in this chapter.

Region support enables the implementation of the following region-based benchmarks: a parallel sorting algorithm called multisort that is based on mergesort, a Fast Fourier Transform, and an LU decomposition with partial pivoting that we call HPL. The multisort algorithm uses regions to allow tasks to operate over data in one dimension with different sizes that have overlapping accesses. The Fast Fourier Transform benchmark makes use of regions to calculate dependencies between tasks that operate over the same data in two regular but different bidimensional shapes. Finally, the HPL benchmark requires the use of regions for tasks that access bidimensional regions with different shapes that overlap and produce irregular dependencies.

4.5.1 Strassen-Winograd

The Strassen-Winograd benchmark has been evaluated in the previous chapter in section 3.6.5 that starts on page 68. This chapter evaluates a version of the algorithm using flat matrices instead of blocked matrices.

Parallelization with SMPSs using Regions and two Simultaneous Blocking Sizes

Blocked data layouts impose restrictions on the granularity of the tasks. For instance, the simple additions and subtractions of the Strassen-Winograd algorithm, when decomposed into tasks, must be performed on one block per operand. While it is possible to implement tasks that operate on more than one block per operand, that would require one type of task for each possible number of blocks per operand. Therefore, that would not be a clean and scalable solution.

The operations of the Strassen-Winograd algorithm can be classified according to their performance characteristics in two groups. One group for the multiplications, and another for the additions and subtractions. While the multiplications have good temporal locality, the additions and subtractions only have good spatial locality. Therefore additions and subtractions would benefit from big block sizes, since they allow to exploit more spatial locality. However, multiplications would not benefit as much. In fact, smaller blocking sizes would produce more parallelism and shorter tasks, which in turn has the potential to produce better balanced schedules. Thus, in such scenarios, it may be desirable to use different logical blocking sizes for each group.

The regions-based implementation of the algorithm has been written using different blocking sizes for each group. The algorithm is recursive and at each level it divides the matrices into four submatrices and performs operations over them. The implementation of the previous chapter defined the base case of the recursion as a multiplication that matches the physical size of the block. The regions-based implementation substitutes physical blocks over hypermatrices by logical blocks over flat matrices. While additions and subtractions were solved iteratively block by block, the regions version uses its own independent decomposition size for those operations. At the lowest levels of the recursion, the decomposition can lead to submatrices that are smaller than the logical blocking size of those operations. In those cases, the logical blocking size becomes the size of the submatrices.

Figure 4.5 shows the performance difference between using the same blocking size for both classes of operations and using different ones. Each column of panels corresponds to a fixed number of cores, and each row of panels corresponds to a fixed matrix size. The vertical axis indicates the multiplication block size, and the

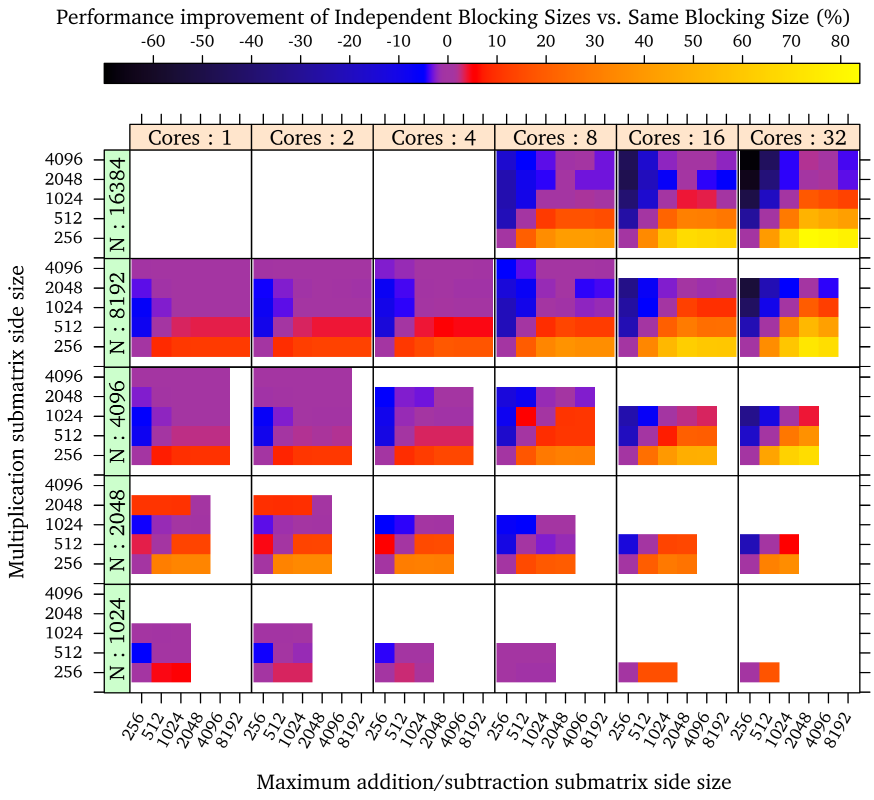


Figure 4.5: Performance improvement of the region-based Strassen-Winograd with different blocking sizes for the multiplication tasks than the rest compared to using the same blocking for all tasks as the multiplication tasks.

horizontal axis the additions and subtraction task size. Colors indicate the performance improvement of the given configuration compared to using the multiplication block size for all tasks.

The lower triangles show the effects of using bigger sizes for the additions and subtractions than the multiplication, and the upper triangles show the effects of using bigger multiplications than additions and subtractions. The figure shows that in general, bigger additions and subtractions perform better, and therefore do not hurt parallelism too much. The improvement can be as high as 82%. This improvement comes from the reduction of runtime overhead and the better task performance.

Runtime Overhead

The runtime overhead of the region version is higher than the version run on the region-unaware runtime, but the difference is much lower than that of the other codes. Figure 4.6 shows the increment of task management overhead in the main thread. While it may raise up to 2.3 times the overhead of the blocked version, figure 4.7 shows that the total task management overhead remains below 30% in most cases, and takes up to 42% only for very small multiplication task sizes.

These figures have been generated with a region tree that uses compression.

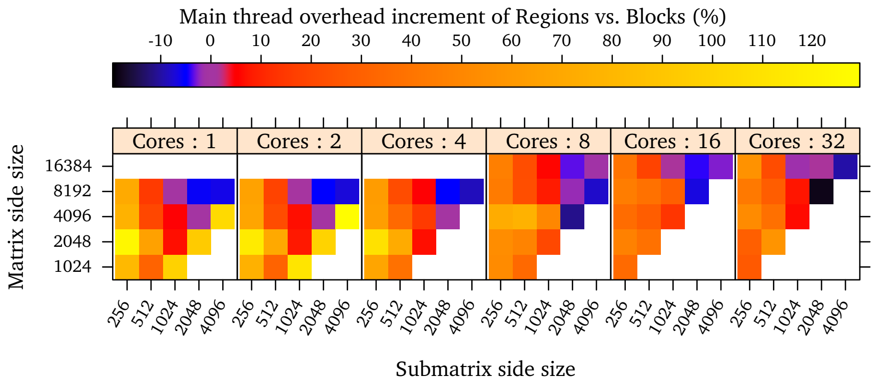


Figure 4.6: Task management overhead increment on the main thread when running the SMPSs implementation of the Strassen-Winograd algorithm with regions with several matrix and submatrix sizes compared to the region unaware version.

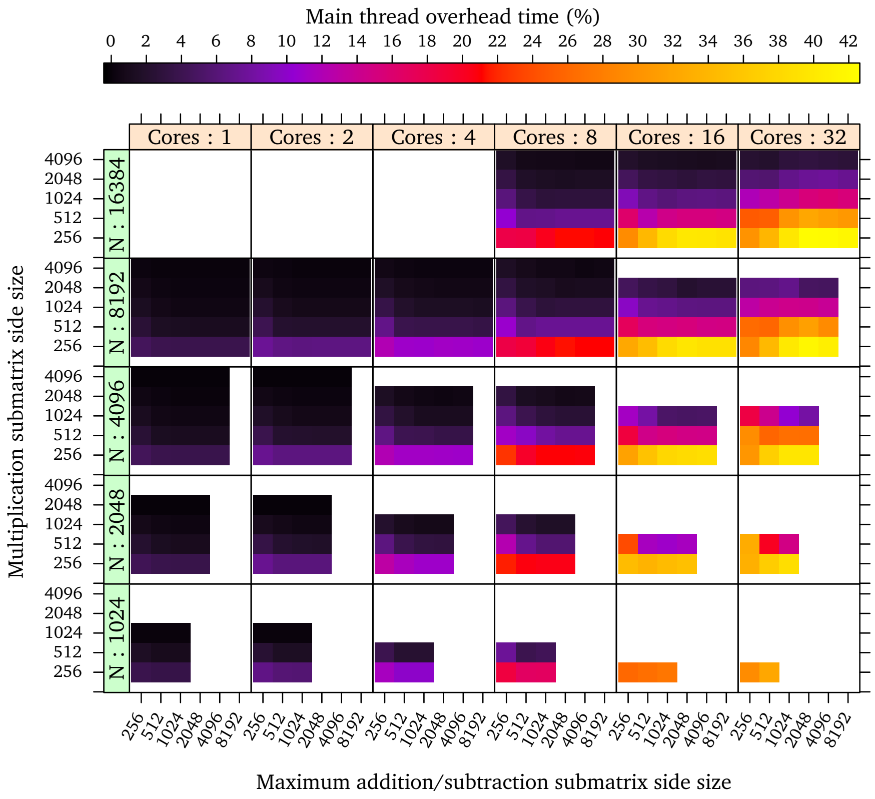


Figure 4.7: Task management overhead on the main thread when running the SMPSs implementation of Strassen-Winograd algorithm with regions with several matrix and submatrix sizes.

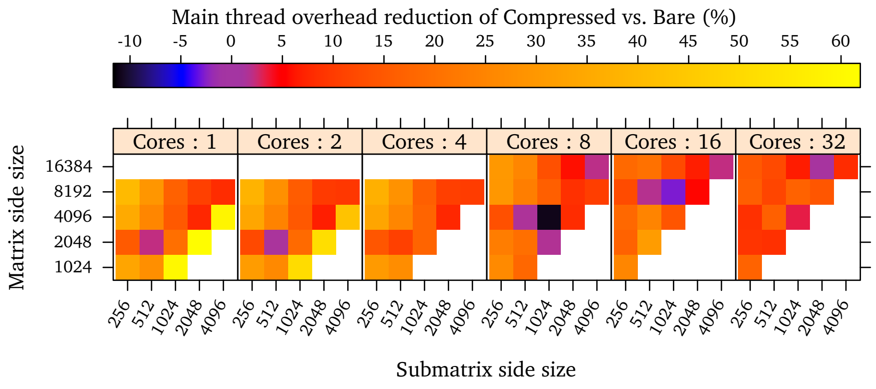


Figure 4.8: Reduction of task management overhead on the main thread when running the SMPSs implementation of the Strassen-Winograd algorithm with a compressed region tree compared to an uncompressed region tree, with several matrix and submatrix sizes.

Compressing the region tree reduces the task management overhead by up to 60%, as figure 4.8 shows.

Scalability and Other Implementations

Figure 4.9 shows the strong scalability of the benchmark with several problem sizes under the blocked SMPSs version, the new flat SMPSs version using regions and the OpenMP 3 version that uses task nesting and barriers. In the previous chapter we showed that the performance of the Strassen implementation using blocks had substantially better performance than the OpenMP implementation. When we use flat matrices and regions, the SMPSs version still has much better scalability due to its ability to extract more parallelism, and thus to keep threads busy for more time, and even use bigger tasks. Figure 4.10 shows the effective parallelism of each version and demonstrates that the SMPSs versions achieve more parallelism.

However, figure 4.9 also shows that the version using regions performs worse than the version using blocks on the smallest problem sizes. In these cases, tasks are small and therefore the flat data layout has lower effective spatial locality. Figure 4.11 shows up to 62% higher mean task floating point operations per cycle in the blocked version compared to the regions-based version.

Table 4.5 summarizes the performance metrics of each implementation. Notice that while in the previous benchmarks, the blocked data layout was advantageous performance-wise, in the Strassen-Winograd code the amount of parallelism is such that the region-aware code can use bigger tasks to recover much of the lost task performance without loosing too much parallelism. In the cases that the regions version uses the same blocking sizes as the blocked version, task performance is clearly the factor that determines its lower performance.

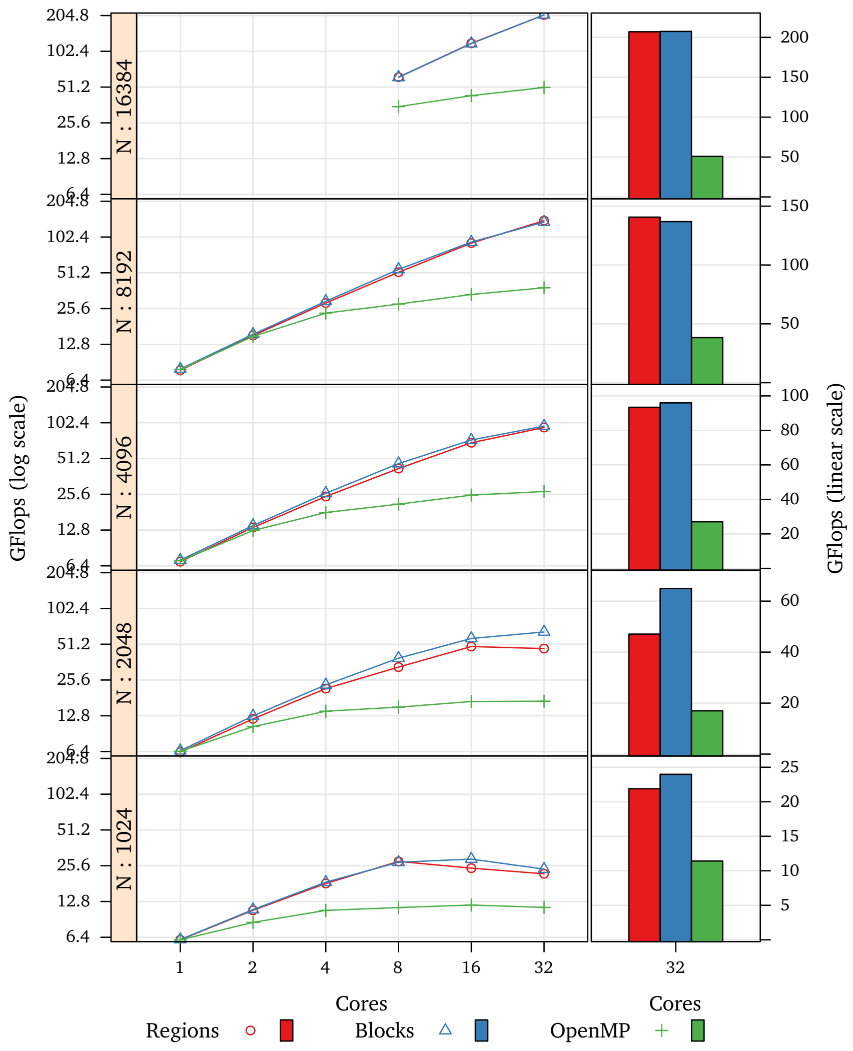


Figure 4.9: Strong scalability of the region-aware SMPSs Strassen-Winograd algorithm, the region-unaware, the OpenMP version and the MKL parallel version and performance with 32 cores.

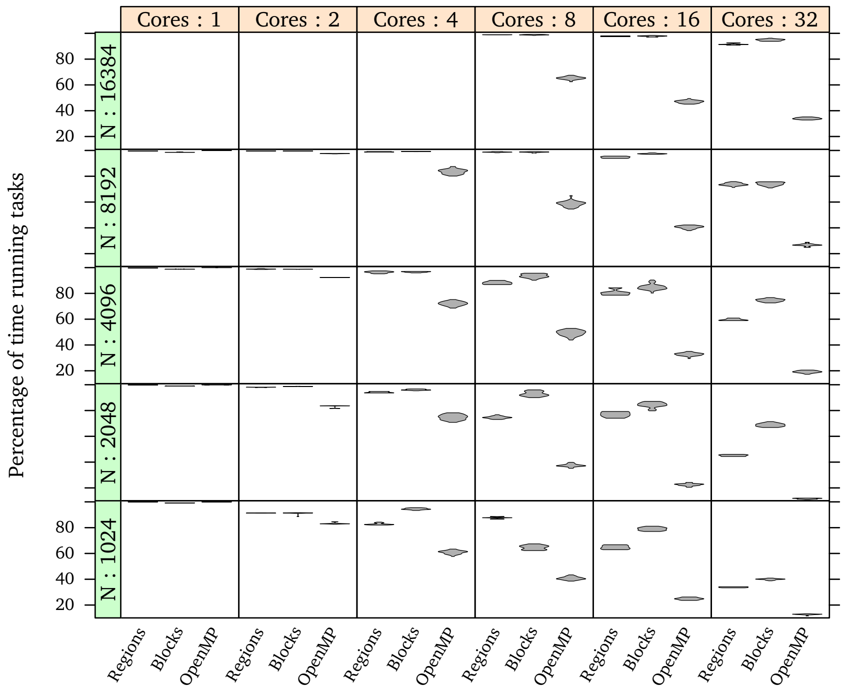


Figure 4.10: Average parallelism of the Strassen-Winograd algorithm implementations.

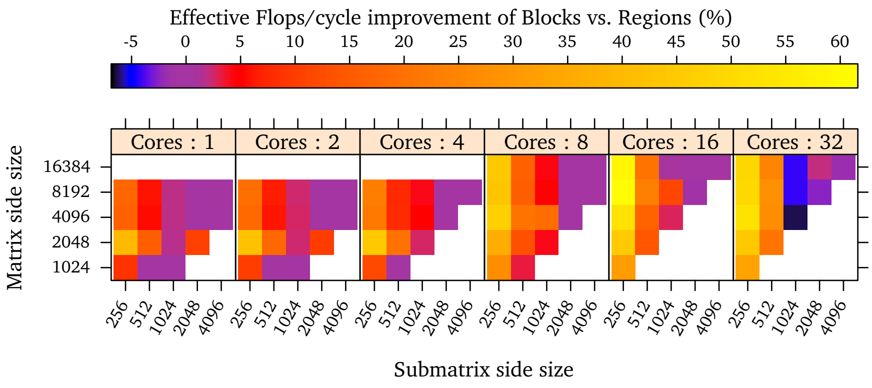


Figure 4.11: Taks performance improvement of Strassen-Winograd when using the blocked data layout compared to the flat data layout.

Cores	N[a]	BS1[b]	BS2[c]	BS[d]	BS[d]	GF[e]	GF[e]	GF[e]	FPC[f]	FPC[f]	FPC[f]	Eff.[g] (%)	Eff.[g] (%)	Eff.[g] (%)
1	1024	1024	1024	512	1024	6.1	6.1	6.1	3.9	3.5	3.9	99	99	99
1	2048	1024	1024	512	1024	6.3	6.5	6.4	3.6	3.3	3.6	99	98	99
1	4096	4096	1024	512	1024	6.9	7.2	7.1	3.4	3.2	3.5	99	98	99
1	8192	4096	1024	512	1024	7.8	7.9	7.9	3.4	3.1	3.4	99	98	99
2	1024	512	512	512	256	10	10	8.5	3.4	3.4	2.8	91	91	83
2	2048	2048	512	512	512	12	12	10	3.1	3.3	3.2	98	98	83
2	4096	1024	1024	512	512	13	13	12	3.4	3.2	3.1	98	98	92
2	8192	2048	1024	1024	512	15	15	14	3.3	3.4	3.0	99	99	97
4	1024	512	512	256	256	18	18	10	3.2	2.7	2.5	82	94	60
4	2048	2048	512	512	256	21	23	13	3.0	3.1	2.3	94	96	74
4	4096	1024	1024	1024	512	24	26	17	3.2	3.4	2.9	96	96	72
4	8192	2048	1024	1024	512	28	29	23	3.2	3.3	2.8	98	99	83
8	1024	256	512	512	256	27	27	11	2.4	3.2	2.1	87	64	40
8	2048	1024	1024	512	512	32	39	15	3.3	2.8	2.8	74	92	37
8	4096	4096	1024	1024	512	42	46	21	3.1	3.2	2.6	88	93	49
8	8192	1024	1024	1024	512	51	54	27	3.0	3.1	2.5	98	98	58
8	16384	2048	2048	2048	512	62	61	35	3.4	3.4	2.5	98	98	65
16	1024	1024	256	256	256	24	29	11	1.7	1.9	1.9	64	79	24
16	2048	2048	512	512	512	48	57	16	2.4	2.6	2.6	76	84	22
16	4096	4096	1024	1024	512	69	73	25	2.9	2.9	2.4	80	84	32
16	8192	2048	1024	1024	512	91	92	33	2.8	2.8	2.3	94	97	40
16	16384	2048	2048	2048	512	119	118	43	3.3	3.3	2.3	97	97	46
32	1024	512	256	256	256	21	23	11	1.5	1.7	1.8	33	39	12
32	2048	1024	512	512	512	47	64	16	2.2	2.2	2.6	45	68	11
32	4096	2048	1024	1024	512	93	95	27	2.7	2.3	2.3	59	74	19
32	8192	2048	2048	2048	512	140	136	38	3.1	3.1	2.1	73	74	26
32	16384	4096	2048	2048	512	207	207	50	3.2	3.1	2.0	91	94	33

Implementation: Regions Blocks OpenMP

[a] Matrix side size.
[b] Maximum submatrix side size for addition and subtraction tasks
[c] Submatrix side size for multiplication tasks
[d] Submatrix side size
[e] GFlops.
[f] Mean floating point operations per cycle while running tasks.
[g] Mean time that threads spend running tasks.

Table 4.5: Performance summary of the Strassen-Winograd algorithm implementations.

4.5.2 Multisort

The multisort benchmark is a variant of the mergesort algorithm. While the original algorithm allocates temporary arrays at each recursion level, the algorithm we present uses only one temporary array. The objective is to minimize the overhead of memory allocation and page initialization. To this extent, instead of dividing the problem each time by 2, we divide it by 4.

Listing 4.5 shows its main algorithm. The code sorts the data in the data parameter using tmp as temporary storage. Since in practice mergesort is slow for small arrays, we use the quicksort algorithm for those cases. The threshold is controlled by the MIN_SORT_SIZE constant. Lines 4–7 perform the recursive calls over each quarter of the data. Then, lines 9 and 10 merge two quarters each from the data array into the tmp array. Finally, line 12 merges the two resulting halves from tmp back into data.

Parallelization with SMPSs using Regions

Parallelizing the multisort code with SMPSs is as simple as converting merge and basicsort into tasks. However, as [Cormen et al., 2009, pg. 797–798] demonstrate, the amount of parallelism that can be obtained by parallellizing only the recursion is small. To achieve more parallelism, the merge function must be also parallelized.

The most commonly used solution in the literature, for instance the one that [Cormen et al., 2009, pg. 798–803] use, consists in finding the median value of one of the input arrays and to use its value to divide both arrays into two parts that can be merged in pairs independently. This formula can be applied recursively as needed.

However, that approach requires that the input arrays are accessible before deciding the pivot. With task nesting models this can be achieved by placing a barrier before the merge. In that case, lines 8 and 11 of listing 4.5 would contain barriers. While task nesting is orthogonal to tasks with dependencies, this chapter only covers solutions based on dependencies alone, and thus does not apply that solution.

Since at task instantiation time the values of the two input arrays of the merge function are not available, instead of deciding a pivot based on their contents, we delay finding the pivot, and instead decide which part of the output will be calculated by each task. With that information, the tasks scan the input arrays to find the pivot. Listing 4.6 shows the main merge code. Parameters first and length determine the portion of the result that is generated.

When the merge task starts, its input values are available and thus it can traverse them to determine the pivoting indexes. These are named leftFirst and rightFirst and are calculated by the find_pivot function in listing 4.7. The pivots are searched using binary search in both arrays until we find the position in both that corresponds to the starting position for first in the result array.

Listing 4.8 shows the task declarations of the whole benchmark. Notice that while the tasks do not make extensive use of the regions syntax, they have dependencies over overlapping data regions with different sizes. These are determined by the n value passed to the calls to the merge and basicsort functions in listing 4.5.

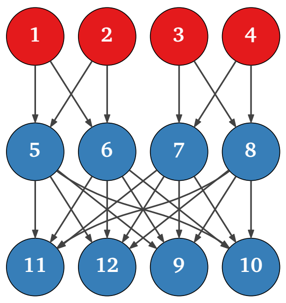
Figure 4.12 shows the task graph for an execution over an array of size N decomposed in 4 parts for the basic sort tasks (MIN_SORT_SIZE=N/4) and for the merge tasks (MIN_MERGE_SIZE=N/4). As we reduce the size of the merge tasks, the graph grows in depth and in complexity.

```
void multisort(long n, T data[n], T tmp[n]) {
   if (n >= MIN_SORT_SIZE*4L) {
      // Recursive decomposition
      multisort(n/4L, &data[0], &tmp[0]);
      multisort(n/4L, &data[n/4L], &tmp[n/4L]);
      multisort(n/4L, &data[n/2L], &tmp[n/2L]);
      multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);

      merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
      merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);

      merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
   } else {
      // base case
      basicsort(n, data);
   }
}
```

Listing 4.5: Main code of the multisort algorithm.

```
void merge(long n, T left[n], T right[n], T result[n*2], long first, long length) {
   if (length <= MIN_MERGE_SIZE) {
      // Base case
      merge_task(n, left, right, result, first, length);
   } else {
      merge(n, left, right, result, first, length/2);
      merge(n, left, right, result, first + length/2, length/2);
   }
}
```

Listing 4.6: Recursive implementation of the merge algorithm that delays the pivoting to the task and divides the work according to the result.



Node	Task
🔴	basicsort
🔵	merge_task

Figure 4.12: Graph of the multisort algorithm when executed over an array that is decomposed in 4 subarrays for the basic sort tasks and the merge tasks.

```
int pivots_are_aligned(T *left, T *right, long n, long leftFirst, long rightFirst) {
    if (leftFirst == 0 || rightFirst == 0 || leftFirst == n || rightFirst == n)
        return 1;

    if (left[leftFirst] <= right[rightFirst] && right[rightFirst-1] <= left[leftFirst])
        return 1;
    if (right[rightFirst] <= left[leftFirst] && left[leftFirst-1] <= right[rightFirst])
        return 1;

    return 0;
}


void find_pivot(T *left, T *right, long n, long first, long *leftFirst, long
  *rightFirst) {
    *leftFirst = first/2L;
    *rightFirst = first/2L;

    if (first == 0)
        return;

    long jumpSize = min(first/2L, n - first/2L) / 2L;
    while (1) {
        if (pivots_are_aligned(left, right, n, *leftFirst, *rightFirst)) {
            return;
        } else if (left[*leftFirst] > right[*rightFirst]) {
            *leftFirst -= jumpSize;
            *rightFirst += jumpSize;
        } else {
            *leftFirst += jumpSize;
            *rightFirst -= jumpSize;
        }
        jumpSize = (jumpSize+1L)/2L;
    }
}
```

Listing 4.7: Function to find the pivots of the merge task.

```
#pragma css task input(n) inout(data)
void basicsort(long n, T data[n]);

#pragma css task input(n, first, length, left, right) output(result{first:length})
void merge_task(long n, T left[n], T right[n], T result[n*2], long first, long
  length);
```

Listing 4.8: Tasks of the multisort benchmark.

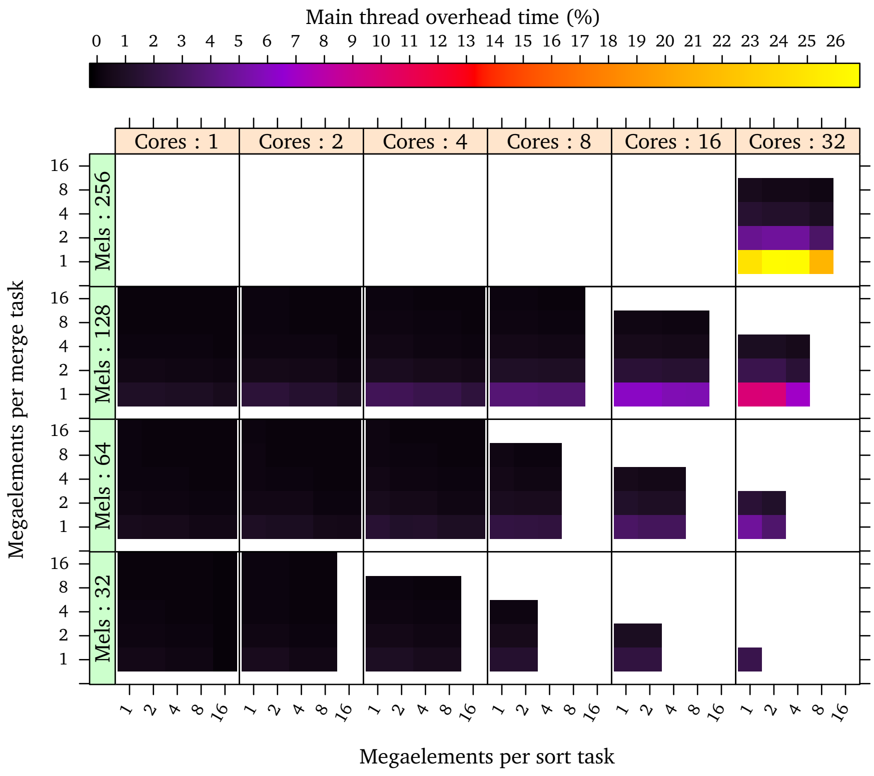


Figure 4.13: Task management overhead on the main thread when running the SMPSs implementation of the multisort algorithm with several problem sizes and several combinations of sort and merge task sizes.

Runtime Overhead

Figure 4.13 shows the overhead of the main thread. To show the effects of the blocking size of the merge and sorting tasks, each panel shows the value for a fixed number of cores and a fixed problem size. The vertical axis determines the size of the merge tasks in megaelements, and the horizontal axis shows the size of the sort tasks, also in megaelements. Each vertical group of panels corresponds to a fixed number of cores, and each horizontal group corresponds to a fixed problem size (N) expressed in megaelements.

The runtime overhead on the main thread is in most cases below 5%. Only when running with 32 cores with very small merge tasks, the overhead gets above 20%. However, as shown in figure 4.14, that amount of overhead does not reduce significantly the amount of time that threads spend running tasks. Instead, the low parallelism is caused by too big blocking sizes.

The previous figures have been generated with a region tree that uses compression. Compression improves the task management overhead compared to executions with uncompressed region trees. Figure 4.15 shows the reduction of task management overhead. Compression reduces the overhead at least 28%, and up to 66%.

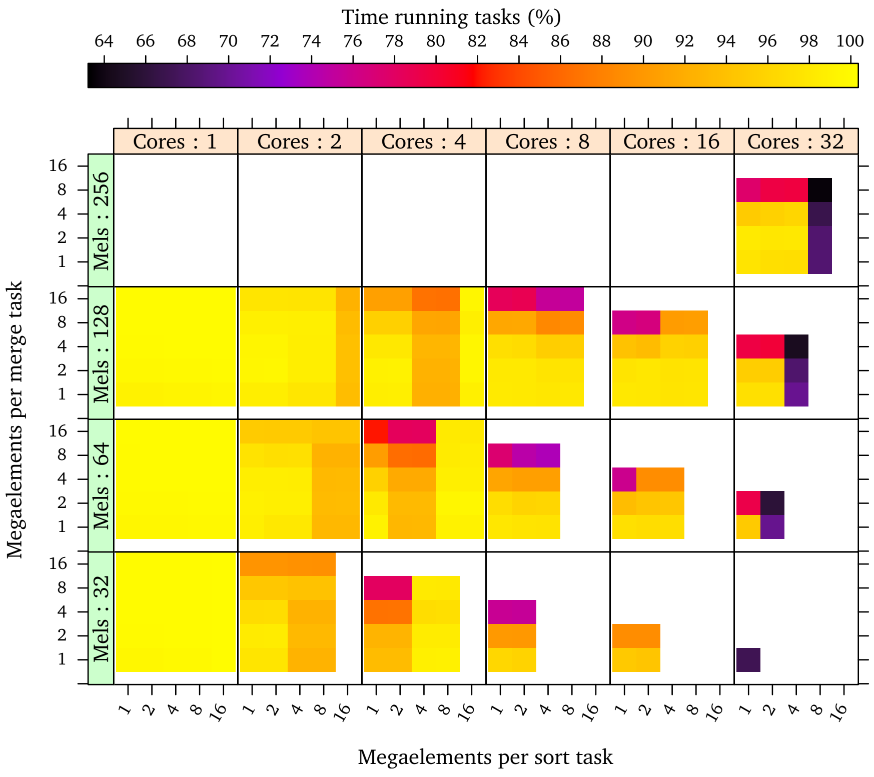


Figure 4.14: Percentage of time time that threads are busy executing task in the SMPSs implementation of the multisort algorithm with several problem sizes and several combinations of sort and merge task sizes.

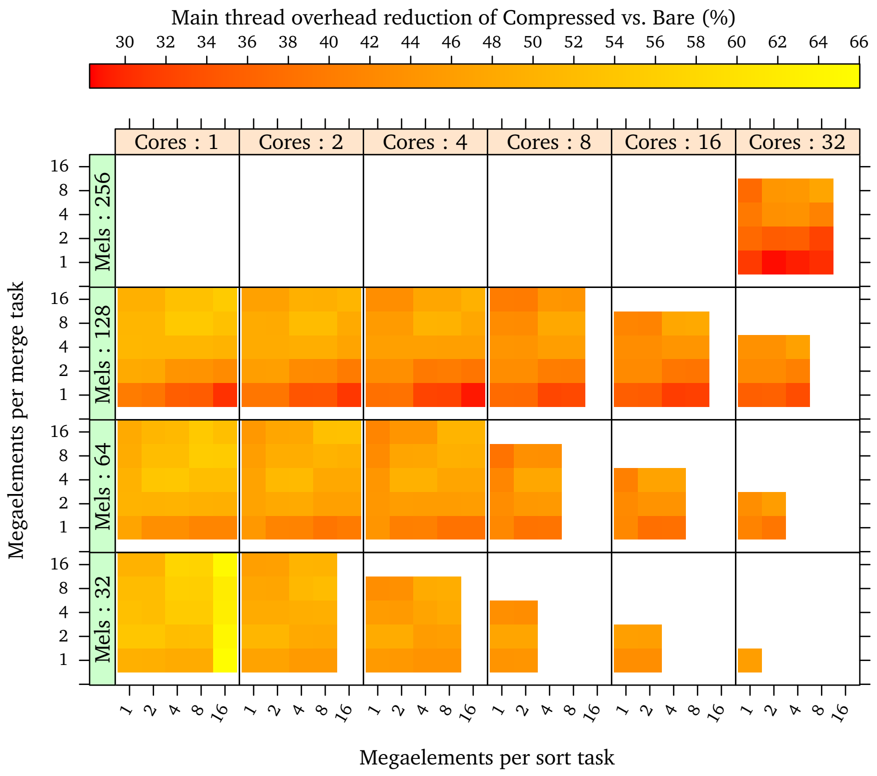


Figure 4.15: Reduction of task management overhead on the main thread when running the SMPSs implementation of the multisort algorithm with a compressed region tree compared to an uncompressed region tree, with several problem sizes and several combinations of sort and merge task sizes.

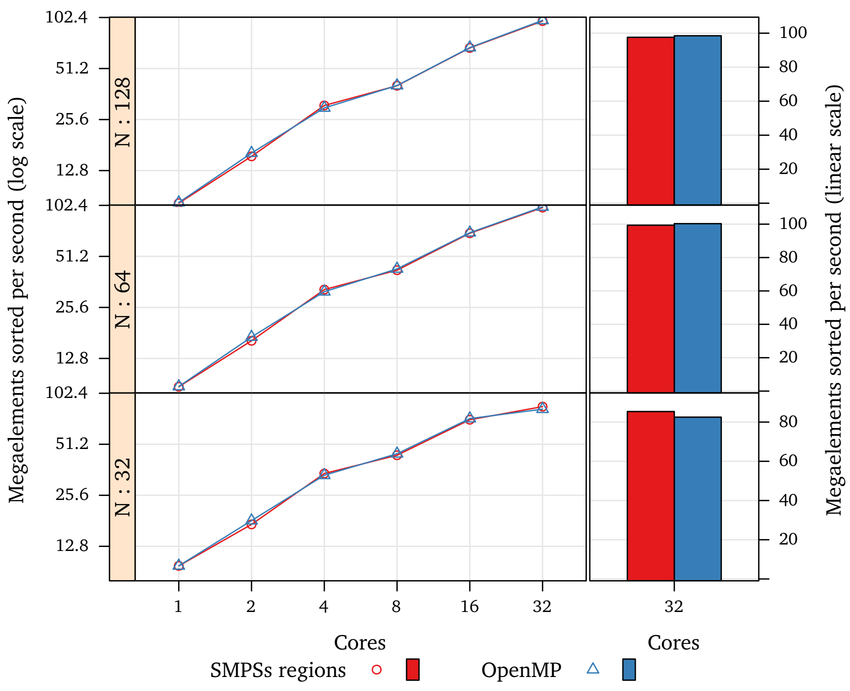


Figure 4.16: Strong scalability of the region-aware SMPSs multisort algorithm and the OpenMP version, and performance of both with 32 cores.

Scalability and Other Implementations

The multisort algorithm is a recursive algorithm and thus it easily fits in task nesting models. We have ported the SMPSs version to OpenMP 3. In the SMPSs version we expand the recursion until the last level, and solve only the base cases with tasks. To preserve its correctness, we take advantage of dependencies to run the partial merge tasks of each nesting level in a valid order.

In the OpenMP version we use the same base code, but instead of only having a flat set of tasks with dependencies, we rely on task nesting. Listing 4.9 show its main code. The core includes barriers to guarantee that data dependencies are respected.

Figure 4.16 shows the strong scalability of the problem with several problem sizes under each implementation. Both perform almost identically. This is a surprising result, since dependency analysis does not uncover additional parallelism but incurs in additional overhead, and the OpenMP version generates the work in parallel by using task nesting.

Table 4.6 summarizes the performance metrics of each implementation.

```
void multisort(long n, T data[n], T tmp[n]) {
    if (n >= MIN_SORT_SIZE*4L) {
        // Recursive decomposition
        #pragma omp task untied
        multisort(n/4L, &data[0], &tmp[0]);
        #pragma omp task untied
        multisort(n/4L, &data[n/4L], &tmp[n/4L]);
        #pragma omp task untied
        multisort(n/4L, &data[n/2L], &tmp[n/2L]);
        #pragma omp task untied
        multisort(n/4L, &data[3L*n/4L], &tmp[3L*n/4L]);

        #pragma omp taskwait

        #pragma omp task untied
        merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
        #pragma omp task untied
        merge(n/4L, &data[n/2L], &data[3L*n/4L], &tmp[n/2L], 0, n/2L);

        #pragma omp taskwait

        #pragma omp task untied
        merge(n/2L, &tmp[0], &tmp[n/2L], &data[0], 0, n);
    } else {
        // base case
        #pragma omp task untied
        basicsort(n, data);
    }
    #pragma omp taskwait
}

void merge(long n, T left[n], T right[n], T result[n*2], long first, long length) {
    if (length <= MIN_MERGE_SIZE) {
        #pragma omp task untied
        merge_task(n, left, right, result, first, length);
    } else {
        merge(n, left, right, result, first, length/2);
        merge(n, left, right, result, first + length/2, length/2);
    }
}
```

Listing 4.9: Code of the multisort benchmark in OpenMP with task nesting.

Cores	N[a]	BS1[b]	BS2[c]	BS1[b]	BS2[c]	Mels/s[d]	Mels/s[d]	IPC[e]	IPC[e]	Eff.[f]	Eff.[f]
										(%)	(%)
1	32	16	16	16	16	9.8	9.8	2.39	2.39	99	99
1	64	8	16	16	1	8.7	8.7	2.30	2.31	99	99
1	128	16	16	16	1	8.2	8.3	2.25	2.27	99	99
2	32	8	8	8	16	17.2	18.2	2.33	2.30	93	99
2	64	16	1	16	1	16.3	17.1	2.31	2.27	93	99
2	128	16	1	16	1	15.5	16.2	2.27	2.23	93	99
4	32	8	1	8	1	34.4	33.6	2.22	2.16	98	99
4	64	16	1	8	2	32.6	31.7	2.17	2.11	99	99
4	128	16	2	16	1	31.0	30.0	2.13	2.06	99	99
8	32	1	1	2	1	44.1	45.0	1.52	1.53	96	97
8	64	2	1	2	1	42.4	43.2	1.50	1.51	97	98
8	128	4	1	4	2	40.5	40.6	1.47	1.47	97	98
16	32	1	1	1	1	71.5	72.6	1.25	1.25	94	96
16	64	4	1	4	1	70.1	70.6	1.24	1.25	96	97
16	128	8	1	4	1	67.7	68.0	1.23	1.22	97	98
32	32	1	1	1	1	85.4	82.5	1.06	1.06	67	64
32	64	1	1	1	1	99.4	100.3	0.93	0.92	94	97
32	128	1	1	2	1	97.6	98.5	0.93	0.93	97	98
32	256	2	2	4	1	95.1	95.7	0.92	0.92	97	99

Implementation: SMPSs regions OpenMP

[a] Megaelements.
[b] Megaelements per sort task.
[c] Megaelements per merge task.
[d] Megaelements sorted per second.
[e] Mean instructions per cycle while running tasks.
[f] Mean time that threads spend running tasks.

Table 4.6: Performance summary of the multisort algorithm implementations.

4.5.3 Fast Fourier Transform

The Fast Fourier Transform (FFT) is a class of algorithms that have $O(n\log(n))$ complexity that calculate the Discrete Fourier Transform of a sequence of floating point complex numbers. The FFT is typically calculated by using a divide-and-conquer approach. FFTs of large number of elements are usually calculated by decomposing the problems into smaller problem sizes and then recomposing the original result from the results of the smaller FFTs.

The FFT can be decomposed using several strategies. Some are only suitable when the number of elements n is a prime number; others only apply to powers of 2; others to composed values; and others work in the general case. Each variant has its requirements and its trade-offs. This topic has been addressed by [Arndt, 2011, pages 410–439] among others.

In this section we parallelize the FFT over double precision complex numbers using the 6-step Matrix Fourier Algorithm described by [Bailey, 1989]. This decomposition strategy is simple and in-place (does not require temporary storage). While this is a unidimensional FFT, the algorithm operates over the data as if it was a bidimensional matrix. The 6 phases consist of 3 transpositions, 2 smaller FFTs over all rows and a multiplication of all elements by so called *twiddle factors*. Figure 4.17 shows the order of the phases and how they access the data.

In our implementation, for performance reasons, we have fused the multiplication phase (3) with one transposition phase (4). Furthermore, to simplify the algorithm we have restricted the data lengths to powers of 4. This way we can lay out the data as a square matrix with a side size that is a power of 2.

Each phase of the algorithm is embarrassingly parallel. Therefore, when this algorithm is parallelized with other programming models, it is usually done following a fork-join strategy. Each phase forks a number of workers that compute independent units of work and then joins them using an implicit or explicit barrier. In SMPSs, the barriers between phases are not necessary since the model can handle the dependencies by itself.

Parallelization with SMPSs using Regions

Let n be the length of the transform and m its square root, which corresponds to the side size of the matrix.

To parallelize the transposition phases we divide the matrix into square submatrices and perform the transposition by swapping and transposing symmetrical submatrices around the matrix diagonal, and just transposing the submatrices of the diagonal. That is, given a matrix A its transposition can be calculated as follows:

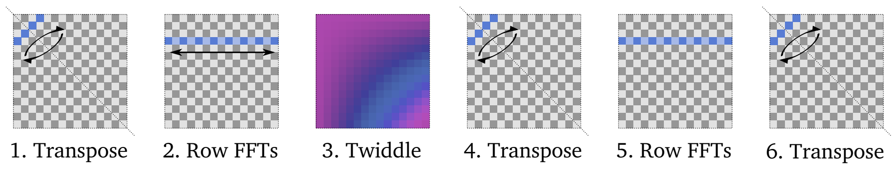


Figure 4.17: Phases of the 6-step Fast Fourier Transform decomposition algorithm.

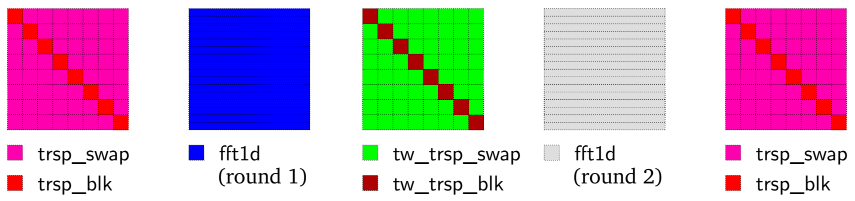


Figure 4.18: Tasks and data layout of the SMPSs version of the FFT.

$$A^t = \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right]^t = \left[\begin{array}{c|c} (A_{11})^t & (A_{21})^t \\ \hline (A_{12})^t & (A_{22})^t \end{array}\right] \tag{4.12}$$

A_{11} and A_{22} would be transposed in-place, and A_{21} and A_{22} would be replaced by the transposed of each other. While this formulation leads to a recursive solution, we have implemented it in an iterative form. The code transposes the matrix in blocks of size tr_bs instead of $m/2$. Listing 4.10 shows the main code of the FFT. Lines 3–7 contain the first transposition. The trsp_blk task transposes in-place a submatrix of the diagonal, and the trsp_swap task takes two submatrices around the diagonal and swaps each by the transposed of the other. Their declarations appear in listing 4.11.

The phase that multiplies the matrix by the twiddle factors does not have any data layout restrictions. Thus, for performance reasons, we have merged it with the transposition phase that follows it. These phases are the third and fourth of the original 6-step algorithm in figure 4.17. The combined phase performs both operations using the same data layout and access patterns as the transposition phase. The code of the phase appears in lines 14-18 of listing 4.10. The tasks, tw_trsp_blk and tw_trsp_swap follow the same data access pattern as the regular transposition tasks, and therefore they have analogous declarations.

The phases that perform the smaller FFT on each row have been parallelized by converting into tasks groups of FFTs that operate over several contiguous rows. These phases appear in lines 10-11 and 21-22 in listing 4.10. The row FFTs are grouped in panels of fft_bs rows and are calculated by the fft1d tasks.

Notice that the data is operated in two different shapes. The transposition phases and the combined phase operate over the matrix in submatrices of tr_bs by tr_bs elements, while the small row FFT phases operates over the matrix in submatrices of fft_bs by m elements. Figure 4.18 shows the data access patterns of the tasks of an execution. In this figure m is 4096, the number of rows per FFT task fft_bs is 256 and the transposition block size fft_bs is 512.

By taking advantage of data dependencies, SMPSs is capable of overlapping the phases of the algorithm. Figure 4.19 shows an execution trace generated with these parameters and 4 cores. Each horizontal colored bar shows a time line of the tasks that a thread has executed. Each color corresponds to a different task, and small flags indicate the beginning and ending of tasks. Notice that tasks from adjacent phases get executed concurrently. This property has the potential to reduce the unbalance that could otherwise appear if there had been barriers between the phases.

To implement the fft1d task we relied on an external sequential implementation of the FFT. The rest of the tasks were fully implemented. In this sense, we manually

```
void fft (double _Complex A[m][m]) {
    // 1. Transpose
    for (long i=0; i<m; i+=tr_bs) {
        trsp_blk (&A[i][i]);
        for (long j=i+tr_bs; j<m; j+=tr_bs)
            trsp_swap (&A[i][j], &A[j][i]);
    }

    // 2. First FFT round
    for (long j=0; j<m; j+=fft_bs)
        fft1d(&A[j][0]);

    // 3 & 4. Twiddle and Transpose
    for (long i=0; i<m; i+=tr_bs) {
        tw_trsp_blk (i, &A[i][i]);
        for (long j=i+tr_bs; j<m; j+=tr_bs)
            tw_trsp_swap (i, j, &A[i][j], &A[j][i]);
    }

    // 5. Second FFT round
    for (long j=0; j<m; j+=fft_bs)
        fft1d(&A[j][0]);

    // 6. Transpose
    for (long i=0; i<m; i+=tr_bs) {
        trsp_blk (&A[i][i]);
        for (long j=i+tr_bs; j<m; j+=tr_bs)
            trsp_swap (&A[i][j], &A[j][i]);
    }
}
```

Listing 4.10: Main code of the 6-step Fast Fourier Transform algorithm in SMPSs.

```
#pragma css task inout(panel{0:fft_bs}{})
void fft1d (double _Complex panel[m][m]);

#pragma css task inout(block{0:tr_bs}{0:tr_bs})
void trsp_blk(double _Complex block[m][m]);

#pragma css task inout(block1{0:tr_bs}{0:tr_bs}, block2{0:tr_bs}{0:tr_bs})
void trsp_swap(
    double _Complex block1[m][m],
    double _Complex block2[m][m]);
```

Listing 4.11: Declaration of the FFT and transposition tasks of the 6-step Fast Fourier Transform in SMPSs.

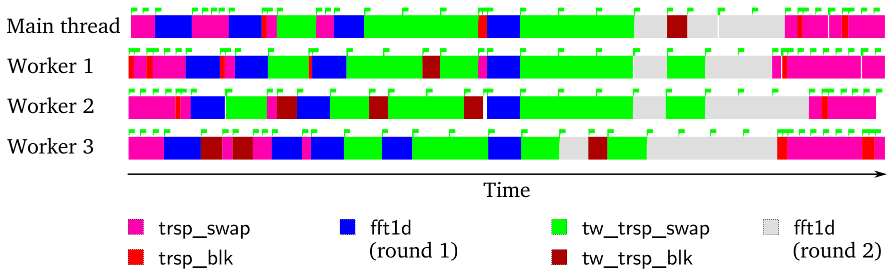


Figure 4.19: Execution trace of the SMPSs version of the FFT with $m = 4096$, $tr\_bs = 512$, $fft\_bs = 256$ and 4 cores.

unfolded the loops of the transpositions and did not cache the twiddle factors. For the sequential FFTs we used the Fastest Fourier Transform on the Web (FFTW) by [Frigo and Johnson, 2005]. This is a library that implements automatically tuned FFT decompositions. The library also provides threaded and MPI versions of the algorithms. We used the sequential calls of version 3.2.2 for the SMPSs case, and in all cases we used “patient” planning.

Performance and Runtime Overhead

To evaluate the performance of the solution, we have measured it with several problem sizes and task granularities. Figure 4.20 shows the floating point performance that we obtained. Notice that the implementation does not scale linearly with the number of cores. The reason is that the FFT algorithms are memory bandwidth demanding and our experiments have a level-3 data cache miss ratio that ranges from 40% to 53%. This problem is made worse by the fact that this chapter does not cover NUMA aspects. Therefore, as the number of NUMA nodes increases, the NUMA affinity decreases and thus performance degrades. This is further confirmed in figure 4.21 which shows that the memory latency increases with the number of cores, and in the worst case it is 7 times higher than in the best case.

The runtime overhead on the main thread is in 76% of the cases below 10%. Figure 4.22 show the metric. Only when running with 32 cores, it gets above 30% and at most 57%, and only for the smallest FFT tasks or transposition tasks. However, as shown in figure 4.23, that amount of overhead is not enough to produce starvation. Instead, starvation only occurs when task granularity is too coarse.

The previous figures have been generated with a region tree that uses compression. Compression improves the task management overhead compared to executions with uncompressed region trees. Figure 4.24 shows that the task management overhead can be reduced by up to 57%.

Scalability and Other Implementations

To rank the performance of the SMPSs implementation we have compared it against a parallel version using threads and another using MPI. All three rely on the FFT implementation from FFTW. The SMPSs version uses FFTW version 3.2.2 compiled without threading. The threaded version uses the same version compiled with threading enabled. And the MPI version uses the MPI implementation of FFTW version

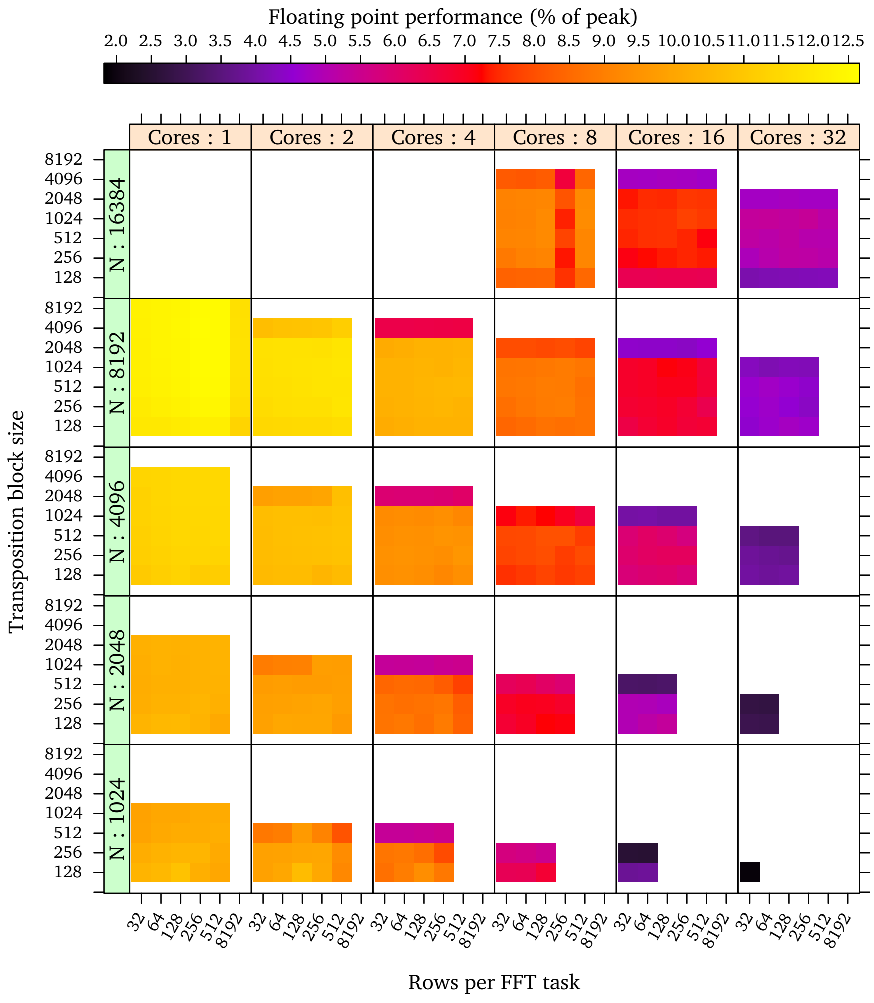


Figure 4.20: Performance of the SMPSs FFT implementation with several problem sizes and blocking sizes.

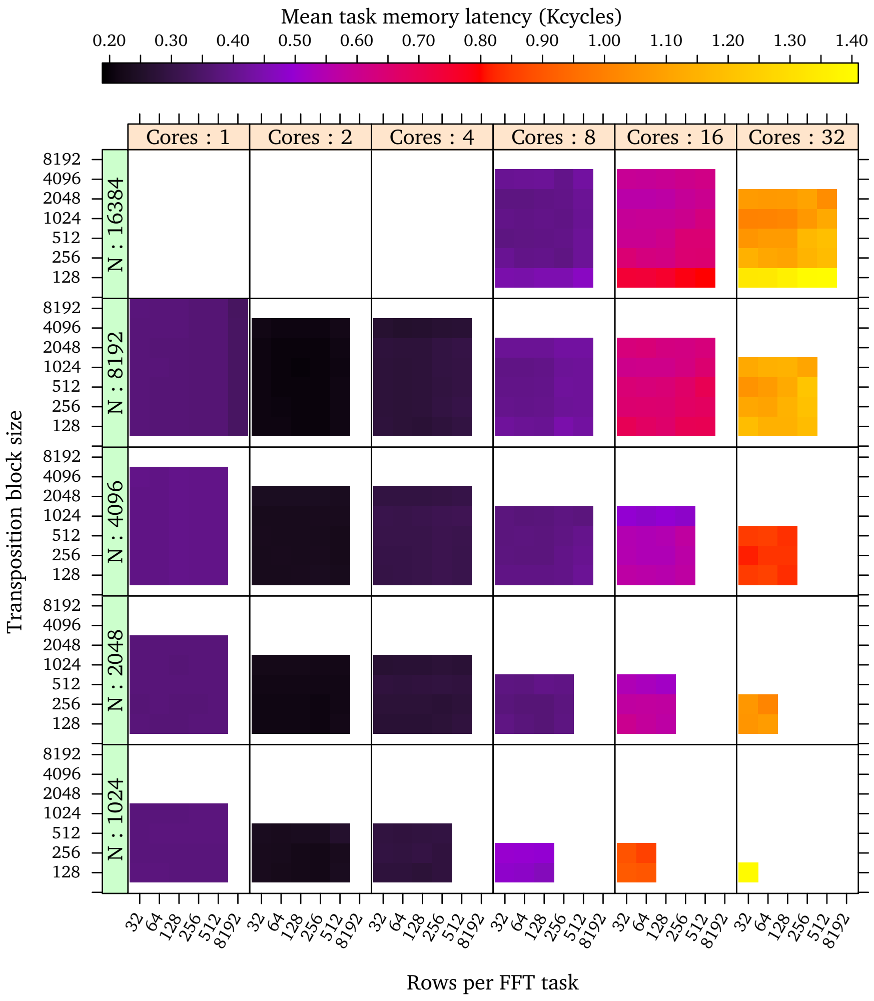


Figure 4.21: Mean task memory latency of the SMPSs FFT implementation with several problem sizes and blocking sizes.

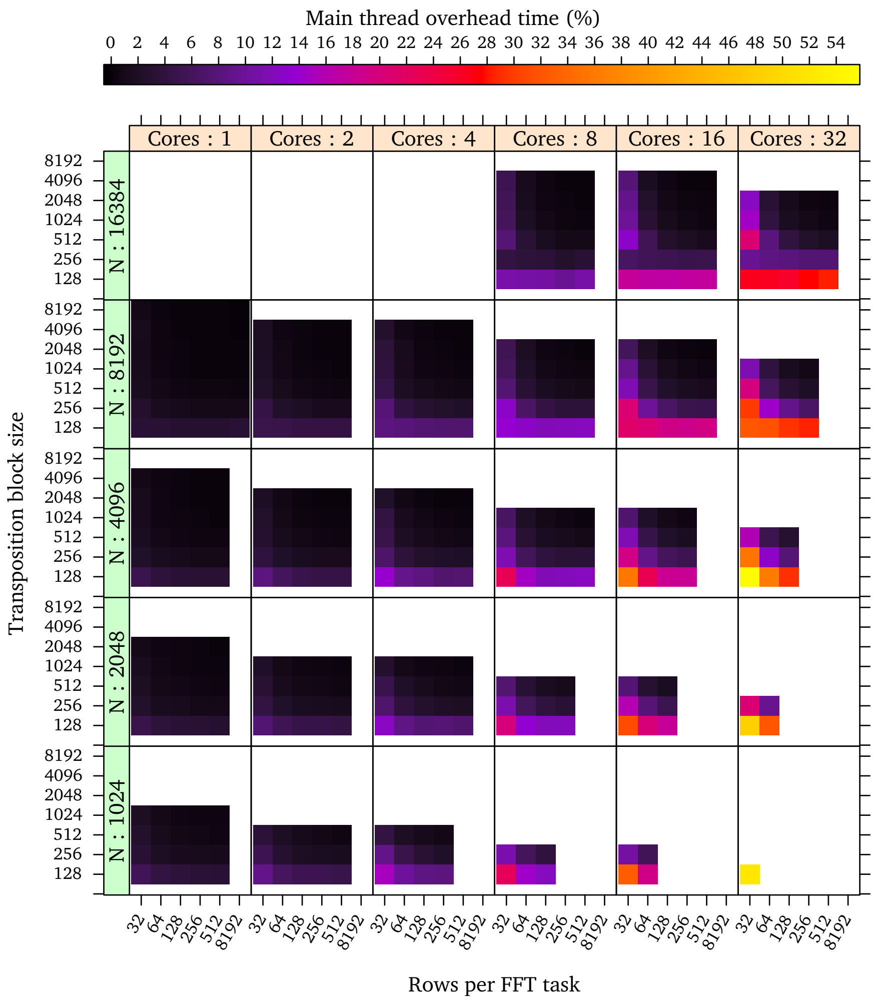


Figure 4.22: Task management overhead on the main thread when running the SMPSs implementation of the FFT algorithm with several problem sizes and several FFT and transpose task sizes.

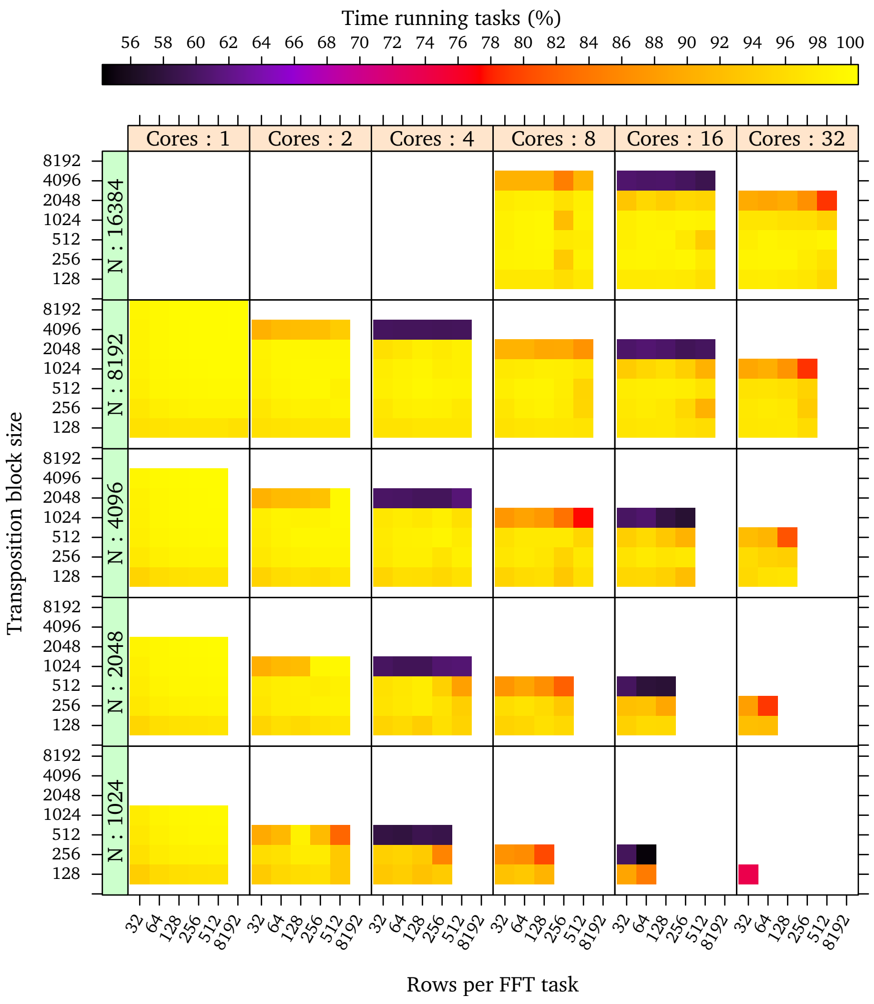


Figure 4.23: Percentage of time time that threads are busy executing task in the SMPSs implementation of the FFT algorithm with several problem sizes and FFT and transpose task sizes.

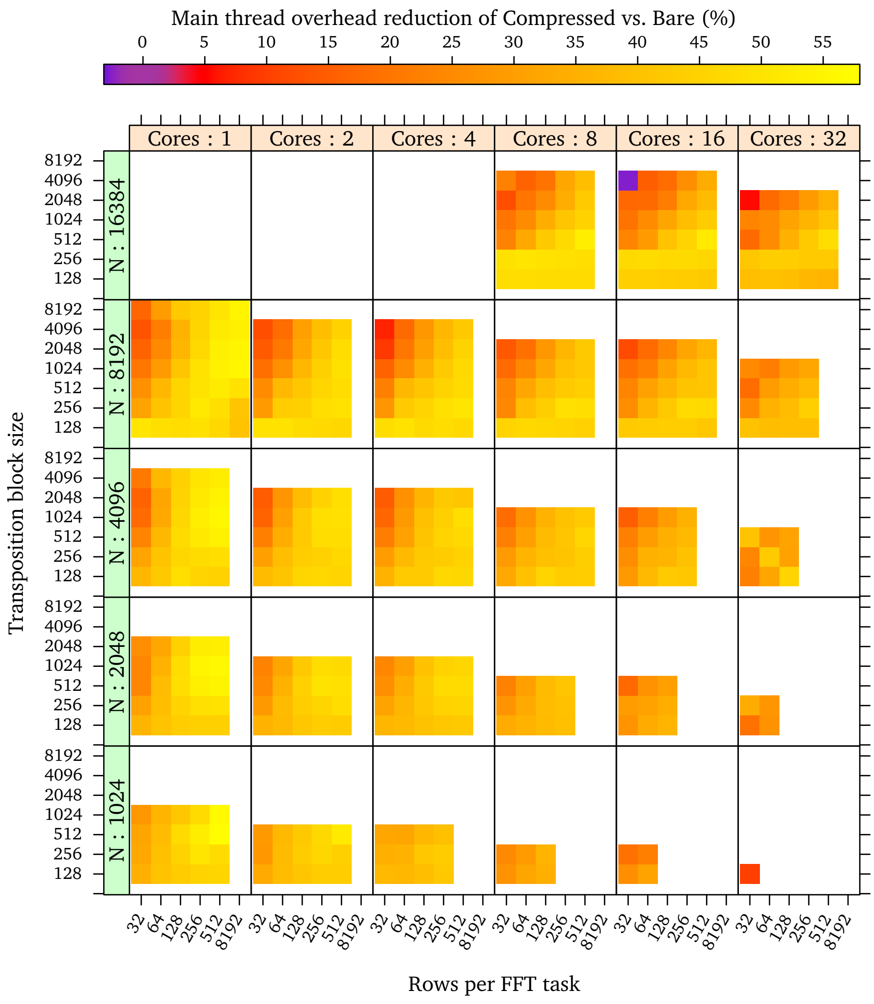


Figure 4.24: Reduction of task management overhead on the main thread when running the SMPSs implementation of the FFT algorithm with a compressed region tree compared to an uncompressed region tree, with several problem and blocking sizes.

3.3alpha1, since version 3.2.2 does not provide any. All of them have been run with “patient” planning.

Since this chapter does not cover NUMA aspects, the SMPSs version and the threaded version have the data page-wise interleaved across the NUMA nodes. The MPI version has the data distributed as defined by the FFTW MPI interface.

Figure 4.25 shows the strong scalability of the problem with several problem sizes under each implementation. The MPI version has the potential to exploit NUMA affinity at the expense of the overhead of data copies. The overhead is only significant when going from 1 to 2 and 4 cores, since in those cases the data copies are redundant. However, this degrades the global scalability of the algorithm, since the redundancy remains within each node. Nevertheless, for the smallest problem sizes it manages to achieve better performance with 32 cores than the rest.

The threaded FFTW version, which uses its own internal threading and work distribution mechanism, in some cases performs worse than the SMPSs version, and in the others it scales worse. Only for the smallest problem size with 1 and 2 cores it outperforms SMPSs.

The SMPSs version with just one core is faster with 4096^2 and 8192^2 elements than the other two implementations. This suggests that our 6-step implementation, even with the overhead of task creation and data dependency calculation, is either a better algorithm than the one that FFTW chooses for that size, or that it is more optimized. For the smallest problem size, the situation is the reverse. However, the scalability of the SMPSs version in that case is better than that of the threaded FFTW, and manages to match it with 4 cores and surpass it with more.

Table 4.7 summarizes the performance metrics of each implementation. Note that the threaded FFTW interface with 1 thread does not use the internal threading infrastructure, and thus we do not have metrics for its parallel efficiency, which should be close to 100%.

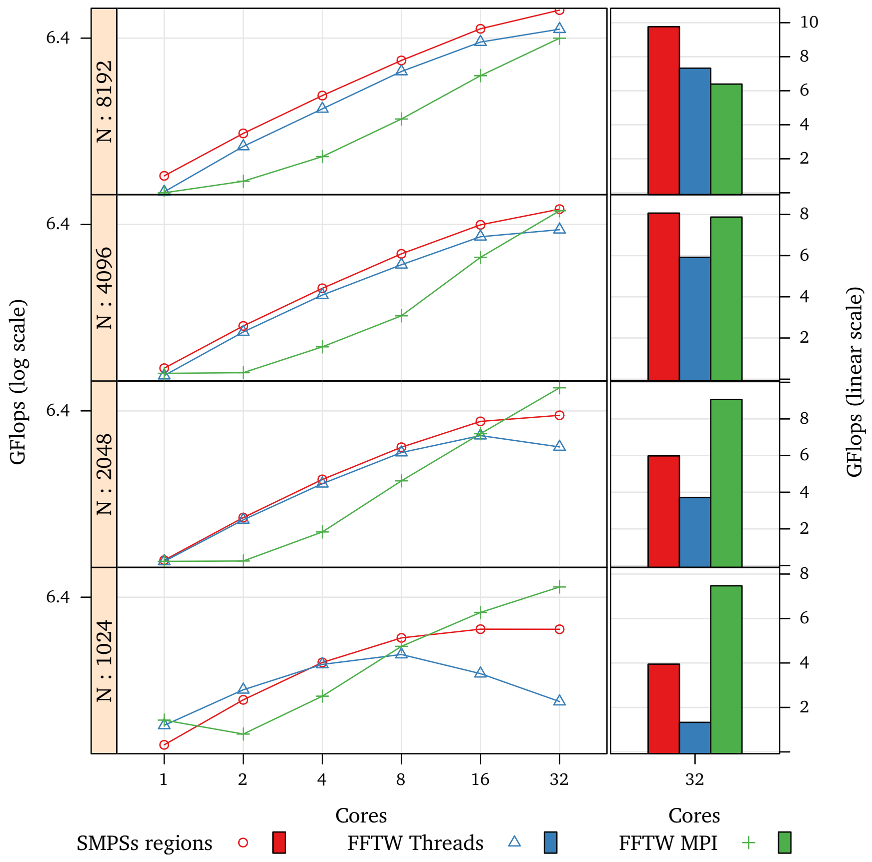


Figure 4.25: Strong scalability of the region-aware SMPSs FFT algorithm and the OpenMP version, and performance of both with 32 cores.

Cores	N[a]	BS1[b]	BS2[c]	GF[d]	GF[d]	GF[d]	IPC[e]	IPC[e]	Eff.[f] (%)	Eff.[f] (%)
1	1024	128	128	0.7	0.9	1.0	2.0	1.7	96	∅
1	2048	128	128	0.7	0.7	0.7	1.9	1.7	96	∅
1	4096	512	4096	0.7	0.7	0.7	1.9	1.7	99	∅
1	8192	256	4096	0.8	0.6	0.6	2.1	1.6	99	∅
2	1024	128	128	1.4	1.6	0.8	1.9	1.5	95	96
2	2048	64	128	1.3	1.2	0.7	1.8	1.5	96	99
2	4096	512	512	1.4	1.3	0.7	1.8	1.7	99	98
2	8192	512	1024	1.5	1.3	0.7	2.0	1.7	99	98
4	1024	128	128	2.4	2.3	1.4	1.8	1.3	92	91
4	2048	256	128	2.3	2.1	1.0	1.6	1.3	96	98
4	4096	512	256	2.4	2.2	1.0	1.6	1.5	98	98
4	8192	512	512	2.7	2.2	1.1	1.8	1.5	98	98
8	1024	128	128	3.5	2.7	3.1	1.3	1.0	90	76
8	2048	128	128	3.7	3.4	2.2	1.3	1.2	96	94
8	4096	256	512	4.1	3.5	1.6	1.4	1.1	97	98
8	8192	256	256	4.6	3.9	1.9	1.5	1.2	97	98
16	1024	64	128	4.0	2.0	5.1	0.8	1.0	84	48
16	2048	128	128	5.5	4.4	4.5	1.0	1.0	95	82
16	4096	256	256	6.4	5.3	3.9	1.1	0.9	97	92
16	8192	128	1024	7.4	6.0	3.6	1.3	1.0	96	94
32	1024	32	128	3.9	1.3	7.5	0.6	0.6	73	33
32	2048	32	128	6.0	3.7	9.1	0.6	0.8	92	51
32	4096	32	128	8.1	5.9	7.9	0.8	0.7	95	79
32	8192	128	128	9.8	7.3	6.4	0.8	0.7	97	81

Implementation: SMPSs regions FFTW Threads FFTW MPI

[a] Matrix side size. The actual size of the FFT is this value squared.
[b] FFT panel height.
[c] Transposition block side size.
[d] Gigaflops per second.
[e] Mean instructions per cycle while running tasks.
[f] Mean time that threads spend running tasks.

Table 4.7: Performance summary of the FFT algorithm implementations.

4.5.4 High Performance Linpack

High Performance Linpack (HPL) is a benchmark that solves a system of linear equations of order n

$$Ax = b \tag{4.13}$$

where $A \in \mathbb{R}^{n\times n}$ is the coefficient matrix, $b \in \mathbb{R}^n$ is the right-hand side, and $x \in \mathbb{R}^n$ is the solution. The benchmark solves the system by first performing the LU factorization with partial pivoting of A as follows:

$$A = PLU \tag{4.14}$$

where P is the pivoting matrix, L is a lower triangular matrix with unit diagonal elements, and U is an upper triangular matrix. Then the algorithm applies the pivoting matrix to b:

$$b^\star = Pb \tag{4.15}$$

From equations 4.13 and 4.14 we obtain

$$PLUx = b \tag{4.16}$$

If we apply the permutation matrix to each side

$$PPLUx = Pb \tag{4.17}$$

and since the permutation matrix is such that $PP = I$, we obtain

$$LUx = b^\star \tag{4.18}$$

If we define

$$y = Ux \tag{4.19}$$

we can calculate y from

$$Ly = b^\star \tag{4.20}$$

by forward elimination. Finally we calculate x from

$$Ux = y \tag{4.21}$$

by backwards substitution.

Parallelization with SMPSs

The algorithm we presented is the same that the dgesv function from the LAPACK library by [Anderson et al., 1990] implements. This function performs the LU decomposition from equation 4.14 by calling the LAPACK dgetrf function. Then it pivots the b vector as in equation 4.15 by calling the dlaswp function. The forward elimination operation from equation 4.20 corresponds to a call to function dtrsm, and the backwards substitution from equation 4.21 corresponds to another call to the same function.

To demonstrate support for incremental parallelization and the ability to leverage existing codes, we have based our implementation on that structure and we have made it parallel by parallelizing the code of some of these functions. The main code is shown in listing 4.12.

```
// LU factorization
tiled_dgetrf(NB, N, N, N, A, IPIV);

// Solve the system with the LU factorization
pivot_vector(N, B, IPIV);        // Pivot vector B
tiled_dtrsv(NB, N, A, B, 0);   // Forward elimination
tiled_dtrsv(NB, N, A, B, 1);   // Backwards substitution
```

Listing 4.12: Main code of the SMPSs implementation of the High Performance Linpack benchmark.

In the listings of this benchmark, all functions whose name begins with tiled_ are functions that decompose a problem into several task calls. Most of those tasks have been implemented as direct calls to the BLAS and LAPACK linear algebra libraries. By using this approach, we can leverage optimized sequential versions of those libraries (i.e. MKL, GotoBLAS, SGI SCSL, ...) as part of our parallel program. Modularity and capability of reuse are important capabilities for productivity and performance. The tiled_dtrsv function is implemented in such a way. The decomposition of simple linear algebra functions using tasks with regions has already been covered in this text.

However, the tiled_dgetrf function is more complex. Listing 4.13 shows the original implementation in Fortran of the dgetrf function from LAPACK. And listing 4.14 shows the SMPSs version. Notice that the code is almost a direct translation from Fortran to C with calls to regular functions replaced by their tiled counterparts. These are implemented similarly to the tiled matrix multiplication and consist of single and double nested loops that call to BLAS and LAPACK tasks.

For performance reasons, the base case and the current panel are handled by calls to tasks that call the dgetrf function instead of dgetf2. The first is a level-3 function and has better performance due to better cache use. Whereas, the second is a level-2 function and in general has lower performance.

The algorithm accesses horizontal and vertical subpanels of the matrix at different stages. Figure 4.26 shows the sequence of regions accessed over matrix A and the pivoting vector IPIV for a given iteration. The sequence from left to right matches the order of the actual operations. Solid colors indicate that the data is updated, and stripped areas that the data is only read. Colors indicate the type of task that updates an area or that has updated it. Notice that the regions are partially overlapping and a given position may be accessed as part of an horizontal panel, a vertical panel and a square in the same iteration. These aspects make this algorithm very hard to write using a tiled data layout, and thus it makes good use of regions.

The granularity of the main decomposition is controlled by the NB parameter, which determines the width of vertical panels and the height of horizontal panels. The other dimensions are dynamic and depend on how many iterations have been executed or how many are left. That is, some are j and some are N—j or N—j—NB. The tiled_dlaswp function in turn is decomposed into tasks that operate over vertical panels that are NB elements wide. The tiled_dtrsm function uses tasks of NB by NB elements. And finally the tiled_dgemm function also uses tasks that operate over blocks of the same size.

Our implementation of the algorithm also includes a verification phase that has also been parallelized using the same strategy. The verification is the one proposed

```
subroutine dgetrf(m, n, a, lda, ipiv, info)
   integer info, lda, m, n, ipiv(*)
   double precision a(lda, *)
```
Function Declaration

```
   nb = ilaenv(...)
```
Determine Block Size

```
   if(nb.le.1 .or. nb.ge.min(m, n)) then
      call dgetf2(..., a, ipiv, ...)
```
Direct Case

```
   else
      do 20 j = 1, min(m, n), nb
```
Main Loop Header

```
         jb = min(min(m, n)–j+1, nb)
         call dgetf2(..., a(j, j), ipiv(j), ...)
```
Handle Current Panel

```
         do 10 i = j, min(m, j+jb–1)
            ipiv(i) = j − 1 + ipiv(i)
   10   continue
```
Adjust Pivots

```
         call dlaswp(..., a, ipiv, ...)
```
Left-Side Row Swaps

```
         if(j+jb.le.n) then
            call dlaswp(..., a(1, j+jb), ipiv, ...)
```
Right-Side Row Swaps

```
            call dtrsm(..., a(j, j), a(j, j+jb), ...)
```
Compute U Rows

```
            if(j+jb.le.m) then
               call dgemm(..., a(j+jb, j), a(j, j+jb), a(j+jb, j+jb))
            end if
```
Update Trailing Matrix

Listing 4.13: LU decomposition as implemented by LAPACK.

```
void tiled_dgetrf(integer NB, integer M, integer N,
   integer LDA, double A[N][LDA], integer IPIV[])
{                                                        Function Declaration
-------------------------------------------------------------------------------
   integer jb = min(min(M, N), NB);                      Determine Block Size
-------------------------------------------------------------------------------
   if (M <= NB || N <= NB)
      dgetrf_tile(..., A, IPIV);                                  Direct Case
-------------------------------------------------------------------------------
   else
      for (integer j=0; j < min(M, N); j += jb) {           Main Loop Header
-------------------------------------------------------------------------------
         jb = min(min(M, N)−j, jb);
         dgetrf_tile(..., &A[j,j], &IPIV[j]);             Handle Current Panel
-------------------------------------------------------------------------------
         if (j != 0) {
            tiled_add_scalar(jb, ..., &IPIV[j]);                Adjust Pivots
-------------------------------------------------------------------------------
            tiled_dlaswp(jb, ..., A, IPIV);
         }                                               Left-Side Row Swaps
-------------------------------------------------------------------------------
         if (j+jb < N) {
            tiled_dlaswp(jb, ..., &A[j+jb,0], IPIV);     Right-Side Row Swaps
-------------------------------------------------------------------------------
            tiled_dtrsm(jb, ..., &A[j,j], &A[j+jb,j]);        Compute U Rows
-------------------------------------------------------------------------------
            if (j+jb < M)
               tiled_dgemm(jb, ...,
                  &A[j,j+jb], &A[j+jb,j], &A[j+jb,j+jb]);
                                                       Update Trailing Matrix
-------------------------------------------------------------------------------
```

Listing 4.14: LU decomposition as implemented in SMPSs.

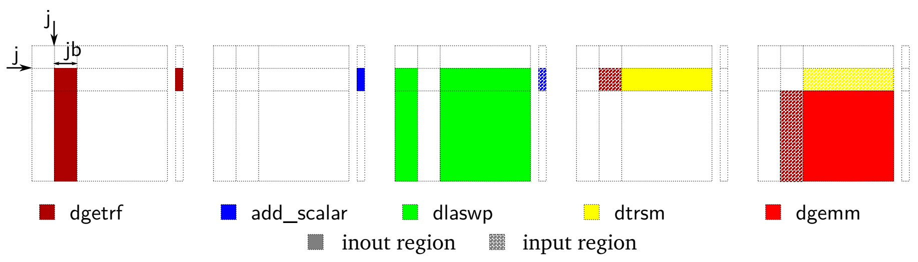


Figure 4.26: Regions accessed by the tasks of one iteration of the LU decomposition over the matrix and the pivoting vector.

by [Luszczek et al., 2006] and consists in verifying that the error of the solution is within a given maximum determined by the arithmetic precision.

Performance and Blocking

When we execute the algorithm as is, we achieve a level of performance that is below our expectations. Figure 4.27(a) shows an execution trace with 32 cores with a matrix of 8192 × 8129 elements and a block size NB of 1024 elements. Each horizontal set of colored bars indicates what kind of task a given thread was executing at a given time. Small notches on top of each bar delimit when a task begins and when it ends. Notice that most segments of each row are white. This indicates that threads spend more time idle than running tasks.

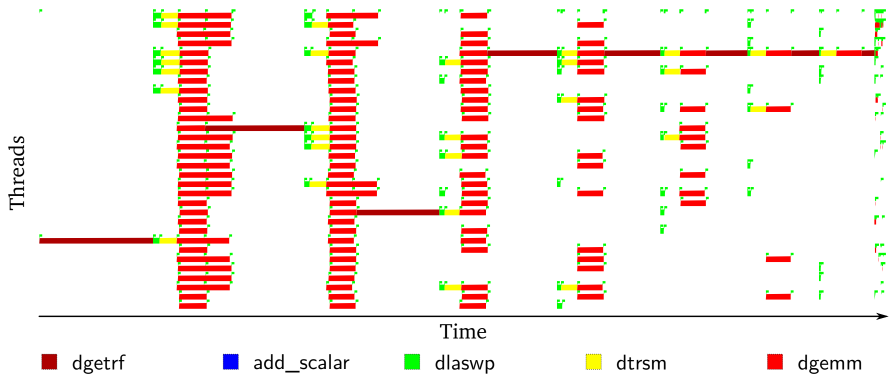
The algorithm is such that every task of an iteration depends either directly or indirectly form the first dgetrf task of that iteration. The dgetrf task, in turn, depends on some of the dgemm tasks of its previous iteration. Thus, whenever the dgetrf task gets delayed, it becomes a bottleneck, and threads starve. Therefore, to achieve acceptable effective parallelism, the dgetrf task must be executed as soon as possible and it must be proportionally small enough compared to the rest of the remaining work of the iteration so that it can finish before the rest that runs in parallel.

In figure 4.27(a), the granularity is too coarse to achieve good parallelism. When we reduce NB to 512, the effective parallelism grows from 18% to 40%. However, that value is still too low. If we reduce NB by half again, we obtain the trace in figure 4.27(b). Note that the time scale is different for each case. The new trace has 75% effective parallelism. This improvement comes at the expense of more runtime overhead and lower task performance. On one hand, the total number of tasks becomes almost 40 times as many as in the first trace. This raises the runtime overhead from less than 1% of the time to 17%. Since tasks perform less computation, their floating point operations per cycle decrease by 30% on average.

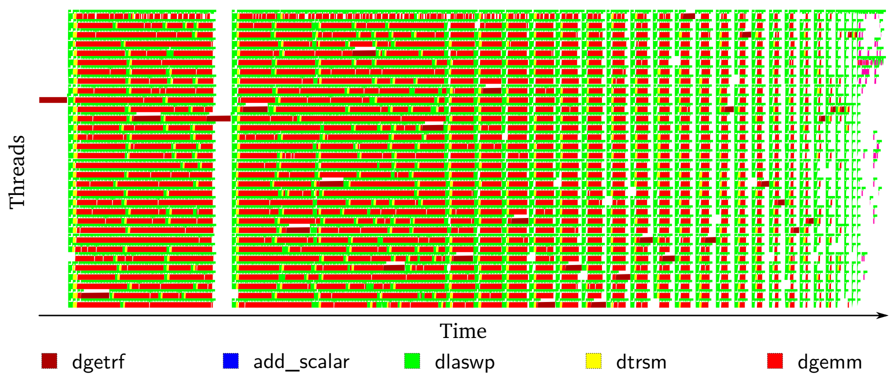
Small tasks also produce scheduling contention which causes the fourth dgetrf task to be delayed, hence the unbalance in the first third of the execution.

The difference between both traces shows that as we reduce the blocking size, the ratio between the duration of the dgemm tasks and the dgetrf tasks grows. Therefore, the blocking strategy that we follow is too coarse for the dgetrf task, and too thin for the dgemm decomposition.

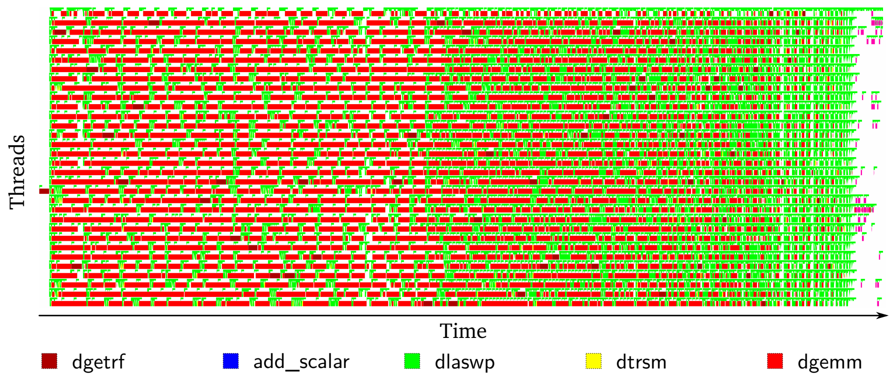
To alleviate the first symptom, we need to decompose the dgetrf_tile calls. However, since the tiled_dgetrf function itself is a valid decomposition, we can alter the function to make it recursive. Therefore, we add a recursion level parameter that



(a)



(b)



(c)

Figure 4.27: Execution traces of HPL: (a) with coarse granularity; (b) with thin granularity; and (c) with adaptive granularity.

we set and control to allow two levels of recursion. Then we change the condition in line 3 of listing 4.14 to include a check for the base case, and we replace the call in line 10 to itself.

To alleviate the second symptom, we need to reduce the amount of tasks generated at each iteration without hurting the potential parallelism. Notice that the dgetrf task is usually the bottleneck. Therefore, to keep threads as busy as possible we need to generate enough tasks, and to execute the dgetrf tasks as soon as possible. Each dgetrf task depends directly on the dgemm tasks that update its region in the previous iteration. From now on, we will refer to that region as the *critical region*.

The key to making the dgetrf tasks execute as soon as possible is to make the tasks that update the critical region also run as early and as fast as possible. To do that, we can set the granularity of those tasks to maximize the parallelism of the region. That is, we make it so that we generate a number of tasks close to the number of threads.

To avoid generating too many small tasks, we adapt the granularity of the other regions to minimize the amount of tasks and to maintain enough parallelism. That is, for each decomposition, we generate also a number of tasks similar to the number of threads for the updates outside of the critical region.

To implement the new granularity policy, we have changed the code to use two granularity variables that are calculated at each iteration. A first one, called jb determines the size of the tasks of the critical region. And a second one jb2 determines the size of the rest of the tasks. To control them we set a maximum block size maxBS and a minimum block size minBS, and derive the actual block sizes of each part from those, the amount of calculations of the current iteration and the number of threads nthr.

Listing 4.15 shows the new code. In line 5 we calculate the block size for the first dgetrf task and use it to decide whether to solve it with the base case or to decompose it. Line 10 determines the block size of the current iteration by taking into account the size of the matrix that will be updated. In line 11 we replaced the regular call to the dgetrf task by a recursive call. In line 17 we calculate the second blocking size that we use for only the dgemm tasks after the critical region.

Figure 4.27(c) shows an execution trace with the new code. Notice that the duration of the dgetrf tasks is much closer to the duration of the dgemm tasks. Moreover, as the algorithm advances, the duration of the tasks gets smaller to increase the amount of parallelism.

These changes allows us to reduce the time that threads spend idle. Figure 4.28 shows the effective parallelism increment. For simplicity, the figure takes the best performing minimum block size of the optimized version and matches the maximum block size with the block size of the unoptimized version. Notice that the effective parallelism of the optimized version can be up to 12 times as much as the unoptimized one.

The additional parallelism comes at the expense of lower task floating point performance. Figure 4.29 shows the difference which can be as high as 44% less floating point task performance. Despite this, the total performance of the algorithm remains up to 10 times higher, as figure 4.30 shows.

Figure 4.31 shows the effect of the maximum and minimum blocking sizes have on the absolute performance. Notice that the minimum blocking size is the most dominating factor since it has the deepest impact on parallelism. However, the maximum blocking size, through modest improvements allows us to to further achieve better performance than would be possible by using the same value for both.

```
void tiled_dgetrf(integer maxBS, integer minNB, integer M, integer N,
   integer LDA, double A[N][LDA], integer IPIV[], integer RL)
{
   integer S = min(M, N);                                  Function Declaration
-------------------------------------------------------------------------------
   integer jb = calculate_small_step(S, nthr, maxBS, minNB);
   if (S <= jb || RL == 0)
      dgetrf_tile(..., A, IPIV);                                      Base Case
-------------------------------------------------------------------------------
   else
      for (integer j=0; j < S; j += jb) {                      Main Loop Header
-------------------------------------------------------------------------------
         jb = calculate_small_step(S−j, nthr, maxBS, minNB);
                                                     Critical Region Block Size
-------------------------------------------------------------------------------
         tiled_dgetrf(maxBS, minNB, ..., &A[j,j], &IPIV[j], RL−1);
                                                 Solve Current Panel Recursively
-------------------------------------------------------------------------------
         if (j != 0) {
            tiled_add_scalar(jb, ..., &IPIV[j]);                  Adjust Pivots
-------------------------------------------------------------------------------
            tiled_dlaswp(jb, ..., A, IPIV);
         }                                                 Left-Side Row Swaps
-------------------------------------------------------------------------------
         if (j+jb < N) {
            integer jb2 = calculate_big_step(S−j, nthr, jb, maxBS, minNB);
                                                        Block Size of the Rest
-------------------------------------------------------------------------------
            tiled_dlaswp(jb, ..., &A[j+jb,0], IPIV);     Right-Side Row Swaps
-------------------------------------------------------------------------------
            tiled_dtrsm(jb, ..., &A[j,j], &A[j+jb,j]);         Compute U Rows
-------------------------------------------------------------------------------
            if (j+jb < M)
               tiled_dgemm(jb, jb2, ...,
                  &A[j,j+jb], &A[j+jb,j], &A[j+jb,j+jb]);
                                                       Update Trailing Matrix
-------------------------------------------------------------------------------
```

Listing 4.15: LU decomposition as implemented in SMPSs with recursion and dynamic granularity.

```
void tiled_dgemm(
    integer smallStep, integer bigStep,
    integer M, integer N, integer K,
    double ALPHA, integer LDA, double A[K][LDA],
    integer LDB, double B[N][LDB],
    double BETA, integer LDC, double C[N][LDC]
) {
    integer j_s = min(smallStep, N);
    for (integer j = 0; j < N; j += j_s) {
        j_s = find_step_progression(j == 0, N − j, j_s, bigStep);
        for (integer l = 0; l < K; l += smallStep) {
            integer l_s = min(smallStep, K−l);
            integer i_s = min(smallStep, M);
            for (integer i = 0; i < M; i += i_s) {
                i_s = find_step_progression(j == 0, M − i, i_s, bigStep);
                dgemm_tile(i_s, j_s, l_s, ALPHA, LDA, &A[l][i], LDB, B[j][l],
                  BETA, LDC, C[j][i]);
            }
        }
    }
}
```

Listing 4.16: Tiled matrix multiplication used in the LU decomposition with dynamic block size.

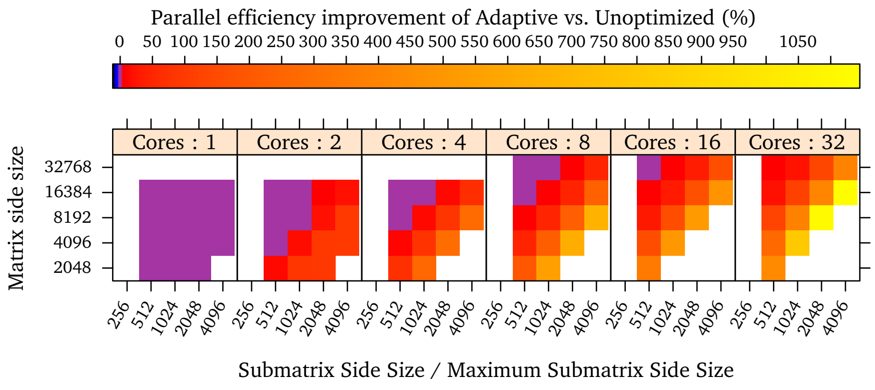


Figure 4.28: Effective parallelism improvement when running the optimized HPL implementation compared to the unoptimized code.

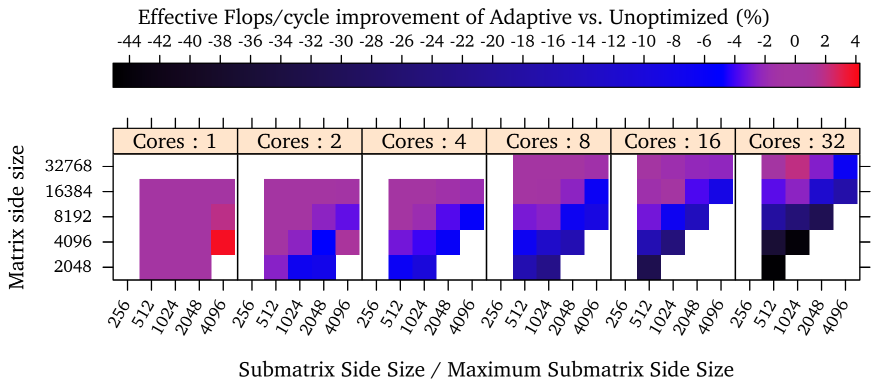


Figure 4.29: Task floating point performance degradation when going from the unoptimized HPL implementation to the optimized code.

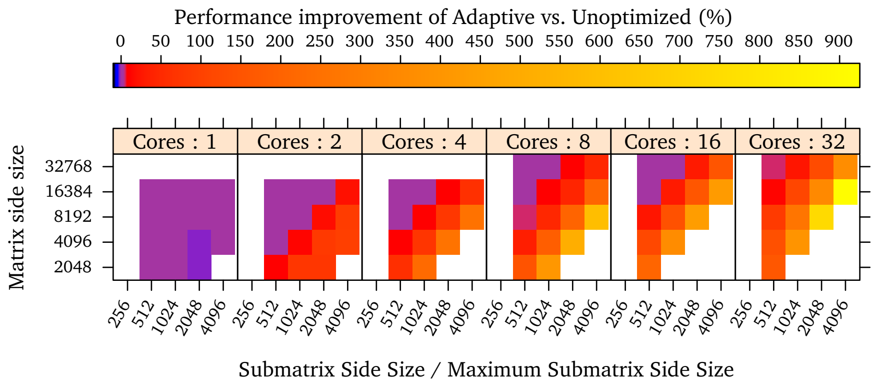


Figure 4.30: Absolute performance improvement when going from the unoptimized HPL implementation to the optimized code.

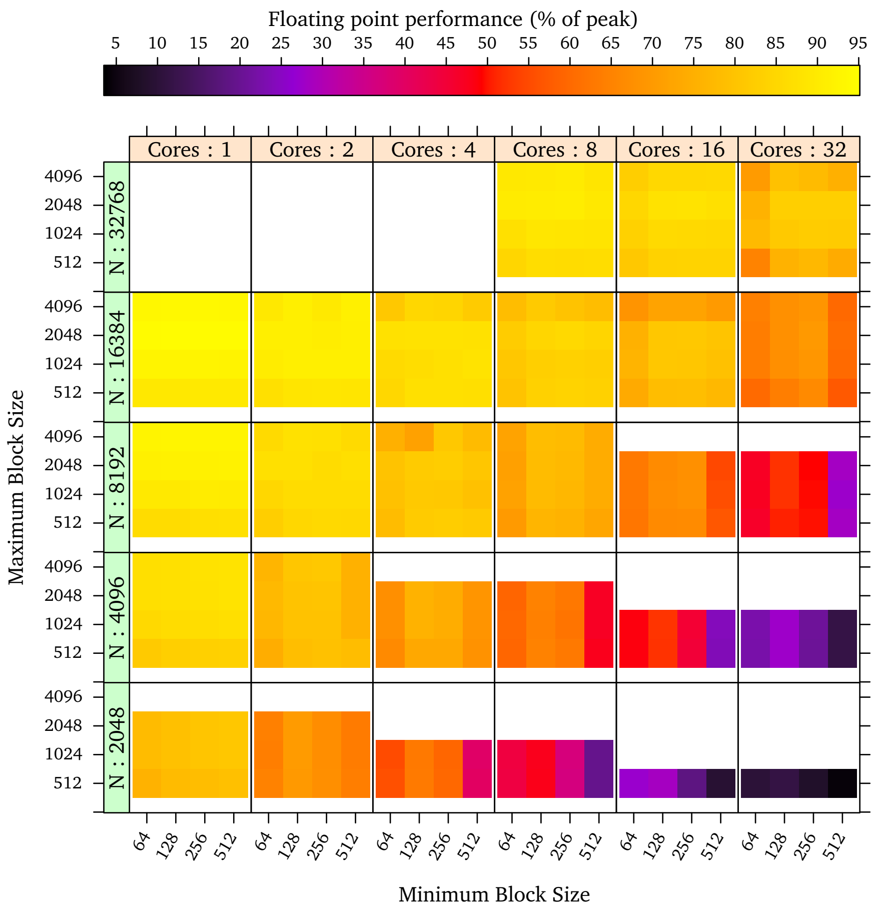


Figure 4.31: Performance of the SMPSs HPL implementation with several problem sizes and blocking sizes.

Scalability and Other Implementations

Since the SMPSs HPL is an implementation of an already existing algorithm, to evaluate its performance we take as reference the original *Linpack TPP benchmark*. This is a highly optimized MPI version of the algorithm that is used for evaluating the performance in the top 500 supercomputer list.

The reference MPI implementation has look-ahead coded by hand. That is, where the SMPSs version overlaps the dgetrf tasks with the dgemm tasks of the previous outer iteration automatically by taking benefit of data dependencies, the MPI version has that behavior coded by hand.

The MPI version also uses a block-cyclic distribution with small block sizes, which improves the balance of this specific problem according to [Dongarra and Walker, 1993]. However, this data distribution leads to a decomposition similar to the LAPACK level-2 dgetf2, which has less locality and is more complex. Moreover, this forces the MPI version to have more frequent communications than a level-3 style implementation would require.

To compare with another shared memory implementation, we have made a third implementation. The code consists of a direct call to dgetrf and two calls to dtrsv. We link this code with the parallel version of the MKL library, which in turn uses OpenMP.

To make the comparison fair, we used the same MKL version for the BLAS and LAPACK functions called in the tasks, the MPI version, and the direct MKL implementation. Figure 4.32 shows the strong scalability of the problem with several problem sizes under each implementation. The SMPSs and the MPI implementations scale and perform on par on most cases. Only for the two smallest problem sizes, the SMPSs version scales worse with more than 8 cores. The general parity of both implementations is a surprising result, since the reference implementation is highly tuned and takes benefit of memory affinity, whereas the SMPSs version is NUMA-unaware.

The parallel MKL version achieves similar performance up to 8 cores, but with more it scales worse.

Table 4.8 summarizes the performance metrics of each implementation. In addition, in the executions for matrices with 16384 elements or more, the SMPSs and the MPI implementations perform similarly, and the difference is always below 10 GFlops. Notice that with 32 cores, the parallel MKL implementation has less parallel efficiency and lower effective floating point instructions per cycle. These symptoms seem to indicate that it does not use look-ahead to reduce the unbalance, and instead relies in a finer-grained decomposition to compensate for it. In addition all the parallel MKL executions with 2 threads produced incorrect results, whereas the same implementation of the operations was used for all other implementations and number of threads.

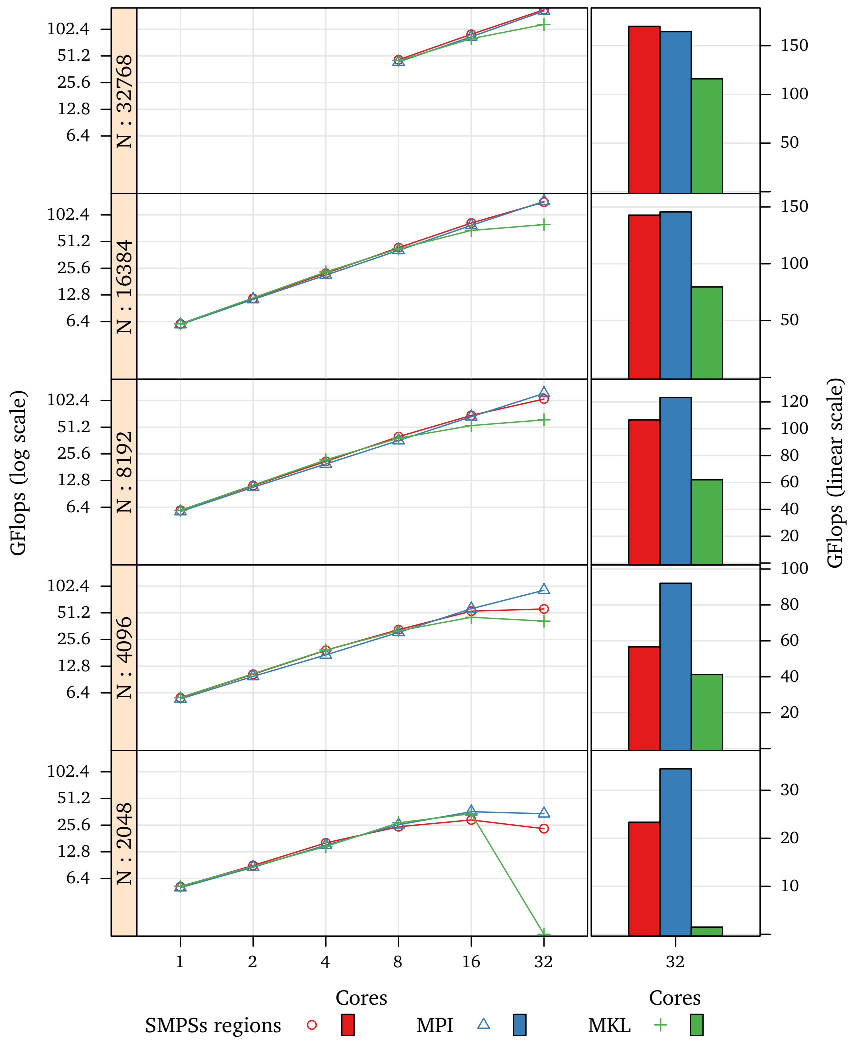


Figure 4.32: Strong scalability of the SMPSs HPL algorithm and the reference MPI implementation, and performance of both with 32 cores.

Cores	N[a]	BS1[b]	BS2[c]	GF[d]	GF[d]	GF[d]	FPC[e]	FPC[e]	Eff.[f] (%)	Eff.[f] (%)
1	2048	512	1024	5.2	5.1	5.2	3.3	3.3	99	99
1	4096	512	2048	5.6	5.5	5.7	3.5	3.6	99	99
1	8192	128	4096	5.9	5.7	5.9	3.7	3.7	99	99
1	16384	128	2048	6.0	5.9	6.0	3.8	3.8	99	99
2	2048	128	2048	9.0	8.6	∅	3.0	∅	93	∅
2	4096	256	4096	10	9.8	∅	3.4	∅	96	∅
2	8192	128	4096	11	10	∅	3.6	∅	97	∅
2	16384	512	4096	11	11	∅	3.8	∅	97	∅
4	2048	128	512	16	15	14	2.8	2.5	90	93
4	4096	128	2048	19	17	19	3.2	3.2	95	96
4	8192	256	512	21	19	21	3.4	3.5	97	98
4	16384	512	1024	22	21	23	3.6	3.7	98	98
8	2048	128	512	24	25	27	2.4	2.7	82	82
8	4096	128	2048	33	30	32	2.8	2.9	91	89
8	8192	128	4096	40	36	38	3.3	3.2	95	93
8	16384	256	2048	43	40	42	3.5	3.5	97	95
8	32768	256	2048	46	43	45	3.7	3.7	98	97
16	2048	128	512	29	36	34	1.9	2.3	63	63
16	4096	128	1024	53	56	45	2.4	2.4	87	77
16	8192	256	1024	69	67	53	3.0	2.5	91	83
16	16384	256	2048	82	77	68	3.4	3.0	95	89
16	32768	256	2048	89	84	80	3.6	3.4	98	92
32	2048	128	512	23	34	1.5	1.4	1.0	37	27
32	4096	128	512	56	92	41	1.8	1.4	65	63
32	8192	128	2048	106	123	61	2.4	1.7	88	73
32	16384	256	2048	142	145	79	3.0	2.1	92	77
32	32768	128	2048	169	164	115	3.4	2.8	97	81

Implementation:	SMPSs regions	MPI	MKL

[a] Matrix side size.
[b] Minimum submatrix side size
[c] Maximum submatrix side size
[d] Gigaflops per second.
[e] Mean floating point operations per cycle while running tasks.
[f] Mean time that threads spend running tasks.

Table 4.8: Performance summary of the HPL algorithm implementations.

Chapter 5

Exploiting Non-Uniform Memory Access

5.1 Introduction

Performance is one of the main driving factors behind the development of High Performance Computing applications. The main aspects that determine the performance of parallel applications are the overheads added by the programming paradigm and runtime, the level of parallelism that is achieved, and the performance of the units of computation. The previous chapters of this thesis cover the first two aspects of the SMPSs programming model. This chapter analyzes task performance from the point of view of memory affinity in Non-Uniform Memory Access (NUMA) systems and proposes techniques to take advantage of them.

The ability to benefit from the potential computational performance of processors is hindered in many applications by the performance gap between the memory system and the execution units of the processor. Some of the physical characteristics of memory system can be improved to achieve better performance. However, changes in this area are limited by physical constraints and cost-effectiveness.

Non-Uniform Memory Access (NUMA) systems increase the total system memory bandwidth by adding more memory buses and physically connecting them in groups. In this text we will refer to *NUMA node* as a group of processors and memory at are connected directly. Each node consists of one or more memory buses connected to one or more processors and memories that are considered local, and an interconnection network that connects them to the other nodes. For brevity, in the rest of this text we will refer to the local memory of a NUMA node, as a *memory*.

The interconnection network allows processors from one node to access the memory of other nodes. However, the performance of these accesses depends on the characteristics of the interconnect, and thus is different to that of local memory. Because of this, the location of the data and the location of the computation that accesses it are important factors that determine the performance of applications in NUMA systems.

This chapter concentrates on NUMA systems that have identical cores and identical nodes. Thus, we differentiate between *local* memory and *remote* memory. A memory is local to a core when its accesses have the least latency on that core. All others are considered remote memories. Those in turn may also have different varying performance characteristics.

5.2 Previous Work

Previous research on software techniques to exploit the performance of NUMA systems has covered mainly operating systems and programming models. An overview of the techniques for operating systems has been published by [LaRowe Jr., 1991]. Most research concentrates on strategies to manage the data, and more specifically on *data migration* and *data replication*, and few cases consider *moving the computation to the data*.

Previous work on programming models includes the work of [Nikolopoulos et al., 2000], which proposes a solution in which the OpenMP runtime monitors page usage and determines page migration and duplication. This research is similar to the research done at the operating system level and applies most of their ideas at the runtime level.

While most of the operating system level solutions, and the transparent OpenMP solution base their strategy on managing the data and do not consider moving the computation to where the data is, the proposal of this chapter for SMPSs is based in the later. This approach has potential in SMPSs because the number of ready tasks at any given time may be orders of magnitude higher than the number of cores. This aspect notably reduces the potential to increase unbalance caused by moving the computation to the data.

NUMA extensions to the OpenMP language have been proposed by [Bircsak et al., 2000]. Their proposal includes directives to force page migration, allows to declare data distributions similarly to High Performance Fortran by [Rice University, 1993], and includes directives to specify computation placement. In this sense, these directives could be considered a mechanism to let the programmer move the computation to where the data is. While the SMPSs scheduling that this chapter proposes is also based on moving the computation to the data, it is not controlled by the programmer and does not rely on user annotations. Another difference is that the proposal for OpenMP uses compiler transformations to allow distributions at granularities smaller than the page.

While distributed systems are different to NUMA systems, some of the ideas and programming languages of distributed systems apply to NUMA systems and achieve better performance than classical NUMA-unaware shared memory programming. The Message Passing Interface (MPI) by the [Message Passing Interface Forum, 2009] is a library standard for distributed programming that is also used in NUMA systems to exploit their performance characteristics. The programming languages that follow the Partitioned Global Address Space (PGAS) paradigm are also tailored for distributed systems. However, their design includes features to handle the distribution of data and computation, that also apply to NUMA systems.

5.3 Data Distribution in Parallel Programming Languages

In NUMA systems the location of the data and the computation that accesses it have an influence on the performance of the applications. In this text we will refer to the mapping between the language-level data declarations and the memories as the data *distribution*. This concept is analogous to the identically named one used in distributed programming models. However, since we are targeting shared memory multiprocessors, all data can be accessed independently of its distribution, whereas in distributed models only local data can be accessed directly.

5.3.1 Granularity, Architectural Limitations and Remapping

The memory of current computers is organized in pages. The page size determines the boundaries at which data can be handled at the physical and virtual levels. For instance, a virtual page may only be valid or invalid as a whole, and a physical page can only be located in one NUMA node. These constraints affect how the data can be distributed at least at the operating system level.

Most PGAS programming languages abstract the indexing of data structures to cope with those limitations. For instance, an array that may be distributed in a block cyclic pattern, may have the indexes that correspond to one node mapped to a contiguous block of memory. To access the data, the compiler adds code to convert the user-side indexing into the indexing required by the actual layout. This can be a source of overhead and complicates the reuse of already compiled codes and libraries, which should be either adapted or recompiled with those transformations.

While data layout changes, if abstracted, can be advantageous for programmability, they can also be advantageous for performance. For instance, the official LINPACK code by [Dongarra, 1988] can distribute the data at sub-page granularity. In this case, this aspect is not transparent and makes the code dependent on the data distribution. However, the implementation takes advantage of this knowledge to improve its performance. The data is laid out in such a way that at each outer iteration, each class of linear algebra operations can be executed in a single call to the BLAS/LAPACK library; whereas if the data was not coalesced, it would require several calls. In addition, the contiguous layout improves spatial locality. These aspects have been described by [Dongarra and Walker, 1993, sec. 7.1.2].

For simplicity and to allow the reuse of external libraries, this study does not include the re-indexing of language-level data structures.

5.3.2 Declaring Distributed Data

Programming language support for data distribution can be decomposed into the means they provide to declare distributed data, and the means to access it.

The distribution of data can be either explicit at the point of its declaration, or it can be implicit. The Chapel language by [Chamberlain et al., 2007] provides a global-view of the memory and uses explicit data distributions. In contrast, Sequoia++ by [Houston et al., 2008], also follows the global-view paradigm, but handles the location of the data dynamically, and thus leaves it out of the control of the programmer.

The Titanium language by [Yelick et al., 1998] and the X10 language by [Charles et al., 2005] follow the Single Program Multiple Data (SPMD) execution model and provide a local or fragmented-view of the data, and thus only allow to declare local data. Therefore, their data distribution is implicit like in MPI.

The Unified Parallel C (UPC) language by the [UPC Consortium, 2005] and Co-Array Fortran (CaF) by [Numrich and Reid, 1998] are also SPMD languages. However, they provide a global-view of the data, and therefore require the specification of the data distribution within their declarations. The implications and differences between local-view and fragmented-view programming paradigms have been further analyzed by [Chamberlain et al., 2007, sec. 2.2.1].

Explicit distribution declarations consist of an assignment of a collection of subsets of the data to the nodes. In this text we will refer to those as *units of distribution*. When they have a shape that conforms to a shape predefined by the language, the mapping is usually part of the variable declaration. Most languages have predefined forms to declare data distributed in *cyclic* and *block-cyclic* forms.

However, some languages allow to define other data distributions by making them and the elements that compose them first-class objects. The units of distribution can be defined by implementing a standard interface, and then they can be used to define the new distributions. This is the case of the Chapel language.

5.3.3 Accessing Distributed Data

The syntax and the conditions necessary to access distributed data are two key factors of the design of data distribution support in parallel programming models.

Chapel and UPC allow to access distributed data transparently and do not impose any restrictions. Titanium also allows transparent access, but requires to get the reference through the communication mechanism first.

X10 allows to express accesses to remote data with the same syntax as local data, but requires the computation to be executed at the location where the data resides. CaF allows to access distributed data without restrictions, however its syntax explicitly differentiates local accesses from possibly remote accesses.

In Sequoia++, data is transparently copied from and to the tasks automatically. However, tasks can only access the data that they specify in their parameters. This condition, and execution model is similar to the CellSs model by [Bellens et al., 2006; Perez et al., 2007], which is based on the model of this thesis.

5.4 A Data Distribution Proposal for SMPSs

The SMPSs programming model provides a simple programming environment with few language constructs but with a powerful runtime. Our proposal for data distribution in SMPSs relies on *on-demand page placement* and *initialization tasks*. This allows us to preserve the simplicity of the model and to move the complexity into the runtime.

5.4.1 Delayed Memory Allocation

Modern operating systems decouple memory allocation requests from the actual allocation. When the user code allocates memory, the operating system marks the corresponding part of the logical memory map of the process as reserved and uninitialized, but does not assign it physical pages. Instead, the first attempt to access each page produces a page fault that the operating system serves by assigning it a new physical memory page. This way, the operating system assigns the memory pages on demand. We refer to this mechanism as allocation on *first-touch*.

While the page faults incur in some overhead, this mechanism has also benefits. First, it allows processes to reserve more memory than they actually need at low cost. This mechanism and its implications is explained by [Rodrigues, 2009]. And second, delaying the actual page allocation enables dynamic page placement policies that depend on the locations of the first access.

5.4.2 Page Placement

To allocate a memory page, the operating system uses a page placement policy to decide in which NUMA node to allocate it. A common default is to allocate the pages locally on first-touch. Another is to interleave the placement between the nodes in round-robin. This policy is simple and balances the use of memory bandwidth. However, local placement on first-touch can perform better under some scenarios. These aspects are reviewed by [Marchetti et al., 1995].

Codes in which the thread or process that initializes each part of the data is the one whose performance is most dependent on its accesses can benefit from the policy if it is combined with pinning threads or processes to the cores. For instance, MPI applications usually benefit, since all their accesses are local. OpenMP applications can also benefit by using static scheduling, and using initializations that are scheduled to the same threads that will later access it.

5.4.3 Declaring a Data Distribution

The definition of a data distribution consists of the units of distribution, and the mapping between them and the memories. In this proposal, instead of declaring both, we only allow the user specify the former, and let the runtime determine the latter.

The units of distribution are defined by initializing the data in parallel with tasks. An *initialization task* is a task that has in its outputs data that it initializes for the first time. That data is the unit of distribution and will be placed together in one node. Instead of forcing a mapping between the initialized data and the NUMA node, the mapping is left to the scheduling policy. This aspect is a major difference between our solution and the previous work.

To illustrate this, listing 5.1 shows three tasks that initialize differently shaped regions of a bidimensional array and a function that initializes three arrays using those. All three arrays are of N by N elements. The first is initialized by horizontal panels of BS rows. The second is initialized by vertical panels of BS columns. And the third is initialized by blocks of BS by BS elements. Notice that this code is a standard SMPSs initialization and that it defines three data distributions by just using the regular elements of the language.

In the rest of this text we will differentiate between *initialization tasks* and *regular tasks*. The first corresponds to task instances that initialize memory addresses for the first time, and the second corresponds to the rest of the task instances.

5.4.4 Distributing the Data

While initialization tasks define the units of distribution, their scheduling determines the placement of the units of data distribution, and thus the mapping to the nodes. Since data placement can have a deep impact on the performance of regular tasks, the runtime automatically detects whether a task is an initialization task and applies different scheduling strategies to each case. The detection is described in section 5.4.7. The following paragraphs discuss the scheduling of initialization tasks, and therefore data distribution.

```
#pragma css task input(BS, N) output(data)
void horizontal_init_task(int N, int BS, double data[BS][N]);

#pragma css task input(BS, N) output(data{}{0:BS})
void vertical_init_task(int N, int BS, double data[N][N]);

#pragma css task input(BS, N) output(data{0:BS}{0:BS})
void block_init_task(int N, int BS, double data[N][N]);

void init(int N, int BS, double horiz_data[N][N], double vert_data[N][N],
  double block_data[N][N]) {
   for (int i=0; i < N; i += BS)
      horizontal_init_task(N, BS, &horiz_data[i][0]);

   for (int j=0; j < N; j += BS)
      vertical_init_task(N, BS, &vert_data[0][j]);

   for (int i=0; i < N; i += BS)
      for (int j=0; j < N; j += BS)
         block_init_task(N, BS, &block_data[i][j]);
}
```

Listing 5.1: Initialization of three bidimensional arrays using tasks by panels of rows, by panels of columns and by blocks.

Both scheduling algorithms consist of two parts: a part that inserts the task in a queue, and a part that retrieves the task from a queue. The first determines what to do with tasks as they become ready. This part selects the preferred NUMA node on which to run the task, and the queue of that node that will contain it. The second part determines which task to run when a thread becomes idle.

The placement policy has been designed to balance the distribution homogeneously between the NUMA nodes. To this end, the scheduler maintains a count of the bytes allocated in each memory and assigns initialization tasks greedily to the memory with the least bytes allocated, and on tie, also the best affinity according to the regular task scheduling policy.

Initialization tasks are inserted in a dedicated queue of the NUMA node. After their insertion, the scheduler updates the number of bytes allocated in that node, and updates a data structure that holds the information about the data that has been initialized and its placement. This queue is only consumed by the threads of the node, and thus initialization tasks are always run by a core that is local to its memory. The runtime relies on the first-touch local page placement policy to enforce the placement of data on the node that runs the initialization task.

Due to the dynamic nature of the programming model, tasks that initialize more than one array can be in flight at any given time. For instance the initialization tasks of the three arrays in listing 5.1 can be scheduled together. While the policy distributes the whole set of data homogeneously, it does not guarantee the same effect on each individual array. To avoid this problem, each block of memory allocated as a single block of memory could have its distribution balanced independently. That is, to distribute all the data allocated by each single call to malloc independently of each other. In our current implementation we do not do so. Instead we place barriers in the application code between the initialization of each array to accomplish the same result.

While the data placement policy is restricted to homogeneous distributions, others could be added by extending the syntax of the language. These aspects are out of the scope of this chapter.

5.4.5 NUMA-aware Scheduling

Tasks that do not initialize any set of memory for the first time are considered regular tasks. These are scheduled using a different algorithm since they do not contribute to the distribution of data, but can take advantage of it.

The original scheduler discussed in chapter 3 has per thread queues. These consist of one queue for high priority tasks, and one for normal priority tasks. Each thread inserts the tasks that it releases in one of its queues, and consumes tasks from its queues as long as they are not empty. The purpose of this design is to exploit the temporal locality inherent to the data dependencies by executing successors as soon as they are freed.

The new scheduler is focused on exploiting NUMA affinity. In that sense, it substitutes per thread queues by per NUMA node queues. The queues of a node are *local* for its threads and *remote* for the rest. Their purpose is to hold tasks that are most affine to that node. Where the old scheduler accessed the queues of the thread, the NUMA-aware scheduler accesses the queues of its NUMA node.

Like the regular SMP scheduler, the NUMA-aware scheduler has also different queues for high-priority tasks and normal-priority tasks, in addition to the queues of the data initialization tasks.

When a task becomes free, the runtime calculates its affinity. This procedure also determines if the task is an initialization task or a regular task. If it is an initialization task, it schedules it using the data distribution policy. Otherwise it sends the task to a queue of the node that contains most of its data. If the task only accesses data for which it does not have information, it sends it to a global queue.

The NUMA scheduler has a task protection mechanism similar to the one of the original scheduler. When freeing up successors, if one is most affine to the node of the thread, the thread will reserve it for itself. Identically to the original scheduler, the objective is to exploit the potential data cache reuse inherent to a data dependency and to prevent excessive task stealing. To further prevent excessive task stealing, the scheduler also keeps track of threads that are idle. During task stealing, threads skip nodes with idle threads.

Threads attempt to retrieve ready tasks by checking several sources in LIFO order until they succeed. Threads check first the initialization task queue of their node. Then they attempt to get a high priority task, and then a normal priority task. High priority tasks and normal priority tasks are looked up in the same order. First the thread checks the protected task, then the queue of their node, then the global queue, and finally it attempts to steal from the queues of the other NUMA nodes.

While the purpose of this scheduling policy is to maximize memory affinity, by doing so, it also favors homogeneous memory bandwidth usage. The effectiveness of both aspects depends on the scheduler, the data dependency structure of the algorithm, and the distribution of the data.

5.4.6 Calculating Memory Affinity

In this text we call *memory affinity* the degree to which the data used in a computation is local to the place where the computation is performed. That is, if we execute a task on a core and all the data that it accesses is local, we say that the execution of the task on that core has an affinity of 100%. On the other hand, if all the data is located on remote nodes, we say that it has 0% affinity.

Modeling affinity and its effects is complex since performance ultimately depends of the real accesses to the memory after discounting the effects of data caches. For simplicity this study does not take into account the effects of the cache, and does not consider the actual memory performance metrics of the code nor the architecture. Instead it considers that all data is accessed a uniform number of times and assumes that local accesses are always faster than remote accesses.

5.4.7 Handling NUMA Information

To calculate the affinity of a task and to determine the new memory that it initializes, the runtime must have the means to retrieve the placement of the memory pages that the task accesses.

While in Linux the get_mempolicy system call allows to query that information, its interface only allows to retrieve the NUMA node of a single page at a time. The Linux move_pages system call also allows to query that information but this time for multiple pages. However, its interface requires to generate a list that contains the starting address of every page that the task accesses.

Fortunately the location of each page is determined as its corresponding initialization task is assigned to a node. Thus, to avoid the overhead incurred by the system calls, the runtime keeps a record as the information is generated. To store and retrieve it efficiently, it has to map regions to NUMA nodes. The *region tree* presented in chapter 4 is a data structure that maps regions to information needed to calculate data dependencies. To map regions to NUMA nodes, we use a *data distribution tree* that uses the same indexing but that stores NUMA node identifiers in its leaf nodes.

To determine whether a task is an initialization task and to calculate its affinity, the runtime analyzes each region access. Since data allocation happens at page size granularity, it rounds regions to the page granularity by setting the least significant p digits to X, where p is the logarithm in base 2 of the page size.

This scheme works even with initializations that have smaller granularity than the page size. When a task initializes a page for the first time, even if partially, the runtime records it as placed. Any following initialization that touches the page will not be detected as the first initialization since the memory will already have been placed. Instead, the regular scheduling algorithm will favor its execution in the same node.

Our implementation calculates memory affinity as the number of bytes that tasks access in each memory over the total number of bytes that they access, and the new memory as the number of bytes that they initialize for the first time.

5.5 Evaluation

This section explores how NUMA affinity affects the performance of a set of benchmarks, and how the scheduler that this chapter proposes improves it. For consistency, the benchmarks, the hardware, the software and the measurement methodology are the same as in the previous chapter. In all cases the variants are the region-based ones unless otherwise specified.

To evaluate data distribution strategies, the initialization code of each benchmark has been changed to allow several unit of distribution shapes, and the code has been run under the NUMA-aware scheduler that this thesis proposes.

For brevity this section only presents the evaluation of the benchmarks whose performance was sensitive to NUMA affinity. The performance of the benchmarks that include matrix multiplications is dominated by the computational complexity of those operations, and thus they do not benefit from NUMA-affine scheduling in big problem sizes. For brevity, these have been moved appendix F. The affected benchmarks are the matrix multiplication, Cholesky, Strassen-Winograd and HPL.

5.5.1 Additional Schedulers

To evaluate the benefits of the NUMA-aware scheduler, the evaluation includes measurements made with the NUMA-unaware scheduler with memory pages interleaved across the NUMA nodes used by the threads in each experiment. This setup has already been described in page 27. In the rest of this text we will refer to it as the NUMA-unaware scheduler. This scheduler represents a compromise solution that ignores NUMA and tries to avoid too much penalization without programming effort.

However, memory interleaving is not the worst case scenario. In addition, we have made measurements with two more schedulers. One penalizes NUMA affinity, and the other the homogeneous distribution of memory bandwidth. These "bad" schedulers represent policies in the opposite direction to the NUMA-aware scheduler and together with the NUMA-unaware executions allow us to evaluate the sensibility of a given code to those parameters.

The first "bad" scheduler is the (computation) *Misplacing* scheduler, that favors the execution of tasks in the NUMA nodes with the worst memory affinity. And the second one is the (memory bandwidth) *Unbalancing* scheduler, that favors the execution of tasks that have most of their data located on the NUMA nodes that have the most data in use at the given moment.

The difference between the misplacing scheduler and the NUMA-aware scheduler is the order in which they extract tasks from the queues. When the NUMA scheduler gathers local tasks, the misplacing scheduler steals tasks, and the other way around. In addition, to further reduce NUMA affinity, the misplacing scheduler steals tasks in two phases. First it scans the remote node queues for tasks for which the node is the least affine. If it fails to get one, then it attempts to steal one for which it is not the most affine.

The unbalancing scheduler attempts maximize the memory bandwidth usage of the most used node and to minimize the rest. To increase the number of memory accesses to the NUMA node with the most loaded memory, it keeps an estimation of the bandwidth usage of each memory. This value is constructed as the sum of the affinities of the running tasks. That is, if the system is only running one task and all its data is located on a single node, then that node will have a load of 1, and the rest will have 0 load. If there is only one task that has half of its data in one node, and half in another, these two will have 0.5 load, and the rest will have 0 load.

To determine which task to execute, the thread orders the NUMA nodes according to their memory load in decreasing order and attempts to get tasks from the queues in that order.

```
#pragma css task input(size) output(a, b, c)
void init_segment(long size, double a[size], double b[size], double c[size]);

void init(long N, long BS, double a[N], double b[N], double c[N]) {
    for (long i = 0; i < N; i+= BS)
        init_segment(BS, &a[i], &b[i], &c[i]);
}
```

Listing 5.2: Initialization task declaration and function of the NUMA-aware Triad implementation.

5.5.2 Triad

The triad benchmark is the simplest code that we evaluate. The code performs simple arithmetic operations over the elements of three unidimensional arrays. Its implementation and performance are presented in the extended benchmarks of the previous chapter that appear in the appendix E in page 225.

Units of Distribution

Since its data is unidimensional, this benchmark has very few possibilities to explore data layouts. For simplicity, we have chosen a layout that matches the data blocks that the tasks use. That is, if the arrays have N elements and we operate in subarrays of BS elements, then we initialize them in subarrays of BS elements too.

To achieve the maximum affinity the three arrays must be distributed identically. Therefore, the main arrays cannot be initialized independently. Instead we initialize each subarray triplet together in a task. Listing 5.2 shows the task declaration and the initialization loop.

Effectiveness of the NUMA Scheduling Policy

To validate the effectiveness of the runtime placement we have measured the performance of the code with the NUMA-aware scheduler and the other three reference schedulers. Figure 5.1 shows the strong scalability of each case with several problem sizes, and table 5.1 shows the performance values numerically. These results show that exploiting NUMA is crucial to achieve good performance in this benchmark. The penalty for not doing so is so high that the performance of the executions using an interleaving placement policy is almost identical to the executions that only use one memory at a time (the unbalancing scheduler), or the executions that run the tasks on a non affine node (the misplacing scheduler).

Performance Compared to Other Implementations

To compare the performance of the NUMA-aware implementation to other programming models, we have made an implementation using MPI. In this case, the timed section of the code begins after a global barrier, includes the local triad and a second barrier.

Figure 5.2 shows the strong scalability with three problem sizes with the NUMA-unaware scheduler, the NUMA-aware scheduler, and the MPI implementation. The NUMA-aware SMPSs version and the MPI version perform almost on par. Table

Cores	N[a]	BS[b]	GB/s[c]	GB/s[c]	GB/s[c]	GB/s[c]	Aff.[d] (%)	Aff.[d] (%)	Aff.[d] (%)
1	32	16	4.8	4.8	4.7	4.8	100	100	100
1	64	32	4.8	4.9	4.8	4.8	100	100	100
2	32	16	5.2	5.2	5.2	5.2	100	100	100
2	64	32	5.2	5.2	5.2	5.2	100	100	100
4	32	8	5.1	5.1	5.1	5.1	100	100	100
4	64	8	5.1	5.1	5.1	5.1	100	100	100
8	32	4	10.2	7.3	7.0	6.6	50	42	11
8	64	8	10.3	7.2	6.8	6.8	50	46	12
8	128	16	10.3	7.1	6.9	6.9	50	40	16
16	32	2	20.1	9.8	9.6	8.9	25	21	4
16	64	4	20.2	9.9	9.5	8.7	25	19	3
16	128	8	20.3	9.9	9.7	8.9	25	24	5
32	32	1	36.9	11.4	11.9	10.6	12	10	2
32	64	2	38.9	11.4	12.1	10.9	12	10	1
32	128	4	39.9	11.5	12.0	11.1	12	10	2

Memory Nodes: NUMA Interleaved Unbalancing Misplaced

[a] Megaelements per array.
[b] Submatrix side size.
[c] Gigabytes per second.
[d] Mean memory affinity.

Table 5.1: Performance summary of Triad with four schedulers with different NUMA policies.

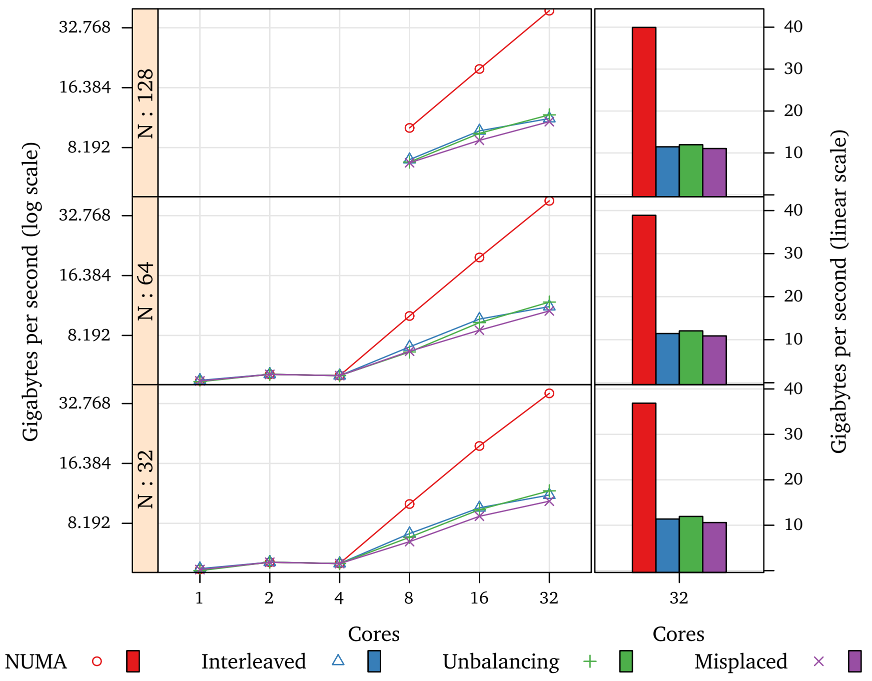


Figure 5.1: Strong scalability of the SMPSs triad benchmark under three different memory affinity scenarios.

5.2 summarizes the performance metrics of each implementation. Notice that the performance of the MPI version for any given number of cores does not vary with the problem size. In contrast, in SMPSs, the overhead gets proportionately smaller as we increase the number of cores, and as a consequence performance improves.

Cores	N[a]	GB/s[b]	GB/s[b]	GB/s[b]	IPC[c]	IPC[c]	Eff.[d] (%)	Eff.[d] (%)
1	32	4.8	4.8	4.9	1.5	1.5	99	99
1	64	4.9	4.8	4.9	1.5	1.5	99	99
2	32	5.2	5.2	5.2	0.8	0.8	99	99
2	64	5.2	5.2	5.2	0.8	0.8	99	99
4	32	5.1	5.1	5.2	0.4	0.4	99	99
4	64	5.1	5.1	5.2	0.4	0.4	99	99
8	32	7.4	10.2	10.3	0.3	0.4	98	99
8	64	7.7	10.3	10.3	0.3	0.4	98	99
8	128	7.5	10.3	10.3	0.3	0.4	99	99
16	32	9.8	20.1	20.6	0.2	0.4	94	96
16	64	9.9	20.2	20.6	0.2	0.4	94	97
16	128	9.9	20.3	20.6	0.2	0.4	95	98
32	32	11.4	36.9	41.0	0.1	0.4	94	82
32	64	11.5	38.9	41.2	0.1	0.4	96	91
32	128	11.5	39.9	41.2	0.1	0.4	97	95

Implementation: SMPSs Interleaved | SMPSs NUMA | MPI

[a] Megaelements per array.
[b] Gigabytes per second.
[c] Mean instructions per cycle while running tasks.
[d] Mean time that threads spend running tasks.

Table 5.2: Performance summary of the Triad implementations.

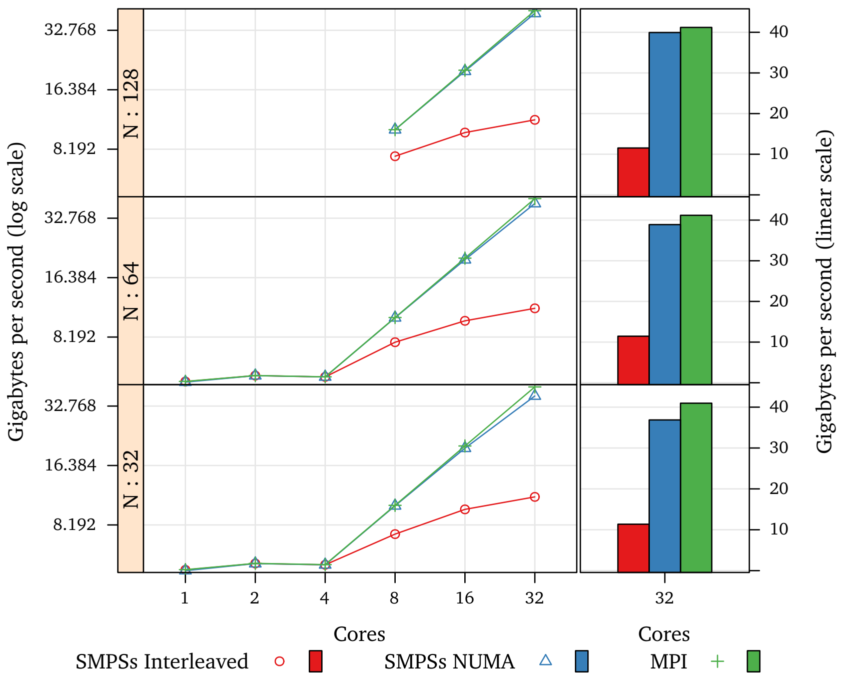


Figure 5.2: Strong scalability of the NUMA-unaware SMPSs triad, the NUMA-aware SMPSs triad, and the MPI implementation and performance with 32 cores.

5.5.3 Gauss-Seidel 2D Heat Transfer

The blocked version of the Gauss-Seidel 2D heat transfer benchmark has been presented and evaluated in the section that starts in page 36. Its regions-based version has been presented in the previous chapter in page 87 and it has been evaluated in an appendix in page 235. This chapter evaluates the NUMA aspects of the regions-based version.

Units of Distribution

This algorithm operates over a bidimensional array. The tasks update non overlapping square regions and read a halo around them of one element wide/high that overlaps the square regions that other tasks update. To establish the shape of the units of distribution we consider only the updated region as the basic shape.

We have tried three distributions. One that initializes the matrix in horizontal panels as wide as the matrix and as high as the updated region; one that does the equivalent with vertical panels; and one that initializes them in blocks with the same shape as the updated region. Throughout this chapter we will call them the *horizontal* distribution, the *vertical* distribution, and the *blocked* distribution.

In this benchmark the blocked distribution allows the NUMA scheduler to favor the affinity of the region that is updated since it contains most of the data of the task. While the halos can reside on remote nodes, they represent a small fraction of the total data, and thus should not affect task performance significantly. The horizontal distribution and the vertical distribution reduce the amount of potentially remote data in the halo in half, but are also more restrictive for the scheduler.

Effects of Data Placement on Performance

To validate the effectiveness of the runtime placement and its effect on performance, we have made measurements with the NUMA-unaware scheduler with memory interleaving and with the NUMA-aware scheduler with the three data distributions. Since the complexity of the algorithm is linear to the amount of data, we expect that memory affinity will have a measurable effect on performance.

Figure 5.3 shows the performance and strong scalability of each case with the best blocking size for each problem size, and table 5.3 shows the absolute numbers. Notice that the NUMA-aware executions perform in general better than the NUMA-unaware executions. And despite the fact that the blocked distribution has the potential to allow more memory affine schedules than the rest, the overall best performer is the horizontal distribution.

The reason for the lower than expected performance of the blocked distribution and the overall low performance of the vertical distribution is the architecture page size. In the experimental machine, the virtual page size is set to 16384 bytes. Since the problem operates over double precision floating point numbers, the minimum effective unit of distribution is at least 2048 elements wide. Initializations with narrower widths end up allocated in contiguous clusters that are as wide as a page. While the runtime takes this into consideration, this problem cannot be avoided without array reindexing.

The blocked distribution suffers from this effect horizontally, but not vertically. However, the consequences for the vertical distribution are critical since it is constrained to just one dimension. When the problem size is narrower than one page per memory, the data cannot be successfully distributed between all of them, and thus the execution suffers from lower memory bandwidth and lower potential affinity.

Figure 5.4 shows the mean task memory affinity in each case. Notice that the horizontal distribution is the one that achieves the best affinity in most configurations and that it is closely followed by the blocked distribution. The vertical distribution is the worst one that is NUMA aware, and the NUMA-unaware execution has homogeneous affinity for any given number of cores since the memory pages are interleaved.

As a result of exploiting the memory affinity, the horizontal execution achieves up to 70% more mean task memory bandwidth as shown in figure 5.5, and up to 90% less mean task memory latency as shown in figure 5.6. These effects when combined allow in some cases to achieve up to 70% more mean task floating point operations per cycle, as shown in figure 5.7.

Effectiveness of the NUMA Scheduling Policy

To further study the effects of NUMA on this code we have made additional executions with the “bad” schedulers and the data distributed in horizontal panels. Figure 5.8 shows the strong scalability of each case. The blocking size used in the measurements of the NUMA scheduler and the NUMA-unaware scheduler are the ones that generated the fastest results in each case. The blocking size of the unbalancing and misplacing executions are the same as the ones of the NUMA-unaware scheduler.

As already shown in the earlier scalability figure, this problem is sensible to memory affinity. The new plot shows that interleaving the memory pages to avoid taking into account NUMA affinity is only slightly better than attempting to get the worst performance out of it, and is far from the performance of a NUMA-aware execution.

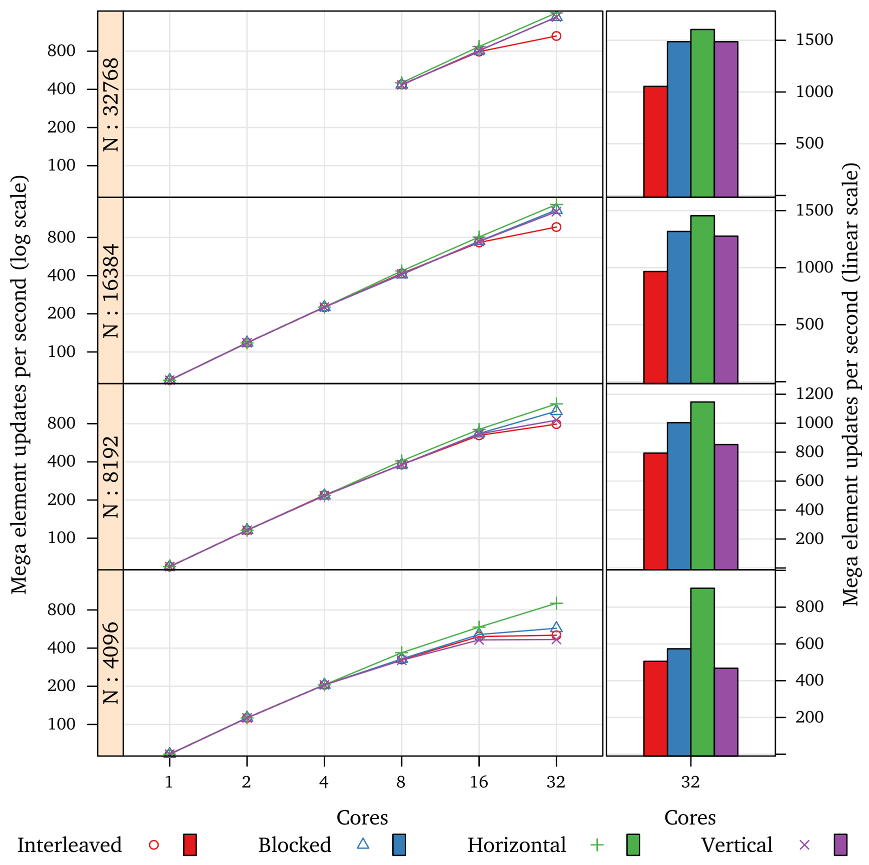


Figure 5.3: Strong scalability and performance with 32 cores of the NUMA-unaware Gauss-Seidel algorithm, the executions with a blocked data distribution, the executions with horizontal distribution, and the executions with vertical distribution.

Cores	N[a]	MU/s[b]	MU/s[b]	MU/s[b]	MU/s[b]	Aff.[c]	Aff.[c]	Aff.[c]	Aff.[c]
						(%)	(%)	(%)	(%)
1	2048	55	55	55	55	100	100	100	100
1	4096	58	58	58	58	100	100	100	100
1	8192	59	59	59	59	100	100	100	100
1	16384	59	59	59	59	100	100	100	100
2	2048	106	106	106	106	100	100	100	100
2	4096	112	112	112	112	100	100	100	100
2	8192	115	115	115	115	100	100	100	100
2	16384	118	118	117	118	100	100	100	100
4	2048	185	185	185	185	100	100	100	100
4	4096	205	205	205	205	100	100	100	100
4	8192	218	216	216	216	100	100	100	100
4	16384	224	225	225	225	100	100	100	100
8	2048	235	226	305	221	50	88	99	69
8	4096	325	328	367	320	50	95	99	90
8	8192	380	379	406	380	50	97	98	94
8	16384	416	407	435	411	50	98	98	98
8	32768	437	434	450	433	50	98	98	98
16	2048	202	210	307	209	25	99	99	82
16	4096	493	512	585	464	25	79	88	59
16	8192	646	669	720	663	25	85	88	76
16	16384	729	747	806	743	25	96	97	89
16	32768	793	806	869	809	25	97	97	97
32	4096	505	573	902	467	12	95	99	54
32	8192	793	1003	1146	852	12	84	99	55
32	16384	966	1317	1455	1276	12	88	93	71
32	32768	1054	1486	1604	1485	12	91	93	92

Distribution: Interleaved Blocked Horizontal Vertical

[a] Matrix side size.
[b] Mega element updates per second.
[c] Mean memory affinity.

Table 5.3: Performance summary of Gauss-Seidel with several data distributions.

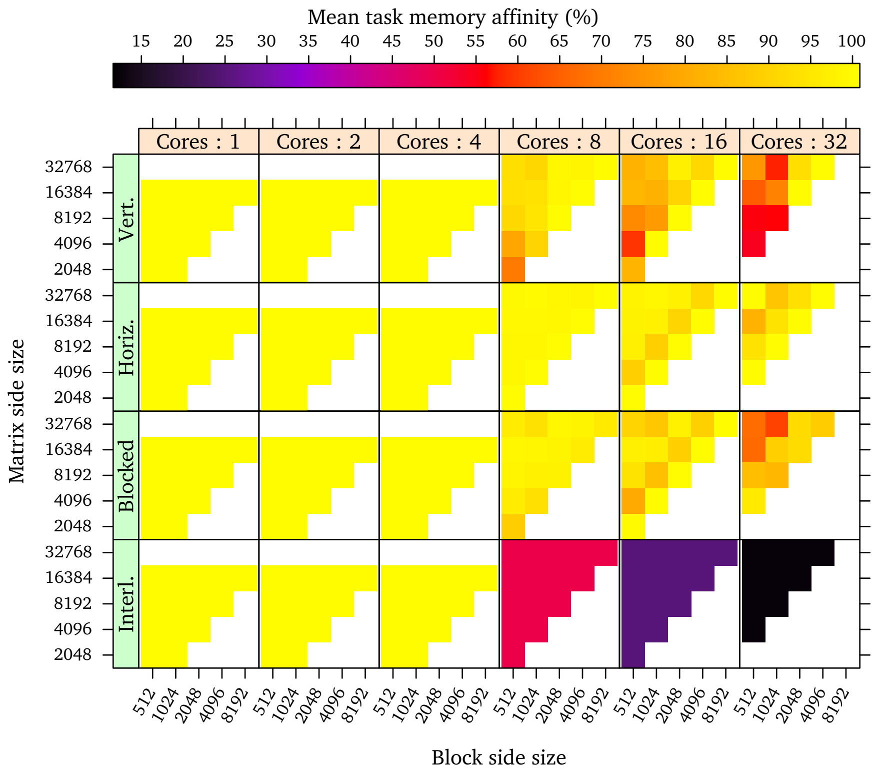


Figure 5.4: Mean memory affinity of the Gauss-Seidel tasks when running with the NUMA-unaware scheduler, and the three distributions.

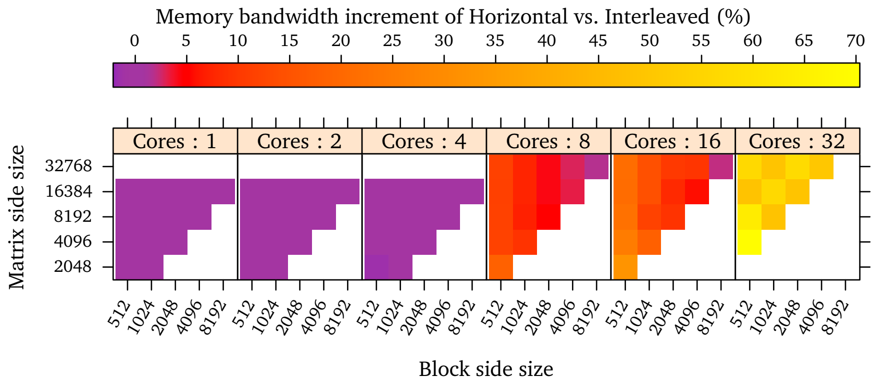


Figure 5.5: Memory bandwidth improvement of the Gauss-Seidel task when the data is distributed horizontally compared to the executions with the interleaved distribution.

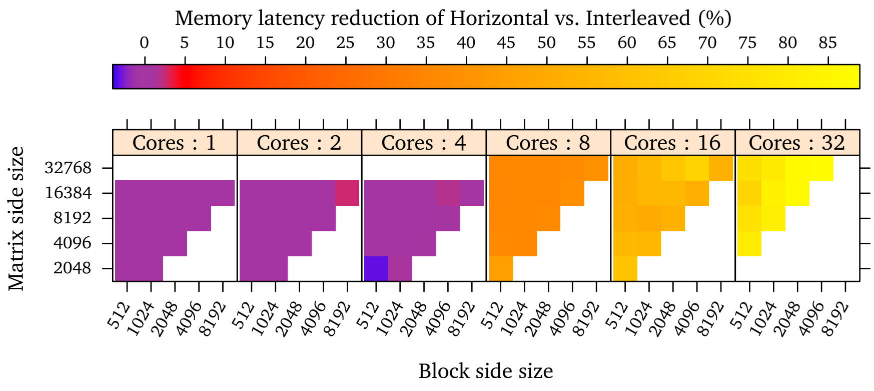


Figure 5.6: Mean memory latency of the Gauss-Seidel task when the data is distributed horizontally compared to the executions with the interleaved distribution.

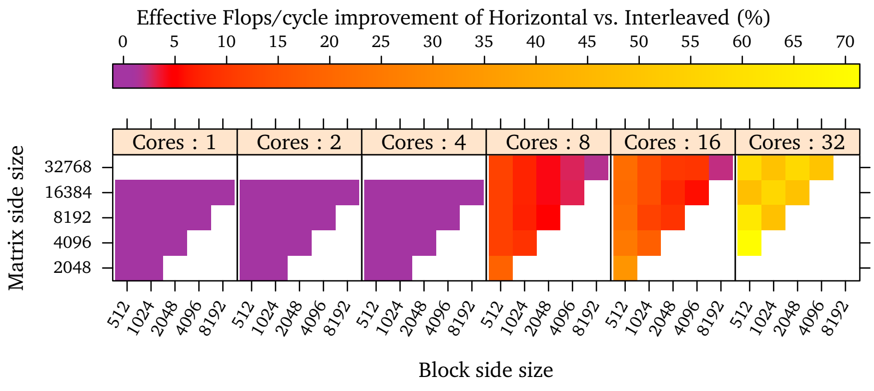


Figure 5.7: Mean floating point operations per cycle improvement of the Gauss-Seidel task when the data is distributed horizontally compared to the executions with the interleaved distribution.

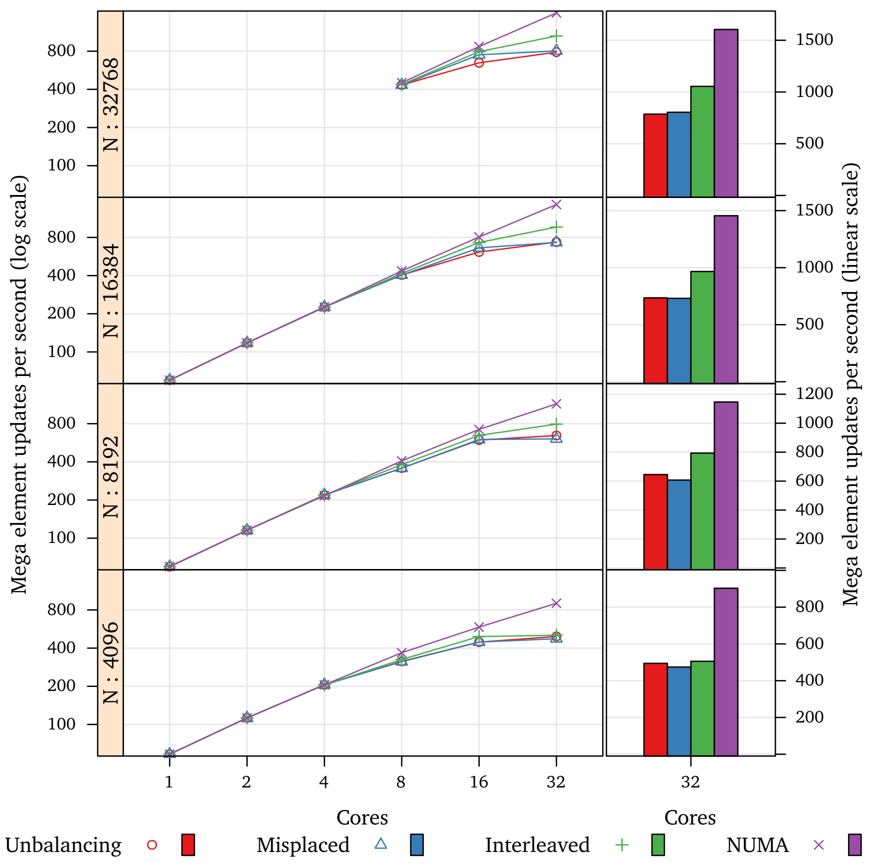


Figure 5.8: Strong scalability and performance with 32 cores of the Gauss-Seidel algorithm with memory overloading scheduler, the misplacing scheduler, the NUMA-unaware scheduler, and the NUMA scheduler.

Performance Compared to Other Implementations

To compare the performance of the NUMA-aware implementation to other programming models, we have made additional measurements with the blocked implementation from chapter 3 that starts in page 36, and the OpenMP version of the same section.

Figure 5.10 shows the strong scalability of each implementation with several problem sizes with the best performing block size in each case. The series labeled "Interleaved" corresponds to the NUMA-unaware executions. The series labeled "Blocked Layout" corresponds to the implementation from the blocks-based programming model with the memory interleaved, and the "OpenMP" series corresponds to an equivalent one to the regions version, but using OpenMP and barriers.

The OpenMP version is the worst performer, since it cannot generate enough parallelism, big enough tasks, and cannot benefit from NUMA affinity.

Notice that while NUMA affinity is important to achieve good performance, the blocked layout measures show that the data layout has also an important impact for the smallest problem sizes. Figure 5.9 shows the floating point performance difference between the NUMA-aware executions with the flat layout and the NUMA-unaware executions with the blocked layout. With 32 cores the effect of NUMA is bigger than that of the blocked layout. With less cores, the blocked layout improves performance over the flat layout more than NUMA-awareness for the smallest problem sizes.

Table 5.4 summarizes those findings numerically.

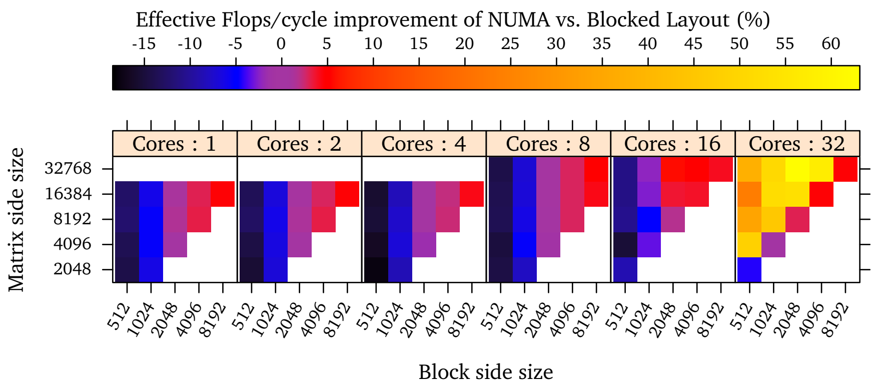


Figure 5.9: Mean Gauss-Seidel task performance difference between the NUMA-aware execution with the flat layout and horizontal distribution and the NUMA-unaware execution with blocked data layout.

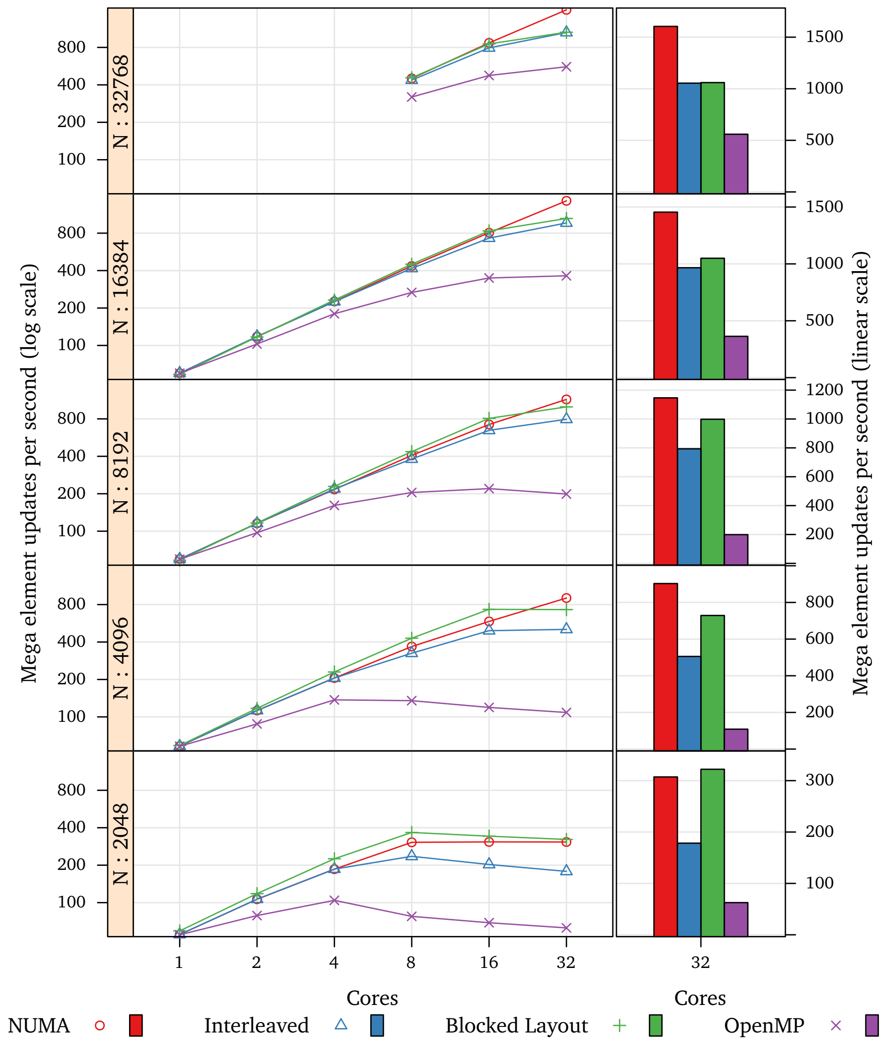


Figure 5.10: Strong scalability and performance with 32 cores of the Gauss-Seidel algorithm with the NUMA-aware scheduler, with the NUMA-unaware scheduler, the NUMA-unaware scheduler with blocked data layout, and the OpenMP implementation.

Cores	N[a]	MU/s[b]	MU/s[b]	MU/s[b]	MU/s[b]	FPC[c]	FPC[c]	FPC[c]	Eff.[d]	Eff.[d]	Eff.[d]
									(%)	(%)	(%)
1	2048	55	55	59	55	0.2	0.2	0.2	99	99	99
1	4096	58	58	58	58	0.2	0.2	0.2	99	99	98
1	8192	59	59	58	59	0.2	0.2	0.2	99	99	99
1	16384	59	59	58	59	0.2	0.2	0.2	99	99	99
2	2048	106	106	118	78	0.2	0.2	0.2	96	96	97
2	4096	112	112	117	87	0.2	0.2	0.2	96	96	98
2	8192	115	115	116	97	0.2	0.2	0.2	96	96	99
2	16384	117	118	117	102	0.2	0.2	0.2	99	99	99
4	2048	185	185	225	104	0.2	0.2	0.2	94	94	94
4	4096	205	205	229	137	0.2	0.2	0.2	94	94	97
4	8192	216	218	229	161	0.2	0.2	0.2	94	95	98
4	16384	225	224	232	179	0.2	0.2	0.2	98	98	98
8	2048	305	235	366	77	0.1	0.1	0.2	79	73	82
8	4096	367	325	429	135	0.2	0.1	0.2	85	93	94
8	8192	406	380	435	204	0.2	0.2	0.2	94	94	96
8	16384	435	416	450	266	0.2	0.2	0.2	95	95	97
8	32768	450	437	456	319	0.2	0.2	0.2	95	95	98
16	2048	307	202	342	68	0.2	0.1	0.2	39	34	40
16	4096	585	493	732	119	0.1	0.1	0.2	81	85	86
16	8192	720	646	807	219	0.2	0.1	0.2	86	86	94
16	16384	806	729	833	348	0.2	0.2	0.2	94	87	96
16	32768	869	793	853	475	0.2	0.2	0.2	95	95	97
32	2048	307	178	322	62	0.2	0.1	0.2	19	16	19
32	4096	902	505	728	108	0.2	0.1	0.1	58	55	69
32	8192	1146	793	998	198	0.2	0.1	0.1	66	87	89
32	16384	1455	966	1049	363	0.2	0.1	0.1	86	89	94
32	32768	1604	1054	1059	559	0.2	0.1	0.1	88	91	93

Implementation: NUMA Interleaved Blocked Layout OpenMP

[a] Matrix side size.
[b] Mega element updates per second.
[c] Mean floating point operations per cycle while running tasks.
[d] Mean time that threads spend running tasks.

Table 5.4: Performance summary of the Gauss-Seidel implementations.

5.5.4 Multisort

The multisort benchmark has been evaluated in the previous chapter in the section that starts in page 106. This chapter evaluates the effects of NUMA on its performance.

Units of Distribution

This algorithm has two types of tasks: the tasks that sort small subarrays using the quicksort algorithm, and the tasks that merge presorted pairs of subarrays of the data. The first type accesses the array in contiguous and non-overlapping segments that have the size of the sorting block size. The second, does not have such a well defined access pattern. Instead it performs a scan over a wide range of data and then consumes a portion of it, that is also variable. Therefore, the affinity that the runtime calculates of the merge task is not representative of the actual data accesses, and thus, the NUMA-aware scheduler may be ineffective for this case.

The data accesses of the original mergesort algorithm match the recursion structure of the algorithm. These can be represented as an inverted binary tree, where each node below the first level corresponds to a merge function that consumes all the data that its parents produce. For this reason we propose to distribute the array in equally sized segments that total the number of NUMA nodes. This way, there will be as many subtrees as NUMA nodes whose data is completely located in only one node, and thus their corresponding tasks will have a chance of having 100% affinity despite the inability to calculate affinity reliably.

Effects of Data Placement on Performance

To validate the effectiveness of the runtime placement and its effect on performance, we have made measurements with the NUMA-unaware scheduler with memory interleaving and with the NUMA-aware scheduler with the proposed data distribution. Since this is a sorting code, its performance is highly dependent on memory bandwidth, and thus it is expected that memory affinity will have an important impact on it.

Figure 5.11 shows the performance and strong scalability of each case with the best blocking size for each problem size, and table 5.5 shows the numerical data. The NUMA-aware scheduling scales better than the NUMA-unaware scheduling and is up to 37% faster. However, scalability degrades in both cases when going from 4 to 8 cores, which corresponds to the transition from 1 to 2 NUMA nodes. This suggests that the NUMA-aware scheduler is not benefiting from all the potential memory affinity.

Figure 5.12 shows the mean task memory affinity, which is much lower for the NUMA-aware executions that can be achieved. Each vertical panel corresponds to executions with the number of cores indicated on top, and each horizontal panel corresponds to executions with the problem size indicated on the left in megaelements. This data suggests that this benchmark may be suffering from excessive task stealing, which penalizes memory affinity.

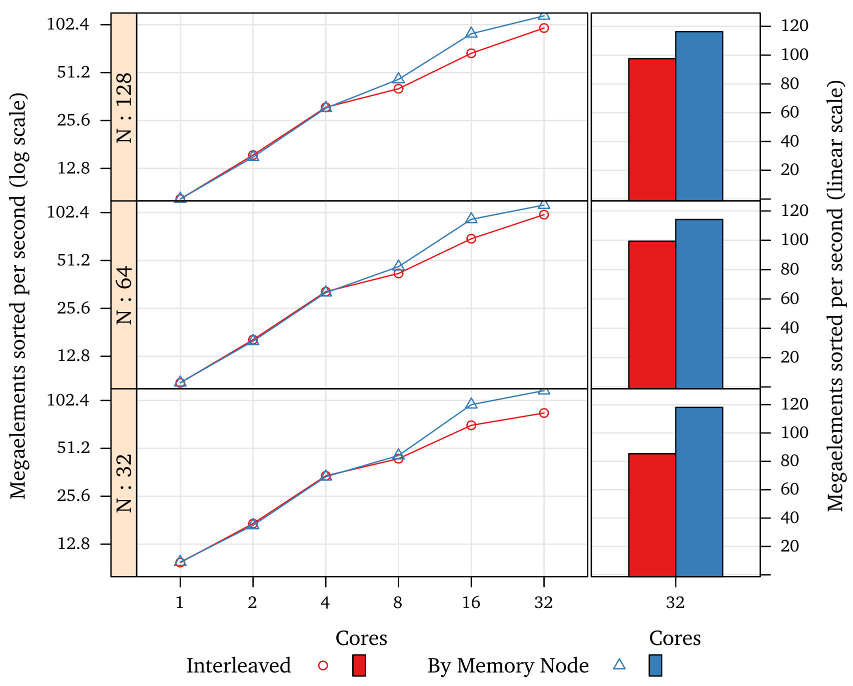


Figure 5.11: Strong scalability and performance with 32 cores of the Multisort algorithm, with the NUMA-unaware scheduler with memory interleaving, and with the NUMA-aware scheduler with the one block per node distribution.

Cores	N[a]	BS1[b]	BS2[c]	BS1[b]	BS2[c]	Mels/s[d]	Mels/s[d]	Eff.[e]	Eff.[e]	Aff.[f]	Aff.[f]
								(%)	(%)	(%)	(%)
1	32	16	16	16	4	9.8	9.9	99	99	100	100
1	64	8	16	16	16	8.7	8.7	99	99	100	100
1	128	16	16	16	16	8.2	8.2	99	99	100	100
2	32	8	8	8	8	17.2	16.8	93	93	100	100
2	64	16	1	8	16	16.3	15.9	93	93	100	100
2	128	16	1	16	16	15.5	15.1	93	93	100	100
4	32	8	1	4	1	34.4	34.0	98	99	100	100
4	64	16	1	8	1	32.6	32.1	99	99	100	100
4	128	16	2	16	4	31.0	30.6	99	99	100	100
8	32	1	1	2	1	44.1	46.2	96	89	50	60
8	64	2	1	4	1	42.4	46.8	97	93	50	61
8	128	4	1	4	1	40.5	46.4	97	96	50	61
16	32	1	1	2	2	71.5	96.3	94	86	25	60
16	64	4	1	2	4	70.1	92.9	96	86	25	60
16	128	8	1	8	1	67.7	89.9	97	95	25	44
32	32	1	1	1	1	85.4	118.1	67	57	12	37
32	64	1	1	2	2	99.4	114.2	94	57	12	37
32	128	1	1	1	1	97.6	116.3	97	90	12	23
32	256	2	2	2	1	95.1	120.8	97	93	12	24

Distribution: Interleaved | By Memory Node

[a] Megaelements.
[b] Megaelements per sort task.
[c] Megaelements per merge task.
[d] Megaelements sorted per second.
[e] Mean time that threads spend running tasks.
[f] Mean memory affinity.

Table 5.5: Performance summary of the Multisort implementations.

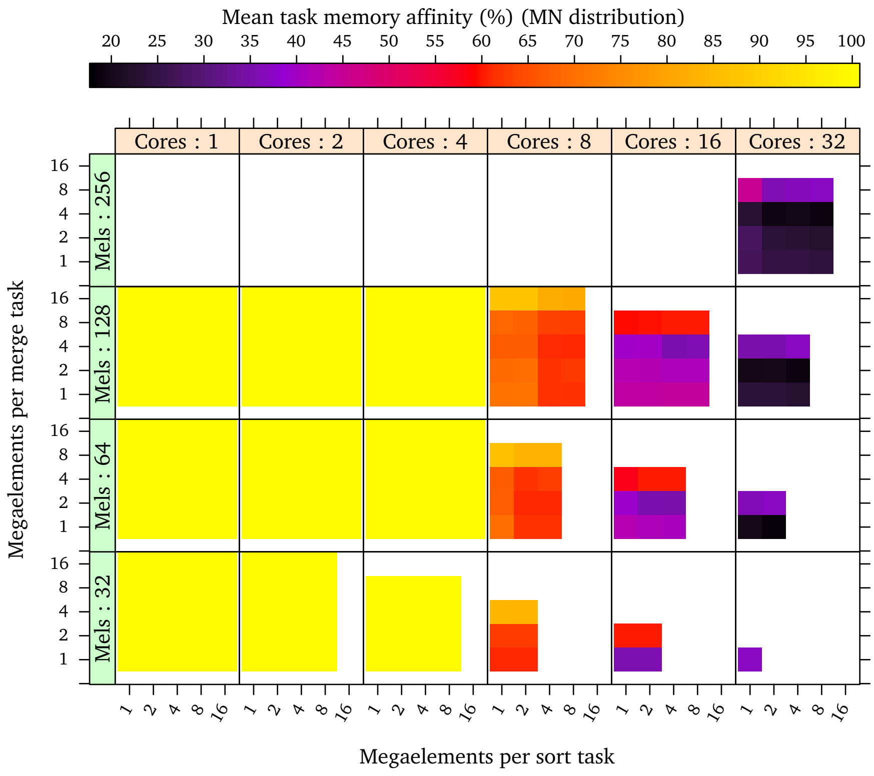


Figure 5.12: Mean task memory affinity when running Multisort with the NUMA-aware scheduler with several number of cores, problem sizes, and blocking sizes for the quick sort tasks and merge tasks.

Effectiveness of the NUMA Scheduling Policy

To further study the effects of NUMA on this code we have made additional executions with the "bad" schedulers and the data distributed by NUMA nodes. To further check if task stealing is reducing the ability to exploit affinity, we have made an additional set of executions with the NUMA-aware scheduler with task stealing disabled. Figure 5.13 shows the strong scalability of each case. The blocking size used in the measurements are the ones that generated the fastest results for the NUMA-aware scheduler.

As already shown in the earlier scalability plot, this problem is sensible to memory affinity. The new plot shows that interleaving the memory pages to avoid taking into account NUMA affinity is almost as bad as the worst case scenarios, and that task stealing in this case is hurting performance.

Performance Compared to Other Implementations

To compare the performance of the NUMA-aware implementation to other programming models, we have made additional measurements with the OpenMP version from chapter 4 that starts in page 112, and the OpenMP version of the same section.

Figure 5.14 shows the strong scalability of each implementation with several problem sizes with the best performing block size in each case. The series labeled

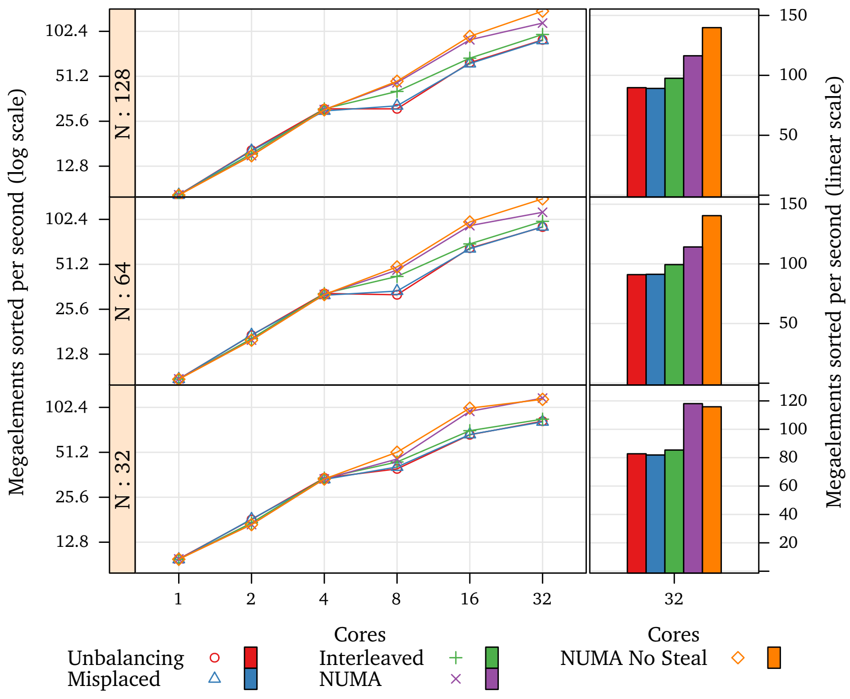


Figure 5.13: Strong scalability and performance with 32 cores of Multisort with the memory overloading scheduler, the misplacing scheduler, the NUMA-unaware scheduler, the NUMA scheduler, and the NUMA scheduler with task stealing disabled.

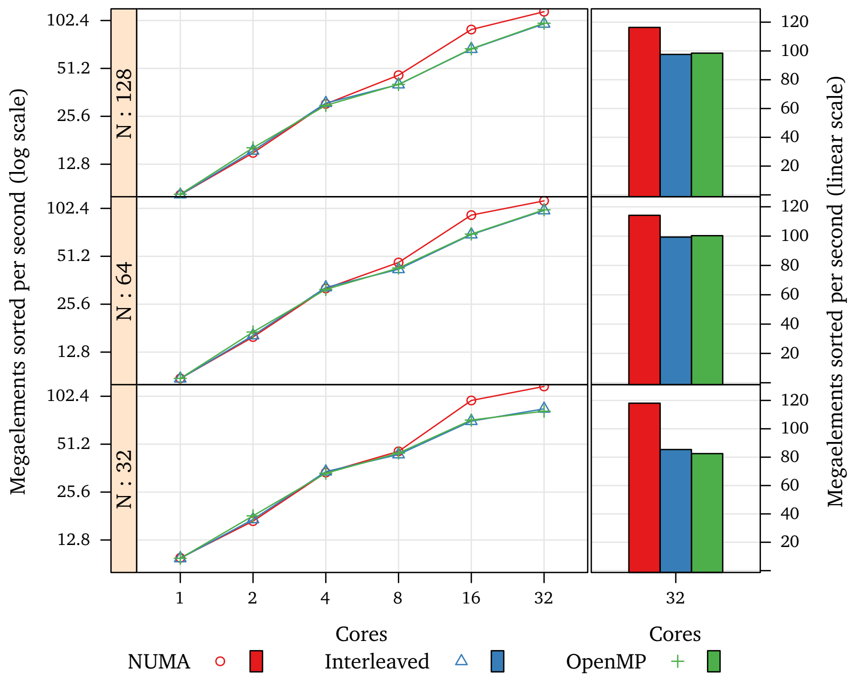


Figure 5.14: Strong scalability and performance with 32 cores of the multisort algorithm with the NUMA-aware scheduler, with the NUMA-unaware scheduler, the NUMA-unaware scheduler with blocked data layout, and the OpenMP implementation.

"Interleaved" corresponds to the NUMA-unaware executions, and the "OpenMP" series corresponds to a version using OpenMP and task nesting. The NUMA-unaware and the OpenMP versions perform almost identically despite the different programming model paradigms. The NUMA-aware version performs noticeably better.

Table 5.6 summarizes those findings numerically.

Cores	N[a]	Mels/s[b]	Mels/s[b]	Mels/s[b]	IPC[c]	IPC[c]	IPC[c]	Eff.[d] (%)	Eff.[d] (%)	Eff.[d] (%)
1	32	9.9	9.8	9.8	2.4	2.4	2.4	99	99	99
1	64	8.7	8.7	8.7	2.3	2.3	2.3	99	99	99
1	128	8.2	8.2	8.3	2.2	2.2	2.3	99	99	99
2	32	16	17	18	2.3	2.3	2.3	93	93	99
2	64	15	16	17	2.2	2.3	2.3	93	93	99
2	128	15	15	16	2.2	2.3	2.2	93	93	99
4	32	33	34	33	2.2	2.2	2.2	99	98	99
4	64	32	32	31	2.1	2.2	2.1	99	99	99
4	128	30	30	29	2.1	2.1	2.1	99	99	99
8	32	46	44	45	1.7	1.5	1.5	89	96	97
8	64	46	42	43	1.7	1.5	1.5	93	97	98
8	128	46	40	40	1.7	1.5	1.5	96	97	98
16	32	96	71	72	1.9	1.2	1.3	86	94	96
16	64	92	70	70	1.8	1.2	1.2	86	96	97
16	128	89	67	68	1.7	1.2	1.2	95	97	98
32	32	118	85	82	1.7	1.1	1.1	57	67	64
32	64	114	99	100	1.7	0.9	0.9	57	94	97
32	128	116	97	98	1.2	0.9	0.9	90	97	98
32	256	120	95	95	1.2	0.9	0.9	93	97	99

Implementation: NUMA Interleaved OpenMP

[a] Megaelements.
[b] Megaelements sorted per second.
[c] Mean instructions per cycle while running tasks.
[d] Mean time that threads spend running tasks.

Table 5.6: Performance summary of the multisort implementations.

5.5.5 Fast Fourier Transform

The FFT benchmark has been evaluated in the previous chapter starting in page 115. This chapter evaluates how NUMA affects its performance.

Units of Distribution

The implementation that this section evaluates is based on the 6-step matrix Fourier decomposition. This algorithm operates over the data as if it was a bidimensional array, and consists of several phases. The data accesses of those can be classified in two groups. One group corresponds to the phases that apply a base FFT algorithm over each row of the matrix. And the other, corresponds to the phases that perform the matrix transposition, and the matrix transposition combined with the twiddle factor multiplication.

For the first group, a distribution in horizontal panels would favor the memory affinity of the FFT tasks. The distribution can match the actual shape of the FFT tasks.

The tasks of the second group operate either over square blocks of the diagonal, or over pairs of square blocks that are located symmetrically with respect to the diagonal. All these are non-overlapping and have constant size. To favor their affinity, each pair should be placed together in the same memory. None of the initialization strategies of the previous benchmarks produce a distribution that meets those constrains. Therefore we define a new distribution that places each pair along the diagonal in the same NUMA node. We will refer to it as the *pairwise* distribution.

Listing 5.3 shows the task declarations and the initialization function to distribute the data in a pairwise manner. The initialization function starts at line 15 and uses two tasks. The task declared in line 1 initializes a pair of blocks, and the task declared in line 9 initializes one block (of the diagonal).

For completeness the evaluation includes measurements with the data distributed in vertical panels, which should produce an affinity close to the interleaved executions; and executions with the blocked distribution. Figure 5.15 shows the strong scalability that was achieved with each distribution and with the NUMA-unaware scheduler with memory interleaving, and table 5.7 shows the data numerically.

The worst performing distribution is the vertical distribution due to two factors. First, its distribution does not favor memory affinity for any of the tasks. For the panel FFT, it has similar affinity to the interleaved distributions, and for the rest, in most cases it will be affine to only one of the blocks.

The second factor is the page size. Since it is set to 16KB, to cover at least one page, the vertical panels must be at least 1024 elements wide. Therefore, to cover the 8 nodes, the problem must have at least 8192×8192 elements. Thus, all measurements with N below 8192 suffer from reduced memory bandwidth under the vertical distribution.

The blocked and the pairwise distributions perform similarly. While they are constrained by the memory page size, they are not as constrained as the vertical distribution. Their almost identical performance suggests that memory affinity does not affect significantly the performance of the transposition tasks. Perhaps because a big blocking size can hide part of the latency of the remote accesses.

Finally, the best scaling distribution is the horizontal distribution. This one favors the memory affinity of the FFT tasks, and is not constrained by the page size.

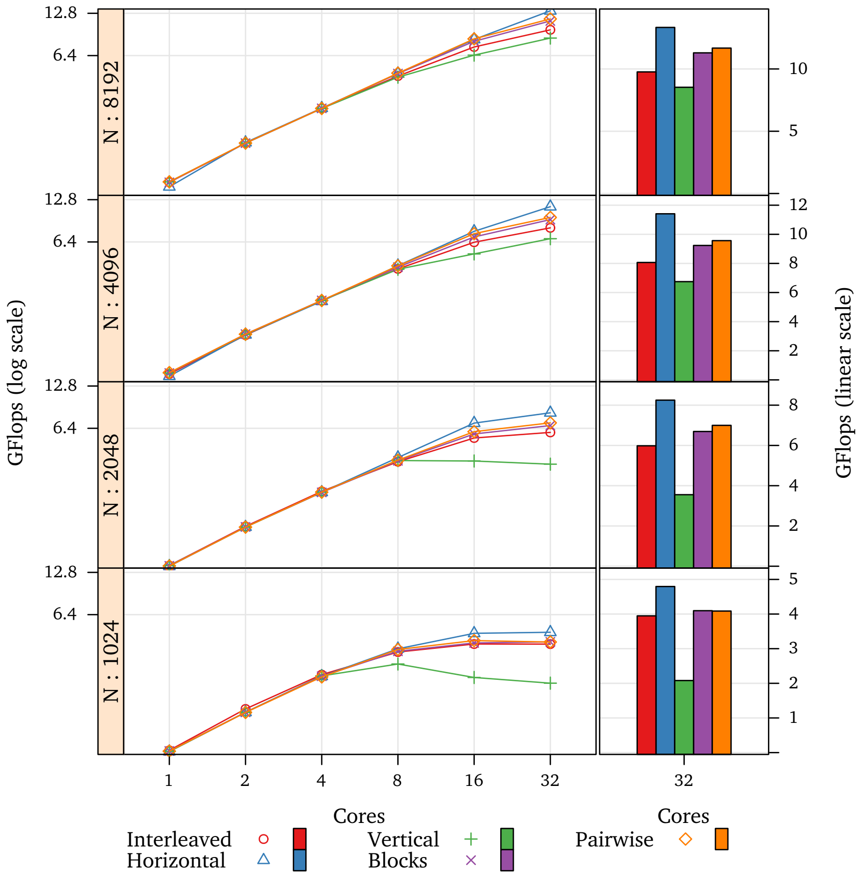


Figure 5.15: Strong scalability of the FFT algorithm with the NUMA-unaware scheduler with memory interleaving, and with the NUMa-aware scheduler with the horizontal distribution, the vertical distribution, the blocked distribution and the pairwise distribution.

```
#pragma css task input(N, BS) \
   output(block1{0:BS}{0:BS}, block2{0:BS}{0:BS})
void initBlockPair(
   long N, long BS,
   double _Complex block1[N][N],
   double _Complex block2[N][N]
);

#pragma css task input(N, BS) output(block{0:BS}{0:BS})
void initBlock(
   long N, long BS,
   double _Complex block[N][N]
);

void initByPairs(long N, long BS, double _Complex data[N][N]) {
   for (long i = 0; i < N; i+=BS) {
      for (long j = 0; j < i; j+=BS)
         initBlockPair(N, BS, &data[i][j], &data[j][i]);
      initBlock(N, BS, &data[i][i]);
   }
}
```

Listing 5.3: Initialization code to distribute a matrix symmetrically along the diagonal.

Figure 5.16 shows the performance improvement of the horizontal distribution compared to the NUMA-unaware executions. Notice that with 32 cores, there is at least one blocking size combination for each problem size that performs at least 25% better with the NUMA scheduler than with the NUMA-unaware scheduler. However, the minor differences between the schedules when running under only one memory make the some of the executions with 1 and 2 cores perform worse under the NUMA scheduler than under the NUMA-unaware scheduler.

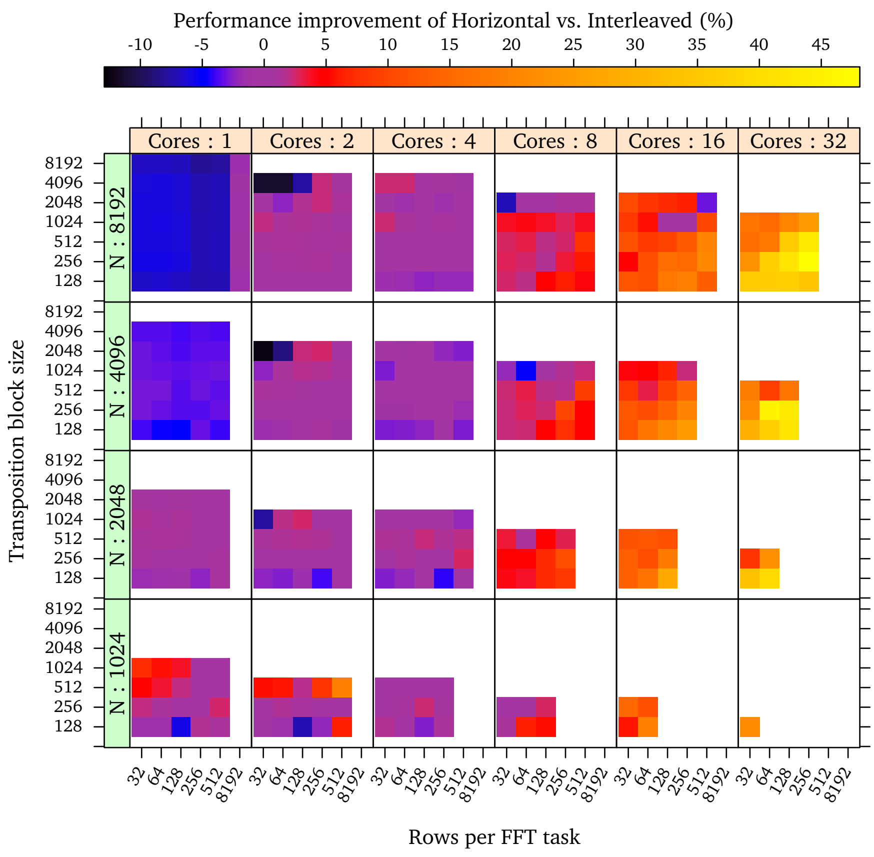


Figure 5.16: Performance improvement when going from the NUMA-unaware scheduling to the NUMA-aware scheduling with the horizontal distribution under the FFTW code.

Cores	N[a]	GF[b]	GF[b]	GF[b]	GF[b]	GF[b]	Aff.[c] (%)	Aff.[c] (%)	Aff.[c] (%)	Aff.[c] (%)	Aff.[c] (%)
1	1024	0.7	0.7	0.7	0.7	0.7	100	100	100	100	100
1	2048	0.7	0.7	0.7	0.7	0.7	100	100	100	100	100
1	4096	0.7	0.7	0.7	0.7	0.8	100	100	100	100	100
1	8192	0.8	0.7	0.8	0.8	0.8	100	100	100	100	100
2	1024	1.4	1.3	1.3	1.3	1.3	100	100	100	100	100
2	2048	1.3	1.3	1.3	1.3	1.3	100	100	100	100	100
2	4096	1.4	1.4	1.4	1.4	1.4	100	100	100	100	100
2	8192	1.5	1.5	1.5	1.5	1.5	100	100	100	100	100
4	1024	2.4	2.3	2.3	2.3	2.3	100	100	100	100	100
4	2048	2.3	2.2	2.2	2.2	2.2	100	100	100	100	100
4	4096	2.4	2.4	2.5	2.5	2.5	100	100	100	100	100
4	8192	2.7	2.7	2.7	2.7	2.7	100	100	100	100	100
8	1024	3.5	3.7	2.8	3.5	3.6	50	82	74	82	83
8	2048	3.7	4.0	3.8	3.8	3.8	50	84	64	70	71
8	4096	4.1	4.3	4.1	4.2	4.3	50	84	64	66	70
8	8192	4.6	4.8	4.5	4.8	4.8	50	84	64	65	72
16	1024	4.0	4.7	2.3	4.0	4.2	25	62	77	65	63
16	2048	5.5	7.0	3.7	5.8	6.1	25	76	47	56	61
16	4096	6.4	7.6	5.3	6.9	7.3	25	76	45	47	53
16	8192	7.4	8.3	6.4	8.1	8.4	25	77	43	48	56
32	1024	3.9	4.8	2.1	4.1	4.1	12	51	72	53	52
32	2048	6.0	8.2	3.6	6.7	7.0	12	58	40	38	42
32	4096	8.1	11	6.7	9.2	9.6	12	73	28	42	47
32	8192	9.8	13	8.5	11	11	12	73	29	41	45

Distribution: Interleaved Horizontal Vertical Blocks Pairwise

[a] Matrix side size. The actual size of the FFT is this value squared.
[b] Gigaflops per second.
[c] Mean memory affinity.

Table 5.7: Performance summary of the FFT distributions.

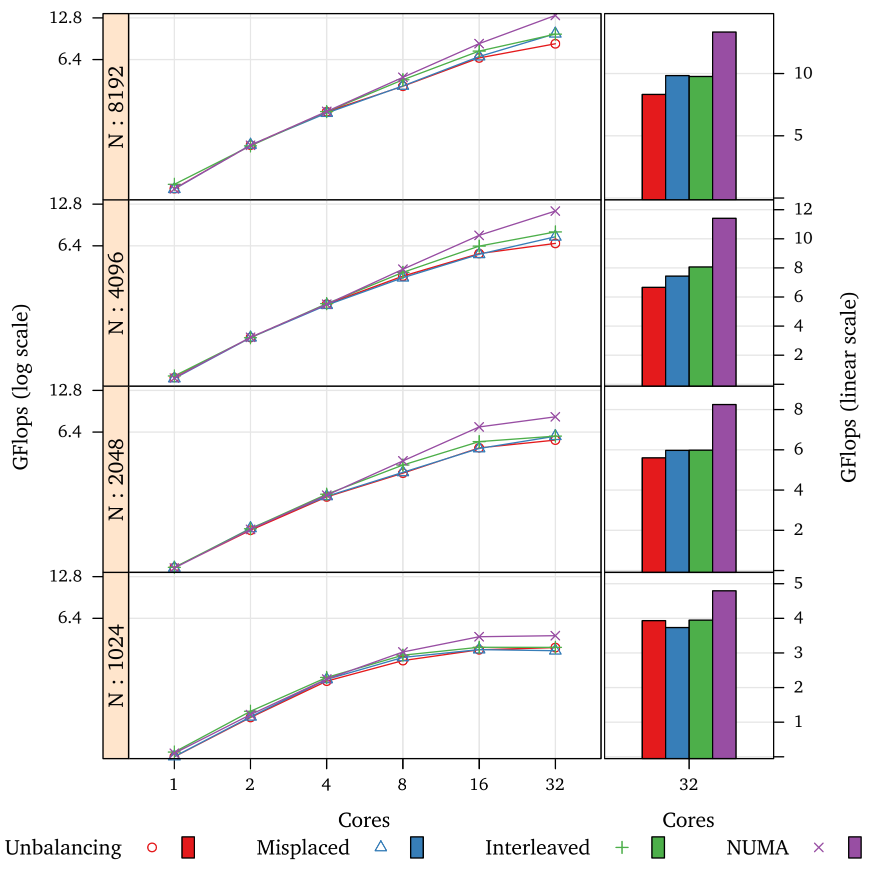


Figure 5.17: Strong scalability of the FFT algorithm with memory load unbalancing scheduler, the misplacing scheduler, the NUMA-unaware scheduler and the NUMA-aware scheduler.

Effectiveness of the NUMA Scheduling Policy

To evaluate how much memory affinity and memory load balancing affect performance on this code, we have selected the horizontal distribution and made additional measurements with the “bad” schedulers. Figure 5.17 shows the scalability of each case with the same blocking size as the NUMA scheduler.

The performance with the “bad” schedulers is not significantly different to the performance with the NUMA-unaware scheduler with memory interleaving. However, all three cases scale much worse than the executions with the NUMA-aware scheduler. These results show that NUMA affinity is important in this code to achieve good performance, and that an interleaved memory placement is not a good compromise between simplicity and performance in this case.

Performance Compared to Other Implementations

To compare the performance of the NUMA-aware implementation against other programming models, we compare it against the additional versions already presented in the evaluation of the previous chapter starting in page 118.

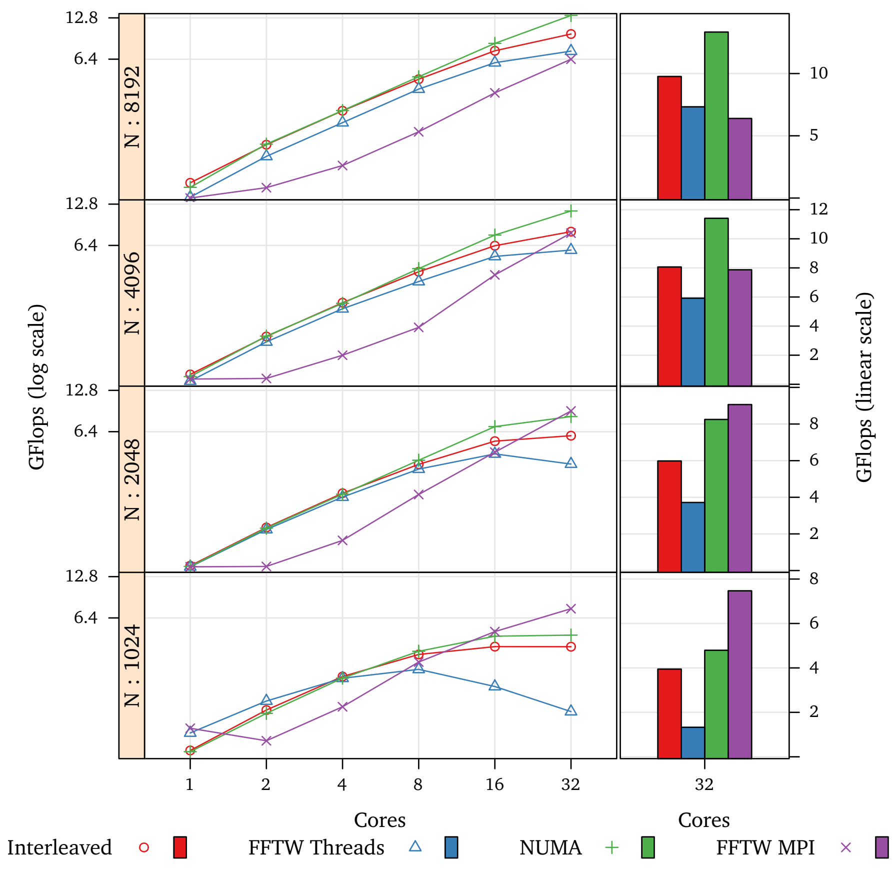


Figure 5.18: Strong scalability and performance with 32 cores of the FFT algorithm with several variants under SMPSs, the threaded FFTW implementation and the MPI FFTW implementation.

Figure 5.18 shows the strong scalability of each implementation with several problem sizes with the best performing block size where applicable. The series labeled "Interleaved" corresponds to the NUMA-unaware executions with memory page interleaving. The series labeled "FFTW Threads" corresponds to the threaded implementation that the FFTW provides with the memory pages interleaved. The "NUMA" series corresponds to executions with the NUMA-aware scheduler and the horizontal distribution. Finally, the series labeled "FFTW MPI" corresponds to executions using the MPI version of the FFTW.

Notice that while the MPI version is the best performer for the two smallest problem sizes, the NUMA-aware executions perform significantly better for the bigger problem sizes, despite the fact that the MPI version is a highly tuned implementation. Moreover, even the NUMA-unaware executions with the biggest problem size manages to perform better than the MPI version.

Table 5.8 summarizes the performance of those experiments numerically.

Cores	N[a]	GF[b]	GF[b]	GF[b]	GF[b]	FPC[c]	FPC[c]	FPC[c]	Eff.[d] (%)	Eff.[d] (%)	Eff.[d] (%)
1	1024	0.7	0.9	0.7	1.0	0.5	0.5	0.5	96	∅	97
1	2048	0.7	0.7	0.7	0.7	0.5	0.4	0.5	96	∅	99
1	4096	0.7	0.7	0.7	0.7	0.5	0.3	0.5	99	∅	99
1	8192	0.8	0.6	0.7	0.6	0.6	0.3	0.5	99	∅	99
2	1024	1.4	1.6	1.3	0.8	0.5	0.4	0.5	95	96	97
2	2048	1.3	1.2	1.3	0.7	0.5	0.3	0.4	96	99	98
2	4096	1.4	1.3	1.4	0.7	0.5	0.3	0.5	99	98	99
2	8192	1.5	1.3	1.5	0.7	0.5	0.3	0.5	99	98	99
4	1024	2.4	2.3	2.3	1.4	0.5	0.4	0.4	92	91	94
4	2048	2.3	2.1	2.2	1.0	0.4	0.3	0.4	96	98	98
4	4096	2.4	2.2	2.4	1.0	0.4	0.3	0.4	98	98	99
4	8192	2.7	2.2	2.7	1.1	0.5	0.3	0.5	98	98	98
8	1024	3.5	2.7	3.7	3.1	0.3	0.3	0.4	90	76	89
8	2048	3.7	3.4	4.0	2.2	0.3	0.3	0.4	96	94	94
8	4096	4.1	3.5	4.3	1.6	0.4	0.2	0.4	97	98	97
8	8192	4.6	3.9	4.8	1.9	0.4	0.3	0.4	97	98	98
16	1024	4.0	2.0	4.7	5.1	0.2	0.2	0.3	84	48	81
16	2048	5.5	4.4	7.0	4.5	0.3	0.2	0.3	95	82	92
16	4096	6.4	5.3	7.6	3.9	0.3	0.2	0.3	97	92	96
16	8192	7.4	6.0	8.3	3.6	0.3	0.2	0.4	96	94	95
32	1024	3.9	1.3	4.8	7.5	0.1	0.1	0.2	73	33	61
32	2048	6.0	3.7	8.2	9.1	0.2	0.2	0.2	92	51	84
32	4096	8.1	5.9	11	7.9	0.2	0.2	0.3	95	79	92
32	8192	9.8	7.3	13	6.4	0.2	0.2	0.3	97	81	95

Implementation: Interleaved FFTW Threads NUMA FFTW MPI

[a] Matrix side size. The actual size of the FFT is this value squared.
[b] Gigaflops per second.
[c] Mean floating point operations per cycle while running tasks.
[d] Mean time that threads spend running tasks.

Table 5.8: Performance summary of the FFT implementations.

Chapter 6

Conclusions, Impact and Future Work

6.1 Conclusions

In this thesis we have presented solutions to problems that appear when programming parallel applications. The first is a general one, and the other two are successive refinements over the first.

First, we have presented a parallel programming model that uses annotations on top of regular C to make it capable of supporting parallelism. The key aspect that differentiates it over other models is that it relies on a powerful runtime that handles data dependencies, and thus removes that burden from the programmer.

Second, we have proposed an extension to the syntax of the language and to the dependency analysis algorithm to support strided and overlapping sets of data. This extension allows the programming model to cover a wider spectrum of applications and data layouts.

Finally, we have proposed a methodology to exploit NUMA affinity within the runtime. One of its key aspects is that it reuses the elements of the language to define the units of data distribution, and relies on runtime policies to determine their placement. These two aspects define together data distributions.

In the following sections we discuss in detail the conclusions that derive from the work performed in each area.

6.1.1 Simplifying programming parallel applications with data dependencies

The first group of contributions of this thesis addresses the programmability of parallel applications with data dependencies and has been presented in chapter 3.

Programming model First, we have proposed a simple programming model that presents very few language elements and that moves as most of the complexity to the runtime. By implementing and comparing the performance to alternatives using other programming models and high performance implementations, we have shown that despite the minimalism, the programming model we propose has several advantages.

Easier parallelization Code can be more easily parallelized since the programmer does not need to think in terms of data dependencies. Moreover, some codes are closer to their sequential implementation than it is possible in other programming models, which require code restructuring to achieve parallelism.

Less imbalance The model is capable of finding more parallelism that other programming models. The availability of more parallelism reduces thread starvation and therefore also imbalance.

Better task performance Applications can take advantage of the additional parallelism that the programming model obtains to make tasks bigger. Doing this potentially allows them to make better use of locality and thus to improve their performance.

Scheduling The scheduler that we presented follows data dependencies to favor the reuse of data already in the cache of the processors. This improves the performance of applications in which the cost of the task first-access data cache misses is significant. In our evaluation we have compared its performance against three other scheduling policies. The results show that in most cases, the task protection mechanism is enough to preserve the benefits of reusing the data in the cache and its effect prevails over the rest of the aspects of the scheduling policy.

Applications We have presented a set of benchmarks that range from embarrassingly parallel, to benchmarks with complex dependencies. The simplicity of their code and their performance demonstrate that making the programming model aware of data dependencies simplifies coding and improves performance. In embarrassingly parallel codes we have shown that our implementation does not incur in too much overhead despite the fact that it analyzes data dependencies at run time. In codes with dependencies we have show that the overhead is offset by the improvement in performance.

The Strassen-Winograd code, which fits naturally in task nesting programming models has shown that our proposal is capable of extracting much more parallelism than task nesting. The Gauss-Seidel benchmark has shown that the model is capable of exploiting the parallelism between several wavefronts. But equally important is that the model removes the synchronism in the advance of the waveform. That is, it starts executing tasks of the following step of the waveform before the tasks of the current have finished. This demonstrates that by being dynamic, the model is capable of exploiting unforeseen parallelism.

6.1.2 Handling dependencies over strided and overlapping data accesses

The second group of contributions of this thesis relates to adding support for applications that have dependencies between strided and overlapping data accesses. This has been discussed in chapter 4.

The main contributions of this part of the thesis are extensions of the programming syntax to support the specification of the strided data accesses, a compact representation to summarize a set of addresses, a data structure to perform data dependency analysis efficiently and an extended evaluation. The benefits of the approach that we proposed are the following:

Improved programmability By extending the language with support for strided and partially overlapping accesses, applications can use flat arrays. This simplifies writing and porting already existing codes. Moreover, by using and internal representation that is independent of array bounds and indexing, we have shown that it is possible to allow the full range of operations that C allows over arrays and pointers. More specifically, the address-based representation allows pointer arithmetic and decaying arrays into pointers. This removes restrictions and thus simplifies porting and reusing already existing codes.

Increased range of applicability By allowing data dependencies over strided and overlapping data accesses, the model allows to parallelize a wider range of applications that could not take advantage of data dependency aware parallelization with the blocked model.

Moderate overhead Our analysis shows that despite the additional overhead, region support in most cases does not hurt performance significantly. Nevertheless, in many cases the tiled data layout has better task performance than the strided layout and thus performs better. The evaluation also compares to alternative implementations with other programming models and highly tuned parallel library implementations. Where we used flat data layouts, the SMPSs performed at least on par to the alternative implementations, and in some cases it performed substantially better.

Better adaptation to task performance characteristics The extended model allows tasks to have different blocking sizes for each type of task. This allows to chose a blocking size that better adapts to the parallelism, the runtime overhead and the performance characteristics of the task. In some codes, this has an important impact on performance.

Better adaptation to the amount of parallelism Blocking size freedom allows the programmer to use different task granularities depending of the expected amount of parallelism of each part of the code. For instance, the amount of work between the earliest and the latest iterations of an algorithm may produce very different number of tasks. By adapting task granularity at each iteration, programmers can reduce overhead in parts with too many tasks, and to increment parallelism in parts with too few tasks.

In addition we have presented a set of codes that were not possible to program without supporting dependencies over strided and partially overlapping sets of data, or that at least had only compromised solutions that negated the advantages of the programming model. The new codes perform on par and in some cases surpass the performance of highly tuned implementations.

6.1.3 Exploitation of NUMA

The third group of contributions is related to making the programming model aware of NUMA and to improve the performance of applications running under that environment. These aspects have been covered in chapter 5 and lead to the following conclusions:

Data distribution does not require special syntax We have demonstrated that the syntax and concepts of the previous chapter are enough to support the distribution of data. Similarly to how SPMD programming models define their data distributions implicitly by allocating each set of data in a different instance of the program, the solution we propose takes advantage of data initialization tasks to define the units of distribution.

Data placement can be dynamic We have demonstrated that on algorithms that do not have special requirements about the exact distribution of the data, the actual placement can be decided by the runtime and thus does not need to be defined by the programmer. Instead the programmer only needs to define the units of data distribution, and the runtime can perform the actual placement with a simple policy that favors placements that equalize the amount of data in each node.

Dynamic data placement can perform better By determining the placement of data dynamically, and by following scheduling policies that favor the execution of tasks in NUMA-affine nodes, temporary data is more likely to end up placed in the same NUMA node as one of the inputs of the task that creates it. This way, the placement of the input data is propagated to temporary data. Thus, the runtime places together data that is correlated. In addition, since placement occurs on-demand, temporary data is always initialized in the local memory, and thus its initialization has maximum affinity.

An abundance of tasks allows scheduling the computations While most of the previous work is based on data centered techniques, our solution is based on scheduling the computations. Since the programming model is in many cases capable of extracting more parallelism than dependency-unaware alternatives, restricting the execution of tasks to certain nodes is less likely of producing starvation. This allows us to approach the problem from the point of scheduling instead of from the point of data management. In addition, task stealing further reduces unbalance.

The region tree data indexing is reusable While the region tree data structure defined in the previous chapter was used to calculate data dependencies, in the chapter dedicated to NUMA we demonstrate that it can also be used to index a cache of the location of the data. This cache is necessary since the operating system interfaces to query the location of data are inadequate for strided ranges.

In addition, the evaluation section has analyzed the sensitivity of a set of benchmarks to NUMA affinity, and the effectiveness of the NUMA-aware scheduling policy. The analysis also includes an evaluation of the effects of the shape of the units of distribution on performance. Moreover, where possible, we compare our solutions to alternative implementations of the benchmarks using other programming models, and high performance reference implementations. The evaluation has lead to the following conclusions:

Some codes do not benefit with our current configuration Due to the limitations on the hardware used in the evaluation, the measurements only cover executions with up to 8 NUMA nodes and 32 cores. Under these conditions, the performance of some benchmarks does not benefit significantly from NUMA affinity. Most of these cases involve matrix multiplication tasks that already use a highly tuned code. Moreover, in some codes, with small problem sizes, a tiled layout seems to be more beneficial to performance that NUMA affinity. However, we expect that NUMA affinity would have a bigger impact with more nodes.

Some codes perform better than highly optimized alternatives In all cases, the NUMA-aware SMPSs implementation performs on par or better than OpenMP and high performance library implementations. Moreover, for big problem sizes, it manages to outperform FFTW with threads, FFTW with MPI, and the reference implementation of High Performance Linpack.

Some codes respond very badly to interleaved page placement The most bandwidth demanding benchmarks of the evaluation represent vector to vector operations, stencil algorithms, sorting algorithms and Fast Fourier Transforms. The measurements show that they are very sensitive to NUMA affinity, and that ignoring NUMA by interleaving the placement of the memory pages does not scale well.

Page-level distribution can be too restrictive Data distribution in its current form only allows page-level data distribution. Abstract array indexing, like some other programming languages do, allows to decouple data distribution from the page size, and thus enables sub-page distributions. This restriction affects specially the distribution across the contiguous dimension, and in this case, since we are using C, vertical distributions, and to a lesser extent, blocked distributions. Under bad conditions, data cannot be distributed between all the nodes.

Best distribution shape candidates The analysis of the shapes of the units of distributions shows that in most cases the distribution shape that performs the best is the distribution in horizontal panels, and that it is closely followed by the distribution by blocks. Notice that the codes are written in C, and thus the horizontal panels correspond to regions whose contiguous dimension is fully covered.

6.2 Publications Produced

The work contained in this thesis has generated publications in each of the three main topics.

6.2.1 General Programming Model

The general programming model was first published by [Perez et al., 2006] in a form tailored for Grids and shared memory. It is further refined by [Bellens et al., 2006;

Perez et al., 2007], which targets the Cell B.E. processor. Its final form for shared memory is published by [Perez et al., 2008].

> Josep M. Perez, Rosa M. Badia, and Jesus Labarta. Including SMP in grids as execution platform and other extensions in GRID superscalar. In *Second IEEE International Conference on e-Science and Grid Computing 2006 (e-Science '06)*, December 2006. doi: 10.1109/E-SCIENCE.2006.261144.

> Pieter Bellens, Josep M Perez, Rosa M Badia, and Jesus Labarta. Cellss: a programming model for the Cell BE architecture. In *Proceedings of the ACM/IEEE SC 2006 Conference*, Tampa, FL, USA, Nov 2006. ISBN 0-7695-2700-0. doi: 10.1109/SC.2006.17.

> Josep M. Perez, Pieter Bellens, Rosa M. Badia, and Jesus Labarta. CellSs: Making it easier to program the Cell Broadband Engine processor. *IBM Journal of Research and Development*, 51(5):593–604, September 2007. ISSN 0018-8646. doi: 10.1147/rd.515.0593.

> Josep M. Perez, Rosa M. Badia, and Jesus Labarta. A dependency-aware task-based programming environment for multi-core architectures. In Causal Productions, editor, *Proceedings of the 2008 IEEE International Conference on Cluster Computing*, pages 142–151, September 2008. ISBN 978-1-4244-2639-3. doi: 10.1109/CLUSTR.2008.4663765.

6.2.2 Extensions to Support Strided and Overlapping Data Accesses

The support for strided and possibly overlapping data accesses was published by [Perez et al., 2010].

> Josep M. Perez, Rosa M. Badia, and Jesus Labarta. Handling task dependencies under strided and aliased references. In *Proc. of the 24th ACM Int. Conf. on Supercomputing*, ICS '10, pages 263–274, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0018-6. doi: 10.1145/1810085.1810122.

It was used as a basis for a submission to the Class 2 HPC Challenge Competition 2010 held as part of the Supercomputing 2010 conference. The submission obtained a honorable mention.

> Josep M. Perez, Rosa M. Badia, and Jesus Labarta. SMPSs Submission to HPCC 2010 Class 2 Competition. Honorable mention in *HPC Challenge 2010 Class 2 Awards*. ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10, New Orleans, LA, USA, Nov 2010.

6.2.3 Exploiting Non-Uniform Memory Access

The work related to exploiting the performance of NUMA architectures is still pending publication.

> Josep M. Perez, Rosa M. Badia, and Jesus Labarta. Exploiting NUMA Locality with Task-Based Programming Models. To be submitted.

6.3 Impact and Future Work

6.3.1 The Programming Model under Other Environments

While the work of this thesis is centered around shared memory multiprocessors, the programming model has been successfully used in other environments. First, [Perez et al., 2006] demonstrate its usage in a mixed environment called GridSs consisting of local shared memory processors and the computing resources provided by a Grid. This environment constitutes a mixture of shared memory and heterogeneous distributed computers.

The model has also been successfully used on the Cell Broadband Engine (Cell B.E.) architecture. [Bellens et al., 2006; Perez et al., 2007] demonstrate the usage and performance on a single node containing 2 processors of that architecture. The Cell B.E. processor described by [Chen et al., 2007] consists of a main core and 8 coprocessors. The main core is in-order and has a Power ISA. It is capable of executing 2 simultaneous threads. The coprocessors are called SPUs or SPEs, and have an exclusively SIMD ISA. Each SPU has a local embedded memory. While the main processor functions as a regular processor that accesses the main memory, the SPUs can only operate over their local memory. They can access other memories only through DMA data copies from or to their embedded memory.

The CellSs programming model that [Bellens et al., 2006; Perez et al., 2007] propose runs the main program in the main processor and the tasks in the SPUs. In this sense, it demonstrates the feasibility of the model over a distributed architecture. Moreover, due to the limited amount of memory of the SPUs and their use of SIMD instructions, CellSs tasks are usually very fine grained. Therefore, that work demonstrates the suitability of the model for very fine grained tasks.

The COMP Superscalar model (COMPSs) presented by [Tejedor and Badia, 2008], is based on the same principles but tailored for applications written in the Java language that must run in parallel in distributed environments. These can be clusters, grids and clouds. COMPSs enhances the model with support for object orientation features, and web services. A key difference to the work of this thesis is that COMPSs uses futures to support changing the state of the objects asynchronously.

The whole set of programming models and the environment-specific implementations have been labeled as *StarSs* by [Labarta, 2010].

The model can also be exported to other emerging technologies. Graphical Processor Units have also been the target of the work by [Ferrer et al., 2011]. In that case, the implementation is based on the OmpSs model, which in turn is based on the work of this thesis. This is further explained in the following section.

6.3.2 Impact on Standards

The OmpSs model by [Duran et al., 2008] is the first attempt at introducing the concepts of the programming model of this thesis into OpenMP. While this work is based on the annotation of functions, the OpenMP tasks are regular blocks of code. In this sense, the OmpSs syntax differs from the SMPSs syntax to cover this difference. In addition, the OmpSs proposal also covers distributed systems. While the proposals of GridSs, CellSs and CompSs consider data and dependencies together, the OmpSs proposal decouples both concepts. In that sense, OmpSs adds primitives to declare data transfers, whereas the other distributed models also based on this thesis used the directionality information to determine them.

The work in OmpSs has been used as test ground for the inclusion of data dependencies in OpenMP tasks. The OpenMP 4 specification by the [OpenMP Architecture Review Board, 2013] is the first version to include dependencies between tasks. While OmpSs includes the work developed in chapter 4 to support dependencies over strided and overlapping subarrays, the OpenMP 4 specification has the syntax to support them, but [OpenMP Architecture Review Board, 2013, pg. 117, lin. 14-15] restricts its usage to the same as the original model explained in chapter 3:

> "List items used in depend clauses of the same task or sibling tasks must indicate identical storage or disjoint storage".

6.3.3 Future and Derived Work

Improved Reductions

While the reductions presented in this thesis are complete, they are just a stop gap to avoid the serialization that otherwise would occur by using **inout** accesses. A more flexible design would automatically construct reduction trees and thus would scale better.

Reductions are the most important bottleneck that we find when we parallelize applications. Some algorithms are resistant to minor precision errors. Some of the ongoing work is related to taking advantage of that fact to reduce the unbalance produced by reductions.

Some optimization algorithms when parallelized can obtain additional speedups by bounding their work based on the current optimum solutions calculated by the other threads. This is an aspect that could be exploited within reductions that have that kind of behavior.

The work on improved reductions has already been started by [Ciesko et al.].

Dynamic Data Reshaping, Data Replication and Renaming

While the work presented in chapter 4 allows to program with flat arrays, the blocked data layout has better locality and in many cases achieves better task performance.

Regions in SMPSs are determined at run time. This allows the programmer to parametrize the array sizes and the regions. In fact all our benchmarks receive as parameters the dimensions of the problem and the dimensions of the block size. These determine the dimensions of the data and the area covered by the task regions, which is passed by parameter and thus determined at the task creation time.

This approach to writing code enables us to take one step further: *dynamic data reshaping*. That is, while tasks receive their data assuming a certain *shape*, if the task parameters define the shape of the data, the compiler can emit alternative versions of the tasks that accept data with a different shape that the runtime can determine dynamically. These alternative versions receive the parameters that determine the data shape with the values that correspond to the actual shape that the runtime passes to them.

By doing this, the runtime receives the task invocations over flat arrays and can transform the layout of the data to a blocked one, which has better spatial locality an thus more potential task performance. Since data accesses can be partially overlapping, this can lead to several tasks accessing different subsets of the data that intersect. To cover that case, this mechanism can be coupled with the ability to manage more than one replica of the data.

Allowing data replicas in that way has two advantages. First, it allows us to exploit NUMA affinity through replication to the memories of the nodes where the computation takes place. And second, it would allow us to re-export the programming model back to distributed environments, this time with region support.

Moreover, data replication and reshaping are good foundations to allow the implementation of region-aware data renaming, which allows to uncover the parallelism that currently cannot be exploited due to “false” dependencies.

More Accurate NUMA and Data Access Modeling

The model used for determining the NUMA-aware scheduling policy presented in chapter 5 presents a series of shortcomings. First, it assumes that each byte of the parameters of a task is accessed a uniform number of times, and does not take into account the effects of data caches nor the order of the accesses. These factors can have a dramatic impact of the task performance, since they determine the sensitivity of the performance of a task to NUMA affinity. Therefore should be taken into consideration. The solution could be either based on user-provided information, compiler-generated information, or on information extracted through profiling.

Second, the scheduling algorithm assumes a one level NUMA system with identical nodes, uniform local memory performance and uniform remote memory performance. The model could be enhanced to take into account cases in that are less restrictive. Both this and the first point would affect the metric used to determine where to assign tasks to memories and how to perform task stealing.

Task Nesting and the Parallel Generation of Work

While tasks in OpenMP allow nesting, the programming model we presented in this thesis does not. Task nesting allows to parallelize the generation of work, and thus has the potential to reduce the task creation bottleneck. In fact, all the benchmarks of this thesis that have been written in OpenMP take advantage of the generation of tasks in parallel, either through nesting or through other outer parallel constructs.

Another benefit is that nesting allows to extract some parallelism in the presence of dependencies. However, handling dependencies directly allows us to extract more parallelism. These aspects have been demonstrated in the Strassen benchmarks that start in page 102.

While task nesting and dependency handling may seem orthogonal aspects, they can be combined to achieve the same amount of parallelism as dependency handling and to allow the parallel generation of work at the expense of additional overhead. To this end, the programming model needs to be extended in two ways. First it needs to allow task nesting as a form of work decomposition in such a way that parent tasks specify the total data accesses of their contents, which includes that of their children. And second, it needs to allow to specify whether a task will further decompose the problem into other tasks, or it will truly make use of their parameters. Alternatively, the first direct access to each parameter within a task could be guarded by a **wait on** pragma to signify the need to access to that memory.

By doing this, the runtime could send potentially decomposable tasks to be executed in parallel before their data dependencies are satisfied. This would allow them to further decompose the problem and generate the information required to calculate the input dependencies to their inner subtasks, and the information required to calculate their output dependencies.

By matching the input and output dependency information of the inner subtasks across the outer tasks, it is possible to calculate the dependencies between all of the subtasks as if they were not created at an inner nesting level. However, it is not clear whether it would still be beneficial after discounting the additional overhead.

Appendix A

SMPSs Extensions to the C99 Grammar

A.1 Initialization and Finalization

(1)	⟨start-directive⟩ →
(1.1)	**#pragma css start**
(2)	⟨finish-directive⟩ →
(2.1)	**#pragma css finish**

A.2 Task

(3)	⟨task-construct⟩ →
(3.1)	⟨task-declaration⟩
(3.2)	| ⟨task-definition⟩
(4)	⟨task-declaration⟩ →
(4.1)	⟨task-pragma⟩ ⟨function-declaration⟩
(5)	⟨task-definition⟩ →
(5.1)	⟨task-pragma⟩ ⟨function-definition⟩
(6)	⟨task-pragma⟩ →
(6.1)	**#pragma css task** ⟨opt-task-clauses⟩ ⟨new-line⟩
(7)	⟨opt-task-clauses⟩ →
(7.1)	⟨task-clauses⟩
(7.2)	|
(8)	⟨task-clauses⟩ →
(8.1)	⟨task-clauses⟩ ⟨task-clause⟩
(8.2)	| ⟨task-clause⟩

(9) ⟨task-clause⟩ →
(9.1) **input (** ⟨task-parameter-list⟩ **)**
(9.2) | **output (** ⟨task-parameter-list⟩ **)**
(9.3) | **inout (** ⟨task-parameter-list⟩ **)**
(9.4) | **reduction (** ⟨task-parameter-list⟩ **)**
(9.5) | **highpriority**

(10) ⟨task-parameter-list⟩ →
(10.1) ⟨task-parameter⟩
(10.2) | ⟨task-parameter⟩ **,** ⟨task-parameter-list⟩

(11) ⟨task-parameter⟩ →
(11.1) ⟨identifier⟩ ⟨opt-task-parameter-dimensions⟩ ⟨opt-region⟩

(12) ⟨opt-task-parameter-dimensions⟩ →
(12.1) ⟨task-parameter-dimensions⟩
(12.2) |

(13) ⟨task-parameter-dimensions⟩ →
(13.1) ⟨task-parameter-dimensions⟩ ⟨task-parameter-dimension⟩
(13.2) | ⟨task-parameter-dimension⟩

(14) ⟨task-parameter-dimension⟩ →
(14.1) **[** ⟨expression⟩ **]**

(15) ⟨opt-region⟩ →
(15.1) ⟨region-specifiers⟩
(15.2) |

(16) ⟨region-specifiers⟩ →
(16.1) ⟨region-specifiers⟩ ⟨region-specifier⟩
(16.2) | ⟨region-specifier⟩

(17) ⟨region-specifier⟩ →
(17.1) **{ }**
(17.2) | **{** ⟨expression⟩ **}**
(17.3) | **{** ⟨expression⟩ **..** ⟨expression⟩ **}**
(17.4) | **{** ⟨expression⟩ **:** ⟨expression⟩ **}**

A.3 Mutual exclusion

(18) ⟨lock-directive⟩ →
(18.1) **#pragma css mutex lock (** ⟨expression⟩ **)**

(19) ⟨unlock-directive⟩ →
(19.1) **#pragma css mutex unlock (** ⟨expression⟩ **)**

A.4 Synchronization

(20) ⟨barrier-directive⟩ →
(20.1) **#pragma css barrier**

(21) ⟨waiton-directive⟩ →
(21.1) **#pragma css wait on (** ⟨waiton-expression-list⟩ **)**

(22) ⟨waiton-expression-list⟩ →
(22.1) ⟨conditional-expression⟩
(22.2) | ⟨conditional-expression⟩ **,** ⟨waiton-expression-list⟩

Appendix B

Compiler and Runtime Integration

The programming environment consists of a compiler and a supporting runtime library. Since the programming model is highly dynamic, the compiler does not perform dependency analysis. Instead, it relays the information of the programming primitives to the runtime. The runtime receives the actual information and performs the task instantiation, the dependency calculation, the scheduling, the task invocation and the synchronization.

B.1 Runtime Interface

The Interface between the transformed code and runtime is bidirectional. On one hand, the transformed code must invoke runtime services like task instantiation and synchronization. On the other hand, the runtime must be able to invoke the task implementations.

B.1.1 Runtime Initialization and Finalization

The application must initialize the runtime before using it. The function for initializing it is the following:

```
void css_init(void);
```

It initializes the internal data structures and forks the worker threads. This function corresponds to the **start** primitive described in section 3.3.1.

Before exiting, the application must finalize the runtime by calling the following functions:

```
void css_finish(void);
```

This function executes a barrier, joins the worker threads, and frees up the resources used by the runtime. It corresponds to the **finish** primitive described in section 3.3.1.

B.1.2 Task Implementation Registration

To allow tasks to execute asynchronously, the runtime must be able to invoke task implementations. Since constructing arbitrary function calls programmatically from C is complex and non-portable, a standard interface has been defined for all tasks that is flexible enough to allow any number of parameters and types.

For each task implementation we define an additional function called the *adapter* that has the standard interface, from now on the *adapter interface*. Its purpose is to call the actual task implementation by passing the parameter values in the format determined by the actual task implementation prototype.

The type declaration of the adapter in C is the following:

```
typedef void (*function_adapter) (void *parameters[]);
```

That is, adapters are functions that return **void** and receive a unique parameter that is an array of pointers to **void**. The array contains an address for each parameter from left to right. Values and C **struct**s are passed to the adapter as a pointer to the actual value or **struct**. Arrays are passed as a pointer to the first element, and pointers are passed as-is.

The adapter passes scalar values and **struct**s by dereferencing their pointers in the parameter array, and passes arrays and pointers by passing the pointers stored in the parameter array.

Adapters must be registered within the runtime to allow it to perform the actual task invocations. The registration function is the following:

```
void css_registerTask(char const *name, function_adapter adapter);
```

It takes the name of the task and a pointer to the adapter function. The name is used by the runtime to identify the task types when it generates execution traces. This is discussed in section 3.5.2 in page 28. The adapter is used to invoke the task implementation asynchronously.

During initialization, all adapters must be registered. Tasks are also identified by a number that corresponds to their registration order, starting from 0.

B.1.3 Task Instantiation

Task invocations are notified to the runtime by calling the following function:

```
void css_addTask(unsigned int functionId, unsigned int flags, unsigned int
   parameterCount, css_parameter_t const *parameters);
```

It receives a task identifier that has been determined by the adapter registration order, flags, the number of parameters and an array with the actual parameter description. The only flag currently available is the high priority flag, which corresponds to a call to a task that has the **highpriority** clause.

The parameter descriptor is the following:

```
typedef struct {
   long flags;
   size_t size;
   void *address;
} css_parameter_t;
```

The value of the flags field indicates the directionality of the parameter and if it is the target of a reduction. The possible values and semantics are the following:

Flag Value	Semantics
1	An input parameter
2	An output parameter
1+2	An inout parameter
1+4	An input parameter that is a constant
1+2+8	An inout parameter that is the target of a reduction

The size field indicates the size in bytes of the data, and the address field points to the data.

B.1.4 Synchronization

The runtime provides the following functions for synchronization:

```
void css_barrier(void);
void css_waitOn(unsigned int variableCount, void **addresses);
```

The first one executes a full barrier and corresponds to the **barrier** directive described in section 3.3.4.

The second one corresponds to the **wait** directive described in the same section. Its first parameter is the number of variables in the **on** clause. The second parameter is an array which one entry for each object over which the user code must wait. Each entry is a pointer that corresponds to the base address in case of a scalar or a **struct**, and a pointer to the first element of the block in case of an array.

B.2 Compiler

The compiler is a source-to-source compiler that relies on a standard C99 compiler for the actual compilation. It also serves as a front end for linking the application transparently with the runtime library.

During compilation it translates C code extended with the programming model annotations into standard C99 code with calls to the runtime library. The compilation process consists of three phases. First it analyzes the code. Then it performs the code transformation using the data gathered during the analysis. And finally, It uses the platform native compiler to generate the actual object files.

B.2.1 Code Analysis

During the analysis phase, the compiler visits the **task** pragmas to extract their information. First, it parses the pragma. Then it verifies that it is semantically correct. Finally it extends the symbol the function with the task information. More specifically, it adds a field that identifies the function as a task, the parameter information, and whether the task has high priority. For each parameter, it stores its directionality and its dimensions.

B.2.2 Code Generation

During the code generation phase, the compiler eliminates the directives of the programming language and generates calls to the runtime where needed. In addition it generates the code necessary to invoke the tasks from within the runtime.

```
double *a, *b, *c; long BSIZE;

#pragma css start
for (long i = 0; i < N; i+= BSIZE)
   initialize_segment(BSIZE, &a[i], &b[i], &c[i], 0.0, 2.0, 1.0);

#pragma css barrier
for (long i = 0; i < N; i += BSIZE)
   triad(BSIZE, &a[i], &b[i], &c[i], scalar);

#pragma css finish
```

Listing B.1: Main code of the triad benchmark.

```
#pragma css task input(size, a_scalar, b_scalar, c_scalar) output(a, b, c)
void initialize_segment(long size, double a[size], double b[size], double c[size],
  double a_scalar, double b_scalar, double c_scalar);

#pragma css task input(size, b, c, scalar) output(a)
void triad(long size, double a[size], double b[size], double c[size], double
  scalar);
```

Listing B.2: Declaration of the tasks used by the triad benchmark.

Listings B.1 and B.2 show a version of the triad benchmark that we will use to illustrate the code conversion process. Listing B.3 shows the main code after applying the compiler transformations.

```
task_registration___cssgenerated();
css_init();
for (long i = 0;i < N; i += BSIZE)
{
   long int parameter_0___cssgenerated = BSIZE;
   double parameter_4___cssgenerated = 0.0;
   double parameter_5___cssgenerated = 2.0;
   double parameter_6___cssgenerated = 1.0;
   css_parameter_t parameters___cssgenerated[7];
   parameters___cssgenerated[0].flags = CSS_IN_SCALAR_DIR;
   parameters___cssgenerated[0].size = sizeof(long int);
   parameters___cssgenerated[0].address = &parameter_0___cssgenerated;
   parameters___cssgenerated[1].flags = CSS_OUT_DIR;
   parameters___cssgenerated[1].size = (BSIZE) * sizeof(double);
   parameters___cssgenerated[1].address = &a[i];
   parameters___cssgenerated[2].flags = CSS_OUT_DIR;
   parameters___cssgenerated[2].size = (BSIZE) * sizeof(double);
   parameters___cssgenerated[2].address = &b[i];
   parameters___cssgenerated[3].flags = CSS_OUT_DIR;
   parameters___cssgenerated[3].size = (BSIZE) * sizeof(double);
   parameters___cssgenerated[3].address = &c[i];
   parameters___cssgenerated[4].flags = CSS_IN_SCALAR_DIR;
23    parameters___cssgenerated[4].size = sizeof(double);
24    parameters___cssgenerated[4].address = &parameter_4___cssgenerated;
25    parameters___cssgenerated[5].flags = CSS_IN_SCALAR_DIR;
26    parameters___cssgenerated[5].size = sizeof(double);
27    parameters___cssgenerated[5].address = &parameter_5___cssgenerated;
28    parameters___cssgenerated[6].flags = CSS_IN_SCALAR_DIR;
29    parameters___cssgenerated[6].size = sizeof(double);
30    parameters___cssgenerated[6].address = &parameter_6___cssgenerated;
31    css_addTask(initialize_segment_task_id___cssgenerated, CSS_NO_FLAG, 7,
        4, parameters___cssgenerated);
32  }
33
34  css_barrier();
35  for (long i = 0; i < N; i += BSIZE)
36  {
37    long int parameter_0___cssgenerated = BSIZE;
38    css_parameter_t parameters___cssgenerated[5];
39    parameters___cssgenerated[0].flags = CSS_IN_SCALAR_DIR;
40    parameters___cssgenerated[0].size = sizeof(long int);
41    parameters___cssgenerated[0].address = &parameter_0___cssgenerated;
42    parameters___cssgenerated[1].flags = CSS_OUT_DIR;
43    parameters___cssgenerated[1].size = (BSIZE) * sizeof(double);
44    parameters___cssgenerated[1].address = &a[i];
45    parameters___cssgenerated[2].flags = CSS_IN_DIR;
46    parameters___cssgenerated[2].size = (BSIZE) * sizeof(double);
47    parameters___cssgenerated[2].address = &b[i];
48    parameters___cssgenerated[3].flags = CSS_IN_DIR;
49    parameters___cssgenerated[3].size = (BSIZE) * sizeof(double);
50    parameters___cssgenerated[3].address = &c[i];
51    parameters___cssgenerated[4].flags = CSS_IN_SCALAR_DIR;
52    parameters___cssgenerated[4].size = sizeof(double);
53    parameters___cssgenerated[4].address = &(scalar);
54    css_addTask(triad_task_id___cssgenerated, CSS_NO_FLAG, 5, 2,
        parameters___cssgenerated);
55  }
56
57  css_finish();
```

Listing B.3: Main code of the triad benchmark after the transformations applied by the compiler.

Runtime Initialization and Finalization

The **start** pragma appears at the beginning of the program and initializes the SMPSs runtime. The compiler translates it into two function calls as follows:

```
task_registration___cssgenerated();
css_init();
```

These appear in lines 1 and 2 of the transformed code in listing B.3. The first function is a function generated during link time that registers the task adapters on the runtime as explained in sections B.1.2 and B.2.3. The second initializes the runtime and is explained in section B.1.1.

The compiler translates the **finish** pragma into a call to css_finish(). The function call appears in line 57 of listing B.3.

Task Adapters

To allow the runtime to invoke task implementations using a standard function prototype, tasks are called indirectly through adapter functions as described in section B.1.2. The compiler generates them.

Functions from external libraries can be declared as tasks. Since they may be precompiled using other compilers, the SMPSs compiler must be able to generate their adapters even if their definition is not available.

They could be generated when linking the application. However, that approach is complex since it would need to capture the task declarations during compilation, including the types they use and the parameter directionality, and then generate the adapters.

Instead, in every C *translation unit* the compiler generates a task adapter for every task that is called within it. To avoid problems due to duplication, the compiler declares them as *weak* symbols. During link time, only one instance of each adapter gets referenced. More information about weak symbols can be found in the ELF specification by the [T.I.S. Committee, 1995].

The body of the adapter code contains a call to the task implementation. Each parameter of the task corresponds to a position in the array that is passed to the adapter as a parameter. The array has type **void** $**$ and contains in each position the base address of the array, for array parameters, or a pointer to the data, if a scalar or a C struct.

For arrays and pointers, the parameter is passed as is. For scalars and C **struct**s, the array element is cast into a pointer to the correct type and then dereferenced.

Listing B.4 shows the adapters generated for the tasks declared in listing B.2.

Task Invocation

During the code conversion phase, the compiler searches task invocations in the code and substitutes them with calls to the css_addTask runtime function as described in section B.1.3. The substitution consists of a C99 *compound-statement* containing the declaration and initialization of the data structures that must be passed to css_addTask, and the actual call. Listing B.3 shows the two task call substitutions. The first task call has been replaced by the compound-statement that starts in line 4, and the second by the one in line 36.

The data structures consist of one css_parameter_t C struct per parameter. The value of its flags field is determined by the directionality clause that contains the parameter. The address field corresponds to the base address of the parameter. If the parameter is a constant, then it is substituted by a variable that is initialized with the same value. Since C99 and SMPSs support variable length arrays, the dimensions of an array parameter can be expressions containing other parameters. This is the case of the triad example.

Thus, when generating the value of the size field of the css_parameter_t C struct, the compiler generates an expression on which it replaces the parameter names that may appear by their corresponding expressions in the task call. Hence, the size

```
void __attribute__((weak)) initialize_segment_adapter__cssgenerated(void
  **parameter_data) {
   initialize_segment(
       *((long int *) parameter_data[0]),
       parameter_data[1],
       parameter_data[2],
       parameter_data[3],
       *((double *) parameter_data[4]),
       *((double *) parameter_data[5]),
       *((double *) parameter_data[6])
   );
}

void __attribute__((weak)) triad_adapter__cssgenerated(void
  **parameter_data) {
   triad(
       *((long int *) parameter_data[0]),
       parameter_data[1],
       parameter_data[2],
       parameter_data[3],
       *((double *) parameter_data[4])
   );
}
```

Listing B.4: Adapters of the tasks of the triad benchmark.

variable that appears in the array dimensions of the task declarations gets replaced by its actual value BSIZE in the size expressions in lines 14, 17, 20, 43, 46 and 49 of listing B.3.

Arrays with variable length can only be used if their length is defined at the time of the task instantiation. That is, their dimension expressions cannot depend on task invocations that may be still pending.

The css_addTask runtime function receives as first parameter an identifier that corresponds to the task that must be instantiated. Since until linking the application the compiler does not have information about other tasks used or defined in other C *translation units*, the compiler declares a variable with external linkage that has a name derived from the task name and that contains the task identifier. Hence, the calls to css_addTask in lines 31 and 54 of listing B.3 receive as first parameter such a variable.

B.2.3 Linking

The runtime identifies tasks by a sequential number starting from 0. Since the tasks of a program cannot be known in advance, identifiers and registering the task implementations can be delayed to link time.

During compilation, the compiler adds an additional table to the object files that it generates. The table contains one entry for each task known to the compiler with its name. During linkage, the linker gathers the task tables of each object and generates a new one that contains one entry for each task in the application. The

```
extern void initialize_segment_adapter___cssgenerated(void **);
extern void triad_adapter___cssgenerated(void **);

int const initialize_segment_task_id___cssgenerated = 0;
int const triad_task_id___cssgenerated = 1;

void task_registration___cssgenerated(void)
{
   css_registerTask("initialize_segment",
     &initialize_segment_adapter___cssgenerated);
   css_registerTask("triad", &triad_adapter___cssgenerated);
}
```

Listing B.5: Link time code generated for the triad benchmark.

order of the task table determines the identifier of each task and their registration order.

The linker generates additional code to define the task identifiers and also the task_registration___cssgenerated function, which registers the adapters. These codes are simple and only require the task names.

Listing B.5 shows the task identifier definitions and the definition of the adapter registration function for the triad example.

Appendix C

Additional Block-Based Benchmarks

This appendix evaluates the performance of the triad and matrix multiplication benchmarks with the block-based programming model presented in chapter 3 and extends its evaluation section that starts in page 29.

C.1 Triad

The purpose, the algorithm and its parallelization in SMP superscalar have been shown in the sections that start in page 32.

While this benchmark is embarrassingly parallel, it is also shows that it can be made parallel with SMPSs in a simple manner and despite the overhead of calculating dependencies, it achieves performance comparable to OpenMP.

Since this benchmark does not use any already programmed high performance library to solve it, the performance we achieve is heavily dependent on the compiler. The binaries of the triad benchmark that the ICC compiler produces are more than 4 times faster than those produced by GCC. For this reason we have used ICC in the SMPSs version and in the alternative implementations.

Parallelization with OpenMP

To compare the performance of the algorithm in SMPSs to other programming models we have made an equivalent implementation in OpenMP with tasks. Listing C.1 shows the main code of the OpenMP implementation. Since OpenMP allows to create tasks in parallel, it uses a parallel loop in line 3 to spawn the tasks in line 5. The `triad` function is identical to the SMPSs task with the task declaration directive removed.

Compared to the SMPSs version, OpenMP is slightly simpler to program since it does not require specifying the task parameter directionalities and does not require outlining the task code to a separate function.

```
for (int rep = 0; rep < NTIMES; rep++) {
    START_TIME();
    #pragma omp parallel for
    for (long i = 0; i < N; i += BS)
        #pragma omp task untied
        triad(BS, &a[i], &b[i], &c[i], alpha);
    STOP_TIME();
    total_time[rep] = GET_TIME();
}
find_best_bandwidth(total_time, NTIMES);
```

Listing C.1: Main triad code in OpenMP with tasks.

Determining the Task Size

One of the key factors that determine the performance of a parallel application is the granularity of the parallel unit. In SMPSs, the parallel unit is always the task, and in this case its granularity is determined by the subvector size. Since the OpenMP version also uses tasks and the same kind of parallelization, the procedure to determine the task size is identical for both cases. In the following paragraphs we discuss the impact of these issues on the SMPSs version.

The Triad problem is a linear and embarrassingly parallel problem, hence, the best subvector size is expected to be the one that assigns only one task per thread. That is, N/P with P equal to the number of threads. While in this case that is a sensible option, in the implementation of the rest of the algorithms, task granularity has deeper correlation to several performance factors. For consistency and completeness we have measured the performance with 1, 2, 4, and 8 tasks per thread.

Figure C.1 shows the memory bandwidth achieved with those configurations. Each panel corresponds to a set of measurements with a fixed number of cores. The vertical axis determines the vector size (N) in megaelements, and the horizontal axis is the subvector size BS, also in megaelements. The figure shows the mean performance of 30 executions of each configuration in gigabytes per second per node. The task granularity of this application does not affect performance substantially as long as there is enough work for all threads.

Notice that since this chapter does not cover NUMA aspects, as we increase the number of cores, performance per node decreases. Figure C.2 shows the mean memory bandwidth that the application consumes. This metric is derived from the level 3 cache miss frequency that we obtained when running the application with a version of the runtime that traces the execution and measures hardware counter metrics. The figure shows that the effective memory bandwidth grows sublinearly to the number of cores, which explains the bad scalability of memory bandwidth per core.

Table C.1 summarizes the performance metrics of the subvector size that achieves the best performance for each number of cores and problem size. In addition to the problem size, the decomposition parameters and the performance, the table also shows information gathered on executions with tracing turned on. The relevant columns are the instructions per cycle (IPC), floating point operations per cycle (FPC), main thread overhead, parallel efficiency, and idle time ratio. The IPC column measures the mean effective IPC, and has been calculated as the total number of

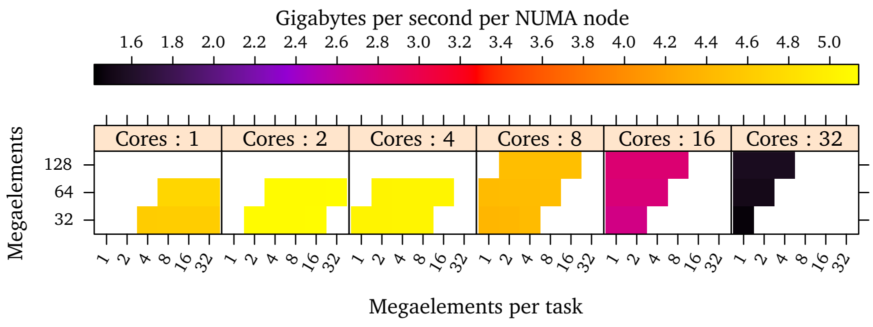


Figure C.1: Performance of the SMPSs Triad implementation with several vector and subvector sizes.

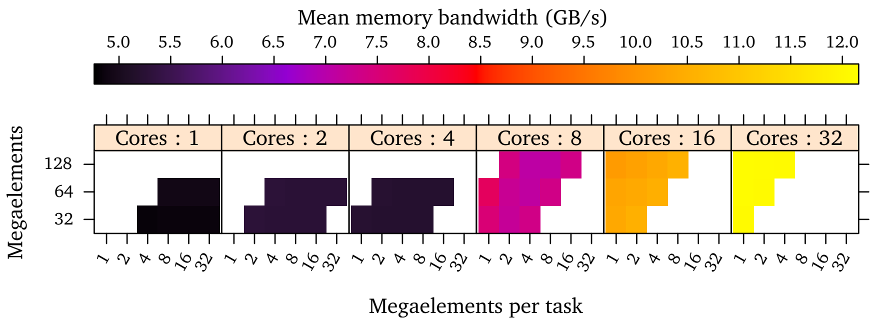


Figure C.2: Average memory bandwidth consumed in the SMPSs Triad implementation with several vector and subvector sizes.

Cores	N[a]	BS[b]	Tasks	GB/s[c]	IPC[d]	FPC[e]	Ovh.[f] (%)	Eff.[g] (%)	Idle (%)	Tr.O.[h] (%)
1	32	32	10	4.73	1.51	0.25	0.04	99	0	-1
1	64	32	20	4.82	1.53	0.26	0.04	99	0	-1
2	32	16	20	5.23	0.83	0.14	0.06	99	0	0
2	64	32	20	5.23	0.83	0.14	0.03	99	0	0
4	32	8	40	5.15	0.41	0.07	0.08	99	0	0
4	64	16	40	5.15	0.41	0.07	0.04	99	0	0
8	32	4	80	9.06	0.29	0.05	0.25	99	0	0
8	64	8	80	9.13	0.29	0.05	0.13	99	0	0
8	128	16	80	9.17	0.29	0.05	0.07	99	0	0
16	32	2	160	11.20	0.21	0.03	0.68	94	5	0
16	64	4	160	11.45	0.21	0.03	0.35	95	4	0
16	128	8	160	11.58	0.21	0.03	0.18	95	4	0
32	32	1	320	11.91	0.12	0.02	1.76	95	3	0
32	64	1	640	12.43	0.12	0.02	1.84	96	3	0
32	128	1	1280	12.73	0.12	0.02	1.94	97	2	0

[a] Megaelements per array..
[b] Submatrix side size.
[c] Gigabytes per second.
[d] Mean instructions per cycle while running tasks.
[e] Mean floating point operations per cycle while running tasks.
[f] Time that the main thread spends generating tasks and idle.
[g] Mean time that threads spend running tasks.
[h] Increment of the execution time when enabling tracing.

Table C.1: Best submatrix side sizes for the SMPSs Triad implementation and their performance characteristics.

task instructions executed divided by the total number of cycles executing tasks in each thread. The FPC metric has been calculated in a similar way and measures the floating point operation rate. The main thread overhead column measures the time that the main thread spends in the runtime. Parallel efficiency measures the mean time that threads spend executing tasks. The idle time column shows the mean time threads spend idling. The tracing overhead has been calculated as the difference in performance, in this case in GB/s, between the runs with tracing enabled and the runs without tracing. In all cases it is within 2% which demonstrates that the tracing mechanism does not introduce a have impact on the performance of these executions.

Scheduling

Since this benchmark is embarrassingly parallel and does not reuse data, the scheduling policy does not affect the performance that we achieve. The choice of the scheduling policy has not yielded statistically significant differences between the executions when the rest of the parameters remained unchanged.

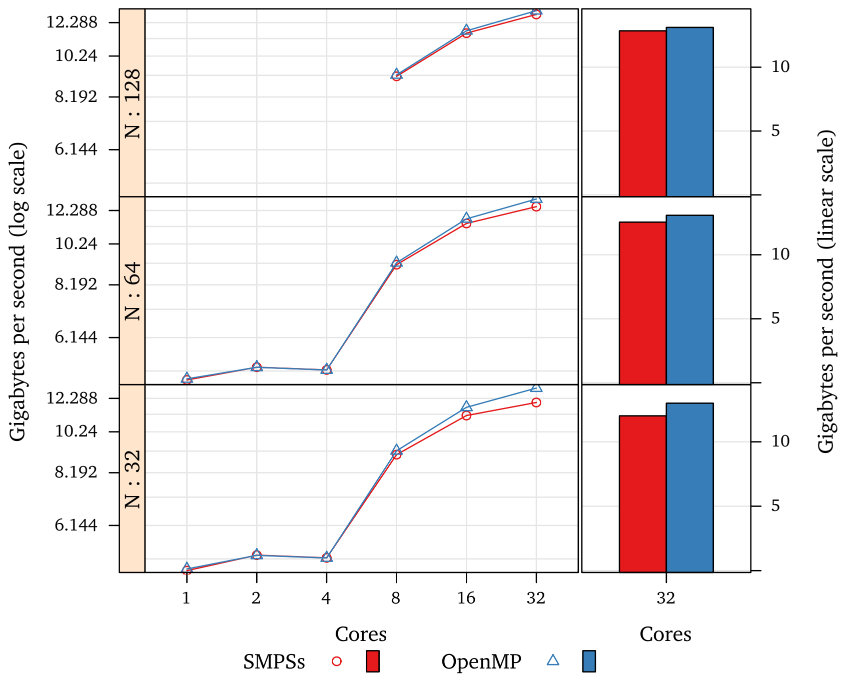


Figure C.3: Strong scalability of each parallel implementation of Triad with several vector sizes and performance with 32 cores.

Performance of the Implementations

The OpenMP version of the algorithm has also been executed with the same problem sizes and granularity and from these we have selected the best performing cases of each implementation. Figure C.3 shows the strong scalability of both implementations on three problem sizes. Each panel corresponds to a problem size N measured in gigaelements. The horizontal axis indicates the number of cores and the vertical axis the number of gigabytes per second with a logarithmic scale. The right side of the figure shows the performance of the configurations with 32 cores using a linear scale.

Table C.2 summarizes the mean values of the main performance metrics of each Triad implementation. Notice that the task IPC dominates the performance difference between the OpenMP and the SMPSs versions. This might be related to how each schedules its tasks.

Cores	N[a]	GB/s[b]	GB/s[b]	IPC[c]	IPC[c]	Eff.[d] (%)	Eff.[d] (%)
1	32	4.80	4.85	1.51	1.52	99	99
1	64	4.88	4.91	1.53	1.54	99	99
2	32	5.23	5.22	0.83	0.83	99	99
2	64	5.23	5.22	0.83	0.83	99	99
4	32	5.15	5.15	0.41	0.41	99	99
4	64	5.15	5.15	0.41	0.41	99	99
8	32	9.04	9.23	0.29	0.37	99	99
8	64	9.14	9.24	0.29	0.37	99	99
8	128	9.17	9.24	0.29	0.37	99	99
16	32	11.19	11.70	0.21	0.23	94	98
16	64	11.45	11.74	0.21	0.23	95	98
16	128	11.60	11.75	0.21	0.23	95	99
32	32	12.02	13.00	0.12	0.13	95	97
32	64	12.55	13.08	0.12	0.13	96	98
32	128	12.85	13.12	0.12	0.13	97	98

Implementation: 🟥 SMPSs 🟦 OpenMP

[a] Megaelements per array.
[b] Gigabytes per second.
[c] Mean instructions per cycle while running tasks.
[d] Mean time that threads spend running tasks.

Table C.2: Performance summary of the Triad implementations.

C.2 Matrix multiplication

The matrix multiplication algorithm and its parallelization have already been explained in the sections that start in page 33. In this appendix we evaluate its performance.

Parallelization with OpenMP

To compare the performance of the algorithm in SMPSs to other programming environments, we have made a parallel implementation using OpenMP tasks. The SMPSs and the OpenMP versions use MKL in sequential mode for the submatrix operations, which allows us to compare them in terms of the programming model without regard to the underlying kernel implementation.

The OpenMP implementation is shown in listing C.2. Notice that the matrices are stored continuously in memory, as opposed to the SMPSs version, which stores them by blocks. Since OpenMP does not allow task dependencies, we have grouped each whole task chain of the SMPSs version into a single task. That is, we have used the decomposition by panels from equation 3.3 instead of the decomposition by blocks from equation 3.2. Another difference is that the OpenMP version generates the tasks in parallel.

This version has been compiled with GCC. However, it has been linked with the Intel OpenMP runtime, which is required by the MKL library even in sequential mode. Nevertheless, the Intel OpenMP runtime also implements the GCC internal OpenMP runtime API.

Parallel Library Implementation

To complete the comparison, we have included an implementation of the algorithm using an already parallel version of the algorithm. The MKL parallel version is implemented as a single call to the dgemm function which is implemented in parallel within the MKL library.

Determining the Submatrix Dimensions

The values of the submatrix dimensions MBS, NBS and KBS determine the number of tasks of the problem decomposition and the computational weight of each task. Big submatrices generate few tasks with big computational weight, while small submatrices generate many tasks with small computational weight.

The performance of the algorithm depends on the ability to keep the cores executing tasks and the performance of the tasks themselves. Both aspects depend on their granularity. A big number of tasks may provide more parallelism than a smaller one, however, the performance of the tasks may be lower due to less potential exploitation of data locality. Moreover, a big number of low computational weight tasks may turn task creation into a bottleneck.

Given the matrix dimensions M, N and K and their respective submatrix dimensions MBS, NBS and KBS that divide them into m, n and k segments, the algorithm in listing 3.6 generates $m \times n$ chains of dependent tasks, each with k tasks that calculate a submatrix of C.

While for a given problem size, it may be simple to find analytically the biggest submatrix size that generates enough task chains to keep all cores busy, the per-

```
void tiled_dgemm(
   int MBS, int NBS, int KBS,
   int M, int N, int K,
   double ALPHA, double const *A, double const *B,
   double BETA, double *C)
{
   #pragma omp parallel for
   for (int i=0; i<M; i += MBS)
      for (int j=0; j<N; j+= NBS)
         #pragma omp task untied
         for (int k=0; k<K; k += KBS) {
            if (k==0) {
               if (BETA == 0.0)
                  dgemm_nobeta_tile(MBS, NBS, KBS, M, N, K, ALPHA,
                    &A[k+i*K], &B[j+k*N], &C[j+i*N]);
               else
                  dgemm_tile(MBS, NBS, KBS, M, N, K, ALPHA,
                    &A[k+i*K], &B[j+k*N], BETA, &C[j+i*N]);
            } else {
               dgemm_tile(MBS, NBS, KBS, M, N, K, ALPHA, &A[k+i*K],
                 &B[j+k*N], 1.0, &C[j+i*N]);
            }
         }
}

void dgemm_nobeta_tile(int MBS, int NBS, int KBS, int M, int N, int K,
   double ALPHA, double const *A, double const *B, double *C)
{
   static const double dzero = 0.0;
   dgemm_("N", "N", &NBS, &MBS, &KBS, &ALPHA, B, &N, A, &K,
     &dzero, C, &N);
}

void dgemm_tile(int MBS, int NBS, int KBS, int M, int N, int K,
   double ALPHA, double const *A, double const *B,
   double BETA, double *C)
{
   dgemm_("N", "N", &NBS, &MBS, &KBS, &ALPHA, B, &N, A, &K,
     &BETA, C, &N);
}
```

Listing C.2: Double precision generalized matrix-matrix multiplication in OpenMP.

formance of the MKL code used in the tasks does not increase consistently with submatrix size. For this reason we have measured the performance with several submatrix sizes. Moreover, to simplify the evaluation, the problem has been reduced to square matrices and square submatrices. That is, $M = N = K$, and MBS=NBS=KBS.

Figure C.4 shows the floating point performance of our implementation. Each panel corresponds to a set of measurements with a fixed number of cores. The vertical axis determines the matrix side size N, and the horizontal axis is the submatrix side size NBS. The figure shows the mean performance of 30 executions of each configuration with respect to the theoretical hardware peak measured in floating point operations per second.

Most configurations perform at higher than 85% of the peak floating point performance. However, with 2, 8 and 32 cores there are configurations in the diagonal with lower performance. To highlight the reasons, we have measured all configurations with tracing. Figure C.5 shows the average idle time per thread. Notice that the diagonal configurations with 2, 8 and 32 cores spend about 50% of their time idling due to too few task chains. This explains the drop in floating point performance.

Figure C.4 also shows a drop in floating point performance for the experiments with 16 and 32 cores when using submatrices of 128 elements per side. While these configurations have the greatest amount of parallelism, they suffer from high management overhead due to the low amount of computational cost per task and low task performance due to low data reuse. Figure C.6 shows the overhead of task management in the main thread. Notice that the overhead with submatrices of 128 by 128 elements is always higher than the other configurations, but it grows with the number of cores and starts to be noticeable with 8 cores. The mean task floating point operations per cycle are shown in figure C.7. The executions with 128 by 128 element submatrices have also lower task floating point performance than the rest and this difference also grows with the number of cores.

Figure C.7 also shows that the floating point performance of the tasks does not grow smoothly with submatrix size. Instead the best performing submatrix side size is consistently 512 for all problem sizes and number of cores.

Since the tests of this appendix and the ones of its corresponding main chapter have been performed with the data interleaved between the NUMA nodes and without NUMA awareness, task performance decreases with the number of threads. This effect is greatest for small submatrix sizes, since they have less temporal locality as shown by their higher cache miss ratio in figure C.8.

To summarize the performance of this application we have selected from each problem size and number of cores the submatrix size with greatest mean performance measured as the total floating point operations per cycle without tracing. Table C.3 summarizes the performance of those configurations. Notice that even with one thread, some configurations that generate more than one task are faster than the ones that generate only one. On the other hand, the smallest problem size with 32 cores performs best with a submatrix size that only produces tasks that only occupy half of the cores.

Scheduling

To verify the effectiveness of the scheduling policy on the matrix multiplication, we compare the performance with the scheduling policy that we presented against the three other policies described in page 30. Since task granularity can affect the performance characteristics of the execution, in the following comparisons we have

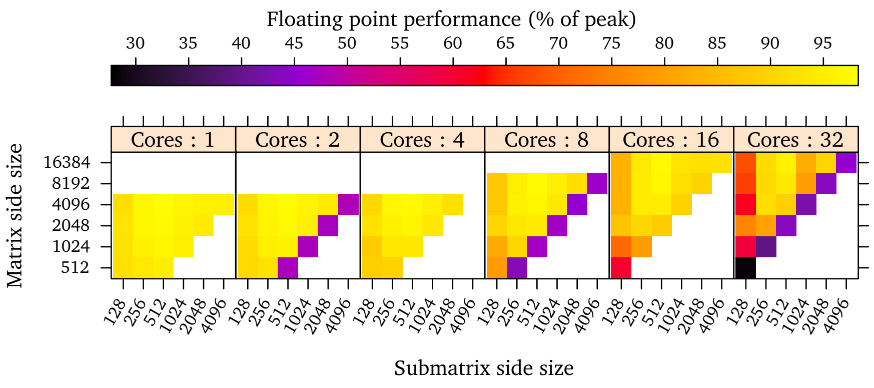


Figure C.4: Performance of the SMPSs matrix multiplication implementation with several matrix and blocking sizes.

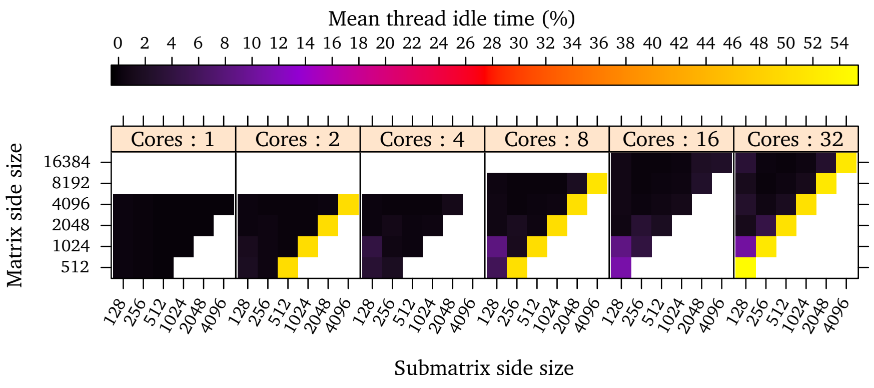


Figure C.5: Average time taken by each thread idling in the SMPSs matrix multiplication implementation with several matrix and blocking sizes.

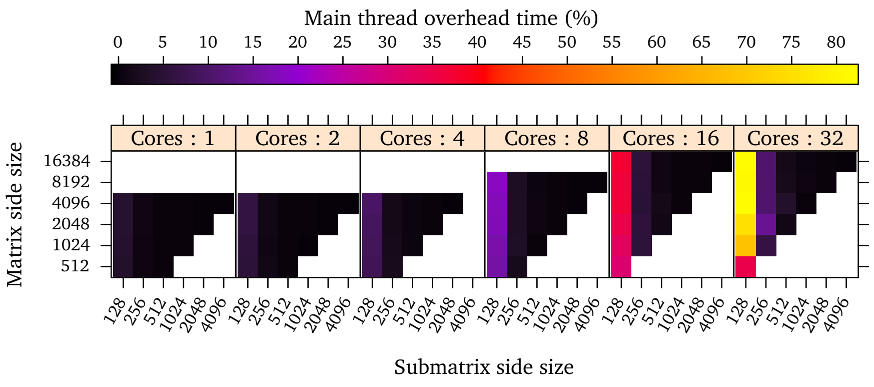


Figure C.6: Percentage of time time that the main thread spends managing tasks when running the SMPSs matrix multiplication implementation with several matrix and blocking sizes.

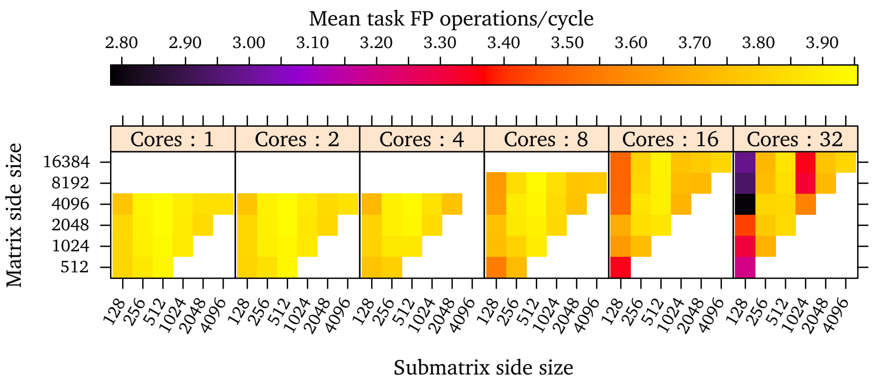


Figure C.7: Mean floating point operations per cycle while running the tasks of the SMPSs matrix multiplication implementation with several matrix and blocking sizes.

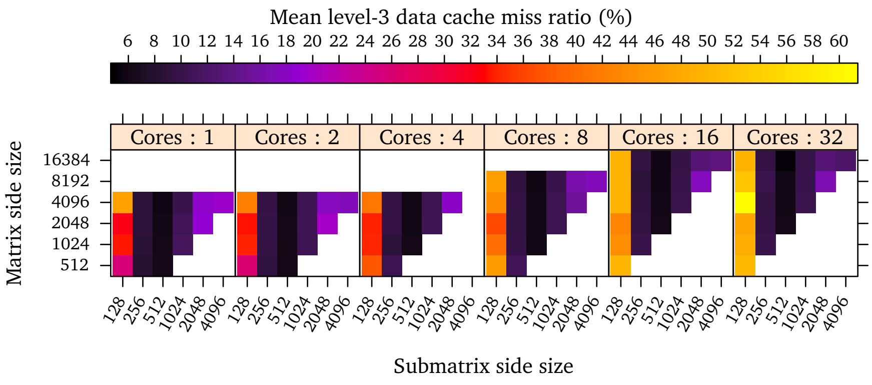


Figure C.8: Average level-3 data cache miss ratio while running the tasks of the SMPSs matrix multiplication implementation with several matrix and blocking sizes.

Cores	N[a]	BS[b]	Tasks	GF[c]	IPC[d]	FPC[e]	Ovh.[f] (%)	Eff.[g] (%)	Idle (%)	Tr.O.[h] (%)
1	512	512	1	6.1	5.93	3.93	0.17	99	0	0
1	1024	512	8	6.2	5.93	3.94	0.16	99	0	0
1	2048	512	64	6.2	5.94	3.94	0.15	99	0	0
1	4096	512	512	6.2	5.94	3.94	0.15	99	0	0
2	512	256	8	11.9	5.84	3.86	0.91	98	0	0
2	1024	512	8	12.3	5.92	3.93	0.19	99	0	0
2	2048	512	64	12.4	5.94	3.94	0.18	99	0	0
2	4096	512	512	12.5	5.94	3.94	0.19	99	0	0
4	512	256	8	22.9	5.75	3.80	1.75	97	1	1
4	1024	256	64	24.0	5.87	3.88	1.49	98	0	0
4	2048	512	64	24.7	5.92	3.93	0.28	99	0	0
4	4096	512	512	24.9	5.93	3.94	0.26	99	0	0
8	512	128	64	40.5	5.42	3.55	15.92	89	5	4
8	1024	256	64	46.1	5.76	3.81	2.96	97	1	0
8	2048	512	64	48.2	5.89	3.91	0.53	99	0	0
8	4096	512	512	49.4	5.91	3.93	0.53	99	0	0
8	8192	512	4096	49.8	5.93	3.93	0.42	99	0	0
16	512	128	64	61.9	5.12	3.35	31.12	84	11	0
16	1024	256	64	81.1	5.66	3.74	5.37	95	3	0
16	2048	256	512	92.5	5.82	3.84	5.44	96	2	1
16	4096	512	512	96.9	5.88	3.90	1.29	99	0	0
16	8192	512	4096	98.7	5.89	3.91	0.78	99	0	0
16	16384	512	32768	99.3	5.89	3.91	0.75	99	0	0
32	512	128	64	58.1	4.88	3.20	34.82	42	54	1
32	1024	128	512	121.8	5.05	3.31	68.27	83	10	0
32	2048	256	512	164.4	5.72	3.78	14.18	94	4	0
32	4096	256	4096	187.4	5.77	3.82	10.47	98	0	0
32	8192	512	4096	192.6	5.80	3.85	1.58	99	0	0
32	16384	512	32768	195.8	5.83	3.87	1.36	99	0	0

[a] Matrix side size..
[b] Submatrix side size.
[c] Gigaflops per second.
[d] Mean instructions per cycle while running tasks.
[e] Mean floating point operations per cycle while running tasks.
[f] Time that the main thread spends generating tasks and idle.
[g] Mean time that threads spend running tasks.
[h] Increment of the execution time when enabling tracing.

Table C.3: Best submatrix side sizes for the SMPSs matrix multiplication implementation and their performance characteristics.

selected the best granularity for the default scheduler in each case and used it for all the schedulers.

Figure C.9 shows the mean floating point performance under several strong scalability scenarios with the four schedulers. Horizontal lines on the left panel correspond to the theoretical peak for 1, 2, 4, 8, and 16 cores respectively. The right side of the figure shows the mean performance of the configurations with 32 cores with a linear scale. While most scheduling policies have similar scalability and performance, only the random policy performs consistently worse, although not by far. Since the task implementation uses the cache very efficiently, the differences in task performance are small.

Figure C.10 shows the reduction of the level 3 data cache miss ratio when replacing the random scheduling policy by the default policy. For block sizes bigger than 256 elements per side, the difference is very small, since the tasks can take better advantage of spatial locality. For smaller block sizes, temporal locality has more impact on the cache miss ratio. In those cases, the default policy, which attempts to pin task chains to threads, reduces the miss ratio by up to 25%.

The low performance with block side sizes of 128 is due by two circumstances. First, under those configurations, the task creation rate is slow compared to the task execution time, as shown previously in figure C.6. And second, the code creates the task graph in breadth order. That is, when the bottleneck is the task creation, the code favors parallelism over data reuse. As we duplicate the problem size, the number of tasks between one and the following in its chain is quadrupled. Therefore, threads cannot follow the dependency chains since by the time they finish a task, the following in the chain has not already been created.

Table C.4 summarizes the mean values of the main performance metrics of the matrix multiplication with each scheduler. They show that the lower performance of the random scheduler is due to lower parallel efficiency. In this case, since the task granularity is coarse, the amount of task chains is small, and thus under the random scheduler, threads have to access several queues before finding one that contains tasks.

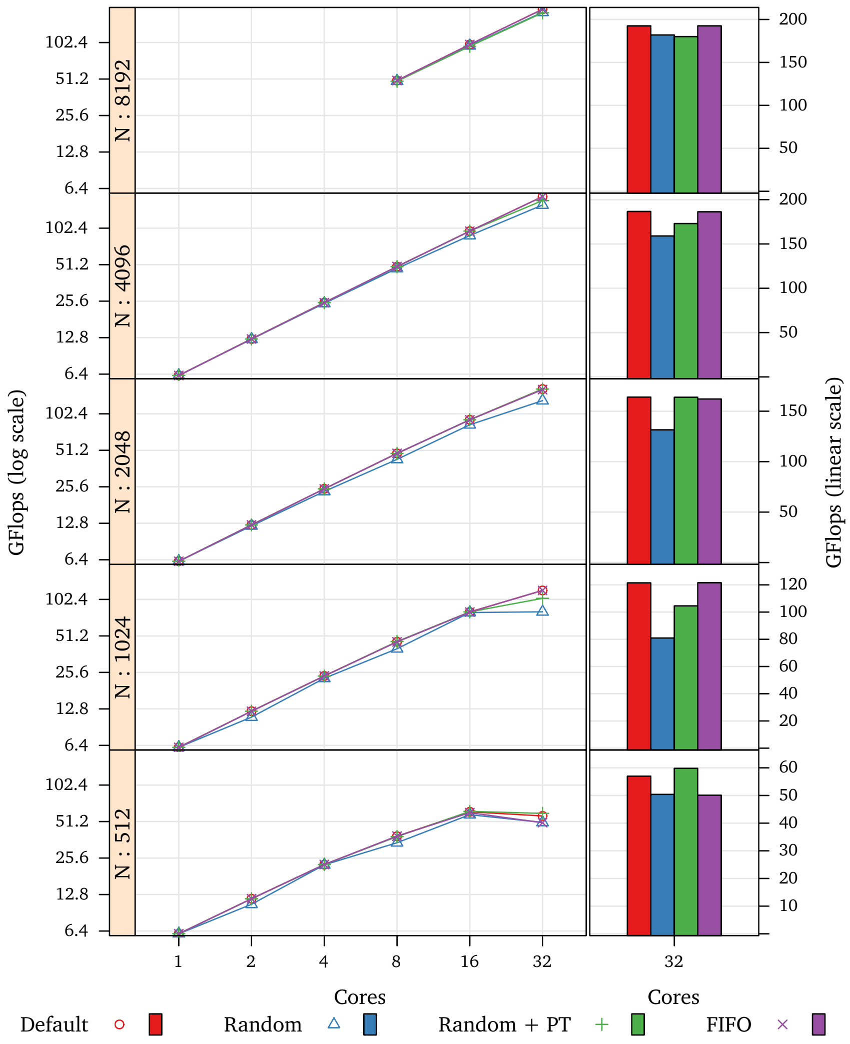


Figure C.9: Strong scalability of the SMPSs matrix multiplication implementation with several matrix sizes under each scheduling policy and performance with 32 cores.

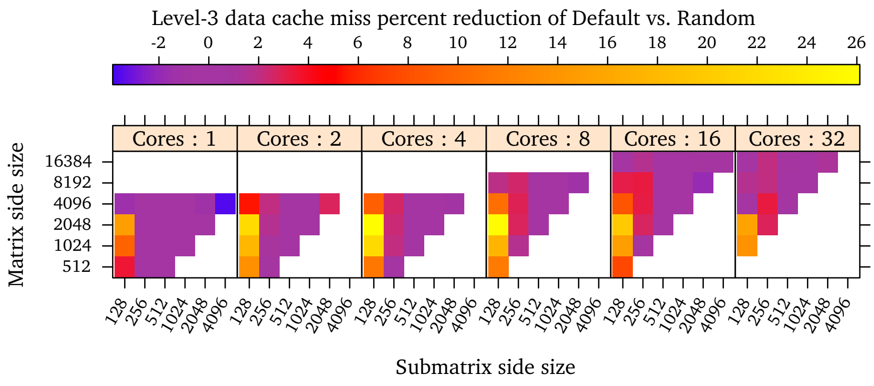


Figure C.10: Data cache miss reduction of the matrix multiplication when changing the scheduling from the random policy to the default policy.

Cores	N[a]	GF[b]	GF[b]	GF[b]	GF[b]	FPC[c]	FPC[c]	FPC[c]	FPC[c]	Eff.[d] (%)	Eff.[d] (%)	Eff.[d] (%)	Eff.[d] (%)
1	512	6.1	6.1	6.1	6.1	3.9	3.9	3.9	3.9	99	99	99	99
1	1024	6.2	6.2	6.2	6.2	3.9	3.9	3.9	3.9	99	99	99	99
1	2048	6.2	6.2	6.2	6.2	3.9	3.9	3.9	3.9	99	99	99	99
1	4096	6.2	6.2	6.2	6.2	3.9	3.9	3.9	3.9	99	99	99	99
2	512	11	10	11	11	3.9	3.9	3.9	3.9	98	88	98	98
2	1024	12	10	12	12	3.9	3.9	3.9	3.9	99	88	99	99
2	2048	12	12	12	12	3.9	3.9	3.9	3.9	99	97	99	99
2	4096	12	12	12	12	3.9	3.9	3.9	3.9	99	99	99	99
4	512	22	22	22	22	3.8	3.8	3.8	3.8	97	97	97	97
4	1024	23	22	23	23	3.9	3.9	3.9	3.9	98	94	98	98
4	2048	24	23	24	24	3.9	3.9	3.9	3.9	99	94	99	99
4	4096	24	24	24	24	3.9	3.9	3.9	3.9	99	98	99	99
8	512	38	34	38	38	3.5	3.4	3.6	3.6	89	82	88	89
8	1024	45	40	46	45	3.8	3.8	3.8	3.8	97	86	97	97
8	2048	48	43	48	48	3.9	3.9	3.9	3.9	99	88	99	99
8	4096	49	47	49	49	3.9	3.9	3.9	3.9	99	96	99	99
8	8192	49	49	49	49	3.9	3.9	3.9	3.9	99	98	98	99
16	512	61	58	62	60	3.4	3.2	3.4	3.3	84	80	81	83
16	1024	80	79	80	80	3.7	3.7	3.7	3.7	95	93	94	95
16	2048	91	83	91	91	3.8	3.7	3.8	3.8	96	90	96	95
16	4096	96	89	96	96	3.9	3.9	3.9	3.9	99	91	99	99
16	8192	98	96	95	98	3.9	3.9	3.9	3.9	99	97	96	99
16	16384	99	98	98	99	3.9	3.9	3.9	3.9	99	99	99	99
32	512	56	50	59	50	3.2	2.7	3.3	2.7	42	40	41	42
32	1024	121	80	104	121	3.3	2.3	3.1	3.3	83	77	73	83
32	2048	163	131	163	162	3.8	3.4	3.8	3.8	94	82	94	94
32	4096	186	159	173	186	3.8	3.4	3.8	3.8	98	94	92	98
32	8192	192	181	180	192	3.8	3.8	3.8	3.8	99	94	92	99
32	16384	195	192	191	195	3.9	3.8	3.8	3.9	99	98	98	99

Scheduler: Default Random Random + PT FIFO

[a] Matrix side size.
[b] Gigaflops per second.
[c] Mean floating point operations per cycle while running tasks.
[d] Mean time that threads spend running tasks.

Table C.4: Performance summary of the scheduler on the SMPSs matrix multiplication.

Performance of the Implementations

For both the SMPSs and the OpenMP implementations we have selected the best performing submatrix sizes for each case. However, since the parallel MKL version has those parameters automatically set up by the MKL library itself, we have left them as is.

Figure C.11 shows the mean floating point performance of the three implementations under several strong scalability scenarios. Horizontal lines on the left panels correspond to the theoretical peak for 1, 2, 4, 8, and 16 cores respectively. The right side of the figure shows the mean performance of the configurations with 32 cores with a linear scale.

In all cases the SMPSs implementation has the best strong scalability. On one hand the SMPSs version has the additional overhead of finding the dependencies, which the other two versions do not, and generates the tasks sequentially, as opposed to the OpenMP version, that generates fewer and in parallel. On the other hand, the SMPSs version operates over blocked matrices, which have better spatial locality and thus obtain better IPC. The scalability of the OpenMP and the MKL versions is similar, however then MKL version exhibits slightly better performance.

Figure C.12 shows the average effective IPC, that is, the IPC while not idling nor executing the SMPSs/OpenMP runtime. Each row of panels corresponds to a different matrix size, and their vertical axis indicates their performance relative to the theoretical hardware peak. Each column of panels corresponds to a number of cores, and the horizontal axis groups the measurements of each implementation. The panels contain violin plots. They are an estimation of the density of the metric, in this case the mean effective IPC of the execution. Wide sections correspond to Y values with high frequency, and narrow sections correspond to IPC values with low frequency. Horizontal lines indicate that the given configuration has very low variability.

The OpenMP and the MKL versions have generally lower effective IPC than the SMPSs version, due to the different memory layout. The OpenMP version also uses smaller block sizes in some cases, since they produce better balance. This is reflected in table C.5, that summarizes the mean values of the main performance metrics of each implementation.

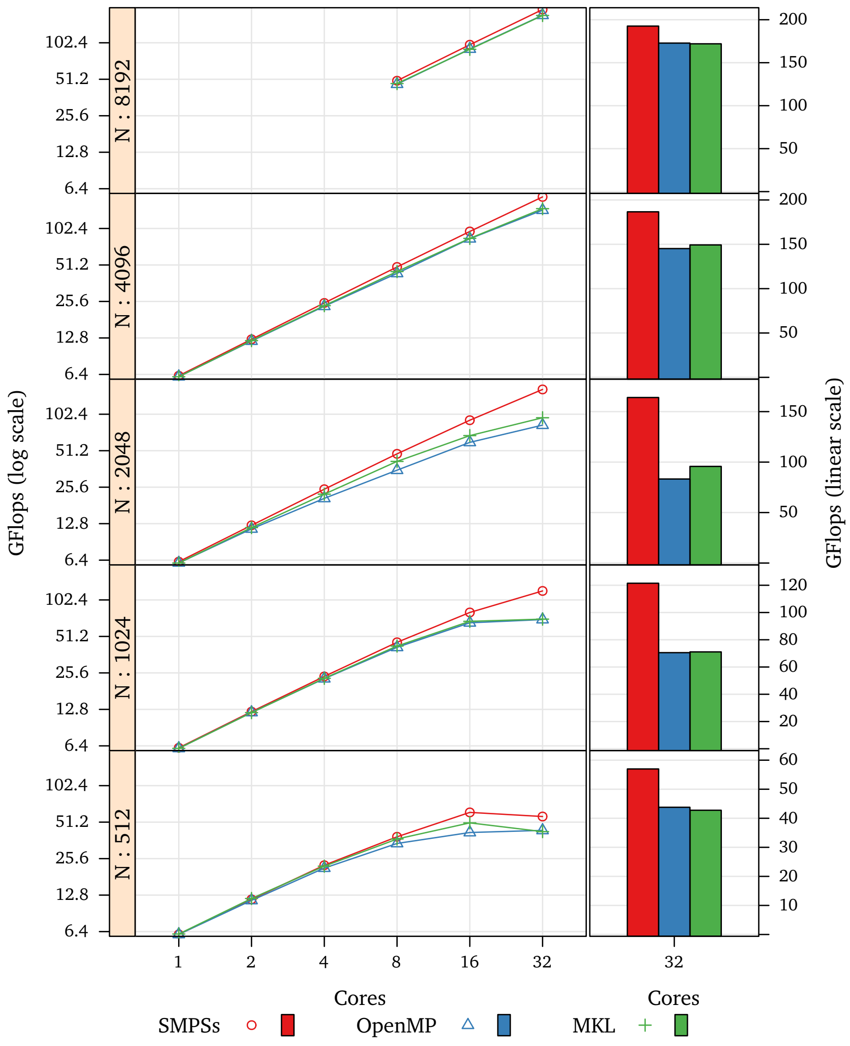


Figure C.11: Strong scalability of each parallel implementation of the matrix multiplication with several matrix sizes and performance with 32 cores.

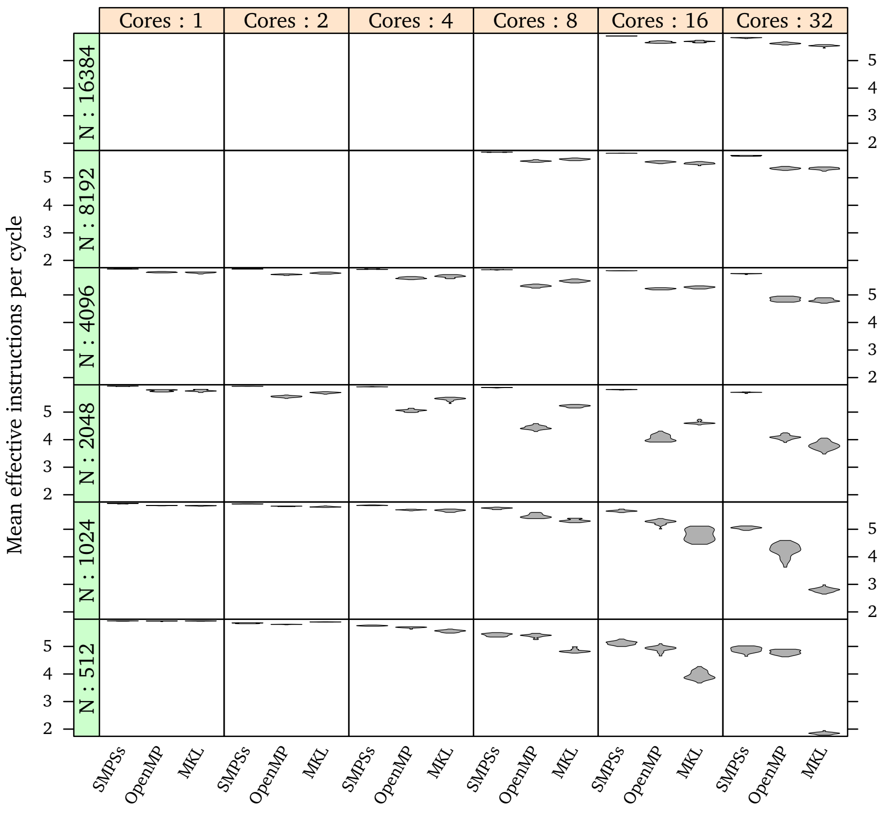


Figure C.12: Average instructions per cycle of each implementation of the matrix multiplication while running effective work with several matrix sizes.

Cores	N[a]	BS[b]	BS[b]	GF[c]	GF[c]	GF[c]	FPC[d]	FPC[d]	FPC[d]	Eff.[e] (%)	Eff.[e] (%)	Eff.[e] (%)
1	512	512	512	6.1	6.1	6.1	3.9	3.9	3.9	99	99	99
1	1024	512	1024	6.2	6.1	6.1	3.9	3.9	3.9	99	99	99
1	2048	512	2048	6.2	6.1	6.1	3.9	3.8	3.8	99	99	99
1	4096	512	4096	6.2	6.1	6.1	3.9	3.9	3.9	99	99	99
2	512	256	256	11	11	12	3.9	3.8	3.9	98	97	99
2	1024	512	512	12	12	12	3.9	3.9	3.9	99	99	99
2	2048	512	1024	12	11	11	3.9	3.7	3.8	99	99	99
2	4096	512	2048	12	12	12	3.9	3.8	3.9	99	99	99
4	512	256	128	22	21	22	3.8	3.7	3.7	97	94	98
4	1024	256	256	23	22	23	3.9	3.8	3.8	98	98	99
4	2048	512	512	24	20	22	3.9	3.4	3.6	99	99	99
4	4096	512	2048	24	23	23	3.9	3.7	3.8	99	99	99
8	512	128	128	38	34	37	3.5	3.5	3.2	89	90	95
8	1024	256	128	45	41	42	3.8	3.6	3.5	97	95	99
8	2048	512	128	48	35	41	3.9	2.9	3.5	99	98	98
8	4096	512	1024	49	43	45	3.9	3.5	3.7	99	98	98
8	8192	512	2048	49	46	47	3.9	3.7	3.8	99	98	99
16	512	128	128	61	42	50	3.4	3.2	2.6	84	85	86
16	1024	256	128	80	66	68	3.7	3.4	3.2	95	91	94
16	2048	256	128	91	59	68	3.8	2.7	3.1	96	96	97
16	4096	512	1024	96	84	84	3.9	3.5	3.5	99	97	98
16	8192	512	2048	98	90	90	3.9	3.7	3.7	99	97	98
16	16384	512	4096	99	93	93	3.9	3.8	3.8	99	97	98
32	512	128	128	56	43	42	3.2	3.1	1.2	42	41	84
32	1024	128	128	121	70	71	3.3	2.8	1.9	83	90	91
32	2048	256	256	163	83	95	3.8	2.7	2.5	94	93	97
32	4096	256	512	186	145	149	3.8	3.2	3.2	98	95	98
32	8192	512	1024	192	172	171	3.8	3.6	3.5	99	97	97
32	16384	512	2048	195	182	180	3.9	3.7	3.7	99	96	97

Implementation: SMPSs OpenMP MKL

[a] Matrix side size.
[b] Submatrix side size
[c] Gigaflops per second.
[d] Mean floating point operations per cycle while running tasks.
[e] Mean time that threads spend running tasks.

Table C.5: Performance summary of the matrix multiplication implementations.

Appendix D

Compiler and Runtime Interface Extensions for Region Support

To support regions, language-level regions must be passed to the runtime where the region-unaware interface had addresses. In addition, the region-aware interface can have more than one (region) access per parameter. These changes affect both the internal runtime interface and the compiler. This appendix describes the extended runtime interface and the changes to the compiler.

D.1 Extensions to the Interface between User Code and Runtime

Array regions provide means to specify subsets of an array. As such, they only affect task instantiation, since they enhance the description of the data accesses, and partial synchronizations, since they allow to synchronize on more precise sets of data.

Thus, initialization and finalization, which was presented in section B.1.1; task implementation registration, which appears in section B.1.2; and full barriers, shown in section B.1.4, remain unchanged under the region-aware runtime.

D.1.1 Task Instantiation

Task calls are notified to the runtime by calling the following function:

```
void css_addTask(unsigned int functionId, unsigned int flags, unsigned int
    parameterCount, css_parameter_t const *parameters);
```

It receives a task identifier that has been determined by the corresponding adapter registration order, flags, the number of parameters and an array with the actual parameter description. The only flag currently available is the high priority flag, which corresponds to a call to a task that has the **highpriority** clause.

While the task instantiation function prototype is identical to the function used in region-unaware runtime, the parameter descriptor is substantially different in the region-aware runtime. The region-unaware parameter descriptor is the following:

```
typedef struct {
    long flags;
    size_t size;
    void *address;
} css_parameter_t;
```

It uses the base address and size to determine the parameter address and the accessed range, and the flags field to determine the directionality and kind of access.

The region-aware parameter descriptor is the following:

```
typedef struct {
    void *address;
    short region_count;
    css_parameter_region_t const *regions;
} css_parameter_t;
```

It also has a field that indicates the base address. However, the accessed ranges and their directionalities are described by a set of region descriptors. The parameter descriptor contains a field that indicates the number of regions and a pointer to the beginning of the region descriptor array.

The region descriptor has the following contents:

```
typedef struct {
    short flags;
    short dimension_count;
    css_parameter_dimension_t const *dimensions;
} css_parameter_region_t;
```

The flags field is identical to the one previously used in the parameter descriptor of the region-unaware runtime. It indicates the directionality of the region and if it is the target of a reduction. The possible values and semantics are the following:

Flag Value	Semantics
1	An input region
2	An output region
1+2	An inout region
1+4	An input region that is a constant
1+2+8	An inout region that is the target of a reduction

The dimension_count and dimensions fields are the number of dimensions, and a pointer to the beginning of an array containing the dimension descriptors starting from the least significant dimension. Each dimension if described as follows:

```
typedef struct {
    size_t size;
    size_t lower_bound;
    size_t accessed_length;
} css_parameter_dimension_t;
```

The fields of the first dimension descriptor are expressed in terms of bytes. Further dimensions are expressed naturally as times the whole previous dimension. The size field indicates the whole size of the dimension. The lower_bound and accessed_length fields represent the first element of the accessed range and the number of elements.

D.1.2 Partial Synchronization

Partial synchronization in the region-unaware runtime uses the following function:

```
void css_waitOn(unsigned int variableCount, void **addresses);
```

Its first parameter is the number of variables in the **on** clause. The second parameter is an array with one entry for each object over which the user code must wait. Each entry is a pointer that corresponds to the base address in case of a scalar or a **struct**, and a pointer to the first element of the block in case of an array.

Since the region-aware runtime accepts partial synchronization on regions, the second parameter has been changed to accept region descriptors. The new function prototype is the following:

```
void css_waitOn(unsigned int variableCount, css_parameter_region_t const
   *variables);
```

While the region descriptor has a field containing the directionality, this field is left unused for partial synchronizations.

D.2 Compiler Extensions

Compilation with region support is very similar to compilation without. The main extensions are related to the increased information that must be kept about task parameters and its late expansion during task call replacement.

Since task parameters may have several regions, each task parameter is decorated by a list of regions instead of just its directionality. And each region is composed by a directionality and one range description for each dimension, in a similar way to the data structures used in the task instantiation function.

Task call replacement fills out the data structures required by the region-aware task instantiation runtime function. While in the region-unaware version, the dimension expressions could depend on the actual values of the task invocation, in the region-aware version, the expressions of the dimension ranges can also depend on the parameters. Thus, during task call replacement, these expressions which use the task prototype parameter names have those names replaced by the actual expressions passed in the invocation.

Appendix E

Additional Region-Based Benchmarks

This appendix evaluates the performance of the region-based implementations of the triad, matrix multiplication, Gauss-Seidel and Cholesky benchmarks. The region-based model is presented in chapter 4 and this appendix extends its evaluation section that starts in page 98.

E.1 Triad

The STREAM benchmark by [McCalpin, 1995] is a synthetic benchmark that measures sustained memory bandwidth and the computation rate for simple vector kernels. This algorithm has been described in chapter 3 in section 3.6.1 that starts on page 32, and evaluated in appendix C. While the algorithm does not require regions, this section compares the execution between the region-unaware runtime and the region-aware runtime to evaluate if the region-aware data dependency analysis has a negative impact on the runtime overhead.

Runtime Overhead

The runtime overhead of the executions with the region-aware runtime is significantly higher than the overhead with the region-unaware runtime. Figure E.1 shows the increment of runtime overhead in the main thread. While the increment ranges from 64% to 134%, the total task management overhead in the main thread remains below 4% of the total time.

The figure has been generated with a region tree that uses compression. Compression improves the task management overhead compared to executions with uncompressed region trees. Figure E.2 shows the reduction of task management overhead due to compression. The overhead is reduced at least by half in all cases.

Scalability and Other Implementations

Figure E.3 shows the strong scalability of the problem with three problem sizes under the region-aware SMPSs runtime, the blocked SMPSs runtime and the OpenMP implementation. The scalability of the region-aware and the blocked SMPSs versions

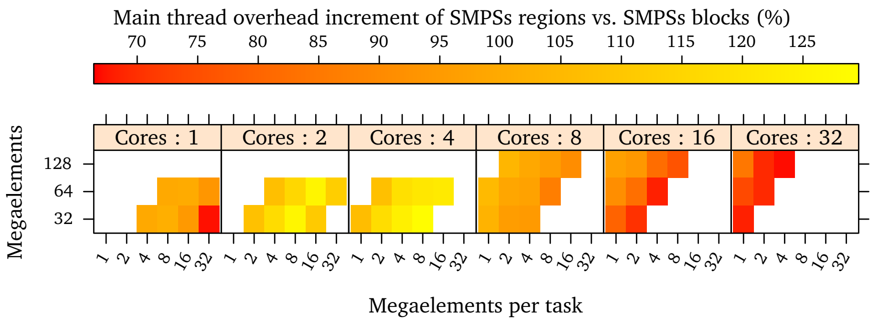


Figure E.1: Task management overhead increment on the main thread when running the SMPSs implementation of triad with regions with several vector and subvector sizes compared to the region unaware version.

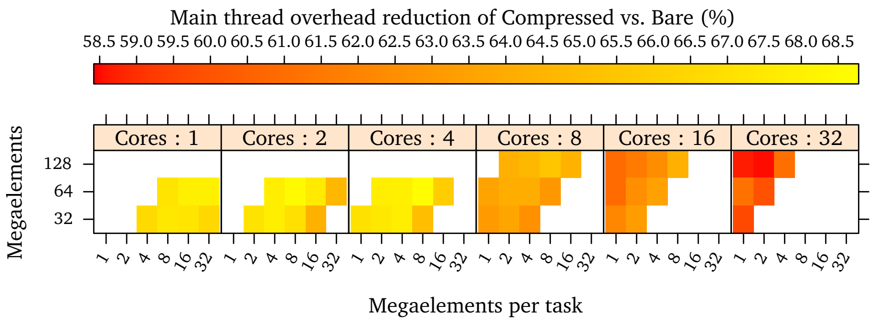


Figure E.2: Reduction of task management overhead on the main thread when running the SMPSs implementation of triad with a compressed region tree compared to an uncompressed region tree, with several vector and subvector sizes.

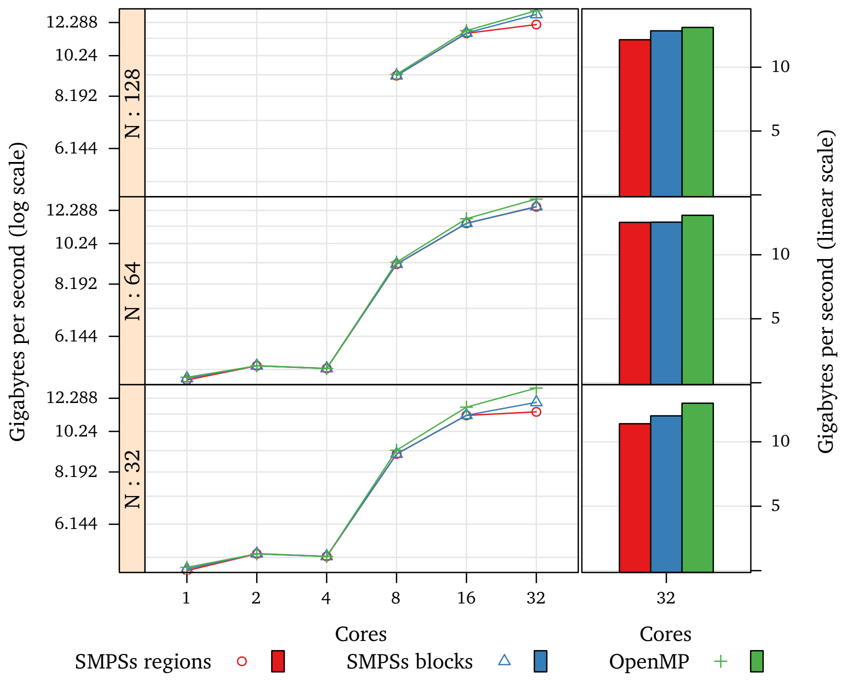


Figure E.3: Strong scalability of the region-aware SMPSs triad, the region-unaware SMPSs triad, and the OpenMP triad and performance with 32 cores.

is very similar. Both scale well, but perform slightly worse than the OpenMP version for the smallest problem sizes and highest number of cores. Table E.1 summarizes the performance metrics of each implementation.

Cores	N[a]	GB/s[b]	GB/s[b]	GB/s[b]	IPC[c]	IPC[c]	IPC[c]	Eff.[d] (%)	Eff.[d] (%)	Eff.[d] (%)
1	32	4.8	4.8	4.8	1.5	1.5	1.5	99	99	99
1	64	4.8	4.9	4.9	1.5	1.5	1.5	99	99	99
2	32	5.2	5.2	5.2	0.8	0.8	0.8	99	99	99
2	64	5.2	5.2	5.2	0.8	0.8	0.8	99	99	99
4	32	5.1	5.1	5.1	0.4	0.4	0.4	99	99	99
4	64	5.1	5.2	5.1	0.4	0.4	0.4	99	99	99
8	32	9.0	9.0	9.2	0.3	0.3	0.4	99	99	99
8	64	9.1	9.1	9.2	0.3	0.3	0.4	99	99	99
8	128	9.2	9.2	9.2	0.3	0.3	0.4	99	99	99
16	32	11.2	11.2	11.7	0.2	0.2	0.2	94	94	98
16	64	11.4	11.5	11.7	0.2	0.2	0.2	95	95	98
16	128	11.6	11.6	11.8	0.2	0.2	0.2	95	95	99
32	32	11.4	12.0	13.0	0.1	0.1	0.1	94	95	97
32	64	12.5	12.5	13.1	0.1	0.1	0.1	96	96	98
32	128	12.2	12.8	13.1	0.1	0.1	0.1	96	97	98

Implementation: SMPSs regions | SMPSs blocks | OpenMP

[a] Megaelements per array.
[b] Gigabytes per second.
[c] Mean instructions per cycle while running tasks.
[d] Mean time that threads spend running tasks.

Table E.1: Performance summary of the Triad implementations.

E.2 Matrix Multiplication

The matrix multiplication benchmark has been described in the chapter 3 in section 3.6.2 that starts on page 33 and its performance has been evaluated in appendix C in page 207. This section evaluates a version of the matrix multiplication using flat matrices instead of blocked matrices.

Parallelization with SMPSs using Regions

Listing E.1 shows the main algorithm and its tasks. Its structure is very similar to the blocked implementation. Since the matrices are flat, they are addressed differently than in the blocked version. While in the blocked version we passed pointers to the beginning of each *physical* block to the tasks, in the flat version we pass pointers to the beginning of the *logical* blocks instead. Hence, the pointers in the task calls of lines 10, 12 and 17.

Tasks also specify the areas of the matrices that are accessed by using region specifiers. For instance, the A matrix in both tasks is accessed as an input region with the A[M][K]{0:MBS}{0:KBS} specification. That is, the matrix dimensions are M by K, and the tasks access an area of MBS by KBS elements starting from the base address that is passed to the task.

Runtime Overhead

The runtime overhead of the regions version is significantly higher than the version run on the region-unaware runtime. Figure E.4 shows the increment of task management overhead in the main thread. While it may raise up to 4.5 times the overhead of the blocked version, figure E.5 shows that the total task management overhead remains below 5% in most cases, and at most takes 17% of the total time of the main thread.

These results have been generated with a region tree that uses compression. Figure E.6 shows the reduction of task management overhead, which can be as high as 84%.

Scalability and Other Implementations

Figure E.7 shows the strong scalability of the problem with several problem sizes under the region-aware SMPSs runtime, the blocked SMPSs runtime, the OpenMP implementation, and the parallel MKL version. The scalability and performance of the region-aware SMPSs version is similar to the scalability and performance of the OpenMP version and the parallel MKL version. However, the performance of the blocked SMPSs version is higher due to the blocked data layout. This is confirmed in figure E.8, which shows the average thread floating point operations per cycle while running effective code in each case.

Table E.2 summarizes the performance metrics of each implementation. It further confirms that the performance differences are essentially due to the data layout.

```
void tiled_dgemm(
    int MBS, int NBS, int KBS,
    int M, int N, int K,
    double ALPHA,
    double const *A, double const *B, double BETA, double *C)
{
    for (int i=0; i<M; i += MBS)
        for (int j=0; j<N; j+= NBS) {
            if (BETA == 0.0)
                dgemm_nobeta_tile(MBS, NBS, KBS, M, N, K, ALPHA,
                  &A[0+i*K], &B[j+0*N], &C[j+i*N]);
            else
                dgemm_tile(MBS, NBS, KBS, M, N, K, ALPHA, &A[0+i*K],
                  &B[j+0*N], BETA, &C[j+i*N]);
        }
    for (int k=KBS; k<K; k += KBS)
        for (int i=0; i<M; i += MBS)
            for (int j=0; j<N; j+= NBS)
                dgemm_tile(MBS, NBS, KBS, M, N, K, ALPHA, &A[k+i*K],
                  &B[j+k*N], 1.0, &C[j+i*N]);
}

#pragma css task input(MBS, NBS, KBS, M, N, K, ALPHA) \
    input(A[M][K]{0:MBS}{0:KBS}, B[K][N]{0:KBS}{0:NBS}) \
    output(C[M][N]{0:MBS}{0:NBS})
void dgemm_nobeta_tile(int MBS, int NBS, int KBS,
    int M, int N, int K,
    double ALPHA, double *A, double *B, double *C)
{
    static const double dzero = 0.0;
    dgemm_("N", "N", &NBS, &MBS, &KBS, &ALPHA, B, &N, A, &K,
      &dzero, C, &N);
}

#pragma css task input(MBS, NBS, KBS, M, N, K, ALPHA, BETA) \
    input(A[M][K]{0:MBS}{0:KBS}, B[K][N]{0:KBS}{0:NBS}) \
    inout(C[M][N]{0:MBS}{0:NBS})
void dgemm_tile(int MBS, int NBS, int KBS,
    int M, int N, int K,
    double ALPHA, double *A, double *B, double BETA, double *C)
{
    dgemm_("N", "N", &NBS, &MBS, &KBS, &ALPHA, B, &N, A, &K,
      &BETA, C, &N);
}
```

Listing E.1: Double precision generalized matrix-matrix multiplication in SMPSs with regions.

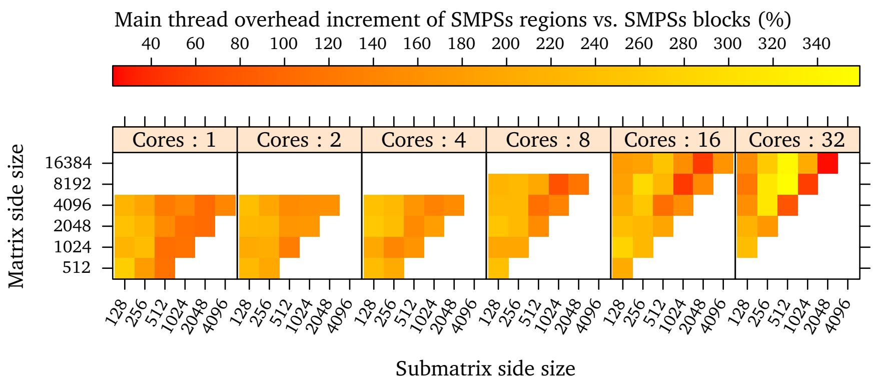


Figure E.4: Task management overhead increment on the main thread when running the SMPSs implementation of the matrix multiplication with regions with several matrix and submatrix sizes compared to the region unaware version.

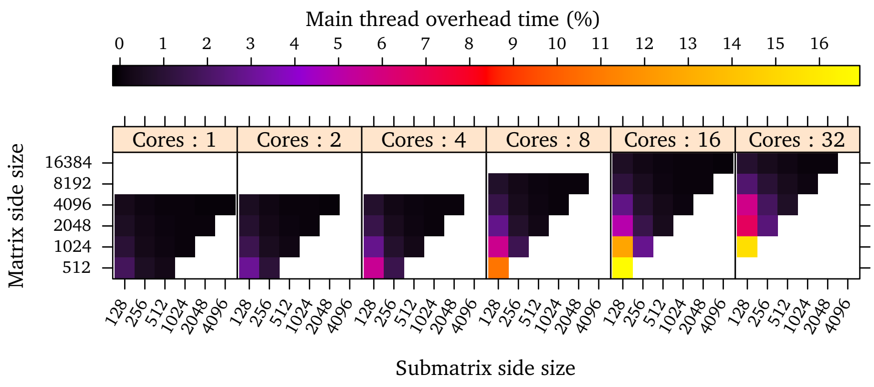


Figure E.5: Task management overhead on the main thread when running the SMPSs implementation of matrix multiplication with regions with several matrix and submatrix sizes.

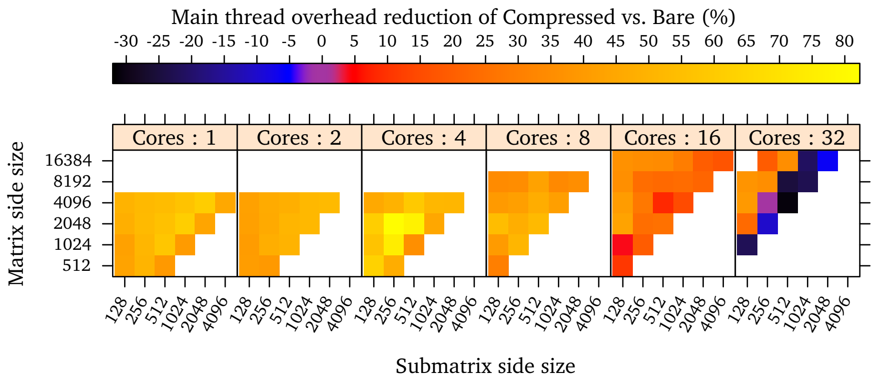


Figure E.6: Reduction of task management overhead on the main thread when running the SMPSs implementation of the matrix multiplication with a compressed region tree compared to an uncompressed region tree, with several matrix and submatrix sizes.

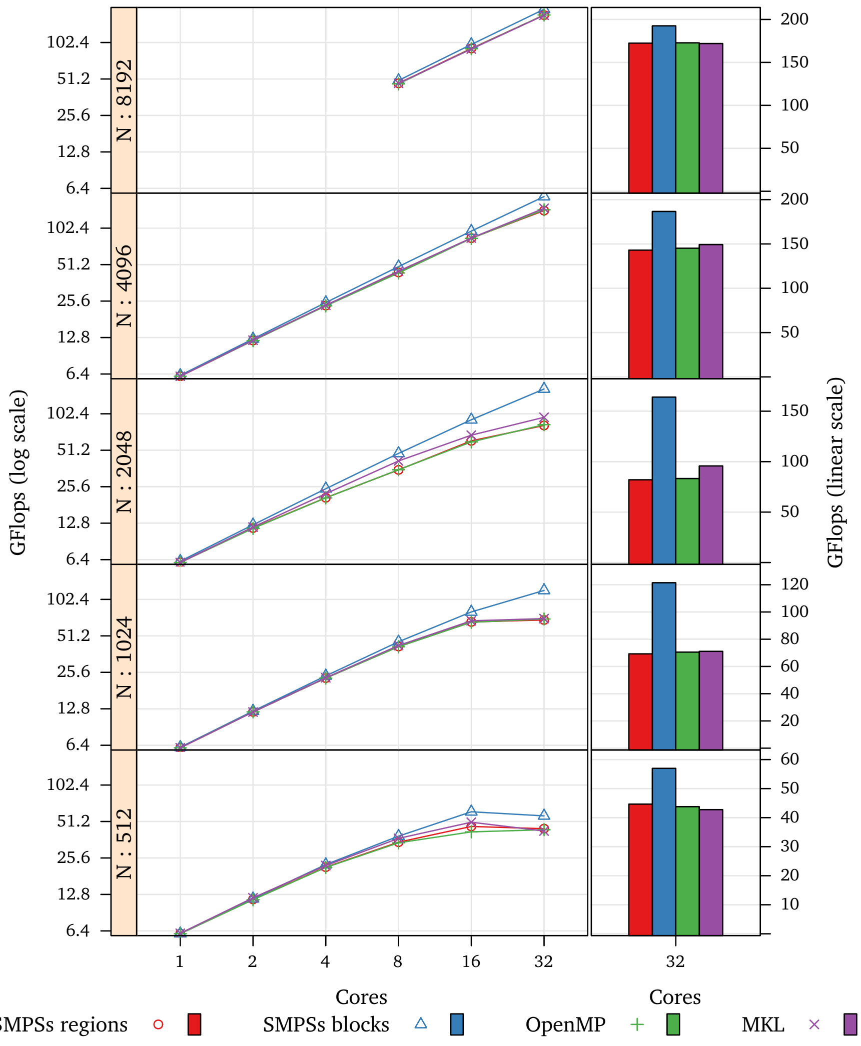


Figure E.7: Strong scalability of the region-aware SMPSs matrix multiplication, the region-unaware, the OpenMP version and the MKL parallel version and performance with 32 cores.

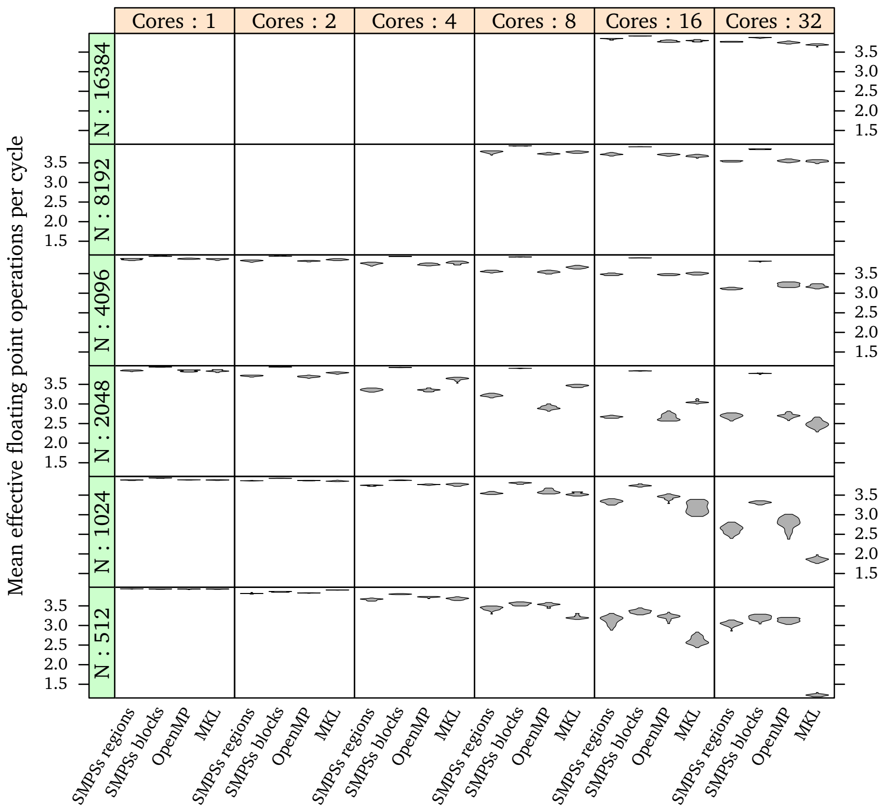


Figure E.8: Average thread effective floating point operations per cycle of the 4 matrix multiplication implementations.

Cores	N[a]	GF[b]	GF[b]	GF[b]	GF[b]	FPC[c]	FPC[c]	FPC[c]	FPC[c]	Eff.[d]	Eff.[d]	Eff.[d]	Eff.[d]
										(%)	(%)	(%)	(%)
1	512	6.1	6.1	6.1	6.1	3.9	3.9	3.9	3.9	99	99	99	99
1	1024	6.1	6.2	6.1	6.1	3.9	3.9	3.9	3.9	99	99	99	99
1	2048	6.1	6.2	6.1	6.1	3.9	3.9	3.8	3.8	99	99	99	99
1	4096	6.1	6.2	6.1	6.1	3.9	3.9	3.9	3.9	99	99	99	99
2	512	11	11	11	12	3.8	3.9	3.8	3.9	98	98	97	99
2	1024	12	12	12	12	3.9	3.9	3.9	3.9	99	99	99	99
2	2048	11	12	11	11	3.7	3.9	3.7	3.8	99	99	99	99
2	4096	12	12	12	12	3.8	3.9	3.8	3.9	99	99	99	99
4	512	21	22	21	22	3.7	3.8	3.7	3.7	96	97	94	98
4	1024	22	23	22	23	3.7	3.9	3.8	3.8	99	98	98	99
4	2048	20	24	20	22	3.4	3.9	3.4	3.6	99	99	99	99
4	4096	23	24	23	23	3.8	3.9	3.7	3.8	99	99	99	99
8	512	34	38	34	37	3.4	3.5	3.5	3.2	92	89	90	95
8	1024	41	45	41	42	3.5	3.8	3.6	3.5	96	97	95	99
8	2048	35	48	35	41	3.2	3.9	2.9	3.5	98	99	98	98
8	4096	44	49	43	45	3.6	3.9	3.5	3.7	98	99	98	98
8	8192	47	49	46	47	3.8	3.9	3.7	3.8	98	99	98	99
16	512	46	61	42	50	3.1	3.4	3.2	2.6	85	84	85	86
16	1024	66	80	66	68	3.3	3.7	3.4	3.2	94	95	91	94
16	2048	61	91	59	68	2.7	3.8	2.7	3.1	96	96	96	97
16	4096	84	96	84	84	3.5	3.9	3.5	3.5	97	99	97	98
16	8192	91	98	90	90	3.7	3.9	3.7	3.7	97	99	97	98
16	16384	95	99	93	93	3.8	3.9	3.8	3.8	97	99	97	98
32	512	44	56	43	42	3.0	3.2	3.1	1.2	43	42	41	84
32	1024	69	121	70	71	2.6	3.3	2.8	1.9	91	83	90	91
32	2048	82	163	83	95	2.7	3.8	2.7	2.5	94	94	93	97
32	4096	143	186	145	149	3.1	3.8	3.2	3.2	98	98	95	98
32	8192	172	192	172	171	3.5	3.8	3.6	3.5	96	99	97	97
32	16384	184	195	182	180	3.8	3.9	3.7	3.7	96	99	96	97

Implementation: SMPSs regions | SMPSs blocks | OpenMP | MKL

[a] Matrix side size.
[b] Gigaflops per second.
[c] Mean floating point operations per cycle while running tasks.
[d] Mean time that threads spend running tasks.

Table E.2: Performance summary of the matrix multiplication implementations.

E.3 Gauss-Seidel 2D Heat Transfer

The Gauss-Seidel 2D heat transfer benchmark has been evaluated in chapter 3 under section 3.6.3 that starts on page 36. This section evaluates a version of the algorithm using flat matrices instead of blocked matrices.

Parallelization with SMPSs using Regions

Listing E.2 shows the sequential version of the stencil code. In chapter 4, we converted the matrix into a blocked representation and we adapted the code of the task into a form suitable for that representation. For reference we reproduce the blocked version in listing E.3.

For the region-based implementation we preserve the original matrix layout. Listing E.4 shows the main algorithm and its task. Notice that in the blocked version (listing E.3) we had to pass a pointer to the beginning of the central block, and one to each of its surrounding blocks to its north, south, east and west (line 16). In the flat version (listing E.4) we pass a pointer to the beginning of the area that is accessed (line 15) and uses several regions to specify the accesses of each area. Figure E.9 illustrates the pointer and the regions.

While the blocked task declaration is simple, the region version is more complex due to the presence of several regions. In listing E.4 we declare in line 2 the north and south halos, in line 3 the west and east halos, and the region that is updated in line 4. However, while the blocked task code contains 4 conditionals, the regions version does not and in general is closer to the sequential code shown in listing E.2.

Runtime Overhead

The runtime overhead of the regions version is significantly higher than the version run on the region-unaware runtime. Figure E.10 shows the increment of runtime overhead in the main thread. While it may raise on average up to 4.3 times the overhead of the blocked version, figure E.11 shows that the total task management overhead remains below 10% in most cases, and takes more only for very small task sizes.

These measurements have been generated with a region tree that uses compression. Figure E.12 shows the reduction of task management overhead that results from compressing the region tree. Notice that the overhead get reduced by at least 30% and at most 51%.

```
1 for (int iter=0; iter < ITERS; iter++)
2   for (long i=1; i <= N; i++)
3     for (long j=1; j <= N; j++)
4       data[i][j] = 0.2 * (data[i][j] + data[i-1][j] + data[i+1][j] + data[i][j-1]
           + data[i][j+1]);
```

Listing E.2: Sequential implementation of the Gauss-Seidel algorithm for the 2D heat transfer problem.

```
#pragma css task input(N, L, top, left, bottom, right) inout(a)
void gs_tile(long N, long L, double a[L][L], double top[L][L], double left[L][L],
  double bottom[L][L], double right[L][L]) {
   for (long i=0; i < L; i++)
      for (long j=0; j < L; j++)
         a[i][j] = 0.2 * ( a[i][j]
            + (i > 0L ? a[i-1][j] : top[L-1][j])
            + (i < L-1L ? a[i+1][j] : bottom[0][j])
            + (j > 0L ? a[i][j-1] : left[i][L-1])
            + (j < L-1L ? a[i][j+1] : right[i][0]) );
}

void gauss_seidel(long N, long L, double data[N/L][N/L][L][L]) {
   for (int iter=0; iter<ITERS; iter++)
      for (long i=1; i < N/L-1; i++)
         for (long j=1; j < N/L-1; j++)
            gs_tile( N, L, data[i][j], data[i-1][j], data[i][j-1], data[i+1][j],
              data[i][j+1] );
}
```

Listing E.3: Blocked SMPSs version of the Gauss-Seidel algorithm for the 2D heat transfer problem.

```
#pragma css task input(N, L) \
   input(a{0}{1:L}) input(a{L+1}{1:L}) \
   input(a{1:L}{0}) input(a{1:L}{L+1}) \
   inout(a{1:L}{1:L})
void gs_tile(long N, long L, double a[N][N]) {
   for (int i=1; i <= L; i++)
      for (int j=1; j <= L; j++)
         a[i][j] = 0.2 * (a[i][j] + a[i-1][j] + a[i+1][j] + a[i][j-1] + a[i][j+1]);
}

void gauss_seidel(long N, long L, double data[N][N]) {
   for (int iter=0; iter < ITERS; iter++)
      for (long i=L; i < N-L; i+=L)
         for (long j=L; j < N-L; j+=L)
            gs_tile( N, L, &data[i-1][j-1] );
}
```

Listing E.4: Region-based SMPSs version of the Gauss-Seidel algorithm for the 2D heat transfer problem.

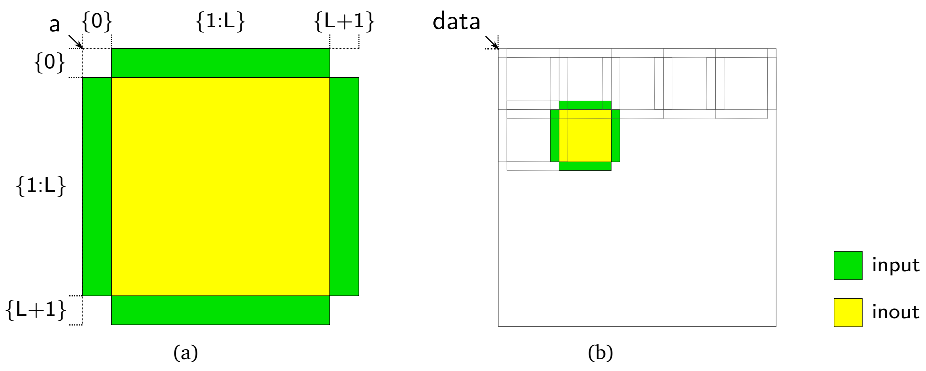


Figure E.9: Regions used in the Gauss-Seidel task: (a) from the point of view of the task; and, (b) from the point of view of the main code.

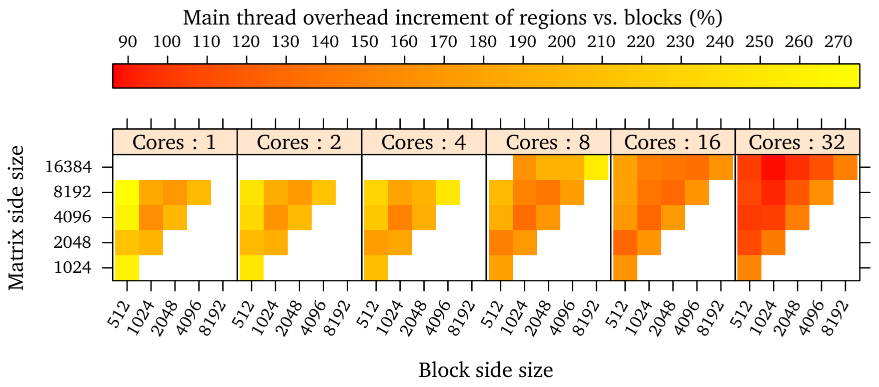


Figure E.10: Task management overhead increment on the main thread when running the SMPSs implementation of the stencil algorithm with regions with several matrix and submatrix sizes compared to the region unaware version.

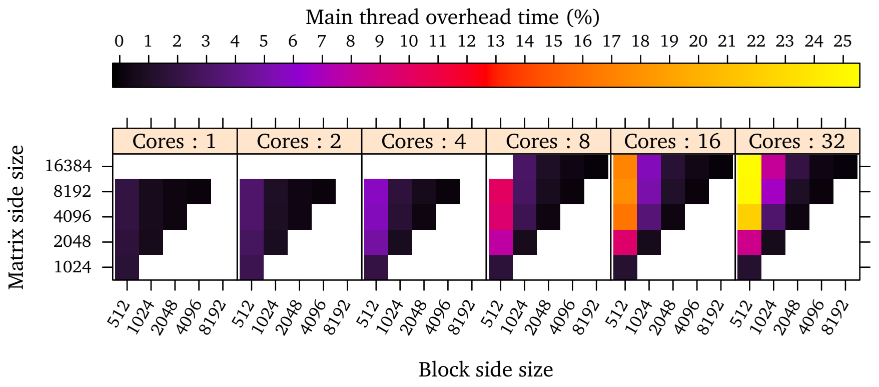


Figure E.11: Task management overhead on the main thread when running the SMPSs implementation of stencil algorithm with regions with several matrix and submatrix sizes.

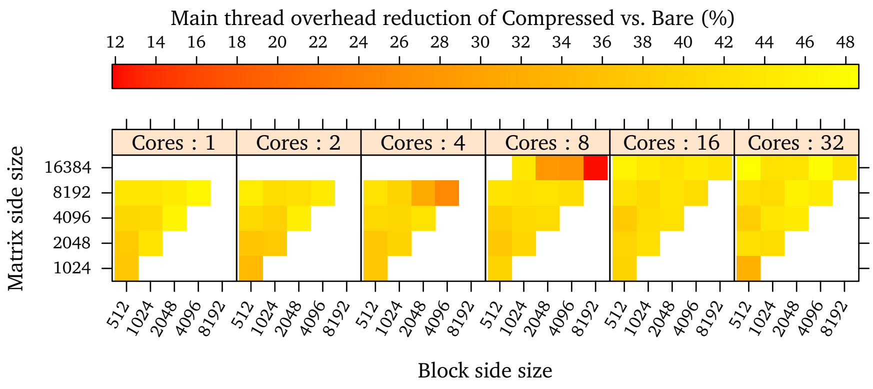


Figure E.12: Reduction of task management overhead on the main thread when running the SMPSs implementation of the stencil algorithm with a compressed region tree compared to an uncompressed region tree, with several matrix and submatrix sizes.

Scalability and Other Implementations

Similarly to the matrix multiplication case, the performance of the blocked implementation of the stencil code has different task performance than the flat implementations due to the different memory layout. In particular, the blocked layout has greater average task instructions per cycle. To illustrate the difference we have made an additional set of measurements with the region-unaware runtime. This time, we have used a flat layout. Since the region-unaware runtime cannot handle that configuration, we pass to the task a pointer to the beginning of each “virtual” block. While the region-aware runtime would allow us to specify a region of L by L elements starting at each location, the region-unaware does not. So, instead we pretend that they are all N by N independent arrays, and rely on the fact that the runtime only checks the base address to find the dependencies and that the problem has a direct, unique, and non-overlapping correspondence between those base addresses and the actual region.

The improvement of floating point operations per cycle when changing the memory layout from the flat representation to blocked representation with the region-unaware runtime is shown in figure E.13.

Also for reference we have added the other two implementations of the previous evaluation: the blocked SMPSs version, and the OpenMP version with barriers. Figure E.14 shows the strong scalability of the problem with four problem sizes under each implementation. The scalability and performance of the region-aware SMPSs version is almost identical to that of the flat version using the region-unaware runtime, which demonstrates that the overhead of handling regions is not significant for this case. The performance of the original blocked version is higher than the rest due to better task performance. However its actual parallel efficiency is very similar in all SMPSs cases. The OpenMP version, like in the previous evaluation, remains the one that has the lowest parallelism. Figure E.15 shows this metric for each version.

Table E.3 summarizes the performance metrics of each implementation.

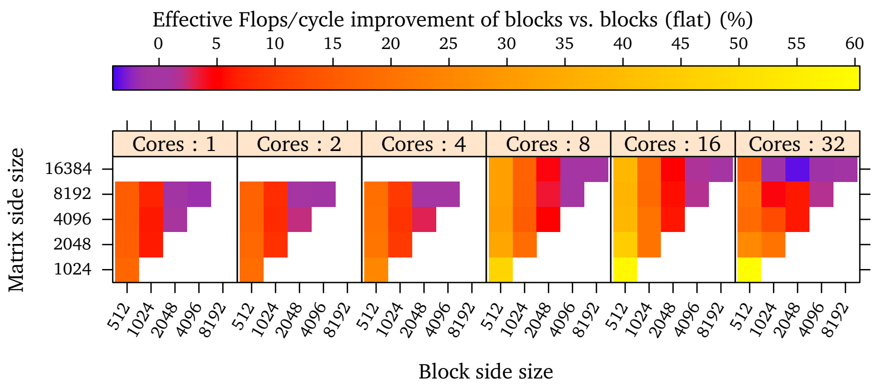


Figure E.13: Mean task floating point operations per cycle increment when changing the memory layout from flat to blocked with the stencil code under the region-unaware runtime.

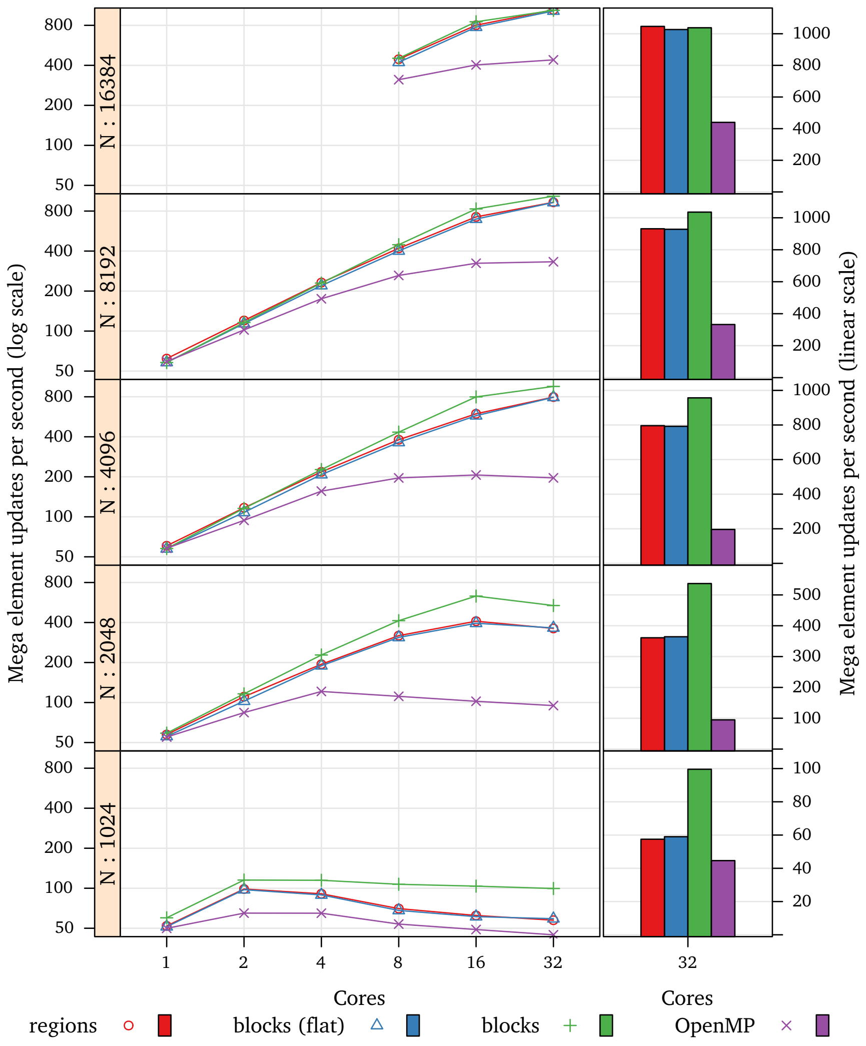


Figure E.14: Strong scalability of the region-aware SMPSs stencil algorithm, the flat layout code using the region-unaware runtime, the blocked implementation with the region-unaware runtime, and the OpenMP version; and performance of each case with 32 cores.

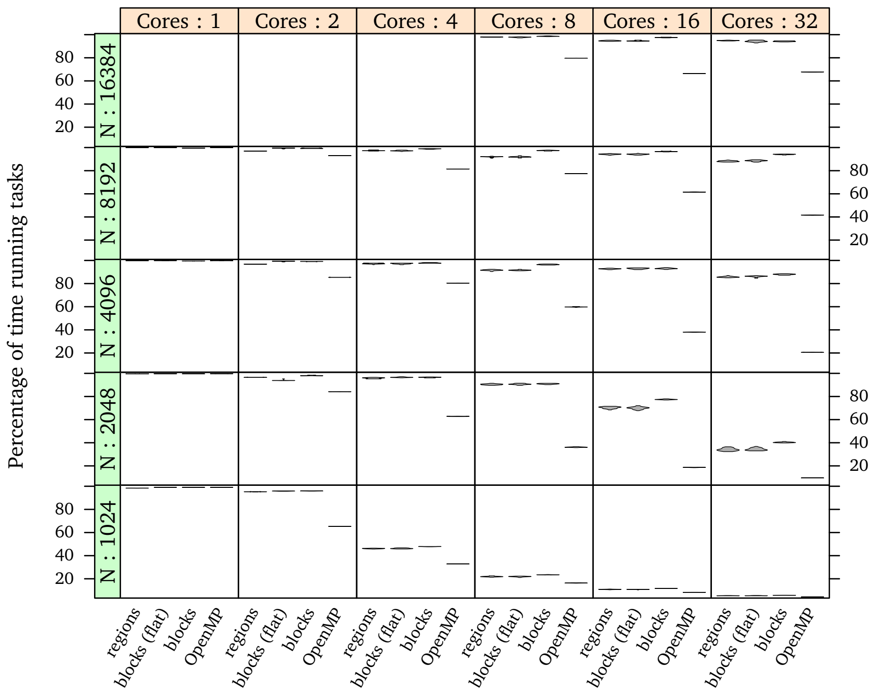


Figure E.15: Average parallel efficiency of the 4 stencil algorithm implementations.

Cores	N[a]	MU/s[b]	MU/s[b]	MU/s[b]	MU/s[b]	FPC[c]	FPC[c]	FPC[c]	FPC[c]	Eff.[d]	Eff.[d]	Eff.[d]	Eff.[d]
										(%)	(%)	(%)	(%)
1	1024	52	51	59	49	0.2	0.2	0.2	0.2	98	99	99	99
1	2048	57	55	58	55	0.2	0.2	0.2	0.2	99	99	99	99
1	4096	60	57	57	58	0.2	0.2	0.2	0.2	99	99	99	99
1	8192	62	58	58	59	0.2	0.2	0.2	0.2	99	99	99	99
2	1024	98	97	115	65	0.2	0.2	0.2	0.2	95	95	95	65
2	2048	110	102	116	84	0.2	0.2	0.2	0.2	96	93	98	84
2	4096	116	108	115	94	0.2	0.2	0.2	0.2	96	99	99	85
2	8192	120	113	116	102	0.2	0.2	0.2	0.2	96	99	99	93
4	1024	90	89	114	64	0.2	0.2	0.2	0.2	46	46	47	32
4	2048	194	189	227	121	0.2	0.2	0.2	0.2	95	96	96	62
4	4096	217	207	226	156	0.2	0.2	0.2	0.2	97	97	97	80
4	8192	231	220	229	175	0.2	0.2	0.2	0.2	97	97	98	81
8	1024	70	68	107	53	0.1	0.1	0.2	0.1	21	21	23	16
8	2048	318	309	412	111	0.1	0.1	0.2	0.1	90	90	90	36
8	4096	380	362	433	196	0.2	0.2	0.2	0.1	91	91	96	59
8	8192	419	401	446	262	0.2	0.2	0.2	0.1	91	91	97	77
8	16384	444	421	452	312	0.2	0.2	0.2	0.2	97	97	98	79
16	1024	62	61	103	48	0.1	0.1	0.2	0.1	10	10	11	8
16	2048	409	395	632	102	0.1	0.1	0.2	0.1	70	69	77	18
16	4096	595	576	798	206	0.1	0.1	0.2	0.1	92	92	93	38
16	8192	723	696	830	323	0.2	0.1	0.2	0.1	94	94	96	61
16	16384	801	775	850	403	0.2	0.2	0.2	0.1	94	94	97	66
32	1024	57	59	99	44	0.1	0.1	0.2	0.1	5	5	5	4
32	2048	360	364	536	94	0.1	0.1	0.1	0.1	34	34	40	9
32	4096	796	792	956	196	0.1	0.1	0.1	0.1	85	86	87	20
32	8192	931	928	1035	332	0.1	0.1	0.1	0.1	88	88	94	41
32	16384	1046	1026	1037	439	0.1	0.1	0.1	0.1	94	94	94	67

Implementation: regions | blocks (flat) | blocks | OpenMP

[a] Matrix side size.
[b] Mega element updates per second.
[c] Mean floating point operations per cycle while running tasks.
[d] Mean time that threads spend running tasks.

Table E.3: Performance summary of the stencil algorithm implementations.

E.4 Cholesky

The Cholesky benchmark has been evaluated in the chapter 3 under section 3.6.4 that starts on page 50. This appendix evaluates a version of the algorithm using flat matrices instead of blocked matrices.

Parallelization with SMPSs using Regions

Listing E.5 shows the main algorithm and one of its tasks. Its structure is very similar to the blocked implementation. Since the matrices are flat, they are addressed differently than in the blocked version. While in the blocked version we passed to the tasks pointers to the beginning of each *physical* block, in the flat version we pass pointers to the beginning of the *logical* blocks instead.

Tasks also specify the area of the matrix that is accessed by using region specifiers. For instance, the A matrix in the dpotrf_tile task is accessed as an inout region with the A[N][N]{0:BS}{0:BS} specification. That is, the matrix dimensions are N by N, and the task accesses an area of BS by BS elements starting from the base address that is passed to the task.

By using regions it is possible to adjust the decomposition size of the tasks to the amount of parallelism they can generate. For instance, the blocking size of the matrix multiplications in lines 3–5 could be reduced for the latest iterations of j. This is not possible to do in the blocked version. For simplicity, this aspect has not been explored for this algorithm, but is explored for the HPL benchmark in section 4.5.4 starting in page 127.

Runtime Overhead

The runtime overhead of the region-aware version is significantly higher than the version run on the region-unaware runtime. Figure E.16 shows the increment of task management overhead in the main thread. While it may raise up to 4.4 times the overhead of the blocked version, figure E.17 shows that the total task management overhead remains below 10% in most cases, and is only high with many threads and very small tasks.

These measurements have been generated with a region tree that uses compression. Figure E.18 shows that compressing the region tree can reduce the runtime overhead up to 64%. However, it does not help with 32 cores and the smallest task size.

Scalability and Other Implementations

Figure E.19 shows the strong scalability with several problem sizes under the region-aware SMPSs runtime, the blocked SMPSs runtime, the OpenMP implementation, and the MKL parallel version. The scalability of the SMPSs versions is very similar. However, the blocked version performs better due to the blocked layout. The MKL version performs similarly to the region-aware SMPSs version with up to 16 threads. However, its performance drops with 32 threads with problems with 4096 elements per side or less. The OpenMP version exhibits very poor performance. The difference between the OpenMP and the MKL version might be due to the MKL version adjusting the decomposition size at each iteration of the outer loop.

```
void cholesky(int N, int BS, double A[N][N]) {
    for (int j = 0; j < N; j += BS) {
        for (int k = 0; k < j; k += BS)
            for (int i = j+BS; i < N; i += BS)
                dgemm_tile( &A[k][i], &A[k][j], &A[j][i], BS, N);
        for (int i = 0; i < j; i+=BS)
            dsyrk_tile( &A[i][j], &A[j][j], BS, N);
        dpotrf_tile( &A[j][j], BS, N);
        for (int i = j+BS; i < N; i+=BS)
            dtrsm_tile( &A[j][j], &A[j][i], BS, N);
    }
}

#pragma css task input(BS, N) inout(A[N][N]{0:BS}{0:BS}) highpriority
void dpotrf_tile(double *A, integer BS, integer N) {
    integer INFO;
    dpotrf_("L", &BS, A, &N, &INFO);
}
```

Listing E.5: Main code of the double precision Cholesky in SMPSs with regions and one of its tasks.

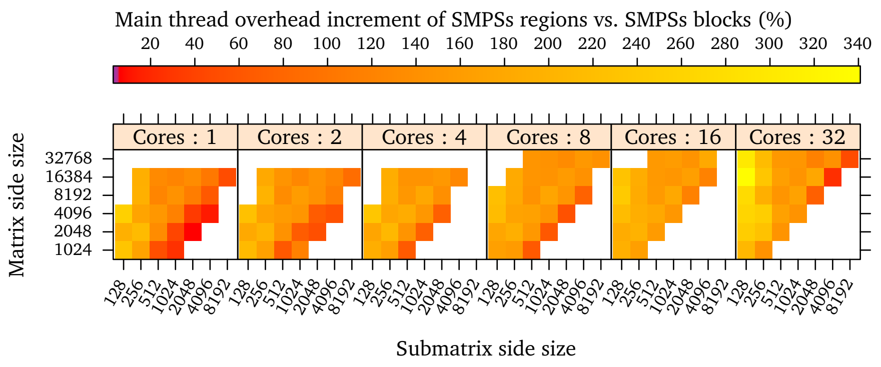


Figure E.16: Task management overhead increment on the main thread when running the SMPSs implementation of the Cholesky with regions with several matrix and submatrix sizes compared to the region unaware version.

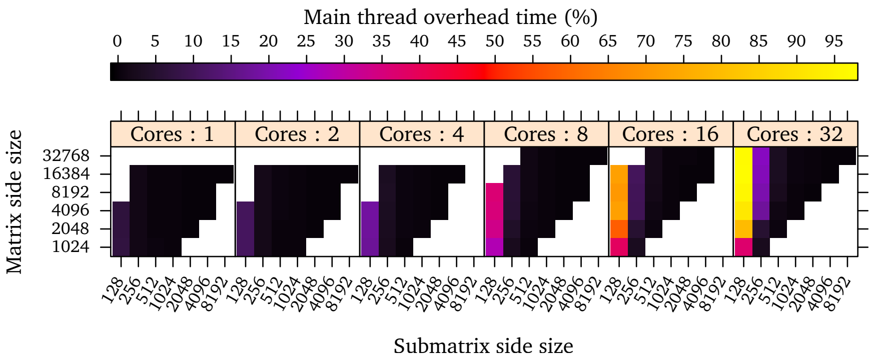


Figure E.17: Task management overhead on the main thread when running the SMPSs implementation of Cholesky with regions with several matrix and submatrix sizes.

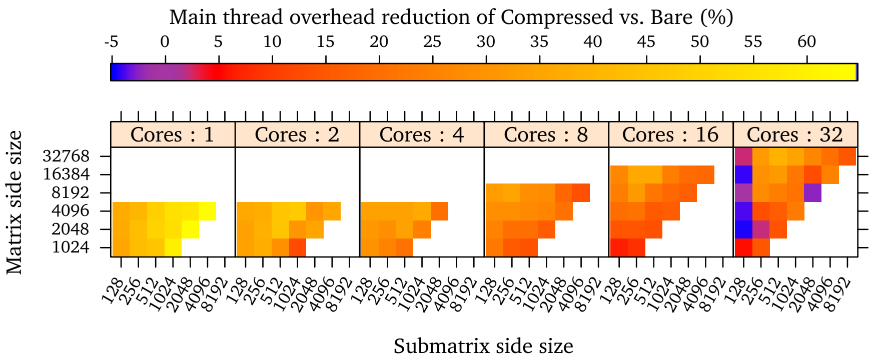


Figure E.18: Reduction of task management overhead on the main thread when running the SMPSs implementation of the Cholesky with a compressed region tree compared to an uncompressed region tree, with several matrix and submatrix sizes.

Figure E.20 confirms the effects of the data layout on the region-aware and the region-unaware SMPSs versions. The figure shows the average thread floating point operations per cycle while running effective code, that is, not idling nor in the runtime.

Table E.4 summarizes the performance metrics of each implementation.

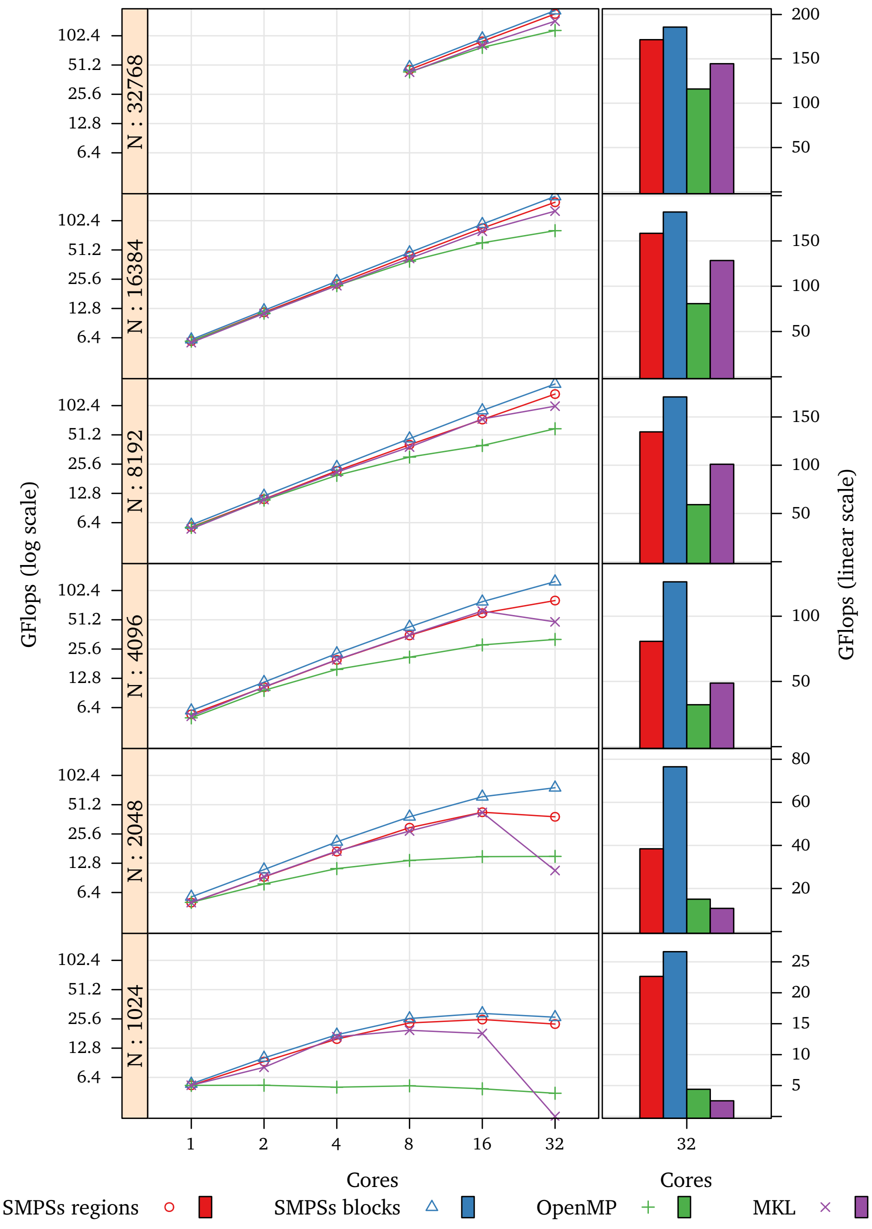


Figure E.19: Strong scalability of the region-aware SMPSs Cholesky, the region-unaware, the OpenMP version and the MKL parallel version and performance with 32 cores.

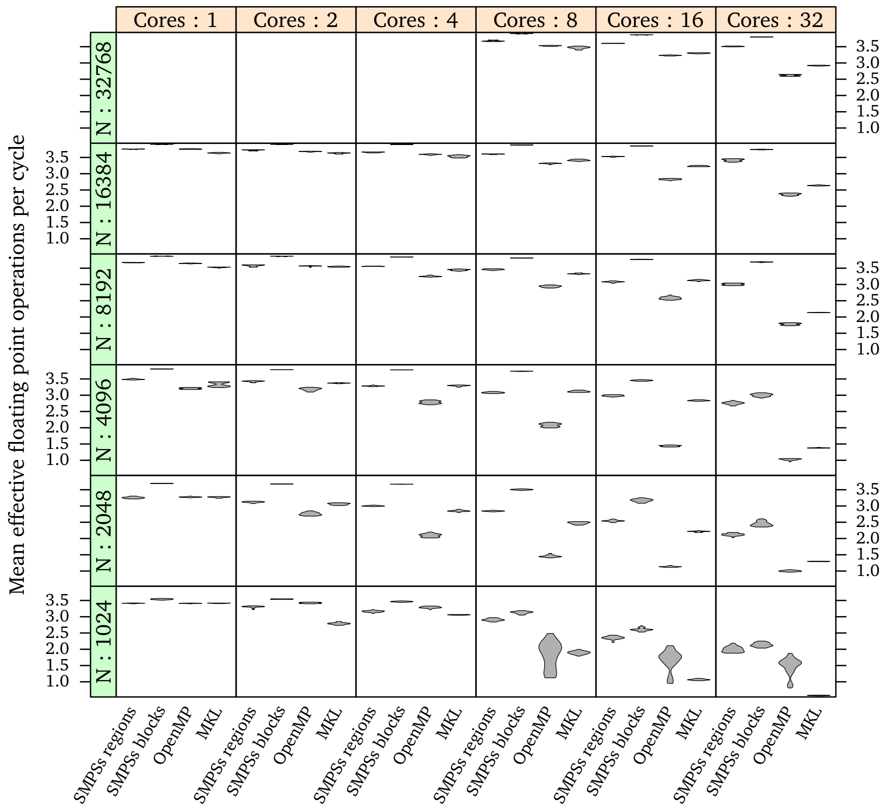


Figure E.20: Average thread effective floating point operations per cycle of the 4 Cholesky implementations.

Cores	N[a]	GF[b]	GF[b]	GF[b]	GF[b]	FPC[c]	FPC[c]	FPC[c]	FPC[c]	Eff.[d] (%)	Eff.[d] (%)	Eff.[d] (%)	Eff.[d] (%)
1	1024	5.3	5.5	5.3	5.3	3.4	3.5	3.4	3.4	99	99	99	99
1	2048	5.0	5.8	5.1	5.0	3.3	3.7	3.3	3.3	99	99	99	99
1	4096	5.5	6.0	5.0	5.2	3.5	3.8	3.2	3.3	99	99	99	99
1	8192	5.8	6.1	5.7	5.5	3.7	3.9	3.6	3.5	99	99	99	99
1	16384	5.9	6.1	5.9	5.7	3.8	3.9	3.8	3.6	99	99	99	99
2	1024	9.3	10	5.3	8.1	3.3	3.5	3.4	2.8	90	92	49	96
2	2048	9.3	11	7.8	9.3	3.1	3.7	2.8	3.1	95	95	93	98
2	4096	10	11	9.6	10	3.4	3.8	3.2	3.4	96	98	96	99
2	8192	11	12	10	11	3.6	3.9	3.6	3.5	99	99	97	99
2	16384	11	12	11	11	3.7	3.9	3.7	3.6	99	99	99	99
4	1024	15	17	5.1	16	3.2	3.5	3.3	3.1	80	81	24	91
4	2048	16	21	11	17	3.0	3.7	2.1	2.8	90	92	88	97
4	4096	19	23	15	19	3.3	3.8	2.8	3.3	96	97	91	97
4	8192	21	23	19	21	3.6	3.8	3.3	3.4	97	98	96	98
4	16384	22	24	21	21	3.7	3.9	3.6	3.5	99	99	97	99
8	1024	23	25	5.2	19	2.9	3.1	1.8	1.9	64	66	23	85
8	2048	29	38	13	27	2.8	3.5	1.5	2.5	85	87	77	92
8	4096	35	43	21	35	3.1	3.7	2.1	3.1	92	92	81	93
8	8192	40	46	30	38	3.5	3.8	2.9	3.3	93	97	82	92
8	16384	44	48	39	41	3.6	3.9	3.3	3.4	98	98	94	98
8	32768	45	48	43	43	3.7	3.9	3.5	3.5	99	99	98	99
16	1024	25	29	4.9	18	2.4	2.6	1.7	1.1	43	45	12	70
16	2048	42	61	14	42	2.5	3.2	1.1	2.2	75	78	55	88
16	4096	59	78	28	62	3.0	3.5	1.4	2.8	82	90	80	91
16	8192	73	91	39	74	3.1	3.8	2.6	3.1	95	96	61	96
16	16384	85	94	60	79	3.5	3.9	2.8	3.2	96	97	85	98
16	32768	89	95	77	82	3.6	3.9	3.2	3.3	98	98	95	99
32	1024	22	26	4.4	2.5	2.0	2.1	1.5	0.6	22	25	6	10
32	2048	38	76	15	10	2.1	2.5	1.0	1.3	48	62	32	20
32	4096	80	126	32	48	2.8	3.0	1.0	1.4	63	84	64	75
32	8192	134	170	59	101	3.0	3.7	1.8	2.1	90	92	66	96
32	16384	158	181	80	128	3.4	3.7	2.4	2.6	92	96	68	97
32	32768	171	185	116	144	3.5	3.8	2.6	2.9	97	97	88	99

Implementation: SMPSs regions | SMPSs blocks | OpenMP | MKL

[a] Matrix side size.
[b] Gigaflops per second.
[c] Mean floating point operations per cycle while running tasks.
[d] Mean time that threads spend running tasks.

Table E.4: Performance summary of the Cholesky implementations.

Appendix F

Additional Benchmarks Using the NUMA-aware Runtime

This appendix evaluates the performance of the region-based benchmarks that do not significantly benefit from NUMA-awareness under the evaluation hardware environment. The NUMA-aware programming model is presented in chapter 5 and this appendix extends its evaluation section that starts in page 148.

F.1 Matrix Multiplication

The matrix multiplication benchmark with regions has been evaluated in the previous appendix in section E.2 that starts on page 229. This section evaluates the NUMA aspects on that implementation.

Units of Distribution

The matrix multiplication operates over three bidimensional arrays. Each matrix is accessed with different orders. Tasks access the first matrix by columns, the second by rows, and the third by blocks. For any given column of the first matrix, the algorithm traverses every row of the second matrix, and updates the value of every position of the result matrix. This access pattern shows that no unit of distribution can be applied simultaneously to each matrix to benefit the NUMA affinity of all three matrices.

We have tried three distributions: a distribution by blocks, a distribution by horizontal panels and a distribution by vertical panels. In all three cases each matrix has been initialized independently by separating their initializations with barriers.

Effects of Data Placement on Performance

To validate the effectiveness of the runtime placement and its effect on performance, we have made measurements with the NUMA-unaware scheduler with memory interleaving and with the NUMA-aware scheduler with the three data distributions.

Since the matrix multiplication algorithm has a high number of computations per element, and we are using the highly tuned MKL kernels for the task implementation, we expect that the NUMA aspects will not have an important effect on performance.

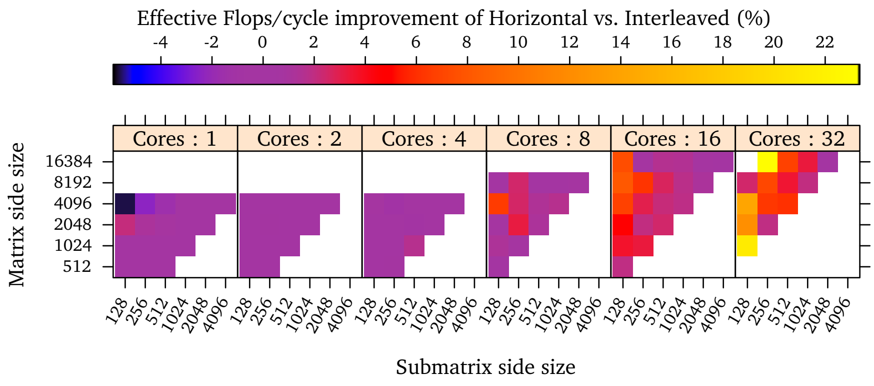


Figure F.1: Average task floating point operations per cycle improvement of the matrix multiplications distributed horizontally compared to the executions with the interleaved distribution.

Figure F.2 shows the performance and strong scalability of each case with the best blocking size for each problem size. Notice that the executions with the smallest problem sizes perform differently depending on the data distribution, but as we increase the problem size the performance difference disappears.

Figure F.1 shows the average task flops per cycle when running the problem with the horizontal distribution compared to the executions with the memory interleaved. Notice that for many executions, task performance is almost identical, and only in two cases it is 23% better.

Effectiveness of the NUMA Scheduling Policy

To evaluate how much NUMA affinity and memory load balancing affect the performance of the matrix multiplication, we have selected the horizontal distribution and made additional measurements with the scheduler that misplaces tasks, and the scheduler that tries to overload the memories. Figure F.3 shows the scalability of each case with the same blocking size as the NUMA scheduler. Notice that the “bad” schedulers do not manage to perform significantly lower than the interleaved case. This shows again that this problem does not benefit at this scale from NUMA-awareness.

Performance Compared to Other Implementations

To compare the performance of the NUMA-aware implementation to other programming models, we have made additional measurements with the blocked implementation presented in page 33 and evaluated in page 207, the OpenMP version, and the parallel MKL version.

Figure F.4 shows the strong scalability of each implementation with several problem sizes with the best performing block size in each case. The series labeled “Interleaved” corresponds to the NUMA-unaware executions. The series labeled “Blocked Layout” corresponds to the implementation from the blocks-based programming model with the memory interleaved. The “OpenMP” series corresponds to an equivalent one to the regions version, but using OpenMP. The series labeled “MKL” corresponds to the executions using the parallel implementation within the MKL library.

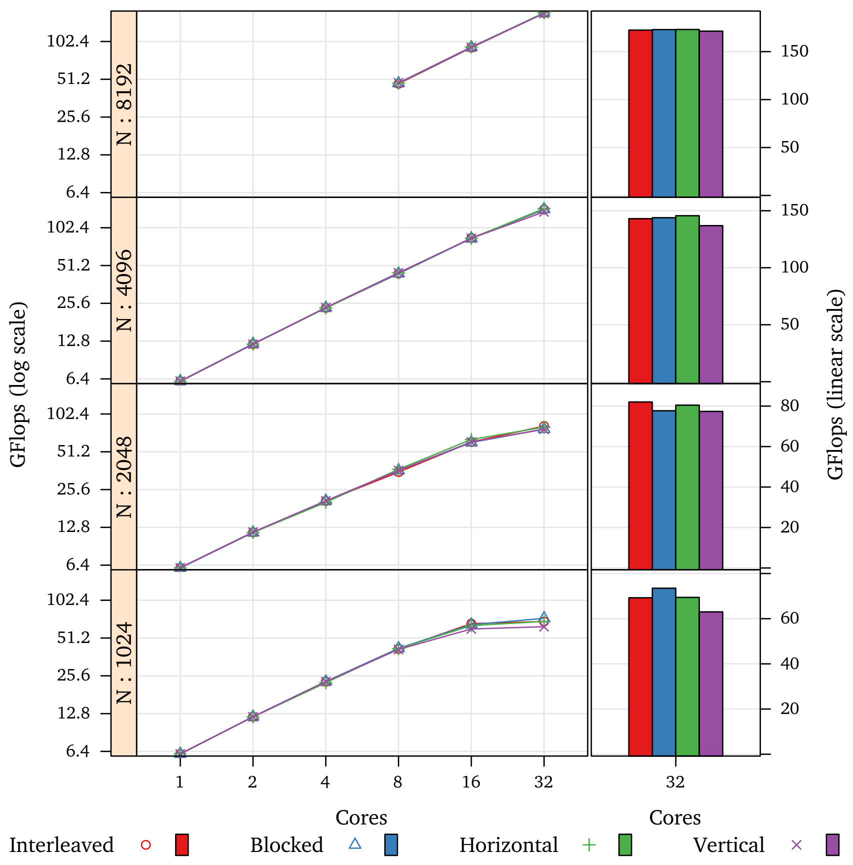


Figure F.2: Strong scalability and performance with 32 cores of the NUMA-unaware matrix multiplication, the executions with a blocked data distribution, the executions with horizontal distribution, and the executions with vertical distribution.

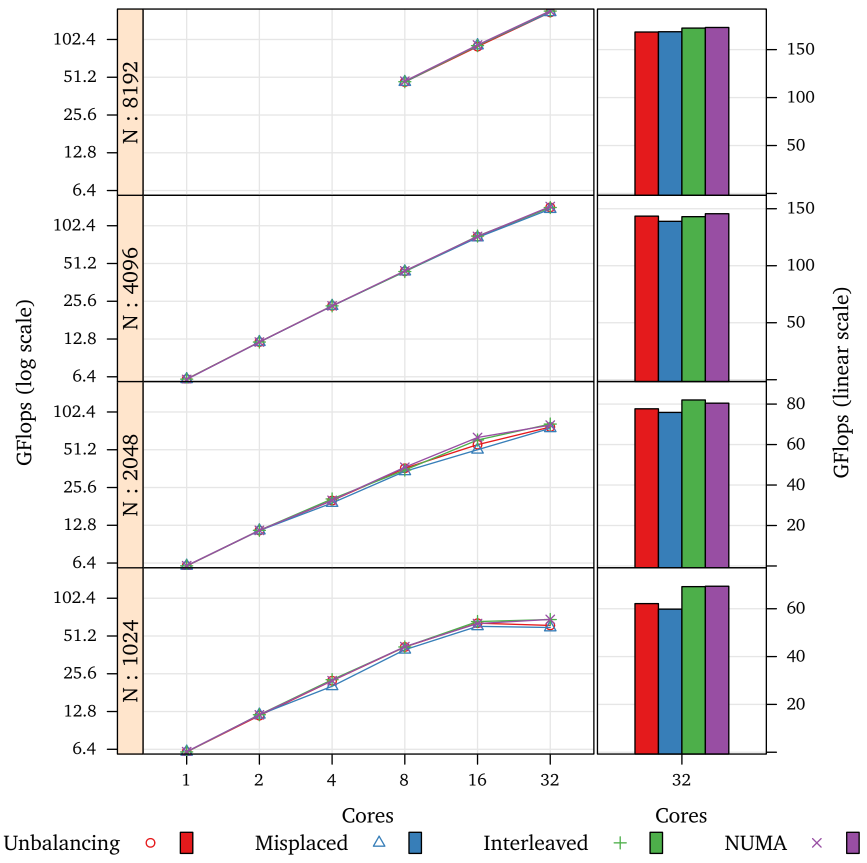


Figure F.3: Strong scalability and performance with 32 cores of the matrix multiplication with memory overloading scheduler, the misplacing scheduler, the NUMA-unaware scheduler, and the NUMA scheduler.

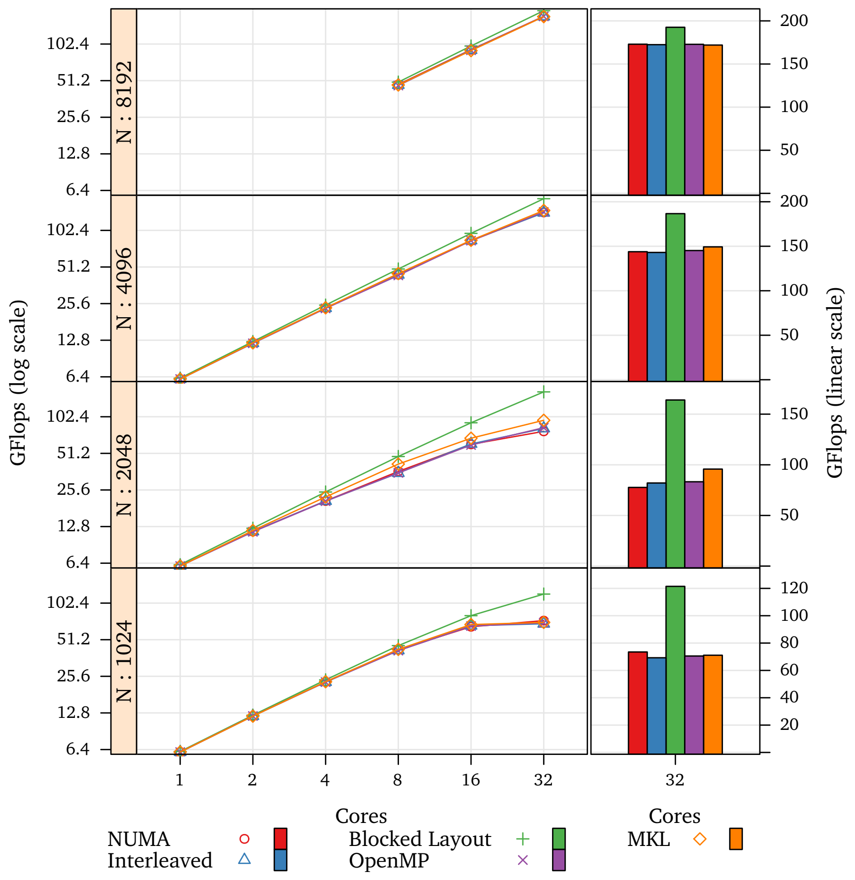


Figure F.4: Strong scalability and performance with 32 cores of the matrix multiplication algorithm with several variants under SMPSs, OpenMP and the parallel MKL.

Notice that NUMA affinity does not have an impact on performance. Instead, spatial locality is clearly the most important factor. Figure F.5 shows the floating point performance difference between the NUMA-unaware executions with the blocked layout and the NUMA-aware executions with the flat layout and the horizontal distribution. For block sizes below 1024 elements per side, the executions with the blocked data layout and the memory interleaved outperform the executions with the flat layout and the NUMA-aware scheduler.

Table F.1 summarizes the scalability performance of each implementation numerically.

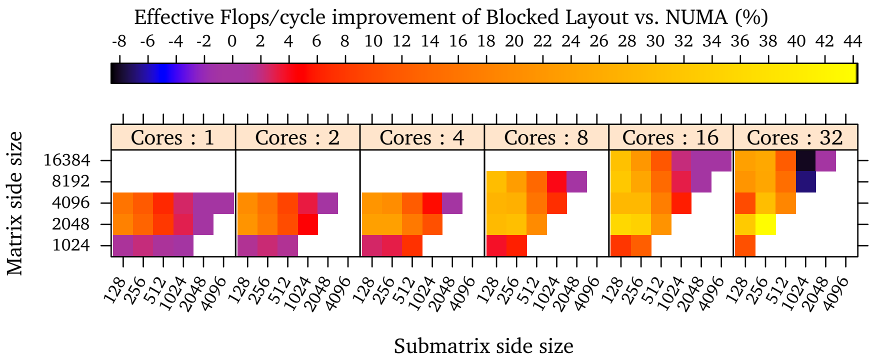


Figure F.5: Mean matrix multiplication task performance difference between the NUMA-unaware execution using the blocked data layout compared to the NUMA-aware execution with the flat data layout and horizontal distribution.

Cores	N[a]	GF[b]	GF[b]	GF[b]	GF[b]	GF[b]	FPC[c]	FPC[c]	FPC[c]	FPC[c]	FPC[c]
1	1024	6.1	6.1	6.2	6.1	6.1	3.9	3.9	3.9	3.9	3.9
1	2048	6.1	6.1	6.2	6.1	6.1	3.8	3.9	3.9	3.8	3.8
1	4096	6.1	6.1	6.2	6.1	6.1	3.9	3.9	3.9	3.9	3.9
2	1024	12	12	12	12	12	3.9	3.9	3.9	3.9	3.9
2	2048	11	11	12	11	11	3.7	3.7	3.9	3.7	3.8
2	4096	12	12	12	12	12	3.8	3.8	3.9	3.8	3.9
4	1024	23	22	23	22	23	3.8	3.7	3.9	3.8	3.8
4	2048	20	20	24	20	22	3.4	3.4	3.9	3.4	3.6
4	4096	23	23	24	23	23	3.8	3.8	3.9	3.7	3.8
8	1024	42	41	45	41	42	3.6	3.5	3.8	3.6	3.5
8	2048	36	35	48	35	41	3.3	3.2	3.9	2.9	3.5
8	4096	44	44	49	43	45	3.6	3.6	3.9	3.5	3.7
8	8192	47	47	49	46	47	3.8	3.8	3.9	3.7	3.8
16	1024	65	66	80	66	68	3.4	3.3	3.7	3.4	3.2
16	2048	60	61	91	59	68	2.7	2.7	3.8	2.7	3.1
16	4096	84	84	96	84	84	3.5	3.5	3.9	3.5	3.5
16	8192	92	91	98	90	90	3.7	3.7	3.9	3.7	3.7
16	16384	95	95	99	93	93	3.8	3.8	3.9	3.8	3.8
32	1024	73	69	121	70	71	3.0	2.6	3.3	2.8	1.9
32	2048	77	82	163	83	95	2.6	2.7	3.8	2.7	2.5
32	4096	143	143	186	145	149	3.2	3.1	3.8	3.2	3.2
32	8192	172	172	192	172	171	3.6	3.5	3.8	3.6	3.5
32	16384	183	184	195	182	180	3.8	3.8	3.9	3.7	3.7

Implementation: NUMA Interleaved Blocked Layout OpenMP MKL

[a] Matrix side size.
[b] Gigaflops per second.
[c] Mean floating point operations per cycle while running tasks.

Table F.1: Performance summary of the matrix multiplication implementations.

F.2 Cholesky

The blocked version of the Cholesky benchmark has been evaluated in chapter 3 in the section that starts in page 50 and in its regions-based version in appendix E in the section thats starts in page 243. This appendix evaluates how NUMA affects the performance of the regions-based version.

Units of Distribution

The algorithm operates over a square matrix with tasks that access non-overlapping square regions. Therefore, this case is suitable for a blocked data distribution that matches the blocks of the tasks, and for horizontal and vertical distributions with their respective height and width that matches the block size.

Since this algorithm contains tasks that operate over pairs of blocks that are aligned horizontally and pairs that are aligned vertically, the horizontal and vertical distributions could potentially perform better than the blocked distribution.

Effects of Data Placement on Performance

To validate the effectiveness of the runtime placement and its effect on performance, we have made measurements with the NUMA-unaware scheduler with memory interleaving and with the NUMA-aware scheduler with the three data distributions. Since most of the tasks are matrix multiplications, we expect that affinity will not have a huge impact on performance.

Figure F.6 shows the performance and strong scalability of each distribution with the best blocking size for each problem size, and table F.2 shows the corresponding numerical data. Notice that there is barely any difference between the distributions nor the interleaved execution. In this case, the matrix multiplication task is optimized so well that the algorithm does not suffer from bad NUMA affinity. Only the smallest problem size with the horizontal distribution performs worse than the rest. In this configuration, the the task affinity is similar to the blocked distribution and much higher than the vertical distribution. However, the MKL library is spending on average 58% more time running the kernel.

Effectiveness of the NUMA Scheduling Policy

To evaluate how much memory affinity and memory load balancing affect performance on this code, we have selected the blocked distribution and made additional measurements with the “bad” schedulers. Figure F.7 shows the strong scalability of each case with the best blocking size of the NUMA scheduler. Notice that the interleaved executions perform similarly to the NUMA-aware executions, and that the “bad” schedulers only perform worse for the smallest problem sizes. This shows that this problem only needs memory interleaving to perform well at this scale.

For completeness, figure F.8 shows the average task memory affinity of the executions and demonstrates that the dependency structure of the problem is not preventing the schedulers from making their NUMA affinity policies effective. Instead affinity is not affecting performance.

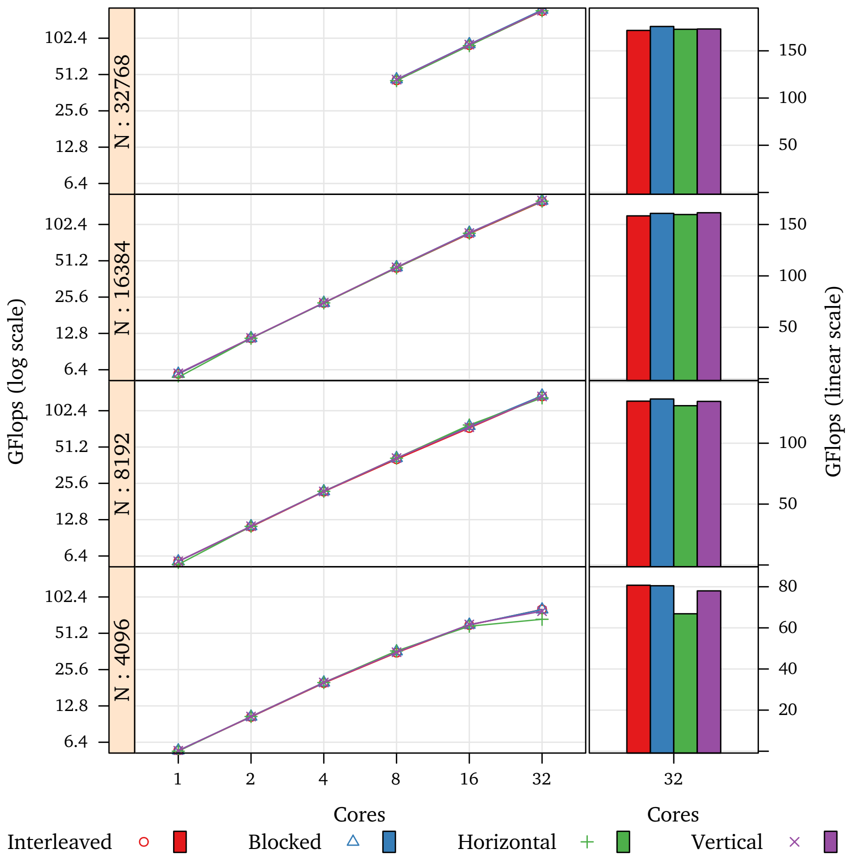


Figure F.6: Strong scalability and performance with 32 cores of the NUMA-unaware Cholesky algorithm, the executions with a blocked data distribution, the executions with horizontal distribution, and the executions with vertical distribution.

Cores	N[a]	GF[b]	GF[b]	GF[b]	GF[b]	Aff.[c] (%)	Aff.[c] (%)	Aff.[c] (%)	Aff.[c] (%)
1	2048	5.0	5.0	5.0	5.0	100	100	100	100
1	4096	5.5	5.5	5.4	5.4	100	100	100	100
1	8192	5.8	5.8	5.4	5.8	100	100	100	100
1	16384	5.9	5.9	5.5	6.0	100	100	100	100
2	2048	9.3	9.3	9.2	9.2	100	100	100	100
2	4096	10	10	10	10	100	100	100	100
2	8192	11	11	11	11	100	100	100	100
2	16384	11	11	11	11	100	100	100	100
4	2048	15	16	15	15	100	100	100	100
4	4096	19	19	19	20	100	100	100	100
4	8192	21	21	21	21	100	100	100	100
4	16384	22	22	22	23	100	100	100	100
8	2048	23	23	22	23	50	71	72	56
8	4096	35	36	36	35	50	63	75	55
8	8192	40	41	41	41	50	72	76	75
8	16384	44	45	44	45	50	80	75	78
8	32768	45	46	45	46	50	83	75	80
16	2048	26	26	26	25	25	53	54	44
16	4096	59	60	58	60	25	52	58	33
16	8192	73	76	77	74	25	47	61	46
16	16384	85	87	86	86	25	63	61	62
16	32768	89	91	88	90	25	70	53	63
32	4096	80	80	66	77	12	41	39	22
32	8192	134	136	130	134	12	39	45	24
32	16384	158	160	159	161	12	41	47	41
32	32768	171	175	172	173	12	57	37	52

Distribution: Interleaved Blocked Horizontal Vertical

[a] Matrix side size.
[b] Gigaflops per second.
[c] Mean memory affinity.

Table F.2: Performance summary of Cholesky with four data distributions.

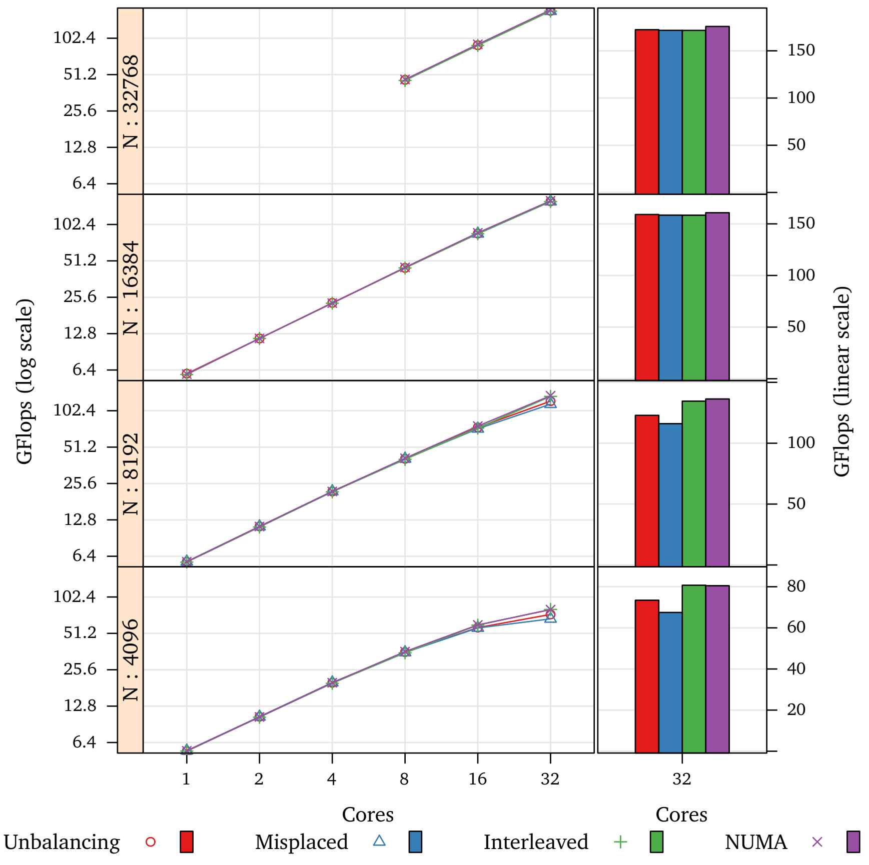


Figure F.7: Strong scalability and performance with 32 cores of Cholesky with memory overloading scheduler, the misplacing scheduler, the NUMA-unaware scheduler, and the NUMA scheduler.

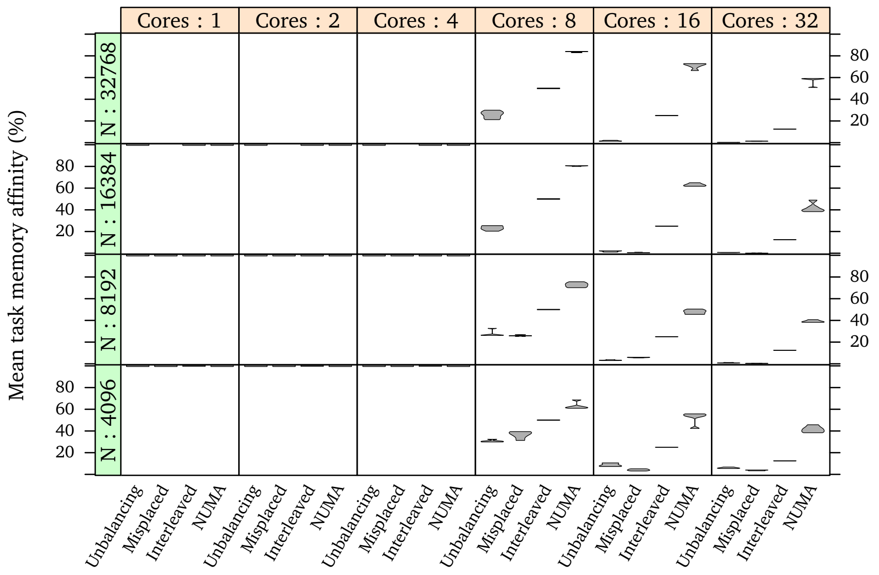


Figure F.8: Mean memory affinity of the tasks of the Cholesky code with four schedulers.

Performance Compared to Other Implementations

To compare the performance of the NUMA-aware implementation to other programming models, we have made additional measurements with the blocked implementation from chapter 3 that starts in page 50, the OpenMP version, and the parallel MKL version.

Figure F.9 shows the strong scalability of each implementation with several problem sizes with the best performing block size in each case. The series labeled “Interleaved” corresponds to the NUMA-unaware executions. The series labeled “Blocked Layout” corresponds to the implementation from the blocks-based programming model with the memory interleaved. The “OpenMP” series corresponds to an equivalent one to the regions version, but using OpenMP and barriers. The series labeled “MKL” corresponds to the executions using the parallel implementation within the MKL library.

Notice that all the SMPSs versions outperform all the other due to its ability to exploit more parallelism at a coarser granularity. However, NUMA affinity does not have an impact on performance. Instead, spatial locality is clearly the most important factor.

Figure F.10 shows the floating point performance difference between the NUMA-unaware executions with the blocked layout and the NUMA-aware executions with the flat layout. For block sizes below 1024 elements per side, the executions with the blocked data layout and the memory interleaved outperform the executions with the flat layout and the NUMA-aware scheduler.

Table F.3 summarizes the scalability performance of each implementation numerically.

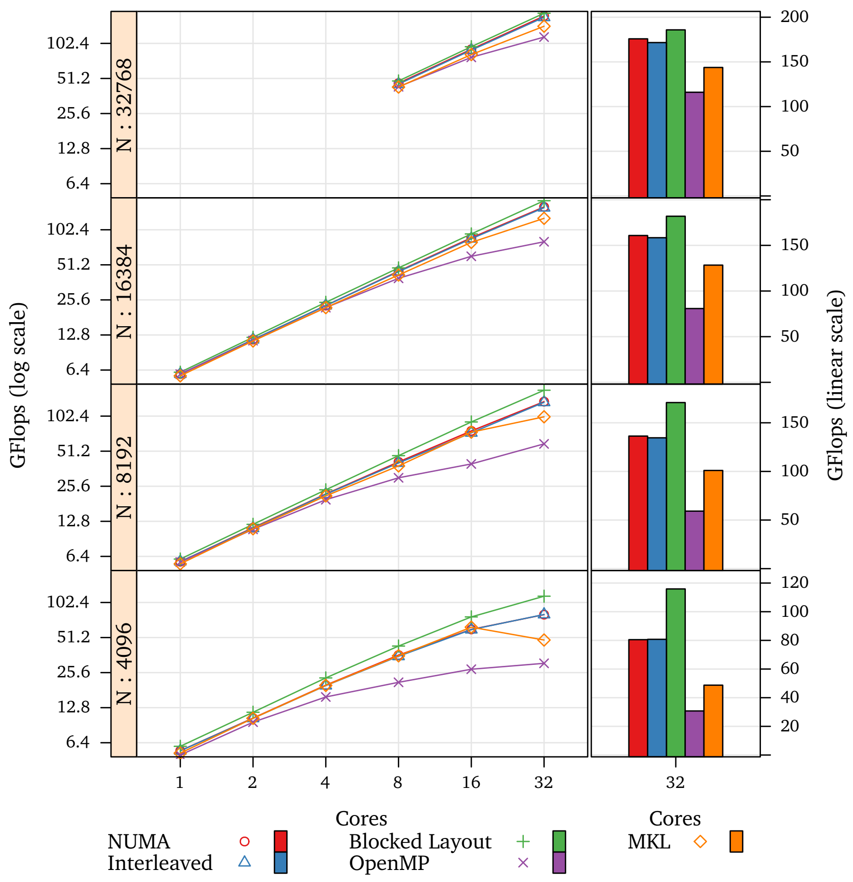


Figure F.9: Strong scalability and performance with 32 cores of the Cholesky algorithm with several variants under SMPSs, OpenMP and the parallel MKL.

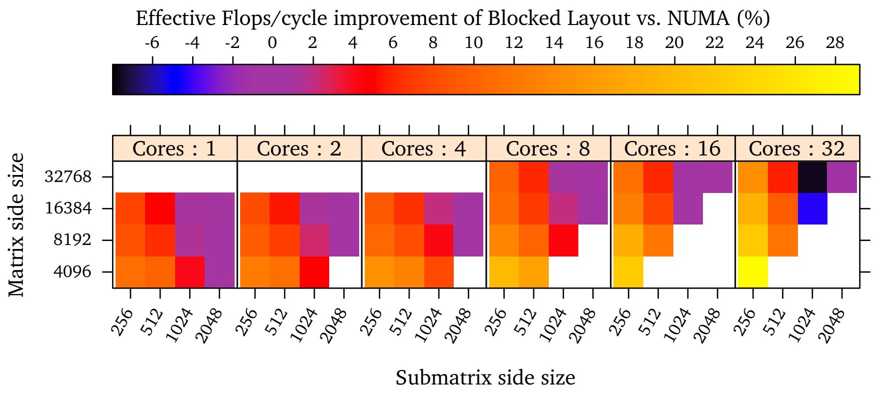


Figure F.10: Mean Cholesky task performance difference between the NUMA-unaware execution using the blocked data layout compared to the NUMA-aware execution with the flat data layout and blocked distribution.

Cores	N[a]	GF[b]	GF[b]	GF[b]	GF[b]	GF[b]	FPC[c]	FPC[c]	FPC[c]	FPC[c]	FPC[c]
1	4096	5.5	5.5	6.0	5.0	5.2	3.5	3.5	3.8	3.2	3.3
1	8192	5.8	5.8	6.1	5.7	5.5	3.7	3.7	3.9	3.6	3.5
1	16384	5.9	5.9	6.1	5.9	5.7	3.8	3.8	3.9	3.8	3.6
2	4096	10	10	11	9.6	10	3.4	3.4	3.8	3.2	3.4
2	8192	11	11	12	10	11	3.6	3.6	3.9	3.6	3.5
2	16384	11	11	12	11	11	3.7	3.7	3.9	3.7	3.6
4	4096	19	19	23	15	19	3.3	3.3	3.8	2.8	3.3
4	8192	21	21	23	19	21	3.6	3.6	3.8	3.3	3.4
4	16384	22	22	24	21	21	3.7	3.7	3.9	3.6	3.5
8	4096	36	35	43	21	35	3.1	3.1	3.7	2.1	3.1
8	8192	41	40	46	30	38	3.5	3.5	3.8	2.9	3.3
8	16384	45	44	48	39	41	3.6	3.6	3.9	3.3	3.4
8	32768	46	45	48	43	43	3.7	3.7	3.9	3.5	3.5
16	4096	60	59	76	27	62	3.0	3.0	3.7	1.9	2.8
16	8192	76	73	91	39	74	3.2	3.1	3.8	2.6	3.1
16	16384	87	85	94	60	79	3.6	3.5	3.9	2.8	3.2
16	32768	91	89	95	77	82	3.7	3.6	3.9	3.2	3.3
32	4096	80	80	115	30	48	2.7	2.8	3.4	1.8	1.4
32	8192	136	134	170	59	101	3.0	3.0	3.7	1.8	2.1
32	16384	160	158	181	80	128	3.5	3.4	3.7	2.4	2.6
32	32768	175	171	185	116	143	3.6	3.5	3.8	2.6	2.9

Implementation: NUMA | Interleaved | Blocked Layout | OpenMP | MKL

[a] Matrix side size.
[b] Gigaflops per second.
[c] Mean floating point operations per cycle while running tasks.

Table F.3: Performance summary of the Cholesky implementations.

F.3 Strassen-Winograd

The Strassen-Winograd benchmark has been evaluated in the previous chapters. The blocked variant has been evaluated in section 3.6.5 that starts in page 68, and the flat variant using regions has been evaluated in section 4.5.1 that starts in page 99. This section evaluates how NUMA affinity affects its performance.

Units of Distribution

The algorithm has been written to allow different blocking sizes for the matrix multiplication tasks and the rest of the tasks, which have different algorithmic cost. It operates over three matrices by non overlapping square blocks. Therefore, this case is suitable for a blocked data distribution that matches the blocks of the tasks, and for horizontal and vertical distributions with their respective height and width that matches the block size. Since there are two blocking sizes, we choose the biggest one.

The access pattern of the algorithm over the source matrices is similar to the one of the standard matrix multiplication. Therefore, the same assumptions apply. However, since Strassen uses temporary matrices, the distribution of those will be determined dynamically and thus the affinity of the tasks that use these as input will depend on the place where their predecessors are executed.

Effects of Data Placement on Performance

To validate the effectiveness of the runtime placement and its effect on performance, we have made measurements with the NUMA-unaware scheduler with memory interleaving and with the NUMA-aware scheduler with the three data distributions. Since some tasks are matrix additions and subtractions, we expect that affinity will have a higher impact on performance than it does with the regular matrix multiplication algorithm.

Figure F.11 shows the performance and strong scalability of each distribution with the best blocking size for each problem size, and table F.4 shows the corresponding numerical data. Note that the floating point performance does not reflect the real number of floating point operations but the equivalent of the regular matrix multiplication algorithm. Therefore, the rate can have values above the theoretical hardware maximum.

Figure F.11 and table F.4 show that the three distributions perform almost on par. However, they scale better than the NUMA-unaware execution. As the problem size gets bigger, the portion of the time spent running matrix multiplication tasks gets also bigger, since they have more computational complexity. However, since their performance is less sensitive to memory affinity, as we increase the problem size, the performance gap between the NUMA-aware and the NUMA-unaware executions gets narrower.

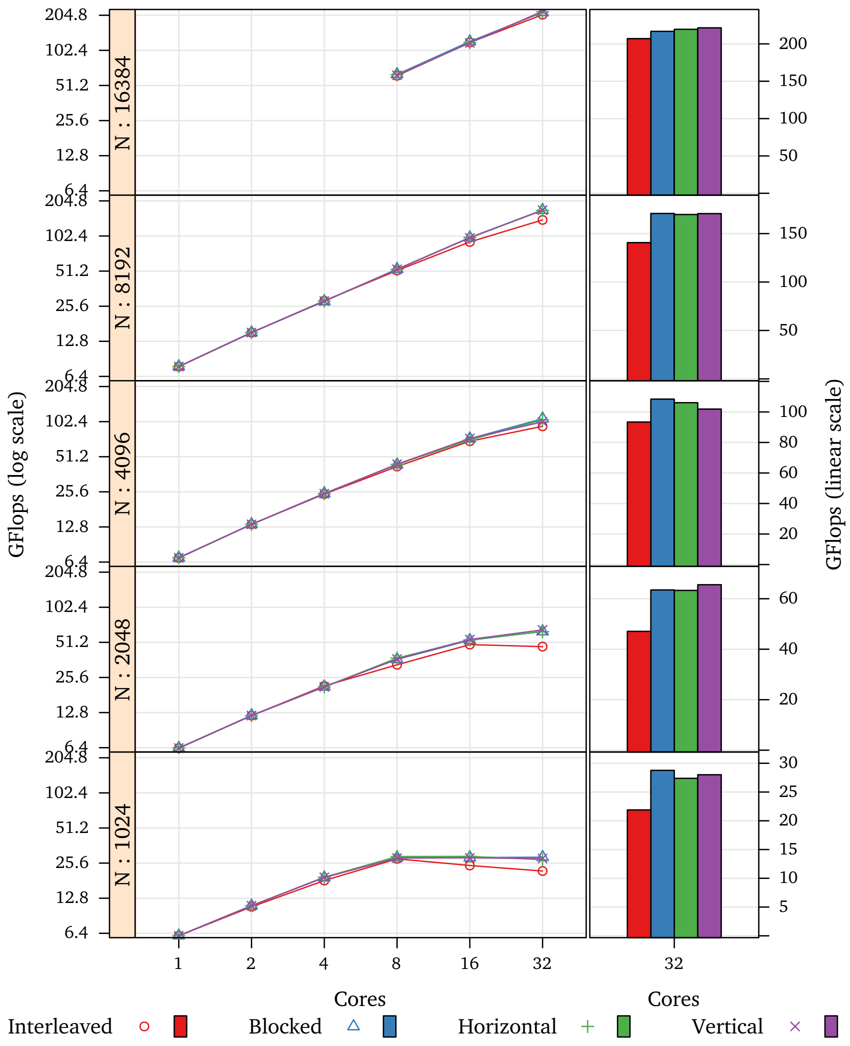


Figure F.11: Strong scalability and performance with 32 cores of the NUMA-unaware Strassen-Winograd algorithm, the executions with a blocked data distribution, the executions with horizontal distribution, and the executions with vertical distribution.

Cores	N[a]	GF[b]	GF[b]	GF[b]	GF[b]	Aff.[c] (%)	Aff.[c] (%)	Aff.[c] (%)	Aff.[c] (%)
1	1024	6.1	6.1	6.1	6.1	100	100	100	100
1	2048	6.3	6.4	6.4	6.4	100	100	100	100
1	4096	6.9	7.0	7.0	7.0	100	100	100	100
1	8192	7.8	7.8	7.8	7.8	100	100	100	100
2	1024	10	11	11	11	100	100	100	100
2	2048	12	12	12	12	100	100	100	100
2	4096	13	13	13	13	100	100	100	100
2	8192	15	15	15	15	100	100	100	100
4	1024	18	19	19	19	100	100	100	100
4	2048	21	21	21	21	100	100	100	100
4	4096	24	24	24	24	100	100	100	100
4	8192	28	28	28	28	100	100	100	100
8	1024	27	28	29	28	50	67	68	67
8	2048	32	36	37	36	50	63	66	63
8	4096	42	43	44	44	50	63	67	66
8	8192	51	53	52	53	50	72	73	72
8	16384	62	63	63	62	50	66	62	65
16	1024	24	28	29	28	25	50	49	44
16	2048	48	53	53	54	25	46	49	52
16	4096	69	73	71	73	25	45	53	48
16	8192	91	99	99	98	25	57	58	56
16	16384	119	121	118	118	25	47	45	45
32	1024	21	28	27	28	12	41	56	39
32	2048	47	63	63	65	12	36	45	34
32	4096	93	108	106	101	12	44	46	40
32	8192	140	170	169	170	12	38	38	36
32	16384	207	216	219	221	12	36	35	37

Distribution: Interleaved | Blocked | Horizontal | Vertical

[a] Matrix side size.
[b] GFlops.
[c] Mean memory affinity.

Table F.4: Performance summary of Strassen-Winograd with several data distributions.

Effectiveness of the NUMA Scheduling Policy

To evaluate how much memory affinity and memory load balancing affect performance, we have selected the blocked distribution and made additional measurements with the “bad” schedulers. Figure F.12 shows the scalability of each case with the same blocking size as the NUMA scheduler.

Notice that on this problem the “bad” schedulers perform better than the NUMA-unaware scheduler. While the NUMA-unaware execution interleaves all the data between the nodes in round-robin, the other two scheduling algorithms have an effect on the placement of the data. In particular, under the “bad” schedulers, the temporary arrays get placed in the memories of the NUMA nodes where their are initializations are executed. Therefore, the accesses to initialize the temporary data will always be affine and thus it is not possible to enforce the full potential of those policies. However, this demonstrates that determining the mapping of the units of distribution dynamically can be beneficial to performance. In fact, the tasks that initialize the temporary data are the most sensitive to affinity. Hence, the benefit surpasses the effects that these schedulers have over the affinity of the rest of tasks and parameters.

Performance Compared to Other Implementations

To compare the performance of the NUMA-aware implementation to other programming models, we have made additional measurements with the blocked implementation from chapter 3 that starts in page 68, and the OpenMP version of the same section that uses task nesting.

Figure F.13 shows the strong scalability of each implementation with several problem sizes with the best performing block size in each case. The series labeled “Interleaved” corresponds to the NUMA-unaware executions. The series labeled “Blocked Layout” corresponds to the implementation from the blocks-based programming model with the memory interleaved, and the “OpenMP” series corresponds to the OpenMP version with task nesting. Notice that all the SMPSs versions outperform the OpenMP implementation due to their ability to exploit more parallelism at a coarser granularity.

NUMA has an important impact on performance for the smallest problem sizes. However it decreases as the problem size increases since the matrix multiplication tasks become more dominating and are less sensitive to NUMA. Figure F.14 shows the floating point performance difference between the NUMA-unaware executions with the blocked layout and the NUMA-aware executions with the flat layout. For block of 256 elements per side, the executions with the blocked data layout and the memory interleaved outperform the executions with the flat layout and the NUMA-aware scheduler. However, for bigger block sizes, the NUMA executions perform better.

Table F.5 summarizes the scalability results numerically.

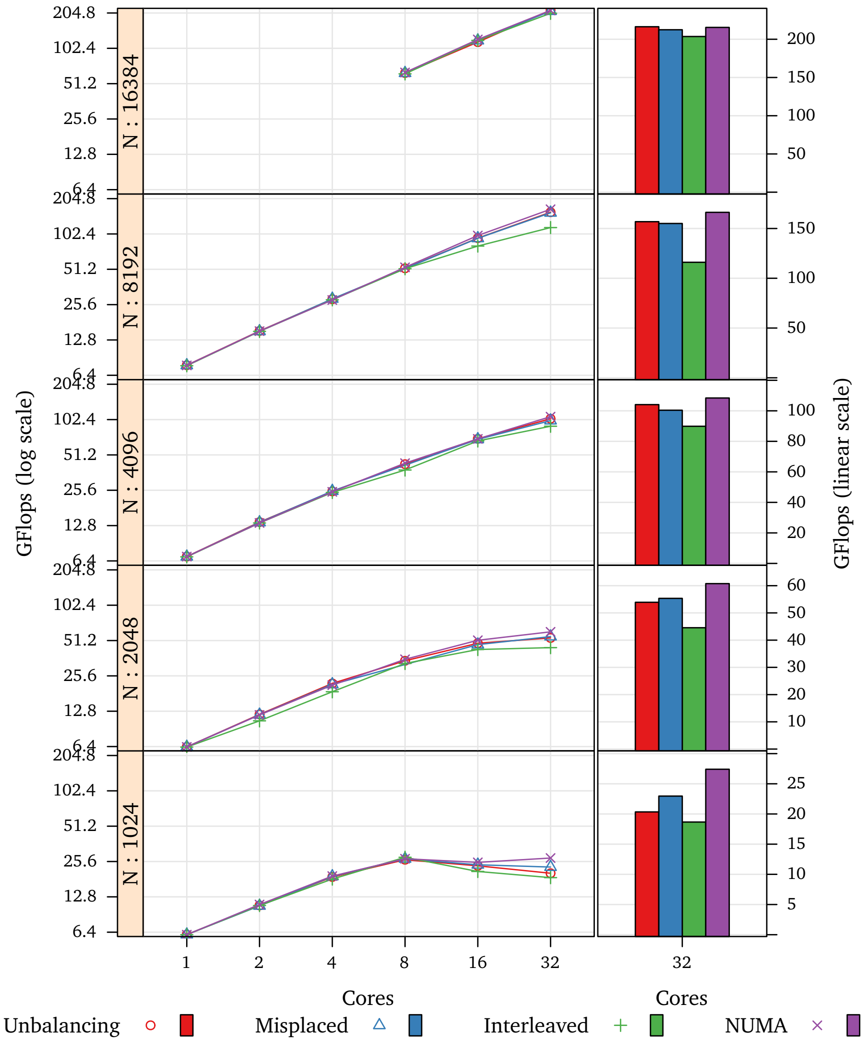


Figure F.12: Strong scalability and performance with 32 cores the Strassen-Winograd algorithm with memory overloading scheduler, the misplacing scheduler, the NUMA-unaware scheduler, and the NUMA scheduler.

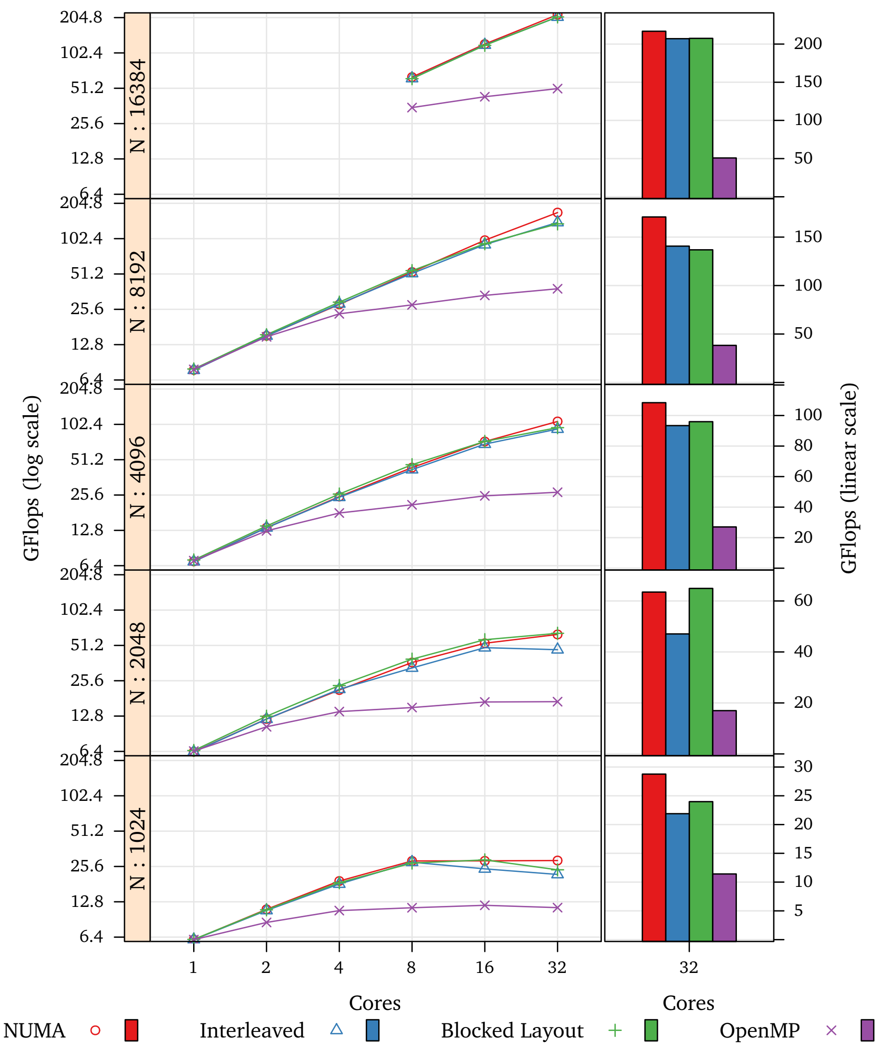


Figure F.13: Strong scalability and performance with 32 cores of the Strassen-Winograd algorithm with memory overloading scheduler, the misplacing scheduler, the NUMA-unaware scheduler, and the NUMA scheduler.

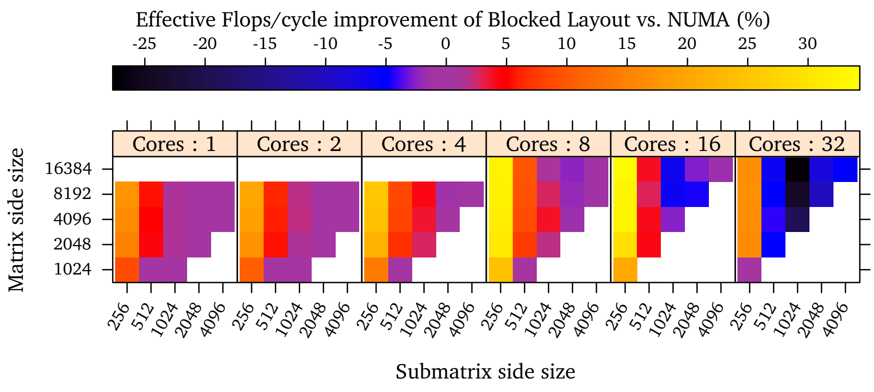


Figure F.14: Difference between the mean Strassen-Winograd task performance of the NUMA-unaware execution with blocked layout and the NUMA executions with horizontal distribution.

Cores	N[a]	GF[b]	GF[b]	GF[b]	GF[b]	FPC[c]	FPC[c]	FPC[c]	FPC[c]	Eff.[d] (%)	Eff.[d] (%)	Eff.[d] (%)	Eff.[d] (%)
1	1024	6.1	6.1	6.1	6.1	3.9	3.9	3.5	3.9	99	99	99	99
1	2048	6.4	6.3	6.5	6.4	3.6	3.6	3.3	3.6	99	99	98	99
1	4096	7.0	6.9	7.2	7.1	3.5	3.4	3.2	3.5	99	99	98	99
1	8192	7.8	7.8	7.9	7.9	3.4	3.4	3.1	3.4	99	99	98	99
2	1024	11	10	10	8.5	3.5	3.4	3.4	2.8	92	91	91	83
2	2048	12	12	12	10	3.2	3.1	3.3	3.2	97	98	98	83
2	4096	13	13	13	12	3.4	3.4	3.2	3.1	98	98	98	92
2	8192	15	15	15	14	3.3	3.3	3.4	3.0	99	99	99	97
4	1024	19	18	18	10	3.2	3.2	2.7	2.5	87	82	94	60
4	2048	21	21	23	13	2.9	3.0	3.1	2.3	95	94	96	74
4	4096	24	24	26	17	3.3	3.2	3.4	2.9	96	96	96	72
4	8192	28	28	29	23	3.2	3.2	3.3	2.8	97	98	99	83
8	1024	28	27	27	11	3.1	2.4	3.2	2.1	68	87	64	40
8	2048	36	32	39	15	2.8	3.3	2.8	2.8	88	74	92	37
8	4096	43	42	46	21	3.1	3.1	3.2	2.6	89	88	93	49
8	8192	53	51	54	27	3.0	3.0	3.1	2.5	98	98	98	58
8	16384	63	62	61	35	3.5	3.4	3.4	2.5	99	98	98	65
16	1024	28	24	29	11	1.9	1.7	1.9	1.9	64	64	79	24
16	2048	53	48	57	16	2.6	2.4	2.6	2.6	75	76	84	22
16	4096	73	69	73	25	3.0	2.9	2.9	2.4	80	80	84	32
16	8192	99	91	92	33	2.9	2.8	2.8	2.3	97	94	97	40
16	16384	121	119	118	43	3.4	3.3	3.3	2.3	96	97	97	46
32	1024	28	21	23	11	1.9	1.5	1.7	1.8	32	33	39	12
32	2048	63	47	64	16	2.5	2.2	2.2	2.6	48	45	68	11
32	4096	108	93	95	27	2.9	2.7	2.3	2.3	72	59	74	19
32	8192	170	140	136	38	2.9	3.1	3.1	2.1	90	73	74	26
32	16384	216	207	207	50	3.3	3.2	3.1	2.0	93	91	94	33

Implementation: NUMA Interleaved Blocked Layout OpenMP

[a] Matrix side size.
[b] GFlops.
[c] Mean floating point operations per cycle while running tasks.
[d] Mean time that threads spend running tasks.

Table F.5: Performance summary of the Strassen-Winograd implementations.

F.4 High Performance Linpack

High Performance Linpack is a benchmark that has been presented in chapter 4 in the section that starts in page 127. The official implementation of the HPL benchmark uses MPI and is used to evaluate the performance of the computers that participate in the Top 500 ranking list. It consists of computation intensive parts and data intensive parts. Therefore, it is a good reference to validate our ability to exploit NUMA affinity.

Units of Distribution

The HPL benchmark is composed of two algorithms. One calculates the LU factorization of a matrix with partial pivoting. This algorithm is the one that takes the most computation time. The other algorithm uses the result to solve the equation system for a particular vector.

Its tasks do not access the matrix in perfectly regular blocks as is the case of Cholesky. Instead, the shape of the regions depends on the type of task, and the iteration of the outermost loop.

On one hand, the most computationally expensive tasks are the ones that perform matrix multiplications. And on the other hand, the most memory bandwidth demanding tasks are the ones that perform the partial pivoting. However, the computation of the blocks of the diagonal are demanding on both ways, since they actually solve the same problem as the main algorithm, but over a smaller data set.

One of the most bandwidth demanding operations of the algorithm is finding the pivots, which consist of scanning the columns of the matrix one by one, and is performed by the task that solves a block from the diagonal. The affinity of this operation could be favored by distributing the data by vertical panels. This distribution also favors the affinity of the pivoting operations, since they are performed over pairs of row segments of the same vertical panels.

For completeness we have measured the performance of the algorithm with the matrix distributed in horizontal panels, in vertical panels and by blocks. The widths and/or heights of each case correspond to the task block size.

Effects of Data Placement on Performance

Figure F.15 shows the performance and strong scalability of each distribution with the best blocking size for each problem size, and table F.6 shows the corresponding numerical data.

Notice that due to the computational complexity of the problem, as we increase the problem size, floating point performance becomes the dominating factor. Therefore, as we increase the problem size, the benefits of exploiting NUMA diminishes, and both the interleaved distribution and the rest tend to perform similarly. However for N up to 16386, the vertical distribution performs better than the rest. The second best performer is the blocked distribution, and then the interleaved executions. The worst performer is the horizontal distribution.

Since this code uses column-major order, the vertical distribution is the least affected by the page size granularity, and the horizontal distribution the most affected. In addition, the vertical distribution favors the NUMA affinity of finding the pivots and performing the pivoting, which are the most memory bandwidth demanding operations. The lower performance of the executions with the horizontal distribution compared to the execution with interleaving are due to the restrictions that the page granularity impose over the distribution and thus the total memory bandwidth.

Effectiveness of the NUMA Scheduling Policy

To evaluate how much memory affinity and memory load balancing affect performance on this code, we have selected the vertical distribution and made additional measurements with the scheduler that misplaces tasks, and the scheduler that tries to overload the memories. Figure F.16 shows the scalability of each case with the best blocking size of the NUMA scheduler. Notice that except for the smallest problem size, NUMA affinity has an important impact on performance. For the biggest problem size, the blocking size is big enough that the multiplication tasks through the exploitation of temporal locality are much less dependent of NUMA affinity.

Performance Compared to Other Implementations

To compare the performance of the NUMA-aware implementation to other programming models, we have made additional measurements with the official MPI version, and the parallel MKL version.

Figure F.17 shows the strong scalability of each implementation with several problem sizes with the best performing block size for the SMPSs versions and the official MPI version, and the automatically chosen one for the MKL version. The series labeled “Interleaved” corresponds to the NUMA-unaware executions. The series labeled “NUMA” corresponds to the SMPSs executions using the NUMA-aware scheduler and the vertical distribution. The series labeled “MPI” corresponds to the official implementation made in MPI. The “MKL” series corresponds to the version that comes with MKL.

Notice that the MPI version has a significant advantage on the two smallest problem sizes. Under those conditions the overhead in the main thread of the NUMA executions with 32 cores ranges between 60% and 80%, and the effective parallelism from 40% to 60%. These parameters clearly show that the NUMA execution is being slowed down by too much overhead. While the MPI performance is also low, the communication overhead seems to be offset its ability to generate the work in parallel. However, for the biggest problem sizes, the NUMA executions manage to slightly outperform the reference implementation.

The MKL version, does not scale as well as all the others, and therefore with 32 cores it ends up last in every case.

Table F.7 summarizes those findings numerically. Note that the parallel MKL executions with 2 cores, despite using the same base linear algebra library, did not produce correct results and therefore have been discarded.

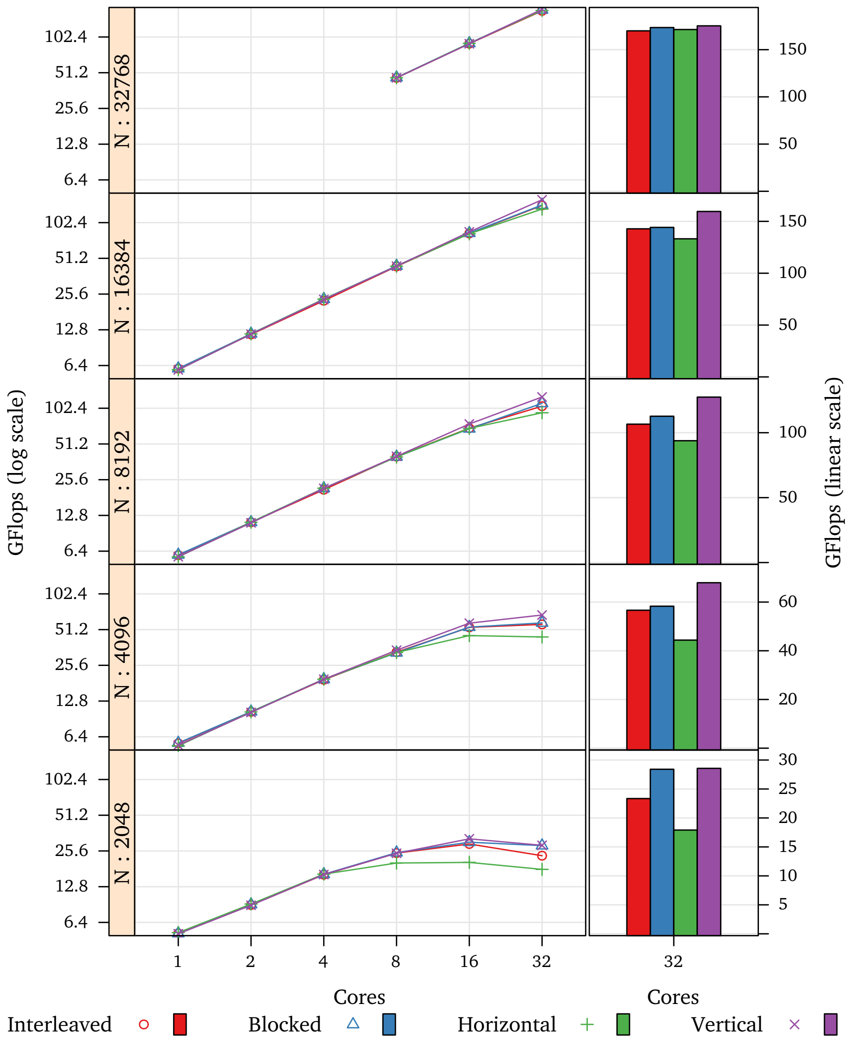


Figure F.15: Strong scalability and performance with 32 cores of the NUMA-unaware HPL algorithm, the executions with a blocked data distribution, the executions with horizontal distribution, and the executions with vertical distribution.

Cores	N[a]	GF[b]	GF[b]	GF[b]	GF[b]	Aff.[c] (%)	Aff.[c] (%)	Aff.[c] (%)	Aff.[c] (%)
1	2048	5.2	5.2	5.3	5.1	100	100	100	100
1	4096	5.6	5.7	5.4	5.4	100	100	100	100
1	8192	5.9	6.0	5.7	5.7	100	100	100	100
1	16384	6.0	6.0	5.9	5.9	100	100	100	100
2	2048	9.0	9.1	9.1	8.9	100	100	100	100
2	4096	10	10	10	10	100	100	100	100
2	8192	11	11	11	11	100	100	100	100
2	16384	11	11	11	11	100	100	100	100
4	2048	16	16	16	16	100	100	100	100
4	4096	19	19	19	19	100	100	100	100
4	8192	21	21	21	21	100	100	100	100
4	16384	22	23	23	23	100	100	100	100
8	2048	24	24	20	24	50	83	62	81
8	4096	33	32	32	34	50	68	62	89
8	8192	40	39	39	40	50	65	62	88
8	16384	43	44	43	43	50	65	64	88
8	32768	46	46	46	46	50	66	64	88
16	2048	29	30	20	32	25	79	52	72
16	4096	53	53	45	58	25	54	34	67
16	8192	69	68	69	75	25	43	36	68
16	16384	82	83	82	85	25	45	42	70
16	32768	89	90	90	89	25	47	47	84
32	2048	23	28	17	28	12	76	54	39
32	4096	56	58	44	67	12	44	28	65
32	8192	106	112	93	127	12	37	21	67
32	16384	142	144	133	159	12	33	26	67
32	32768	169	173	171	175	12	35	31	64

Distribution: Interleaved Blocked Horizontal Vertical

[a] Matrix side size.
[b] Gigaflops per second.
[c] Mean memory affinity.

Table F.6: Performance summary of HPL with four data distributions.

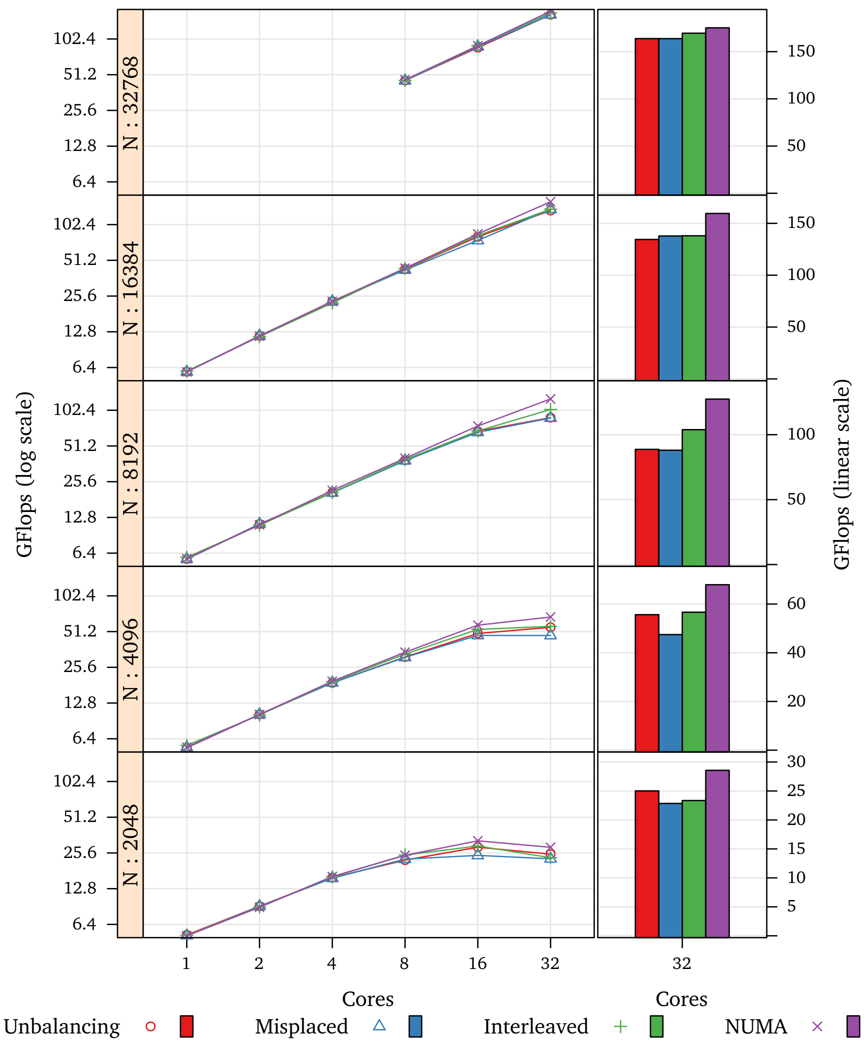


Figure F.16: Strong scalability and performance with 32 cores of HPL with memory overloading scheduler, the misplacing scheduler, the NUMA-unaware scheduler, and the NUMA scheduler.

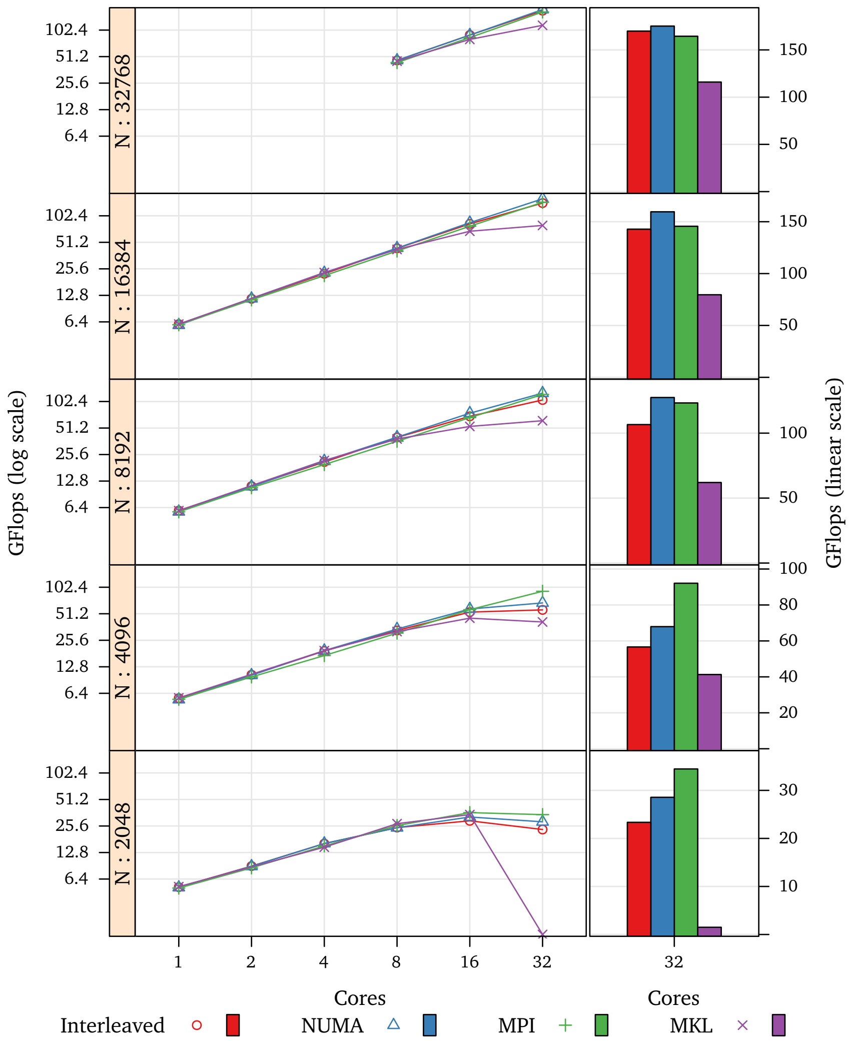


Figure F.17: Strong scalability and performance with 32 cores of the HPL algorithm with several variants under SMPSs, OpenMP and the parallel MKL.

Cores	N[a]	GF[b]	GF[b]	GF[b]	GF[b]	FPC[c]	FPC[c]	FPC[c]	Eff.[d]	Eff.[d]	Eff.[d]
									(%)	(%)	(%)
1	2048	5.2	5.1	5.1	5.2	3.3	3.3	3.3	99	99	99
1	4096	5.6	5.4	5.5	5.7	3.5	3.4	3.6	99	99	99
1	8192	5.9	5.7	5.7	5.9	3.7	3.6	3.7	99	99	99
1	16384	6.0	5.9	5.9	6.0	3.8	3.7	3.8	99	99	99
2	2048	9.0	8.9	8.6	∅	3.0	3.0	∅	93	93	∅
2	4096	10	10	9.8	∅	3.4	3.3	∅	96	98	∅
2	8192	11	11	10	∅	3.6	3.5	∅	97	98	∅
2	16384	11	11	11	∅	3.8	3.7	∅	97	99	∅
4	2048	16	16	15	14	2.8	2.8	2.5	90	91	93
4	4096	19	19	17	19	3.2	3.2	3.2	95	96	96
4	8192	21	21	19	21	3.4	3.5	3.5	97	97	98
4	16384	22	23	21	23	3.6	3.7	3.7	98	98	98
8	2048	24	24	25	27	2.4	2.5	2.7	82	78	82
8	4096	33	34	30	32	2.8	2.9	2.9	91	92	89
8	8192	40	40	36	38	3.3	3.4	3.2	95	94	93
8	16384	43	43	40	42	3.5	3.6	3.5	97	96	95
8	32768	46	46	43	45	3.7	3.7	3.7	98	98	97
16	2048	29	32	36	34	1.9	2.2	2.3	63	59	63
16	4096	53	58	56	45	2.4	2.7	2.4	87	85	77
16	8192	69	75	67	53	3.0	3.2	2.5	91	93	83
16	16384	82	85	77	68	3.4	3.5	3.0	95	96	89
16	32768	89	89	84	80	3.6	3.7	3.4	98	96	92
32	2048	23	28	34	1.5	1.4	1.8	1.0	37	35	27
32	4096	56	67	92	41	1.8	2.3	1.4	65	58	63
32	8192	106	127	123	61	2.4	2.9	1.7	88	87	73
32	16384	142	159	145	79	3.0	3.3	2.1	92	95	77
32	32768	169	175	164	115	3.4	3.6	2.8	97	96	81

Implementation: Interleaved NUMA MPI MKL

[a] Matrix side size.
[b] Gigaflops per second.
[c] Mean floating point operations per cycle while running tasks.
[d] Mean time that threads spend running tasks.

Table F.7: Performance summary of the HPL implementations.

Bibliography

Tarek S. Abdelrahman and Sum Huynh. Exploiting task-level parallelism using pTask. In *Proceedings of the 2009 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 1996)*, pages 252–263, 1996.

Erika Ábrahám, Immo Grabe, Andreas Gürner, and Martin Steffen. Behavioral interface description of an object-oriented language with futures and promises. *Journal of Logic and Algebraic Programming*, In Press, Corrected Proof, 2009. ISSN 1567-8326. doi: 10.1016/j.jlap.2009.01.001.

Eitan Ben Amos. *Cilk on CC-NUMA Machines*. Tel Aviv University, 2006.

Bjarne S. Andersen, John A. Gunnels, Fred G. Gustavson, John K. Reid, and Jerzy Waśniewski. A fully portable high performance minimal storage hybrid format cholesky algorithm. *ACM Transactions on Mathematical Software*, 31:201–227, June 2005. ISSN 0098-3500. doi: 10.1145/1067967.1067969.

Ed Anderson, Zhaojun Bai, Jack Dongarra, A. Greenbaum, A. McKenney, Jeremy Du Croz, Sven Hammarling, James Demmel, Christian H. Bischof, and Danny C. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In Joanne L. Martin, Daniel V. Pryor, and Gary Montry, editors, *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, Supercomputing '90, pages 2–11, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press. ISBN 0-89791-412-0.

Jörg Arndt. *Matters Computational: Ideas, Algorithms, Source Code*. Springer, 1st edition, 2011. ISBN 978-3-642-14763-0.

David H. Bailey. FFTs in external or hierarchical memory. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, Supercomputing '89, pages 234–242, New York, NY, USA, 1989. ACM. ISBN 0-89791-341-8. doi: 10.1145/76263.76288.

Henry G. Baker, Jr. and Carl Hewitt. The incremental garbage collection of processes. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*, pages 55–59, New York, NY, USA, 1977. ACM. doi: 10.1145/800228.806932.

Vasanth Balasundaram and Ken Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. *ACM SIGPLAN Notices*, 24 (7):41–53, 1989. ISSN 0362-1340. doi: 10.1145/74818.74822.

Utpal K. Banerjee. Data dependence in ordinary programs. Master's thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, November 1976. Technical report number UIUCDCS-R-76-837.

Utpal K. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Norwell, MA, USA, 1988. ISBN 0898382890.

Pieter Bellens, Josep M Perez, Rosa M Badia, and Jesus Labarta. Cellss: a programming model for the Cell BE architecture. In *Proceedings of the ACM/IEEE SC 2006 Conference*, Tampa, FL, USA, Nov 2006. ISBN 0-7695-2700-0. doi: 10.1109/SC.2006.17.

Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975. ISSN 0001-0782. doi: 10.1145/361002.361007.

John Bircsak, Peter Craig, RaeLyn Crowell, Zarka Cvetanovic, Jonathan Harris, C. Alexander Nelson, and Carl D. Offner. Extending OpenMP for NUMA machines. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7803-9802-5.

William Joseph Blume. Success and limitations in automatic parallelization of the Perfect Benchmarks^{TM} programs. Master's thesis, University of Illinois at Urbana-Champaign, 1992.

Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. Dag-consistent distributed shared memory. In *Proceedings of the 10th International Parallel Processing Symposium*, IPPS '96, pages 132–141, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7255-2.

Brice Boyer, Jean-Guillaume Dumas, Clément Pernet, and Wei Zhou. Memory efficient scheduling of Strassen-Winograd's matrix multiplication algorithm. In *Proceedings of the 2009 international symposium on Symbolic and algebraic computation*, ISSAC '09, pages 55–62, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-609-0. doi: 10.1145/1576702.1576713.

Michael G. Burke and Ron K. Cytron. Interprocedural dependence analysis and parallelization. *ACM SIGPLAN Notices*, 39(4):139–154, 2004. ISSN 0362-1340. doi: 10.1145/989393.989411.

Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007. ISSN 1094-3420. doi: 10.1177/1094342007078442.

Bradford L. Chamberlain, Sung-Eun Choi, Steven J. Deitz, and Angeles Navarro. User-defined parallel zippered iterators in Chapel. In *PGAS 2011: Fifth Conference on Partitioned Global Address Space Programming Models*, October 2011.

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: 10.1145/1094811.1094852.

Arunodaya Chatterjee. Futures: a mechanism for concurrency among objects. In *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 562–567, New York, NY, USA, 1989. ACM. ISBN 0-89791-341-8. doi: 10.1145/76263.76326.

T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation: A performance view. *IBM J. Res. Dev.*, 51(5):559–572, September 2007. ISSN 0018-8646. doi: 10.1147/rd.515.0559.

Jan Ciesko, Sergi Mateo, Xavier Teruel, Vicenç Beltran, and Xavier Martorell. Task-parallel reductions in OpenMP and OmpSs. To be published in IWOMP14.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. ISBN 0262033844, 9780262033848.

Jack J. Dongarra. The LINPACK benchmark: An explanation. In *Proceedings of the 1st International Conference on Supercomputing*, pages 456–474, London, UK, 1988. Springer-Verlag. ISBN 3-540-18991-2.

Jack J. Dongarra and David W. Walker. The design of linear algebra libraries for high performance computers. LAPACK Working Note 58, Department of Computer Science, University of Tennessee, Knoxville, TN 37996, USA, Jun 1993. UT-CS-93-188.

Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16:1–17, March 1990. ISSN 0098-3500. doi: 10.1145/77626.79170.

Alejandro Duran, Josep M. Pérez, Eduard Ayguadé, Rosa M. Badia, and Jesús Labarta. Extending the OpenMP tasking model to allow dependent tasks. In Rudolf Eigenmann and Bronis R. de Supinski, editors, *OpenMP in a New Era of Parallelism*, volume 5004 of *Lecture Notes in Computer Science*, pages 111–122, 4th International Workshop, IWOMP 2008, West Lafayette, IN, USA, May 2008. Springer-Verlag Berlin, Heidelberg. ISBN 978-3-540-79560-5. doi: 10.1007/978-3-540-79561-2.

Alejandro Duran, Roger Ferrer, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. A proposal to extend the OpenMP tasking model with dependent tasks. *International Jornual of Parallel Programming*, 37(3):292–305, June 2009. ISSN 1573-7640. doi: 10.1007/s10766-009-0101-1.

Rudolf Eigenmann, Jay Hoeflinger, and David A. Padua. On the automatic parallelization of the Perfect Benchmarks®. *IEEE Transactions on Parallel and Distributed Systems*, 9(1):5–23, 1998. ISSN 1045-9219. doi: 10.1109/71.655238.

Tarek El-Ghazawi and Lauren Smith. PGAS: The Partitioned Global Address Space programming model. Birds of a Feather session at Supercomputing 2013, November 2013.

Roger Ferrer, Judit Planas, Pieter Bellens, Alejandro Duran, Marc Gonzalez, Xavier Martorell, Rosa M. Badia, Eduard Ayguade, and Jesus Labarta. Optimizing the exploitation of multicore processors and gpus with openmp and opencl. In Keith Cooper, John Mellor-Crummey, and Vivek Sarkar, editors, *Languages and Compilers for Parallel Computing*, volume 6548 of *Lecture Notes in Computer Science*, pages

215–229. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-19594-5. doi: 10.1007/978-3-642-19595-2_15.

Daniel P. Friedman and David S. Wise. Aspects of applicative programming for parallel processing. *IEEE Transactions on Computers*, 27(4):289–296, 1978. ISSN 0018-9340. doi: 10.1109/TC.1978.1675100.

Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on “Program Generation, Optimization, and Platform Adaptation”.

Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN ’98 Conference on Programming Language Design and Implementation*, volume 33 of *ACM SIGPLAN Notices*, pages 212–223, Montreal, Quebec, Canada, June 1998. ACM. doi: 10.1145/277652.277725.

M. Girkar and C. D. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Trans. Parallel Distrib. Syst.*, 3(2):166–178, 1992. ISSN 1045-9219. doi: 10.1109/71.127258.

Marc Gonzalez, Xavier Martorell, Eduard Ayguade, Jesus Labarta, and Nacho Navarro. Precedence relations in the OpenMP programming model. In *Proceedings of the second european workshop on OpenMP, EWOMP 2000*, pages 58–67, Edimburgh, UK, September 2000.

Marc Gonzalez, Eduard Ayguade, Xavier Martorell, and Jesus Labarta. Exploiting pipelined executions in OpenMP. In *Proceedings of the 2003 International Conference on Parallel Processing (ICPP’03)*, pages 153–160, Los Alamitos, CA, USA, October 2003. IEEE Computer Society. doi: 10.1109/ICPP.2003.1240576.

Paul Havlak and Ken Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2 (3):350–360, 1991. ISSN 1045-9219. doi: 10.1109/71.86110.

Adolfy Hoisie, Olaf Lubeck, and Harvey Wasserman. Performance and scalability analysis of teraflop-scale parallel architectures using multidimensional wavefront applications. *International Journal of High Performance Computing Applications*, 14(4):330–346, Winter 2000. doi: 10.1177/109434200001400405.

Mike Houston, Ji-Young Park, Manman Ren, Timothy Knight, Kayvon Fatahalian, Alex Aiken, William Dally, and Pat Hanrahan. A portable runtime interface for multi-level memory hierarchies. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP ’08, pages 143–152, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-795-7. doi: 10.1145/1345206.1345229.

Sum Huynh. Exploiting task-level parallelism automatically using pTask. Master’s thesis, Department of Electrical and Computer Engineering, University of Toronto, 1996.

International Organization for Standardization and International Electrotechnical Commission. International standard; ISO/IEC 9899:1999, International Organization for Standardization, Geneva, Switzerland, 1999.

Darren J. Kerbyson, Michael Lang, and Scott Pakin. Adapting wave-front algorithms to efficiently utilize systems with deep communication hierarchies. *Parallel Computing*, 37(9):550 – 561, 2011. ISSN 0167-8191. doi: 10.1016/j.parco.2011.02.008. Emerging Programming Paradigms for Large-Scale Scientific Computing.

David J. Kuck, Robert H. Kuhn, David A. Padua, B. Leasure, and Michael Wolfe. Dependence graphs and compiler optimizations. In *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–218, New York, NY, USA, 1981. ACM. ISBN 0-89791-029-X. doi: http://doi.acm.org/10.1145/567532.567555.

Jesus Labarta. Starss: a programming model for the multicore era. In *PRACE Workshop: New Languages & Future Technology Prototypes*, Leibniz Supercomputing Centre, Garching, Germany, March 2010.

Jesus Labarta, Sergi Girona, Vincent Pillet, Toni Cortes, and Luis Gregoris. DiP: A parallel program development environment. In *Proceedings of the 2nd International Euro-Par Conference*, volume 2, pages 665–674, August 1996.

Richard P. LaRowe Jr. *Page placement for non-uniform memory access time (NUMA) shared memory multiprocessors*. Number 13 in Technical report (Duke University. Dept. of Computer Science). Department of Computer Science, Duke University, March 1991. PhD. Dissertation.

Ewing Lusk and Anthony Chan. Early experiments with the openmp/mpi hybrid programming model. In *OpenMP in a New Era of Parallelism*, pages 36–47. Springer, 2008.

Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The HPC Challenge (HPCC) benchmark suite. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 213, New York, NY, USA, 2006. ACM. ISBN 0-7695-2700-0. doi: 10.1145/1188455.1188677.

Michael Marchetti, Leonidas I. Kontothanassis, Ricardo Bianchini, and Michael L. Scott. Using simple page placement policies to reduce the cost of cache fills in coherent shared-memory systems. In *Proceedings of the 9th International Symposium on Parallel Processing*, IPPS '95, pages 480–485, Washington, DC, USA, 1995. IEEE Computer Society. ISBN 0-8186-7074-6.

Vladimir Marjanović, Jesús Labarta, Eduard Ayguadé, and Mateo Valero. Overlapping communication and computation by using a hybrid mpi/smpss approach. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 5–16, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0018-6. doi: 10.1145/1810085.1810091.

John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

Message Passing Interface Forum. MPI: A message-passing interface standard, version 2.2. Specification, September 2009.

Donald R. Morrison. PATRICIA—practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, October 1968. ISSN 0004-5411. doi: 10.1145/321479.321481.

Dimitrios S. Nikolopoulos, Theodore S. Papatheodorou, Constantine D. Polychronopoulos, Jesús Labarta, and Eduard Ayguadé. A transparent runtime data distribution engine for OpenMP. *Sci. Program.*, 8:143–162, August 2000. ISSN 1058-9244.

Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998. ISSN 1061-7264. doi: 10.1145/289918.289920.

OpenMP Architecture Review Board. OpenMP application program interface, version 3.0. Specification, May 2008.

OpenMP Architecture Review Board. OpenMP application program interface, version 4.0. Specification, July 2013.

Yunheung Paek, Jay Hoeflinger, and David Padua. Simplification of array access patterns for compiler optimizations. *ACM SIGPLAN Notices*, 33(5):60–71, 1998. ISSN 0362-1340. doi: 10.1145/277652.277664.

Yunheung Paek, Jay Hoeflinger, and David Padua. Efficient and precise array access analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24 (1):65–109, 2002. ISSN 0164-0925. doi: 10.1145/509705.509708.

Liang Peng, Weng Fai Wong, Ming Dong Feng, and Chung Kwong Yuen. SilkRoad: a multithreaded runtime system with software distributed shared memory for SMP clusters. In *Cluster Computing, 2000. Proceedings. IEEE International Conference on*, pages 243 –249, 2000. doi: 10.1109/CLUSTR.2000.889067.

Josep M. Perez, Rosa M. Badia, and Jesus Labarta. Including SMP in grids as execution platform and other extensions in GRID superscalar. In *Second IEEE International Conference on e-Science and Grid Computing 2006 (e-Science '06)*, December 2006. doi: 10.1109/E-SCIENCE.2006.261144.

Josep M. Perez, Pieter Bellens, Rosa M. Badia, and Jesus Labarta. CellSs: Making it easier to program the Cell Broadband Engine processor. *IBM Journal of Research and Development*, 51(5):593–604, September 2007. ISSN 0018-8646. doi: 10.1147/rd.515.0593.

Josep M. Perez, Rosa M. Badia, and Jesus Labarta. A dependency-aware task-based programming environment for multi-core architectures. In Causal Productions, editor, *Proceedings of the 2008 IEEE International Conference on Cluster Computing*, pages 142–151, September 2008. ISBN 978-1-4244-2639-3. doi: 10.1109/CLUSTR.2008.4663765.

Josep M. Perez, Rosa M. Badia, and Jesus Labarta. Handling task dependencies under strided and aliased references. In *Proc. of the 24th ACM Int. Conf. on Supercomputing*, ICS '10, pages 263–274, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0018-6. doi: 10.1145/1810085.1810122.

Rolf Rabenseifner. Hybrid parallel programming: Performance problems and chances. In *Proceedings of the 45th CUG Conference 2003*, May 2003.

Rice University. High performance fortran language specification. *SIGPLAN Fortran Forum*, 12(4):1–86, Dec. 1993. ISSN 1061-7264. doi: 10.1145/174223.158909.

Goldwyn Rodrigues. Taming the OOM killer. LWN, February 2009. URL `http://lwn.net/Articles/317814/`.

Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. Hybrid analysis: static & dynamic memory reference analysis. In *ICS '02: Proceedings of the 16th international conference on Supercomputing*, pages 274–284, New York, NY, USA, 2002. ACM. ISBN 1-58113-483-5. doi: 10.1145/514191.514229.

Valentina Salapura, Matthias Blumrich, and Alan Gara. Improving the accuracy of snoop filtering using stream registers. In *MEDEA '07: Proceedings of the 2007 workshop on Memory performance*, pages 25–32, New York, NY, USA, 2007. ACM. ISBN 978-1-9593-807-7. doi: 10.1145/1327171.1327174.

Oliver Sinnen, Jsun Pe, and Alexei Vladimirovich Kozlov. Support for fine grained dependent tasks in OpenMP. In *IWOMP '07: Proceedings of the 3rd international workshop on OpenMP*, pages 13–24. Springer-Verlag Berlin, Heidelberg, 2008. ISBN 978-3-540-69302-4. doi: 10.1007/978-3-540-69303-1_2.

James E. Smith and Gurindar S. Sohi. The microarchitecture of superscalar processors. *Proceedings of the IEEE*, 83(12):1609–1624, dec 1995. ISSN 0018-9219. doi: 10.1109/5.476078.

Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 14 (3):354–356, 1969.

Enric Tejedor and Rosa M. Badia. COMP Superscalar: Bringing GRID Superscalar and GCM Together. In *Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, CCGrid '08, Lyon, France, pages 185–193, May 2008. ISBN 978-0-7695-3156-4. doi: 10.1109/CCGRID.2008.104.

T.I.S. Committee. Tool interface standard (TIS) executable and linking format (ELF) specification version 1.2, May 1995.

UPC Consortium. UPC language specifications, v1.2. Tech Report LBNL-59208, Lawrence Berkeley National Lab, 2005.

Shmuel Winograd. On multiplication of 2 X 2 matrices. *Linear Algebra and Application*, 4:381–388, 1971.

Michael Wolfe. Engineering a data dependence test. *Concurrency: Practice and Experience*, 5(7):603–622, 1993. ISSN 1096-9128. doi: 10.1002/cpe.4330050706.

Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance Java dialect. *Concurrency: Practice and Experience*, 10(11–13):825–836, Sep. 1998. ISSN 1096-9128. doi: 10.1002/(SICI)1096-9128(199809/11)10:11/13<825::AID-CPE383>3.0.CO;2-H. Special Issue: Java for High-performance Network Computing.