Cyril Banino-Rokkones

Algorithmic and Scheduling Techniques for Heterogeneous and Distributed Computing

Thesis for the degree philosophiae doctor

Trondheim, March 2007

Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics
and Electrical Engineering
Department of Computer and Information Science

NTNU
Innovation and Creativity

NTNU
Norwegian University of Science and Technology

Thesis for the degree philosophiae doctor

Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Computer and Information Science

ISBN 978-82-471-1075-1 (printed version)
ISBN 978-82-471-1089-8 (electronic version)
ISSN 1503-8181

Doctoral theses at NTNU, 2007:49

Printed by NTNU-trykk

Abstract

The computing and communication resources of high performance computing systems are becoming heterogeneous, are exhibiting performance fluctuations and are failing in an unforeseeable manner. The Master-Slave (MS) paradigm, that decomposes the computational load into independent tasks, is well-suited for operating in these environments due to its loose synchronization requirements. The application tasks can be computed in any order, by any slave, and can be resubmitted in case of slave failures. Although, the MS paradigm naturally adapts to dynamic and unreliable environments, it nevertheless suffers from a lack of scalability.

This thesis provides models, techniques and scheduling strategies that improve the scalability and performance of MS applications. In particular, we claim that deploying multiple masters may be necessary to achieve scalable performance. We address the problem of finding the most profitable locations on a heterogeneous Grid for hosting a given number of master processes, such that the total task throughput of the system is maximized. Further, we provide distributed scheduling strategies that better adapt to system load fluctuations than traditional MS techniques. Our strategies are especially efficient when communication is expensive compared to computation (which constitutes the difficult case).

Furthermore, this thesis investigates also the suitability of MS scheduling techniques for the parallelization of stencil code applications. These applications are usually parallelized with domain decomposition methods, that are highly scalable, but rather impractical for dealing with heterogeneous, dynamic and unreliable environments. Our experimental results with two scientific applications show that traditional MS tasking techniques can successfully be applied to stencil code applications when the master is used to control the parallel execution. If the master is used as a data access point, then deploying multiple masters becomes necessary to achieve scalable performance.

Preface

This doctoral thesis is submitted to the Norwegian University of Science and Technology (NTNU) in partial fulfillment of the requirements for the degree *Doktor Ingeniør*.

The work herein was performed at and funded by the Department of Computer and Information Science (IDI) at NTNU, under the supervision of Professor Lasse Natvig.

The thesis consists of two parts. The first part contains an introduction to the topics of the study, a summary of our contributions, and opens up for future work directions. The second part gathers the main contributions, presented as a collection of seven research papers. The layouts of the papers have been modified from their original form for the sake of presentation. Their contents have not been modified.

Acknowledgments

I am grateful to my supervisors Professor Lasse Natvig, Doctor Jørn Amundsen, and Professor Einar Rønquist for the support and advises they have given me. I thank Doctor Anne Catherine Elster for her invitation to Norway and for helping me to get started with my PhD thesis. I would also like to thank Einar M. R. Rosenvinge and Eivind Smørgrav whom I had the pleasure to work with on parts of this thesis. I want also to thank my friends and colleagues at IDI, who contributed to create a pleasant atmosphere during these four years.

I cannot thank enough Doctor Olivier Beaumont for having introduced me to the field of heterogeneous scheduling, for his guidance and support throughout the last four years, and for having invited me to a stay at the University of Bordeaux (LaBRI) that resulted in a fruitful collaboration. Thanks also to Doctor Arnaud Legrand for invaluable technical advice regarding the simulator used in this thesis.

I could never have surmounted all the obstacles that arose during these four tumultuous years, without the support of my family. I am indebted to my mother for her love and devotion during all my life. I thank my brother Jérémy for his love and friendship. I am deeply grateful to *svigers* Klara and Steinar for having included me in their family from the very beginning. And of course, I would like to thank the most important person in my life, my wife Nina, for her unconditional love and support.

Finally, I want to extend my deepest love to the two persons that I miss profoundly: To my father who guided me into life with love and kindness, and to Håvard that opened my eyes with his happiness and self-irony. Without them, I would not be who I am today. To these two persons I, with love, dedicate this thesis.

Cyril Banino-Rokkones
Trondheim, November 2006.

Contents

Part I

Context

Chapter 1

Introduction

1.1 Evolution of parallel computers

In the last decades, the field of high performance computing has seen a rapid evolution in terms of architectures, technologies and system utilization. The need for always more computing power in a multitude of domains (e.g. scientific and engineering simulations, data mining, signal and image processing, etc) has impulsed the emergence of parallel computing systems. The idea of parallel computing can be illustrated by the well-known motto: *Unity is strength*, i.e. grouping small individual forces together results in a powerful single force. Based on this principle, the precursor parallel computing machines - called *vector computers* - were born in the 1960's, and raised enthusiasm within the scientific community. These vector supercomputers were very expensive due to their special design for performance, but became quickly obsoletes because of rapid technological improvements [175].

In the mid 1990's, *massively parallel processing* systems (MPP) appeared on the high performance computing market, and became really strong competitors for traditional vector supercomputers. Most MPPs are distributed memory systems composed of hundreds or thousands relatively inexpensive processors connected together with custom designed fast interconnects. Each processor is self-contained, with its own cache and memory chips and additional supporting hardware. In the same genre, a *cluster* is a collection of workstations/PCs interconnected by a local network. The broad adoption of cluster systems results from their cost to performance ratio unmatched by any other computing system.

Figure 1.1 depicts the recent architectural trends within the top500 supercomputers [1]. Note that single processor systems cannot match the performance achieved by parallel systems, and logically disappeared from the top500 list. MPP systems on the other hand constitute the most powerful architecture, albeit the share of cluster systems is rapidly increasing. One can also observe a diminution

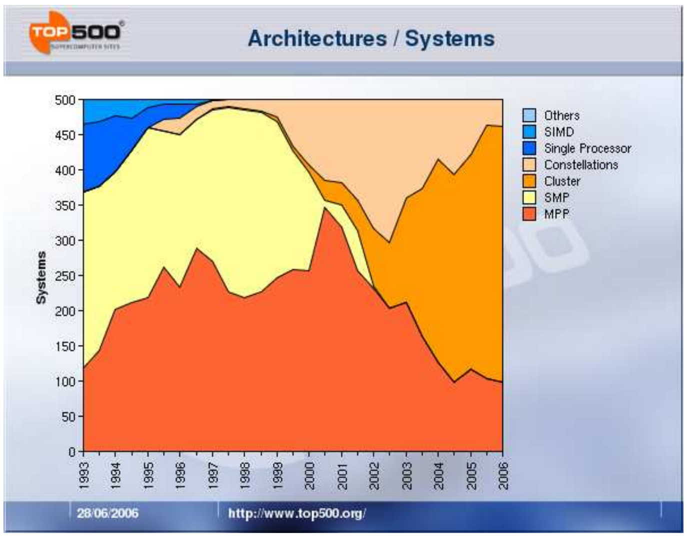


Figure 1.1: Architectures share over time (obtained from [1]).

of *constellation*[1] systems in the list.

It should be noted that this evolution has been possible only at the cost of constant and massive efforts in the development of standard software and notably of high performance libraries such as BLAS [129], MPI [94], LINPACK, LAPACK or ScaLAPACK [74]. While greater performance requires technological improvements, *portability*, that is the ability of a program to be executed on different systems, requires the use of interfaces in form of libraries.

1.2 Towards network-based computing

Although parallel computers have proven to deliver good and stable performance, financial or technical constraints limit the scale of these systems, and consequently the amount of computing power they can supply. Reproducing the successful motto at a bigger scale, the seek for always more computing power conducted to the interconnection of geographically dispersed parallel computers, and the *Grid* was born [88]. Grid computing provides the ability to perform computations at unprecedent scale by taking advantage of several networked supercomputers.

[1]The difference between a cluster and a constellation comes from the relationship between the number of nodes and the number of processors in each node of the system. If there are fewer processors per node than there are nodes in the system then you have a cluster. But if there are more processors per nodes than there are nodes in the system, then you have a constellation. This differentiation is motivated by the different approaches for programming cluster and constellation systems.

The aggregation of personal computing resources can also supply large amounts of computing power. For instance, every large institution owns hundreds of PCs interconnected by a LAN, and has hence a huge amount of potential computing power at its disposal. Livny showed that most of the workstations are often idle, and proposed Condor [178], a system for exploiting this "wasted" computing power. The Condor system has been very successful, and similar commercial solutions are now available to enterprises [60].

With the growing popularity of the Internet and the advance in technology, this idea has been expanded world wide. *Internet computing* seeks to exploit otherwise idle workstations and PCs spread over the Internet. One of the most successful Internet computing applications is the *SETI@home* project [6] whose goal is to analyze radio telescopic data, searching for signs of extraterrestrial intelligence. The success of SETI has inspired many other @home applications and impulsed an ever-growing research trend for supporting this kind of applications [92, 167, 174]. Similarly, *Peer-to-Peer* networks have also been used to run large scale applications on geographically dispersed computing resources [55, 61, 147].

The early adoption of network-based computing platforms is focused on applications that expose a large degree of parallelism with little or no coupling, and whose high computational demands cannot be met by single parallel computers [60]. They are typically implemented under the Master-Slave (MS) paradigm (depicted in greater details in Section 1.4.1). Many applications have been or can be implemented under the MS paradigm: They include Monte Carlo simulations [22], collaborative computing efforts such as SETI@home [6, 174], biological sequence comparisons [171], or also distributed problems organized by companies like Entropia [60].

It should finally be noted that, although harnessing computing power spread over computer networks is much cheaper than buying a new parallel computer, utilizing efficiently these dispersed and volatile resources turns out to be a much more complex task.

1.3 The challenges of high performance computing

1.3.1 Data movements

Delivering huge amounts of computing power is a difficult task. Bringing data quickly to the processing units is probably one of the most important and challenging issues that high performance computing must face. Interestingly, this problem is not a new one:

"In my opinion this problem of making a large memory available at reasonably short notice is much more important that of doing operations such as multiplication at high speed.” — *(Alan Turing, 1947)*

If we look back in history, the source of the problem is clear: In the last decade, processor performance has been steadily improving at a much more higher rate (55%) than memory performance (7%) [101]. This huge gap between memory performance and processor performance is popularly known as the *memory wall* [190]. Modern CPUs are so fast that memory transfers constitute the practical limitation on the processing speed. Therefore, modern computer architectures rely on a hierarchical arrangement of memory (caches) to help bridging that widening gap. Each level of the hierarchy is of higher speed and lower latency, but is of smaller size than lower levels.

In addition to the memory wall, parallel computers disclose another gap. The computational speed of the processors is typically much faster than the communication speed of the interconnect [66]. Therefore, a wealth of efforts have been made to design parallel algorithm that lessen the impact of communication. The situation is exacerbated for Grid applications communicating over high latency WAN links.

Thus, efficiently managing data movements is of tremendous importance at all the levels of modern computing systems. Undoubtedly, this issue will remain of paramount importance for future computing systems as well.

1.3.2 Heterogeneity, variability and unreliability

High performance computing systems are becoming *heterogeneous*, interconnecting resources with different hardware and software. This heterogeneity makes resource selection paramount in order to increase the sustained performance. Balancing the computational load among several processors is already difficult on homogeneous systems, it becomes even harder on heterogeneous systems [113,114].

System load fluctuations are caused by applications that compete for shared resources within the system (e.g. processors and network links). Consequently, the load and availability of the resources fluctuate over time, due to the unpredictable interactions of the users with the system. Several adaptation techniques have been elaborated with the ultimate goal being self-adaptation where the application adapts to its environment without user intervention. For instance, the AppLeS methodology [39] consists in deploying a scheduling agent that monitors the system load, utilizes performance predictions and application-specific information to dynamically generate a schedule application. Casanova et al. [53] present a task farming strategy for scheduling independent task applications onto Grid environments that adapts dynamically the number of tasks submitted to the system in function of system load fluctuations. At last, Heymann et al. [102], propose a strategy that adjusts dynamically the number of slave processes that participate to a Master-Slave computation.

When increasing the number of computing and communication resources composing the system, the number of resource failures that are likely to occur increases accordingly. Large-scale systems - composed of hundreds or thousands of processors - are hosting applications that may run for days, and in these conditions, one expects resource failures (both hardware and software) to be the rule rather than the exception. This has a direct impact for applications that must survive to resource failures [73] (more on fault tolerance in Section 3.3.3).

Thus, large-scale computing systems are heterogeneous, dynamic and unreliable environments that require adapted, flexible and robust techniques and algorithms.

1.4 Research focus

This section begins with a description of the MS paradigm, with an emphasis on why - we believe - this paradigm is well-suited for distributed, heterogeneous, dynamic and unreliable computing systems. Thereafter, we identify and expose the shortcomings that come with the MS paradigm, state the research questions studied in this thesis, and present our main research methods. At last, we conclude with the organization of the thesis.

1.4.1 The master-slave paradigm

The Master-Slave paradigm, also called Master-Worker or task farming paradigm, consists of two entities: A master process and several slave processes. The master is responsible for decomposing the computational domain into a number of smaller independent work units, usually called *tasks*, which are delegated to the slaves for parallel remote computation. The main asset of the MS paradigm is its robustness to resource failures. Its loosely coupled structure presents only one *single point of failure* - whose failure will cause an interruption of the computation - in the form of the master process. If some slave processes die, the computation can carry on with the remaining slaves. Hence, the number of slaves can be adapted dynamically to the number of available resources. If new resources appear during the computation, they can be incorporated as new slaves, and if a resource disappears (e.g. fails or is reclaimed by its owner) the tasks that were allocated to this machine are simply reallocated to other slaves [20]. Moreover, fast slaves with nothing left to do towards the end of the computation can receive unfinished tasks already delegated to other slaves. Redundant results or *replicas* are simply discarded [64]. This mechanism increases the chance to assign tasks to fast slaves, but comes at the expense of wasted computing power. Nevertheless, it has been shown that replicating the tasks only twice, leads to significant improvements [64].

In its simplest form, the MS paradigm works as follows. The master initially distributes one task to every slave, then the slaves compute their tasks and send the results back to the master, which triggers the latter to send additional tasks. As slaves execute tasks at their own paces, they will automatically request tasks proportionally to their computing speeds. This is popularly known as *self-scheduling* (also called *demand-driven* or *work-queue*). By construction, self-scheduling adapts well to the performance fluctuations of the computational resources. If a slave suddenly gets some external load, it will process tasks less rapidly, and hence request tasks less frequently. When the conditions get back to normal, the slave will ask for tasks at its maximal pace.

However, self-scheduling is not efficient for platforms composed of heterogeneous networks. When heterogeneity applies also to the communication links, resource selection strategies become necessary in order to efficiently utilize the available computing resources. Consider for instance the case where a fast slave is connected to the master via a slow communication link. The slave will process tasks faster than it receives them, and will occupy most of the master communication bandwidth. In these conditions, it may be more advantageous to serve slower slaves but that are interconnected with rapid communication links.

This brings about the communication to computation (C-C) ratio issue, common to all parallel programs. Applications being implemented under the MS paradigm must exhibit a large C-C ratio, meaning that the time required to send a task to a slave is much smaller than the time required for the slave to process it. Otherwise there is no possible benefits from a parallel implementation under the MS paradigm.

Finally, the centralization of the data in one single place clearly limits the scalability of the system. Adding more slaves than the master can handle introduces slave starvation, and worse, might cause contention at the master site, degrading the overall performance. Several studies have proposed strategies to automatically and dynamically adjust the number of slaves involved in the computation to optimize the master utilization [86, 102, 149].

1.4.2 Research questions

In appreciation of the problems pointed out above, we believe that the main issue that needs to be addressed, is the lack of scalability of the MS paradigm. Hence, the main research question identified and explored in this thesis is:

Q-1 **How should the MS paradigm be enhanced to improve its scalability?**

On the one hand, sending - and receiving - all the data in a single place constitutes the bottleneck of the MS paradigm. On the other hand, centralizing the scheduling decision making process might be inefficient when dealing with large-scale dynamic systems. The amount of information that needs to be gathered at a central location may require a prohibitive amount of time. System conditions may have changed by the time the information has been gathered to the scheduler. These two observations imply that (i) the master location(s) should be carefully selected, and (ii) scheduling decisions should be made in a decentralized fashion.

This thesis investigates also the suitability of MS techniques to applications that are not usually implemented as such, but that could nonetheless benefit from it. In particular, we think of adaptation to system load fluctuations and resilience to resource failures. Hence, the second research question explored in this thesis has been formulated as follows:

Q-2 **Can MS scheduling techniques be applied to stencil code applications?**

Answering this question implies to:

- Identify stencil code application candidates.
- Implement and evaluate a MS implementation of the selected applications.

We have chosen stencil code applications as candidates for such investigation because several applications of this kind are used at NTNU. Usually, stencil applications are parallelized with *domain decomposition* (DD) methods, that decompose the computational domain into sub-domains assigned to the processors. However, the DD methods make it difficult to account for system heterogeneity, to adapt to system fluctuations, and to handle system failures (more on this in Chapter 3).

1.4.3 Research methods

Static models for conceiving dynamic strategies

Some people believe that static models are inappropriate for designing dynamic scheduling strategies. We mean, on the contrary, that every dynamic environment can be considered as a succession of static contexts. Our research philosophy consists in studying heterogeneous static environments, in order to identify which property or aspect of the problem is the determinant factor that directly impacts on system performance. Then, knowledge that has been acquired on static networks can be embedded within dynamic scheduling strategies.

A subtle point similar to the *bounded irregularity* pointed out by Bast [23] corroborates our research methodology. If the system load fluctuates in an unforeseeable manner throughout the entire execution time, then it becomes impossible to guarantee anything, and the straightforward *self-scheduling* strategy comes out as an optimal strategy. It is therefore important to design algorithms that are efficient when the system stabilizes, and in this context, it makes sense to work with static models in the first place.

Real experiments vs. simulations

The evaluation and comparison of different scheduling strategies can be done either via real experiments or via simulations. Real experiments are important because they allow to test the behavior of an algorithm on a computer which is more complex than the model used to design the algorithm. However, large-scale experiments are difficult to reproduce because of the intrinsic instability of the platform. It is indeed impossible to guarantee that a large-scale platform will remain exactly in the same state between two tests, thereby forbidding any rigorous comparison between two scheduling strategies.

In contrast, simulations allow to fully control the experimental process. One can guarantee that two scheduling strategies were run in the exact same system conditions. Besides, Grid simulators are becoming more and more realistic. SimGrid [51] for instance, the simulator used in this study, allows to model real network topologies and their associated resource characteristics, such as CPU speed, network bandwidth and latency. In addition, it accounts for the congestion generated by multiple connections taking place simultaneously on the same link or on the same machine. The dynamic behavior of the system is described within trace files (CPU load and availability, network bandwidth and latency) that can be artificially generated, or that can be captured on real systems by Grid monitoring tools such as the Network Weather Service [187, 188].

The final advantage that simulations offer over real experiments is diversity. Setting up real experiments is a time-consuming process, which in the end gives results only for the test-bed system. In contrast, simulations allow to study a wide variety of computing systems with little additional efforts. For all these reasons, we used the SimGrid simulator toolkit for testing and comparing our scheduling strategies intended for computational Grids.

On the contrary to Grid environments, the reproducibility of experiments is possible on parallel computers. Most supercomputers provide dedicated access to their computational nodes, such that applications running simultaneously on the system get exclusive access to the nodes. There might still be interferences on the network or with the file system due to applications competing for shared resources, but these interferences can be attenuated if the experiments are repeated a sufficiently large number of times.

Problem analogies

Another research method used in this thesis is the search for analogies between problems. Indeed, different problems may actually share the same objectives, or face the same intrinsic difficulty. Hence, models and techniques used for one kind of problems can successfully be applied to another kind of problems. For instance, we highlight in Paper 1 the analogy between Facility Location problems and resource location problem. In particular, we use a Facility Location model to formulate our resource location problem, and to ultimately derive an efficient heuristic.

Identifying or building transformations from one problem to another is very useful for attacking new problems [89]. In Paper 1, we prove the NP-hardness of our resource location problem by a reduction from the Maximum Knapsack problem. And in Paper 3, we use a 2-dimensional Cartesian representation to derive an optimal principle for independent-task scheduling onto heterogeneous tree-shaped platforms.

1.5 Thesis outline

The rest of this thesis is organized as follows. Chapter 2 provides an introduction to scheduling theory, an overview of the state-of-art of scheduling independent task applications, and concludes by exposing how the contributions of this thesis fit within previous work. Chapter 3 presents stencil code applications, highlights the main issues for efficiently implementing these applications, and concludes by exposing our contributions when working with stencil code applications. Chapter 4 concludes the first part of the thesis, by summarizing our main contributions, discussing some limitations and proposing future work directions.

Our contributions are gathered in the second part of this thesis. Paper 1 addresses the problem of efficiently deploying multi-master MS applications onto heterogeneous platforms. Paper 2 presents a lightweight distributed method for building asynchronous schedules. Paper 3 presents distributed scheduling techniques intended for asymmetric networks. Paper 4 introduces a new MS scheduling strategy tested with an image filtering application on a low-cost PC cluster. Papers 5 and 6 present respectively a DD and a MS implementations of a Lattice Gauge Theory model. Finally, Paper 7 presents cache-efficient optimization techniques for improving the performance of stencil code applications.

Chapter 2

Independent-task scheduling

This chapter starts with an introduction to scheduling theory by presenting the general concepts common to most scheduling problems. Then, we present the state-of-the-art regarding independent task scheduling, the application class of interest for the framework of this thesis. We present theoretical results for the most common optimization objective, namely minimizing the total execution time (or makespan), and show how modifying the scheduling objective (by considering throughput maximization) helps deriving asymptotically optimal algorithms. Finally, we present the contributions of this thesis for the independent task scheduling problem, and how do they fit within previous work.

2.1 Introduction to scheduling theory

One of the challenges in exploiting the power of parallel computers is to map, or *schedule*, the parallelism contained in a program onto a set of processors, in order to achieve performance goals such as minimizing execution time, minimizing communication delays, or maximizing system throughput [54]. Scheduling problems are difficult. There are many factors that affect the decision process, such as the number and nature of the tasks to execute, task priority, current system load, affinity between tasks and machines, or resource usage policies. It is not too surprising then, that most scheduling problems turn out to be NP-complete [89], as they consider optimal execution schedules under a number of constraints. Consequently, one must often resorts to heuristics in order to generate efficient schedules in a reasonable amount of time [82].

2.1.1 DAG, makespan and Gantt-chart

Application programs are composed of different tasks that must be executed in a certain order to produce the desired results. The tasks that are independent of each other are the ones whose execution order can be changed without modifying the result of the program. Hence independent tasks can be executed simultaneously by different processors.

A *directed acyclic graph* (DAG) is used to represent the task dependencies of a program, where transitivity edges are omitted (see Figure 2.1 (a)). Usually, T_{begin} and T_{end} are fictive tasks used to facilitate the identification of the start and the end of the program. The usual scheduling objective is to minimize the total execution time of the schedule ($T_{end} - T_{begin}$), also called *makespan*.

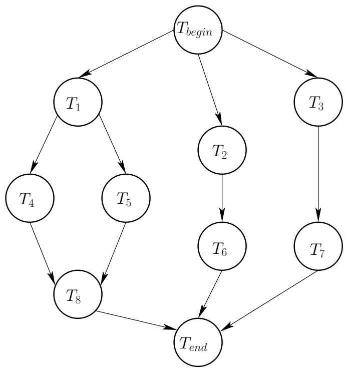
The *macro-dataflow model* [62, 82, 170] has been widely used in the literature for modeling communication costs associated to task dependencies. If two dependent tasks u and v have been assigned to different processors, a communication delay occurs. More precisely, if task u is completed at time-step t, then the execution of task v cannot start before time-step $t + c(v_1, v_2)$. But if two dependent tasks reside on the same processors, the data transfer between the predecessor to the successor occurs via memory accesses. Since memory accesses are typically much faster than inter-processor communications, it is reasonable to neglect them. Figure 2.1 depicts a DAG example with the associated task execution times and inter-task communication times, and a Gantt-chart for visualizing a possible schedule with three processors.

The major drawback of the macro-dataflow model is the lack of realism when modeling communication operations. A processor can send and receive any number of messages concurrently, and the number of messages that can simultaneously circulate on the network is not bounded in any way. These assumptions are not realistic for modeling modern computing systems, and more advanced communication models are required.

2.1.2 Heuristics for DAG scheduling

Scheduling arbitrary DAGs with the makespan minimization as objective is known to be NP-hard [166]. Consequently, a profusion of heuristics have been proposed, including list scheduling heuristics [83] and task clustering schemes [151]. Two versions of this problem have been investigated, depending on whether or not task duplication is allowed. The obvious benefit of task duplication, is to spare communication overhead by allowing several copies of a task to be executed by different processors. In general, scheduling with task duplication produces shorter makespans than without [151].

List scheduling heuristics are greedy algorithms that try to allocate as many



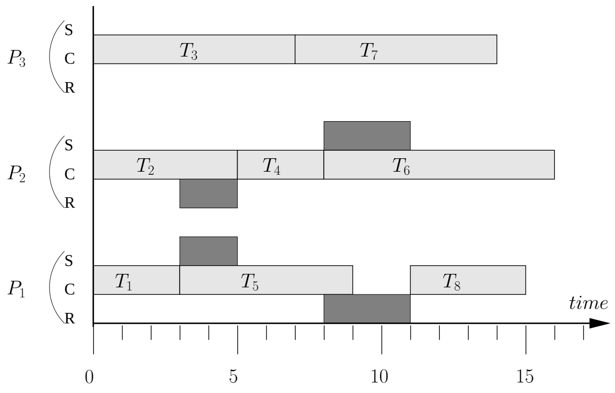
(a) DAG example.

T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
3	5	7	3	6	8	7	4

(b) Task execution times.

(T_1, T_4)	(T_1, T_5)	(T_2, T_6)	(T_3, T_7)	(T_4, T_8)	(T_5, T_8)
2	6	2	5	3	1

(c) Communication times.



(d) A schedule with 3 processors (S: Send, C: Compute, R: Receive).

Figure 2.1: DAG example (a), task execution times (b), inter-task communication times (c) and Gantt-chart of a schedule (d).

tasks as possible at any given time-step. In a first phase, a priority is attributed to every task based on the number of their predecessors, and the tasks are inserted in a list by decreasing priorities. In a second phase, the algorithm iterates over the list built in the first phase and schedules the tasks on the processor that allows the earliest starting time of the task. The attribution of task priorities in the first phase has a tremendous impact on the performance of list scheduling heuristics. Not surprisingly, most list scheduling heuristics assign higher priorities to tasks located on the critical path of the DAG (i.e. located on the longest path of the DAG).

On the other hand, task clustering heuristics [90, 151] try to shorten the critical path of the DAG. Paths are shortened by clustering several tasks into coarser grain tasks. This admittedly reduces the communication overhead, but comes at the expense of increased processing times of the coarser tasks. This new tradeoff implies to determine adequate task granularity to achieve load balancing [151].

2.2 Makespan minimization

In order to minimize the makespan of a schedule, one must distribute the tasks to the processors in a way to achieve optimal load balance, that is, all the processor finishing times must differ as little as possible. Depending on the framework of the study, different model assumptions are made. Common to all studies is (i) the assumption that all the tasks initially reside in one place of the system and (ii) scheduling tasks onto a processor incurs an overhead.

2.2.1 Independent task applications

Independent task applications exhibit a very simple DAG structure (see Figure 2.2). In the early research on independent task scheduling, much attention has been devoted to homogeneous multiprocessor systems composed of identical processors interconnected with homogeneous networks [98, 109, 127]. The scheduling complexity of these problems lies in the *task irregularity* assumption, stating that the task processing times vary in an unpredictable manner. Hagerup [98] claims that in practice, task irregularity can arise from *algorithmic variance* - where the nature of the data being processed leads to different execution times - and from *system induced variance* - provoked by external events such as cache misses, operating system interference, clock interrupts, etc.

On the other hand, heterogeneous platforms are becoming widespread, and their efficient utilization requires a good understanding of the added complexity that heterogeneity introduces. For instance, it has been shown that greedy protocols that delegate as much work as possible to the fastest processors are not adapted to the heterogeneity of the platform [36, 161]. Therefore, more advanced techniques are required. In this context, the difficulty of the scheduling problem has been moved from task irregularity to platform heterogeneity. Hence, the research focus has been devoted to scheduling regular independent task applications onto heterogeneous platforms [36, 80, 161].

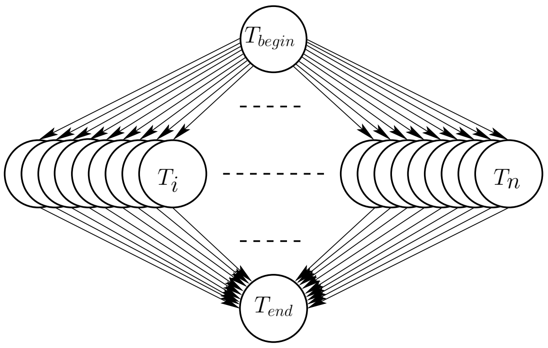


Figure 2.2: DAG of an independent task application.

Scheduling irregular tasks on homogeneous systems

Scheduling irregular tasks gives often rise to bin-packing problems known to be NP-complete. Most of the well-known scheduling heuristics work under the assumption that a fixed overhead incurs each time a bunch of tasks is delegated to a processor, no matter how large the bunch is.

Under this assumption, it appears that *static chunking*, which consists in delegating all the tasks intended to a processor at once minimizes the scheduling overhead. What limits the efficiency of static chunking in practice is the difficulty to accurately estimate the task execution times, which often leads to load unbalance among the processors.

Because one may not have accurate information - or no information at all - on the different task processing times, it might be a good idea to not put all its eggs in the same basket, that is, not send all the tasks intended to a processor at once. This approach is popularly known as *online scheduling*. The simplest strategy of this kind is the *self-scheduling* strategy [64, 98], where tasks are handed out on a one-by-one basis. By construction, the self-scheduling strategy produces a schedule in which the finishing times of the processors differ by at most the processing time of a single task. This is, in a sense, very satisfactory considering that this quantity is assumed to be very small compared to the overall execution time. However, this strategy comes at the expense of a large scheduling overhead in the form of excessive communications. Actually, *self-scheduling* is just the opposite of *static chunking* which minimizes scheduling overhead at the risk of a large load imbalance.

To achieve better tradeoffs, hybrid schemes have been proposed that schedule not all, but several tasks at a time. The idea being that early chunks should be large in order to keep the scheduling overhead small, while smaller chunks are required towards the end of the computation to achieve a good load balance among the processors. Following this principle, a multitude of heuristics using chunks of decreasing sizes have been proposed. Hagerup [98] gives a comprehensive overview of the most popular heuristics.

Bast stresses a subtle point about the task irregularity assumption [23]. When nothing is known about the task processing times, no advantage lies in scheduling several tasks at a time, since in principle a single task might take an equally long time. In these conditions, *self-scheduling* comes out as an optimal scheduling strategy. Therefore, hybrid schemes are meaningful only when there is a kind of *bounded irregularity* of the task processing times, by which it is assumed that a larger number of tasks incurs a larger total processing time than a smaller number of tasks. This bounded irregularity is accounted by so-called *stochastic scheduling* techniques, where the task execution times are modeled as independent, identically distributed random variables with a common probability distribution *D* having a mean $\mu > 0$ and a variance $\sigma > 0$ [98, 109, 127]. Within these settings, Kruskal and Weiss [127] studied the *fixed size chunking heuristic*, that consists in using chunks of a fixed number of tasks. However, the stochastic model complicates greatly the design and performance analysis of heuristics, and very few studies have reported theoretical results under this model.

Scheduling regular tasks on heterogeneous systems

When dealing with heterogeneous computing systems, more efforts have been putted into the computing and communication models, at the expense of a simpler task execution times model.

Initially, all the tasks reside on one processor called the master processor P_m. The tasks will be sent over a network for remote computation by a set of k slave processors $P_1, \ldots, P_k$. Further, it is assumed that the master can communicate with the slaves only one at time (single-port model), requiring c_i time units to communicate with slave P_i. The number of tasks that can be communicated within c_i time units depends on the framework of the study. At last, it takes w_i time units to slave P_i to process one task.

Two optimization problems have been formulated in the literature [36, 161]. The first one is the traditional makespan minimization problem, while the second one aims at maximizing the throughput of the system, i.e. the number of tasks executed within a given time frame T. Interestingly, if a polynomial time algorithm $\mathcal{A}$ can be formulated for the throughput maximization problem, then it is possible to solve the makespan minimization problem in polynomial time using algorithm $\mathcal{A}$ combined with a dichotomic search on T [36]. And vice versa for solving the throughput maximization problem using an algorithm that solve the makespan minimization problem combined with a dichotomic search on the total number of tasks to be processed.

The throughput maximization problem with heterogeneous processors interconnected by a bus (see Figure 2.3 (a)) is polynomial in the case when there is only one initial communication per slave [36] (similar to static chunking). In this scenario, the master pays a fixed communication delay c for sending an unbounded number of tasks intended to a slave. The goal is to find the best permutation σ that determines the order in which the slaves should be served. The problem becomes NP-complete when a final communication between the slaves and the master is necessary to send back the computational results. The problem complexity increases significantly because two permutations σ_1 and σ_2, one for sending the tasks and one for receiving the results, must now be determined.

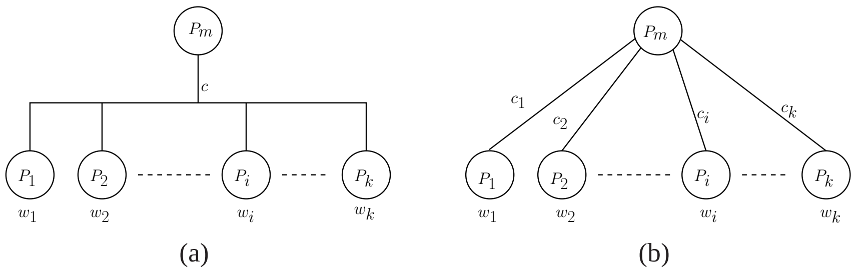


Figure 2.3: Bus (a) and “star” (b) networks.

The throughput maximization problem under the assumptions that (i) each task sent to slave P_i incurs a communication cost of c_i time units, and (ii) the processors are interconnected by a heterogeneous “star” (see Figure 2.3 (b)), is also polynomial [36]. This result has been extended for heterogeneous linear daisy chains and “spider” graphs (see Figure 2.4), with a polynomial time algorithm for solving the makespan minimization problem [80]. Finally, the problem becomes NP-complete for heterogeneous tree-shaped platforms [81].

2.2.2 Divisible load theory

Application model

The *divisible load* model [43,45,176] embodies applications whose computational workload is composed of a large number of homogeneous low-granularity computations called *work units*. There are no communication dependencies between the work units which can therefore be processed in parallel. The total application workload can hence be split into *chunks* of arbitrary size (each chunk corresponding to a given number of work units), and this in a linear fashion, i.e. the computation and communication time requirements of a chunk are proportional to its size.

The divisible load model has been widely studied, and the *divisible load theory* (DLT) has emerged as a new scheduling paradigm for distributed computing

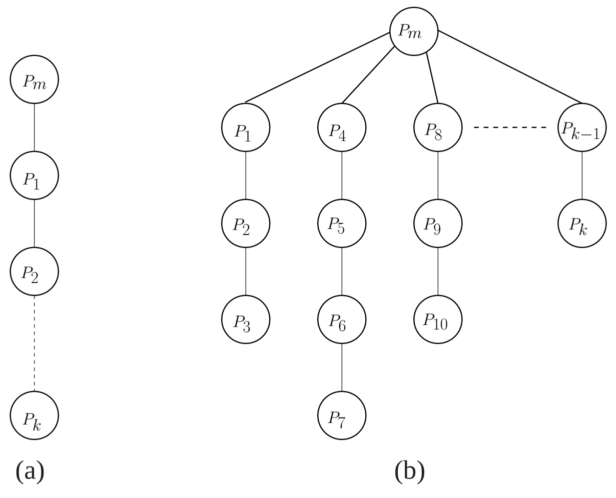


Figure 2.4: Linear daisy chain (a) and “spider” graph (b) networks.

platforms [43]. Many applications have been implemented under this paradigm including image processing (e.g. edge detection [176]), processing of massive experimental data set, signal processing applications [45], pattern searching, file compression, joining operation in relational databases operations, graph coloring or genetic search [76]. This section presents fundamental DLT concepts which are closely related to some of the work presented in this thesis.

Framework

The basic DLT assumptions are the following. A system composed of $p + 1$ processors $P_0, P_1, \ldots, P_p$ is considered. The processor P_0 called *originator* or *master* plays a particular role. At the beginning of the computation, the whole workload is stored in the memory of the originator processor P_0. The originator then scatters the workload over the network to the p remote processors, that will process their shares of the workload in parallel. It is widely accepted in the DLT that the return of the computational results to the master can be neglected. This assumption is made for the sake of simplicity, and may not be realistic for some applications. Still, gathering the results to the originator has been incorporated in the DLT model for special cases as shown in [38, 49]. The data scattering and gathering parts are highly dependent of the underlying platform topology and scheduling policy adopted.

Theoretical model

Typical divisible load models target heterogeneous platforms. We describe below the standard notations used in the DLT literature [45]:

- α_i is the fraction of workload allocated to processor P_i,
- $\alpha = (\alpha_0, \alpha_1, \ldots, \alpha_p)$ is the load distribution vector,
- $T(\alpha)$ is the makespan associated to load distribution α,
- T_{cp} is the time taken to process a work unit by the standard processor,
- w_i is the ratio of the time taken by processor P_i to compute a given load, to the time taken by a standard processor to compute the same load,
- T_{cm} is the time taken to communicate a work unit on a standard link,
- z_i is the ratio of the time taken by link L_i to communicate a given load, to the time taken by a standard link to communicate the same load.

The *standard* processor and link might be any processor or link of the platform used as a reference. With these notations, it takes $\alpha_i w_i T_{cp}$ time units for processor P_i to process its share α_i of the load. Similarly, it takes $\alpha_i z_i T_{cm}$ time units to send the load fraction α_i over the network link L_i.

Scheduling policies

Divisible load models can usually be solved algebraically for optimal allocation to processors and links under a certain *scheduling policy*. The scheduling policy adopted depends of the characteristics of the computing platform targeted in the study. Traditionally, scheduling policies have three components:

- *Load distribution model*: Processors may or may not distribute the load concurrently to several other processors. Under sequential load distribution a processor can communicate load fractions to other processors only one at a time. Conversely, under parallel load distribution, data transmission may happen simultaneously on all communication links.
- *Processor operating mode*: Processors might be equipped *with front-end* or not. A processor equipped with front-end is capable to overlap computation with communication, while a processor without front-end cannot. The *with front-end* model is widely used in the literature, because it seems more representative to actual computing platforms characteristics, albeit the *without front-end* model has been studied as well [35].
- *Communication model*: In most DLT studies, processors are able to communicate only with their neighbors in a *store and forward* fashion. Other communication models such as *store and bypass* [119], circuit or cut-through switching [99] have also been studied.

Optimality principle

The *optimality principle* is a fundamental result, that grounds the DLT, stating that to obtain optimal processing time - i.e. a minimal makespan - all the participating processors must stop computing at the same time [43, 45]. Intuitively, if the processors stop computing at different times, it is possible to redistribute some of the load from the late processors to the early processors.

The first architecture to be applied the optimality principle is the daisy chain because of its simplicity. Consider a daisy chain, where processors are equipped *with front-ends*, communicating under the store and forward model, and with the originator located on a exterior node of the chain. The originator P_0 must send all the load intended to the rest of the chain to its neighbor P_1, which will subtract its share and forward the rest to its neighbor P_2, and so on until the last processor has received its share of the load. Hence, we get the following equation set, illustrated by the Gantt-chart given in Figure 2.5:

$$\textstyle\sum_{i=0}^{p} \alpha_i = 1 \tag{2.1}$$

$$\forall 0 \leq i < p, \quad \alpha_i w_i T_{cp} = (1 - \textstyle\sum_{j=0}^{i} \alpha_j) z_{i+1} T_{cm} + \alpha_{i+1} w_{i+1} T_{cp} \tag{2.2}$$

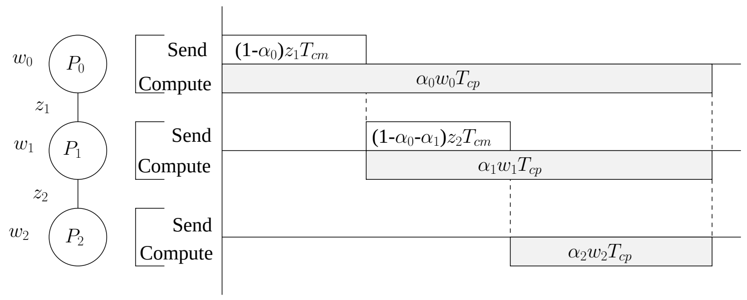


Figure 2.5: Load distribution on a 3-processor daisy chain.

Different scenarios have been considered for the daisy chain network, such as the load originating at an interior or exterior node of the chain, and with or without gathering the computational results back to the originator [45].

Load distribution sequence

Tree and bus networks allow for better performance than daisy chains, because generating much less communication. Indeed, in a daisy chain, the load intended to processor P_k must travel through $k - 1$ links before reaching its destination. The amount of communication generated by the chain topology is hence equal to:

$$\begin{aligned}
\mathcal{C}_{chain} &= [(1-\alpha_0)z_1 + (1-\alpha_0-\alpha_1)z2 + \cdots + (1-\alpha_0-\cdots-\alpha_{k-1})z_k]T_{cm} \\
&= [\sum_{i=1}^{k}\alpha_i z_1 + \sum_{i=2}^{k}\alpha_i z_2 + \cdots + \alpha_k z_k]T_{cm} \\
&= T_{cm}\sum_{j=1}^{k}\sum_{i=j}^{k}\alpha_i z_j
\end{aligned}$$

In contrast, the amount of communication generated by a single level tree (or star) network is equal to $\mathcal{C}_{star} = T_{cm}\sum_{i=1}^{k}\alpha_i z_i$.

As opposed to daisy chains, tree-shaped networks open up the possibility for varying the order or *sequence* of the load distribution among the child processors. This brings about the problem of resource selection, as it has been shown that the optimality principle leads to optimal schedules only for a carefully chosen and ordered subset of child nodes [37, 45].

A recent theorem [37] states that an optimal load distribution for the star network topology is obtained by utilizing all the processors of the network, with a load distribution sequence ordered by increasing link capacities z_i (fast links first).

But if the load distribution sequence is fixed a priori, then it has been shown that the optimal processing time can be achieved by distributing the load only to “fast” processor-link pairs. An exact expression that distinguishes the fast processor-link pairs from the slow ones has been derived [43], and a *reduced* network can be obtained by removing the slow processor-link pairs. The load is then distributed among the remaining processors using the optimality principle.

Installments and sequencing

Under the *with front-end* model, sending all the load intended to the processors in a single message leads to poor utilization of the processors. Indeed, the sequential load distribution imposes that the last processors to be served are waiting idle, while their predecessors in the load distribution sequence are receiving their share of the load. To address this problem, *multi-round* or *multi-installment* algorithms have been proposed [42, 43, 192]. These algorithms dispatch the load in multiple rounds and thus improve overlap of communication with computation.

The two main questions that must be answered when designing multi-round algorithms are: What should the chunk sizes be at each round? And how many rounds should be used? Most of multi-round algorithms assume a fixed number of rounds. A review of multi-round algorithms can be found in [37], but the main observations therein are: (i) dividing the workload into large chunks reduces communication overhead, (ii) sending small chunks at the beginning of the execution makes it possible to overlap communication with computation and (iii) sending small chunks at the end of the execution leads to better load balance among processors. Not surprisingly, observations (i) and (iii) - that proved useful for scheduling irregular tasks onto homogeneous processors - hold also within DLT settings. Based on these three observations, Casanova and Yang [191] have proposed an algorithm that starts by sending larger and larger chunks, and ends by sending smaller and smaller chunks.

Finally, the linear DLT model may lead to flawed solutions as there is no prohibitive cost for sending large numbers of very small messages [52, 192]. Indeed, the linear model implies that an infinite number of rounds where an infinitesimal amount of work is sent out at each round gives an optimal task allocation. Although an affine model - that accounts for network latencies - addresses this issue and renders the model more realistic, it nevertheless increases significantly the complexity of the problem [37, 192].

Network equivalence

A useful concept in DLT is the notion of *network equivalence*, that makes possible to assimilate some network topologies to a unique processor of equivalent power via closed-form expressions or numerical procedures [24]. For instance, these network equivalences can be utilized to show that speedup is bounded from above by a quantity independent of network size, but dependent of network topology [68]. This feature provides a mean to compare the performance of finite configurations of processors against infinite ones [25, 160]. The network equivalence concept proves also to be useful for theorem proving [21, 37].

The network equivalence concept has been successfully applied to several network topologies, including daisy chains [25, 160], arbitrary trees [25, 142] or 2-dimensional meshes [46]. Daisy chains - although not very common in practice - prove useful in this context, as they serve as a good basis for studying more complex architectures such as 3-dimensional [75] and k-dimensional meshes [134] via network equivalence transformations.

Also, the network equivalence principle allowed to formally identify the impact of sequential load distribution on the performance saturation within DLT networks [25]. Although the speedup increases as the number of processors and installments increase, it nevertheless tends to saturation because of the repetitive overhead in propagating the load into the network. The situation is somewhat similar to Amdahl's law [95] as the communication overhead associated to the load transfer takes place in a sequential fashion [45]. On the contrary, speedup is scalable under the parallel load distribution model, that is when nodes can transmit load simultaneously to all their neighbors [176].

Model extensions

The divisible load model has been applied successfully to a wide variety of interconnection topologies including daisy chains [160], star graphs [37], hypercubes [135], two, three and k-dimensional meshes [48, 75, 134], and arbitrary trees [142].

The original DLT model has been extended in many ways, including finite memory processor [78], memory hierarchy design of recent computers [79], network latencies [34, 47], processor latencies for initiating a computation [191], processor release times [41], unknown network resources [120], and adaptation to Grid computing [194, 196].

2.3 Throughput maximization

2.3.1 Performance metric

For applications with very long execution times - typically days or weeks -, two schedules whose durations differ only by a few minutes can be considered as equivalent [32]. In this case, *makespan* minimization might not be the appropriate performance metric. Besides, we saw that makespan minimization greatly complicates the scheduling problem, which can hinder algorithmic design.

A more meaningful and more practical objective function is *throughput maximization*, that is maximizing the number of tasks executed per time unit [14, 32, 107, 169]. Indeed, deriving asymptotically optimal schedules is very satisfactory for such lengthy applications.

Lengthy executions of independent task applications can be decomposed into three phases: A *start-up* phase, a *steady-state phase* and a *wind-down* phase [126]. During start-up, the computation begins with the master starting to delegate tasks to the slaves. Then, the steady-state regime sees the master sending tasks and receiving results from the slaves in a somewhat regular fashion. And finally, the computation terminates with the *wind-down* phase during which the master collects the last results from the slaves. Since the target applications are expected to run for a very long time, the steady-state phase will dominate the total execution time, such that the *start-up* and *wind-down* phases can reasonably be neglected. This is the main argument for focusing on optimizing the *steady-state* regime.

During steady-state, the initial integer formulation can be relaxed, and replaced by a continuous time model. The goal is to describe the activities of the resources during each time unit: What fraction of time is spent communicating (and with who) and what fraction of time is spent computing. To some extent, this continuous time model is similar to the divisible load model, since both domains can be divided into infinitesimally small quantities. The main difference being that one needs to construct a valid periodic schedule (where an integer number of tasks is treated per time period) based on the resource activity descriptions.

2.3.2 Theoretical model

The theoretical models employed in the literature are very similar to DLT models. Usually, the execution time of a task on a processor P_i is modeled by a single value w_i, such that processor P_i requires $\alpha_i w_i$ time units to compute α_i tasks. Similarly, it takes c_{ij} time units to send a task from processor P_i to processor P_j. It is possible to have $c_{ij} \neq c_{ji}$, that is bandwidth asymmetry on the network links. This linear model is the most common model found in the literature due to its simplicity [30, 106, 161, 169].

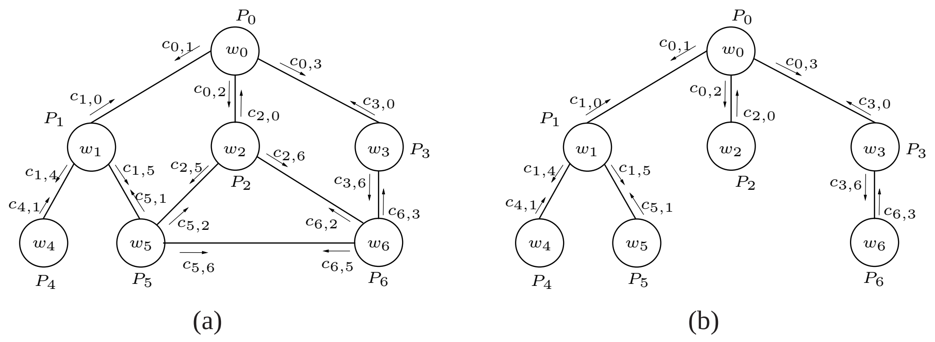


Figure 2.6: Graph network (a) and spanning tree (b).

The two most popular topologies used for modeling complex and large-scale platforms are undirected graphs and spanning trees (see Figure 2.6) [14, 104–106, 126, 169]. Both topologies model the network as sequences of network links or *paths*, that may be shared by several routes, which is necessary to obtain somewhat realistic platform models [52]. The vertices or nodes of the platform represent computing resources capable of computing and/or communicating with their *neighbors* at (possibly) different rates.

Although graph networks provide more general platform models, they nevertheless introduce routing decision making, which greatly complicates the scheduling problem. In contrast, the hierarchical topology of tree networks has the advantage to remove routing decision problems [30, 81, 108, 126, 168]. However, it has been shown that the problem of extracting the best spanning tree from a given network is NP-complete, and that even though such a tree could be found, there exist networks for which the performance of the optimal tree is arbitrarily worse than the whole network performance [14]. Nevertheless, these unusual networks have been constructed to prove the superiority of graphs over trees, and might not be representative of realistic networks.

2.3.3 Scheduling policies

Just like DLT models, steady-state models are governed under a certain scheduling policy. Steady-state scheduling policies have also three components:

- *Network interface*: Very similar to the DLT *load distribution model*, a processor may or may not communicate concurrently to several neighbors. The *single-port* model for both incoming and outgoing communication, restricts a processor to open one communication in emission and one in reception simultaneously [14, 30, 108, 126]. At the other end of the spectrum, processors can communicate simultaneously with all their neighbors in emission or reception, which amounts to the *multi-port* model, or *network-flow* model [169]. In between is a model allowing an unbounded number of incoming and outgoing communications to happen simultaneously, but at a restricted rate amounting to the hardware limitation of the network interface [104–107].

- *Processor operating mode*: Processors can perform three basic operations, sending messages, receiving messages and performing computation. The degree of simultaneity and concurrency between these actions depends on the capacity of the target machine. If all activities can be performed simultaneously, then we speak of a *full-overlap* model [14, 106, 169]. At the other end of the spectrum is the *sequential* model where a machine can perform only one activity at a time. Beaumont et al. [29] define a variety of models that cover all the possible combinations of concurrency between the processor activities.

- *Communication model*: In most literature studies, processors are able to communicate only with their neighbors under the *store and forward* model. Thus, a task can be processed only after receiving all the data associated with that task. More precisely, if P_i sends a task to P_j at time-step t, then P_j cannot start executing the associated task, or forwarding it before time-step $t + c_{i,j}$.

Interestingly, Beaumont et al. [29] showed that any operating model, resulting from a combination of the different overlap and network interface characteristics of a machine can be reduced to the *single-port, full-overlap* model. This is presumably the most powerful argument for the utilization of this model, albeit the communication serialization that comes with it greatly complicates the scheduling problem [32]. Another reason for using this model is that standard communication libraries like MPI [94] and PVM [96], usually use sequential or binomial tree based schemes to support collective communications [181], relying hence only on point-to-point communications.

2.3.4 Formulation of the steady-state regime

One of the strengths of steady-state scheduling, is the possibility to derive an optimal solution using linear programming techniques. Not only this solution can be used to construct an asymptotically optimal schedule, but it can also be used to evaluate the performance of decentralized heuristics against the optimal solution.

Let us formally express the steady-state scheduling problem on a graph platform $G = (V, E, w, c)$, where each processor $P_i \in V$ operates under the *single-port, full-overlap* model. Let P_m denote the master processor, where all the tasks reside initially. To ease the presentation, assume that the size of the task output data is much smaller than the size of the task input data, such that the results collection at the master site can be neglected. We aim at determining the constraints induced by our problem during steady-state.

Processor operations. Let $n(i)$ denote the index set of the neighbors of processor P_i. During one time unit let:

- α_i be the fraction of time spent by P_i computing,
- $s_{i,j}$ be the fraction of time spent by P_i sending input files to each neighbor processor $P_j, j \in n(i)$,

As these variables correspond to the activity during one time unit, we have the following constraint sets:

$$\forall i, \quad 0 \le \alpha_i \le 1 \tag{2.3}$$

$$\forall i, \forall j \in n(i), \quad 0 \le s_{i,j} \le 1 \tag{2.4}$$

One port model for outgoing communications. Because send operations to the neighbors of P_i are assumed to be sequential, we have the equation:

$$\forall i, \sum_{j \in n(i)} s_{i,j} \le 1 \tag{2.5}$$

One port model for incoming communications. Because receive operations from the neighbors of P_i are assumed to be sequential, we have the equation:

$$\forall i, \sum_{j \in n(i)} s_{j,i} \le 1 \tag{2.6}$$

The master P_m should not receive unprocessed tasks from its neighboring processors, which gives the following equation:

$$\forall j \in n(m), s_{j,m} = 0 \tag{2.7}$$

Conservation Laws. During one time unit, for every processor P_i except the master, the number of tasks received from the neighboring processors ($\frac{s_{j,i}}{c_{j,i}}$), should be equal to the number of tasks processed ($\frac{\alpha_i}{w_i}$) plus the number of tasks sent to the neighboring processors ($\frac{s_{i,j}}{c_{i,j}}$). We hence have the following constraint:

$$\forall i \neq m, \sum_{j \in n(i)} \frac{s_{j,i}}{c_{j,i}} = \frac{\alpha_i}{w_i} + \sum_{j \in n(i)} \frac{s_{i,j}}{c_{i,j}} \tag{2.8}$$

It is important to underline that equation (2.8) really applies to the steady-state regime. For this, assume that a start-up phase already took place during which some tasks have been forwarded to the processors, but no computation has been performed, such that each processor received $\frac{\alpha_i}{w_i} + \sum_{j \in n(i)} \frac{s_{i,j}}{c_{i,j}}$ tasks. At the end of the start-up phase, each processor disposes of enough tasks to enter the steady-state regime.

All the aforementioned constraints can be gathered into a linear program, whose objective is to maximize the throughput $n_{task}(G)$ of the platform graph G.

Maximize

$$n_{task}(G) = \sum_{i \in V} \frac{\alpha_i}{w_i},$$

Subject to

$$\begin{cases} \forall i,\ 0 \leq \alpha_i \leq 1 \\ \forall i, \forall j \in n(i),\ 0 \leq s_{i,j} \leq 1 \\ \forall i,\ \sum_{j \in n(i)} s_{i,j} \leq 1 \\ \forall i,\ \sum_{j \in n(i)} s_{j,i} \leq 1 \\ \forall j \in n(m),\ s_{j,m} = 0 \\ \forall i \neq m,\ \sum_{j \in n(i)} \frac{s_{j,i}}{c_{j,i}} = \frac{\alpha_i}{w_i} + \sum_{j \in n(i)} \frac{s_{i,j}}{c_{i,j}} \end{cases}$$

Because we have a linear programming problem in rational numbers, we can utilize well known polynomial time algorithms [117,118] to obtain rational values for all the variables .

2.3.5 Schedule reconstruction

Theoretical feasibility

Once we dispose of the descriptions for all the resource activities, it remains to build up a periodic schedule where an integer number of tasks are sent and/or executed. We can obtain a time period T during which an integer number of tasks is treated, by taking the least common multiple of all the denominators of the activity variables (α_i and $s_{i,j}$) [32]. Then we need to orchestrate the communication and computation events such that the constraints imposed by the network interface and processor operating mode hypotheses are satisfied.

The construction of such a valid schedule is straightforward for powerful processor operating models and network interface possibilities. For instance, the *multi-port, full-overlap* model presents no particular difficulties since all the activities can occur concurrently.

On the other hand, the *single-port, full-overlap* model complicates the problem as one needs to synchronize and order the incoming and outgoing data transfers with neighboring processors. A procedure based on edge-coloring a bipartite graph allows to extract independent communications - i.e. involving disjoint pairs of senders and receivers - and hence to implement the final schedule [14, 32].

But schedule reconstruction is difficult (NP-hard) for models that do not allow for overlapping incoming and outgoing communications [32]. Although it is easy to modify the linear program to account for the sequential operating mode [14], constructing the final schedule amounts to edge-color an arbitrary graph [32].

Finally, the complexity of constructing a valid schedule comes also to a great extent from the platform topology. Indeed, arbitrary graph topologies allow for task allocations to the same machines via different paths. Restrict the topology such that there is only one path between any pairs of nodes (as for example in tree networks), and the problem becomes much simpler [14].

Practical value of centralized schedules

Although centralized procedures can provide asymptotically optimal algorithms [14], their contributions are rather theoretical than practical. Effectively, their use in practice is extremely limited due to the large-scale nature of the system. The amount of information that needs to be gathered at a central location requires prohibitive amounts of resources (e.g. time and memory). Besides, the network topology and system load (in form of resource performance) are typically changing throughout the course of the computation. Thus, there is a need for lightweight adaptive techniques that can respond quickly and efficiently to changing conditions.

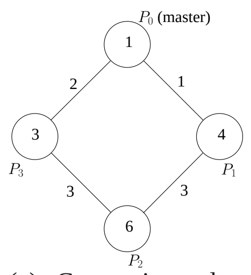
Autonomous scheduling strategies [106, 126, 150] are strategies that rely solely on information measurable locally. Obviously, autonomous strategies may make wrong scheduling decisions because by definition they lack of global knowledge on the system state. But nevertheless, a good autonomous strategy should tend towards optimal throughput rates when the system stabilizes.

2.3.6 Example

Consider the example taken from [14] depicted in Figure 2.7 with four processors operating under the *single-port, full-overlap* model. Assume further that collecting the computational results is neglected, and that the network link have equal bandwidth in both directions.

If we fill the values w_i and $c_{i,j}$ into the linear program, we obtain the solution depicted in Figure 2.7 (b). By taking the least common denominators of all the variables, we obtain a time period $T = 12$ during which:

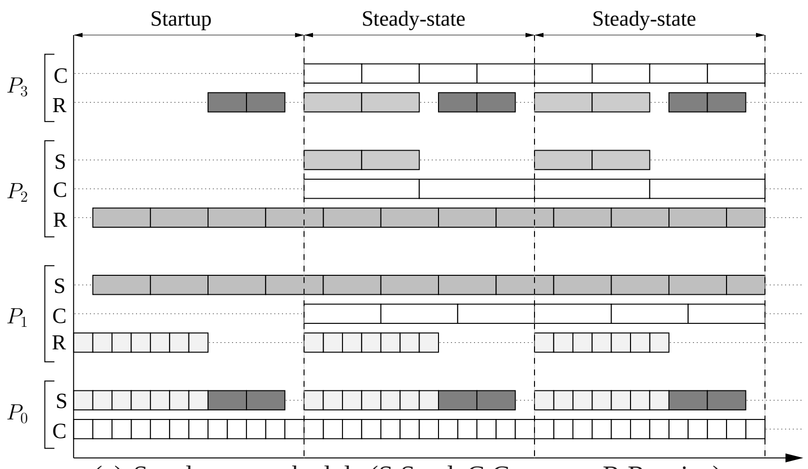
- P_0 computes 12 tasks, sends 7 tasks to P_1 and 2 tasks to P_3;
- P_1 receives 7 tasks from P_0, computes 3 of them and forwards 4 tasks to P_2;
- P_2 receives 4 tasks from P_1, computes 2 of them and forwards 2 tasks to P_3;
- P_3 receives 2 tasks from P_0 and 2 tasks from P_3, and computes 4 tasks.



(a) Computing platform

$$\begin{aligned}
\alpha_0 &= 1\\
s_{0,1} &= 7/12\\
s_{0,3} &= 1/3\\
\alpha_1 &= 1\\
s_{1,2} &= 1\\
\alpha_2 &= 1\\
s_{2,3} &= 1/2\\
\alpha_3 &= 1
\end{aligned}$$

(b) Linear programming solution



(c) Steady-state schedule (S:Send, C:Compute, R:Receive)

Figure 2.7: An example with four processors.

Thus, the platform can process 21 tasks every 12 time units, which corresponds to the $\frac{7}{4}$ value obtained from the linear program ($\sum_i \frac{\alpha_i}{w_i}$). Note that all the processors are executing tasks all the time, so the solution achieves a full utilization of the computing resources. It is interesting to point out that P_3 receives tasks along two different paths, directly from the master P_0, and indirectly via P_1 and P_2. Finally, an optimal steady-state schedule is depicted in Figure 2.7 (c).

2.3.7 The bandwidth-centric principle

A simple but yet efficient scheduling principle for heterogeneous star networks has been formulated under the assumption that sending the computational results back to the master was negligible [30]. The *bandwidth-centric* principle states that if enough bandwidth is available to a parent node, that is, if it can deliver tasks faster than its children can compute them, then all the children can be kept busy. However, if bandwidth is limited, then tasks should be allocated only to the children which have sufficiently fast communication times regardless of their computing speeds. Nonetheless, the computing speed of the children plays a role for determining the amount of work to delegate to each child, while the communication speed of the links interconnecting the parent to its children settles the sequence in which the children should be served. The bandwidth-centric closed-form expression given in [30] permits to identify which children should be fully utilized, which unique child should be partially utilized, and which children should remain unutilized. This result is somewhat similar to the one presented in the DLT context, stating that under the sequential load distribution, processors should be served in the order of increasing communication times.

Thus, the bandwidth-centric principle provides a simple yet effective way for scheduling on large-scale tree-shaped platforms in a decentralized manner. Each node of the tree makes scheduling decisions based solely on information that is measurable locally. The bandwidth-centric principle has been incorporated within autonomous scheduling strategies by Kreaseck et al. [126] that address the practical problem of attaining the maximum steady-state rate after some start-up and maintaining that rate until wind-down.

2.3.8 Network equivalence

Network equivalence is also useful within steady-state scheduling settings. Based on the bandwidth-centric principle, Beaumont et al. [30] conceived a bottom-up method that iteratively determines the steady-state throughput of a heterogeneous tree. At each time step of the procedure, the leaves of the tree are reduced together with their parent into a single node of equivalent computing power determined by the bandwidth-centric principle. The procedure ends when there remains an unique node having a computing power equivalent to the entire tree. These successive network equivalence reductions are more efficient (linear in the number of processors) than solving a linear programming problem for determining the optimal throughput of a tree-shaped platform.

The network equivalence principle is also very convenient when conducting simulations of large-scale systems. Effectively, since a tree can be reduced to a single super-node of equivalent processing power, it is not necessary to employ thousands of nodes to simulate large-scale systems [106].

Within the steady-state scheduling framework, we are only aware of the network equivalence result for tree-shape networks [30]. Extending this result to arbitrary graphs would be a valuated research direction. The work on *topology aggregation* [8, 130] for hierarchical PNNI (Private Network-to-Network Interface) routing in ATM (Asynchronous Transfer Mode) networks may give some inspiration on the matter. Topology aggregation is motivated by the need for compressing the information to reduce the complexity of topology advertisement, and by the need to hide network topology for security reasons [130].

2.3.9 Extensions

The steady-state scheduling problem has been devoted a lot of attention during the last few years. Many model and application extensions have been proposed. A way to include network latencies in the model is explained in [32]. Steady-state scheduling has also been targeting problems like optimizing the pipeline of broadcast [31], scatter and reduce [131] operations in heterogeneous environments. Finally, the case where multiple applications are executed concurrently, and hence competing for CPU and network resources has recently been investigated [28].

2.4 Contributions

This section presents the contributions of this thesis for the problem of scheduling independent task under the MS paradigm onto heterogeneous platforms. The scheduling objective being to maximize the steady-state throughput of the system. We argue for the chosen research directions by showing how do they fit within previous work.

2.4.1 Location-aware master-slave tasking

In Chapter 1, we advertised the fact that the MS paradigm has serious scalability issues, as the master cannot manage efficiently an unlimited number of slaves. Although scheduling strategies that adjust the number of slaves automatically and dynamically throughout the computation allow to elude master congestion [86, 102, 149], they do not provide a mean for delivering scalable performance.

The natural solution to achieve scalable performance consists simply in deploying several masters, that is, increasing the number of data access points. This technique can be implemented relatively easily on parallel computers [121], but the topology of large-scale computing platforms introduces complications such as resource selection and resource placement problems. To date, most studies on the MS paradigm within Grid environments (and on DLT as well) assume the use of a single master whose location is fixed a priori [36, 37, 105, 106, 126, 161]. To the best of our knowledge, the problem of determining how many masters should be deployed, and where should these masters be located on the Grid in order to maximize the system throughout has not been studied yet.

The most closely related work is presented in the paper of Shao et al. [169] who consider resource selection problems within the steady-state MS scheduling framework. The aim is to select performance-efficient hosts for both the master and slave processes. For that end, an exhaustive search is performed, consisting in solving n network flow problems, where n is the number of processors composing the platform. Then the configuration that achieved the highest throughput is selected. Unfortunately, this approach is not applicable when using several masters. There are indeed $\binom{n}{p}$ possible master locations sets, where p is the number of masters to be located on the platform. For this reason, we cannot simply compute the best scheduling strategy for each set, and then select the best result. As an example, for $n = 50$ and $p = 10$, the resulting number of possibilities is $10,272,278,170$. Clearly, even for moderate values of n and p, such enumeration is not realistic, and we need more advanced techniques.

Interestingly, our problem is remarkably similar to *Facility Location* problems [67, 125]. A classic Facility Location problem is a spatial resource allocation problem in which one or more service facilities have to be located to serve a geographically distributed set of population demands according to some objective function. If we show correspondence between (1) the facilities and the master nodes, (2) the service and the computational tasks, and (3) the set of demands and the computing resources of the Grid, then our master-slave tasking problem can be expressed as a Facility Location problem.

Consequently, lessons can be drawn from the design of algorithms for solving different versions of Facility Location problems. To the best of our knowledge, few studies have considered Facility Location theory within Grid computing settings. Maheswaran et al. [140, 141] present *MetaGrid*, an architecture that uses a Fixed Charge Location Problem [67] for resource provisioning for WAN-enabled applications. Ko and Rubenstein [122] present a distributed protocol to place replicated resources in large-scale networks such that each vertex is “close” to some copy of any object. Similarly, Theys et al. [179] notice that the *data staging* problem in a distributed heterogeneous networking environment, presents a high level of similarity with Facility Location problems.

Our contribution to this problem is as follows. We show that the problem of finding the most profitable locations for hosting the master processes in order to maximize the platform throughput is NP-hard. Then, we introduce an efficient heuristic for placing several masters on the platform. Because our heuristic takes into account the platform topology, we term it *location-aware*. Thus, our contributions provide an important step for efficient deployments of large MS applications on computational Grids. This work has been published in [11, 16] and model extensions are presented in [12].

2.4.2 Distributed schedule construction

Once the decision about the platform location(s) for the master(s) has been made, the attention can be devoted to the design of efficient distributed algorithms. Recently, Hong et al. [105–107] proposed an elegant algorithm, based on decentralized versions of flow algorithms. This algorithm, however, works under the assumption of powerful network interfaces that allow for the *multi-port* model, albeit the amount of incoming and outgoing data transfers is limited by the network interface of the machine. Under the *multi-port* model - bounded or not - there is no need to orchestrate the communications since they can occur simultaneously. Things get complicated under the *single-port* model, where only one incoming and one outgoing communications can happen concurrently.

In [126], the distributed orchestration of the communications is governed under the *demand-driven* paradigm, where nodes are regularly requesting tasks to their parents. This mechanism aims at dealing with resource performance fluctuations by allowing any node to participate to the computation. The scheduling strategy is fully autonomous and makes use of the bandwidth-centric principle for prioritizing the children requesting for tasks. Although the work of Kreaseck et al. [126] is a first step towards a distributed implementation of the bandwidth-centric principle, their autonomous protocol might take non-optimal decisions, generating hence long start-up phases as well as unnecessary large numbers of tasks buffered at node locations.

Our contribution to this problem is as follows. We present a lightweight distributed communication procedure which strictly follows the bandwidth-centric principle. Then we show how each node can build up its local schedule autonomously in order to attain the maximum steady-state throughput of the tree. Our procedure is an efficient, practical and scalable implementation of the bandwidth-centric principle. This work has been published in [13].

2.4.3 Collection-aware master-slave tasking

Most studies on independent-task scheduling make the assumption that returning the results to the master node can be neglected [14,23,30,81,98,106,107,109,126]. This assumption is acceptable when the output data are much smaller than the input data (e.g. the answer of the computation is of the type "Yes or No"). This simplification is very convenient as it considerably reduces the complexity of the scheduling problem, but fails to represent a very natural and important practical aspect of many master-slave applications.

Similarly, one finds relatively few studies that take into account the collection of the results by the master within the DLT literature [2, 21, 37, 48, 161]. When return messages are taken into account within the DLT framework, two permutations must then be determined (one for tasks distribution and one for results collection) [21, 37, 48]. Barlas [21] concentrates on two model simplifications, that correspond to *query processing* - when data transfers costs are fixed and independent of load size - and *image processing* - when the communication costs are linear in load size. In both cases, the optimal sequence of messages is given, and a closed-form solution to the DLT problem is derived. Rosenberg et al. [2] address the DLT model with return messages, and restrict the framework to bus networks. The theoretical results are stated under an affine communication cost model and under a linear computation cost model. Additionally, they allow worker processors to be slowed down during the computation by incoming external load.

Our contributions to this problem are as follows. We extend the state-of-art on the master-slave tasking problem by incorporating the results collection in the problem formulation. As a first step in this research direction, we restrict ourselves to tree-shaped networks, extending the works presented in [29, 30, 126]. We show how to obtain and build an asymptotically optimal schedule using a linear program. Further, we extend the bandwidth-centric principle to account for results collection. This theoretical knowledge is then embedded within autonomous heuristics that can respond to system load fluctuations. This work has been published in [18], and is presented in greater details in [19].

Chapter 3

Stencil code applications

This chapter starts with an introduction to stencil computations and presents the main optimization techniques for improving sequential performance of stencil codes. Thereafter, we proceed with the main parallelization technique of stencil codes, namely domain decomposition (DD). We emphasize the reasons that make DD not adapted to heterogeneous and dynamic computing systems, and conclude with a summary of our contributions.

3.1 Introduction to stencil computations

Stencil codes form the basis for a wide range of scientific applications: Iterative solvers, Monte Carlo simulations and image filtering applications all rely on some form of stencil computation. These programs are called *stencil codes* because each element in a multidimensional array is updated with contributions from a subset of its neighbors (see Figure 3.1). For each iteration, the stencil kernel is applied to all the array elements - usually the boundaries receive a special treatment. Stencil codes are among the most time-consuming routines of the aforementioned applications, and therefore it makes sense to aspire for ultimate performance.

There are two types of *locality* that can be exploited to improve the performance of stencil codes [185]. There is *spatial* locality when accessing neighboring points (in address space), and there is *temporal* locality when array elements are reused several times before being evicted from cache. Roughly speaking, spatial locality deals with the data layout, i.e. how the multidimensional array is mapped into address space, while temporal locality deals with the data access patterns.

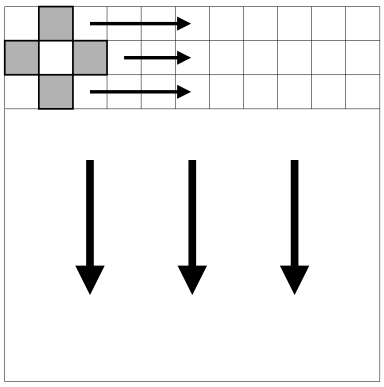


Figure 3.1: Access pattern for a 4-point stencil applied on a 2D array.

3.2 Sequential optimizations

Stencil codes exhibit a particularly poor performance with respect to memory caches. This poor performance is imputed to the fact that each array element is accessed a small, constant number of times per iteration (equal to the number of points in the stencil kernel). For large problem sizes, array elements must be brought into cache several times per iteration, degrading dramatically the overall performance. Reorganizing these computations in order to efficiently utilize the memory hierarchy of modern computer architectures has been the subject of a wealth of research.

Cache blocking or *tiling* is the standard transformation technique which improves locality by moving reuses to the same data closer in time [3, 65, 123, 124, 128, 132, 133, 136, 148, 152, 157, 158, 173, 177, 183, 186, 193]. Tiling reduces the working sets by grouping the updates into rectangular blocks that are processed one after another, in order to reduce capacity misses. Most research on tiling has focused on single-level cache and has been applied to perfectly-nested loops [65, 128, 177]. Multi-level memory hierarchy has been considered by Kodukula et al. [123] and by Yi et al. [193] that propose a compiler technique for transforming loop nests into recursive form.

Li and Song [173] proposed tiling schemes that interleave multiple iterations so that reuse can be exploited across multiple iterations of the time-step loop. Their technique is a combination of loop skewing and tiling, and improves temporal data reuse in secondary cache. They also provide a compiler framework that automates such transformations [136]. Wonnacott [189] proposes a similar approach called *time skewing* that combines blocking in both data and time domains. Jin et al. [112] push this idea further by presenting *recursive prismatic time skewing*, which partitions the iteration space of loop nests into skewed prisms with both spatial and temporal dimensions. In the same genre, Rastello and Robert [157] provide results for minimizing the number of accessed data throughout the computation of a tile (called the *cumulative footprint* of the tile) by utilizing parallelepiped tiles. Additional optimization techniques are reported in Paper 7.

3.3 Parallel implementation issues

3.3.1 Domain decomposition

Usually, stencil applications are parallelized with the domain decomposition (DD) paradigm. Domain decomposition consists in distributing the computational domain of the problem across the processors (see Figure 3.2). Then, during the execution, computation and communication phases alternate, as neighboring processors (in the logical decomposition) need to periodically exchange data located on the boundaries of their local domains.

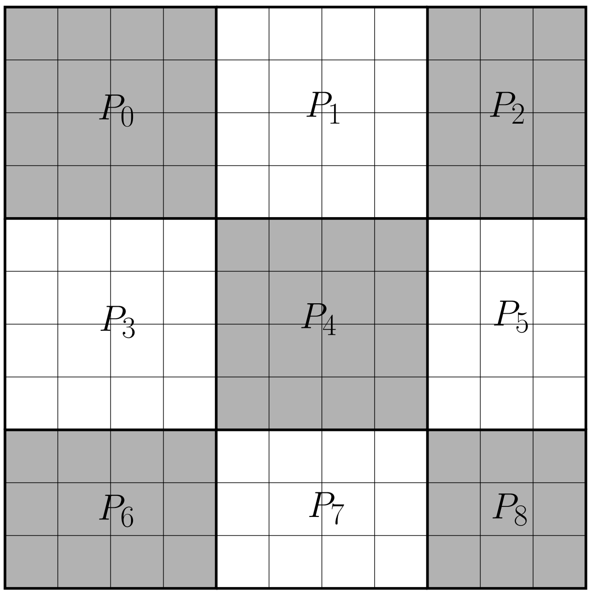


Figure 3.2: 11×11 2D array decomposed across 9 processors.

Domain decomposition methods have been studied extensively because of their utility in a wide range of application areas such as, chemistry, solid and fluid mechanics, or weather forecast simulations. Domain decomposition methods are efficient only when the computational load is well balanced among the processors. Effectively, the processors being tightly coupled by the communication phases, the execution proceeds at the pace of the slowest processor. On homogeneous and stable systems, DD methods may be the simplest and yet the most efficient way of parallelizing stencil code applications. In heterogeneous and dynamic environments on the other hand, things get complicated since the load must be balanced among the processors proportionally to their computational speeds throughout the execution.

3.3.2 Heterogeneous load distributions

Several studies have been conducted on the problem of distributing computations proportionally to processor speeds [26, 27, 33, 113, 114, 139]. In most cases, the problem is reduced to the problem of partitioning some mathematical objects, such as matrices, sets or graphs [73]. The main difficulty resides in the combinatorial nature of the problem which typically turns out to be NP-complete.

Although promising, these efforts have solely focused on the heterogeneous aspect of the underlying platform but not on its dynamic aspect. Hence, even though efficient - i.e. polynomial - heuristics can be derived, the dynamic nature of the underlying platform makes static strategies not well suited over time. In dynamic environments, the processor speeds and network contention will fluctuate during the execution requiring online load redistribution mechanisms.

Hence, the dynamic aspect of the problem is somewhat more challenging than the heterogeneous one, as it introduces the problem of online load redistribution, that is, when and how should the load be redistributed to respond to performance resource fluctuations. Besides, how can one measure the quality of a load distribution? Beaumont et. al. [33] consider the matrix multiplication problem in heterogeneous and dynamic environments, and propose to redistribute the load periodically. Still, one must find a good load redistribution frequency, because a too conservative approach would not result in significant improvements, whereas a too aggressive approach would incur too much overhead.

An important point stressed by Beaumont et. al. is the necessity to minimize the amount of communication when redistributing the load. The amount and location of the data should be taken into account in order to keep the relative position of the processors, that is, to maintain the number and identity of the neighbors of each processor. Otherwise the cost of the redistribution may be prohibitive. Similarly Mahanti and Eager [139] conclude that data migration costs should be minimized for efficient redistribution, and propose redistribution policies that try to leave the relative position of the nodes unaltered. In their work, they consider data redistribution following addition/removal of processing nodes. They find that allocating data to a new node from the center of the computational domain reduces data migration costs compared to allocation from the edge, and addition in groups is beneficial compared to repeated single additions.

Although these studies on DD methods within heterogeneous environments present interesting results that give some insights on the problem complexity, there are still major issues that have not been addressed. First, the different strategies typically rely on a centralized algorithm to (re)distribute the work among the heterogeneous processors. This clearly poses the question of the scalability of the approach when online redistribution cannot be ignored. On the other hand, the problem of online load redistribution frequency is difficult to address without disposing of some form of centralized information about the platform state. Finally, and maybe more importantly, fault tolerance mechanisms are still needed either within homogeneous or heterogeneous environments.

3.3.3 Fault tolerance

Currently, the most common technique for handling fault tolerance within DD methods is checkpoint/restart. That is, checkpoints are saved to disk periodically, and if a processor fails, then the computation halts and restarts from the last consistent checkpoint. For applications that have very long execution times on a very large number of processors, failures are more likely to be the rule rather than the exception. For those applications, the checkpoint/restart technique could take longer than the time to the next failure. There is therefore a need to survive failures without relying on global recovery operations. In other words, if a processor fails, it is desirable that the other processors are able to continue the computation, and do not have to wait for the faulty processor to recover. This constitutes (we believe) the main drawback of DD methods, which does not allow for an efficient way to handle resource failures.

Engelmann and Geist [85] present *naturally fault tolerant* algorithms that are scalable and resilient to failures. Such algorithms must have the ability to tolerate failures through the mathematical properties of the problem by continuing the computation without the failed processors. They show how *chaotic relaxation* and *meshless* methods can be used to derive naturally fault tolerant finite difference methods. However, they advertise that multiple failures of neighboring processors could render the final solution quite incorrect. Moreover, not all scientific problems have the natural fault tolerance property, which urges the need for alternative fault tolerant mechanisms.

3.4 Contributions

This section presents our contributions to the problem of implementing stencil code applications on modern computing systems. Our goal is to investigate the performance of the MS paradigm when implementing stencil applications. Instead of working with trivial applications such as Mandelbrot fractals for instance, we decided to work with stencil applications used by researchers at NTNU. In this way, we could obtain real life applications for our experiments, and provide parallel implementations that might be useful for the researchers working on the applications.

3.4.1 Image filtering application

The first attempt for applying the MS paradigm to stencil code applications was on an image filtering application, known as *matched filtering* [58]. This application is used within medical imaging, in order to detect blood vessels in Computer Tomography (CT) images. A CT image is a cross-section of the human body. By detecting blood vessels in multiple CT images that are piled up, one is able to construct a 3D representation of the blood vessel networks (see Figure 3.3).

Ole Christian Eidheim - a PhD student of the Department of Computer and Information Science at NTNU - has implemented a sequential matched filtering application. The input of the application is a pile of gray-scale images that are filtered through an image correlation step. The correlation kernel is a Gaussian hill, which is rotated in all directions and scaled to several sizes. The sequential implementation is rather slow, and a parallel implementation would allow to use higher-resolution CT images, to treat more pictures, and thus to obtain higher-quality 3D representations.

The matched filtering application is, in principle, quite easy to parallelize. The input images can be divided into blocks that can be processed in parallel, independently of each other. When all the parts of the image have been filtered, the application requires the global pixel minimum and maximum, in order to normalize the pixel values within the range $[0, 255]$. The processors must hence perform two *Allreduce* operations, before starting to normalize their data and write them to disk. Therefore, it is very important that all the processors finish computing at the same time, in order to not wait idle for the collective communications.

Thus, we are facing a scheduling problem with makespan minimization for objective. We designed a new MS scheduling strategy, that hands out batches of tasks of decreasing sizes, and compared it to other strategies suggested in the literature. The parallel implementation involves parallel I/O in order to circumvent the possible bottleneck incurred by the master, and multi-threading to prevent possible processor idleness. This work has been published in [163], and was conducted during the Master thesis of Einar M. R. Rosenvinge, under the supervision of Anne C. Elster and myself.

3.4.2 Lattice gauge theories

This work relates the main issues associated with the porting of Markov-chain based Monte Carlo (MC) simulations from shared-memory architecture systems

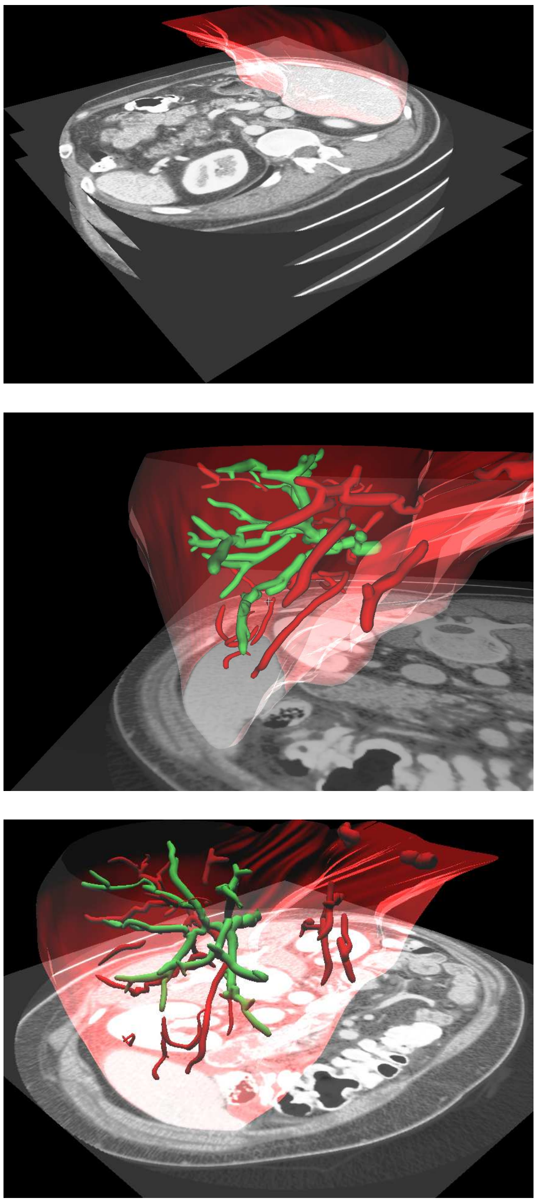


Figure 3.3: 3D reconstructions of blood vessel networks of a human liver after filtering a pile of 2D CT images. Reconstructions performed by Ole Christian Eidheim.

to distributed-memory architecture clusters. The application, called London, is a leading-edge application developed by physicists at NTNU to study the possibility of a new state of matter for hydrogen [9]. Liquid metallic hydrogen may be a superconductor or superfluid that features dissipationless electrical currents or mass flow. The London application is one of the most time consuming of all the high performance applications developed at NTNU.

Like for many high performance computing legacy codes, the main challenge of the port onto distributed memory systems is to depart from the shared-memory programming style. In our case, the code had about 5000 lines with 100 subroutine calls involving sending and receiving of data as well as over 60 calls to SHMEM barrier routines. In addition, it relied heavily on shared memory concepts through about 50 SHMEM_PUT and SHMEM_GET calls.

A previous port onto clusters performed by Lund [137] revealed that one needs to significantly change the computing paradigm in order to obtain a successful port. Indeed, Lund replaced all the SHMEM calls by one-sided MPI subroutines, and obtained an implementation that worked in the same way as the original shared-memory implementation, i.e. with a lot of small messages. As current implementations of MPI-2 calls such as MPI_PUT and MPI_GET continue to have serious performance issues on distributed memory systems with relatively slow interconnect, this approach resulted in a huge communication overhead.

The presence of these numerous small messages is due to the celebrated Metropolis algorithm [145] that is employed within the application. Effectively, Metropolis MC dynamics come with the *detailed balance condition* that prevents the simultaneous update of adjacent sites (according to the application stencil). On shared-memory machines, synchronization is cheap, and one can afford to use a large number of small messages. However, the detailed balance condition poses a serious challenge for the parallelization in distributed memory environments, as communication becomes very expensive compared to computation. Hence, one must find mechanisms that ensure at all times that processors do not update simultaneously adjacent sites, with a minimum communication overhead.

Since our aim was to evaluate and compare the performance of the MS paradigm against a more traditional DD method, we needed to provide two implementations of the London application: One with the MS paradigm, and one based on DD methods. When looking in the literature, we did not find any satisfactory parallel implementation that efficiently handled the detailed balance condition. Most studies [97, 111, 138, 165] rely on DD methods that generate several data transfers between neighboring processors per iteration, albeit conventional wisdom argues that data should be grouped for communication [97, 156].

Our first contribution consists in new parallel algorithms based on DD methods that are scalable, and that minimize the amount of messages exchanged throughout the execution. This work has been published in [17]. Then, we implemented a MS version of the code, and compared the two implementations. This work has been published in [15].

3.4.3 Stencil code optimization

Throughout the work done with the London application, we have first focused on parallel performance, but became interested in sequential optimization as well. The data dependencies implied by the detailed balance condition brought the early London developers to use the well-known Red-Black checkerboard ordering scheme. The Red-Black checkerboard algorithm accesses all the "red" array elements (where sum of coordinates is even) to compute values for the "black" array elements (where sum of coordinates is odd), then it does the other way around using black array elements to update red array elements [154, 159, 183]. This memory access pattern reduces the number of data dependencies, which results in greater parallelism exposed to the compiler. However, the Red-Black ordering scheme harms the performance for sufficiently large problem sizes, because of its non-contiguous data access pattern.

The final piece of work presented in this thesis contributes to the sequential optimization of stencil code computations by presenting techniques that improve spatial and temporal localities of the data. In particular, we study the use of skewed data layouts, that turn out to be more cache-friendly than traditional row-major or column-major storage orders. We provide theoretical and experimental results that validate the superiority of skewed data-layouts for two simple, but fundamental stencil kernels. Further, we show how to automate the detection of situations where the use of skewed data layouts are beneficial.

Chapter 4

Conclusion

This thesis has emphasized the problems that heterogeneity, variability and unreliability exhibited by high performance computing systems introduce at the scheduling level. Even simple problems such as independent-task scheduling require the deployment of sophisticated algorithms. In addition, efficiently managing data movements remains an omnipresent challenge at all the levels of the computing system. This thesis provides algorithmic and scheduling techniques that help addressing some of these challenges.

4.1 Contributions

4.1.1 Master-slave tasking

We advocate in Paper 1 the necessity to deploy several masters to achieve scalable performance. We demonstrated that the problem of finding the most profitable locations for hosting the masters is NP-hard, but nonetheless proposed an effective location-aware heuristic. We also highlighted the strong connections that this problem has with Facility Location problems, and provided a model for establishing and operating the master locations.

We presented in Paper 2 a distributed method, which is an efficient, practical and scalable implementation of the bandwidth-centric principle [30]. We proposed a local scheduling strategy that reduces the amount of tasks buffered during steady-state, and thus reduces the length of the startup and wind-down phases.

We then considered in Paper 3 computing systems where the communication links exhibit bandwidth asymmetry. We derived theoretical results that extend the bandwidth-centric principle from one to two dimensions, i.e. when the cost of returning the computational results to the master is represented in the problem formulation. We then conceived a rendezvous protocol for enabling decentralized schedules. We also provided a task-flow control mechanism, that automatically regulates the flows of tasks and results that circulate in the system according to system load fluctuations.

We showed how to eliminate the startup phase description required to enter the steady-state regime (see Papers 2 and 3). The idea is simply to make the nodes operate as if they were already in steady-state, with the help of fake data transfers if necessary. We applied this technique successfully to tree-shaped platforms, and conjecture that it can be applied to arbitrary graphs as well. This technique contributes to ease the implementation and deployment of MS applications.

Finally, in Paper 4, we presented a novel MS scheduling strategy for minimizing the application makespan, that is well suited for heterogeneous and dynamic computing systems.

4.1.2 Stencil code applications

We considered in Papers 5 and 6 the parallelization of a LGT model on a SMP cluster. We first designed parallel algorithms based on domain decomposition that are scalable, that reduce the amount of communication messages to the minimum, and that are adapted for the peculiarity of LGT models. Then, we investigated the suitability of another parallelization method by comparing these parallel algorithms to a MS implementation.

As we gained insight into stencil computations, we focused on sequential optimization, and presented new transformation techniques intended to better utilize the memory hierarchy of modern computers. In Paper 7, we demonstrate - and quantify - how spatial locality can be improved by using skewed data layouts as opposed to the traditional row-major and column-major storage orders. The other main contribution is the stencil decomposition transformation that improves temporal locality. Overall, we conclude that efficient data access patterns alone are not sufficient, and one must change the data layout in order to match these data access pattern in order to improve performance.

Finally, we helped the *Correlated Condensed Matter Systems* group and the *Algorithm and Visualization Group* group at NTNU in their work by providing efficient parallel implementations of two forefront stencil code applications.

4.1.3 Summary

The initial research questions stated in this thesis were the following:

Q-1 **How should the MS paradigm be enhanced to improve its scalability?**

Q-2 **Can MS scheduling techniques be applied to stencil code applications?**

Our answer to question Q-1 can be summarized as follows. If the master is the bottleneck of the application, then deploying several masters is the only solution to improve the scalability. However, in heterogeneous environments, the different masters should be placed at strategic locations in the system in order to efficiently exploit the computing resources. Further, we showed that the scheduling problem becomes much more complicated when the results collection is taken into consideration, as it becomes necessary to synchronize all the nodes of the system to construct an asymptotically optimal schedule. We demonstrated that synchronizing the entire system is rather impractical and inadequate for dynamic environments. Instead, distributed autonomous strategies can better handle and respond to system load fluctuations. Thus, even though several masters are deployed, efficient distributed scheduling strategies are still needed to deal with the scale, heterogeneity and variability of the system.

Our answer to question Q-2 can be summarized as follows. Stencil code applications implemented under the MS paradigm can achieve scalability if the master is used to control the execution (i.e. tells who computes what), instead of being a data access point. Otherwise, scalability can only be achieved by deploying multiple masters.

4.2 Future work

The theoretical work on independent task scheduling within heterogeneous environments presented in this thesis has been possible only by restraining the framework. We focused essentially on tree-shaped platforms, mostly to avoid routing decision making. An obvious direction for future work would be to consider arbitrary graphs. At first sight, the scheduling problem becomes more complex with graph-shaped platforms. We believe that task-flow control mechanisms such as the one presented in this study is the way to go for conceiving efficient distributed scheduling strategies.

On the practical side, a direction for future work would be to design the hybrid approach presented in Paper 6, where the MS and DD paradigms would be used in concert. Each master would manage a group of slaves while DD methods would be utilized to assign parts of the computational domain to the different masters.

This idea is close in spirit to the hierarchical master-slave technique, but we are not aware of its application to stencil code applications. Such hybrid approach combines the benefits of the two paradigms: The MS flexibility and robustness to the DD scalability. Adaptation to system load fluctuations would consist in balancing the load - or the slaves - between the masters. The strength of this hybrid approach lies in its flexibility: Slaves can be affiliated to any master whenever needed, and can even act as masters on demand.

Bibliography

[1] URL: http://www.top500.org.

[2] M. Adler, Y. Gong, and A. L. Rosenberg. Optimal Sharing of Bags of Tasks in Heterogeneous Clusters. In *15th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA'03)*, pages 1–10. ACM Press, 2003.

[3] N. Ahmed, N. Mateev, and K. Pingali. Tiling Imperfectly-Nested Loop Nests. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 31, Washington, DC, USA, 2000. IEEE Computer Society.

[4] K. Aida, W. Natsume, and Y. Futakata. Distributed Computing with Hierarchical Master-worker Paradigm for Parallel Branch and Bound Algorithm. In *3rd International Symposium on Cluster Computing and the Grid*, page 156, 2003.

[5] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model – One step closer towards a realistic model for parallel computation. In *SPAA '95: Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 95–105, New York, NY, USA, 1995. ACM Press.

[6] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti@home: An Experiment in Public-Resource Computing. *Communications of the ACM*, 45(11):56–61, 2002.

[7] G. Ausiello, M. Protasi, A. Marchetti-Spaccamela, G. Gambosi, P. Crescenzi, and V. Kann. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer-Verlag New York, Inc., 1999.

[8] B. Awerbuch and Y. Shavitt. Topology Aggregation for Directed Graphs. *IEEE/ACM Trans. Netw.*, 9(1):82–90, 2001.

[9] E. Babaev, A. Sudbø, and N. W. Ashcroft. A Superconductor to Superfluid Phase Transition in Liquid Metallic Hydrogen. *Nature*, 431:666, 2004.

[10] S. B. Baden and S. J. Fink. Communication overlap in multi-tier parallel algorithms. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–20, Washington, DC, USA, 1998. IEEE Computer Society.

[11] C. Banino. Optimizing Locationing of Multiple Masters for Master-Worker Grid Applications. In *PARA'04: International Conference on Applied Parallel Computing*, LNCS 2367, pages 1041–1050. Springer Verlag, 2004.

[12] C. Banino. Optimizing Locationning of Multiple Masters for Master-Worker Grid Applications: A Thorough Study. Technical Report 09/04, Dept. of Computer and Info. Science, Norwegian University of Science and Technology, September 2004. URL: http://www.idi.ntnu.no/∼banino.

[13] C. Banino. A Distributed Procedure for Bandwidth-Centric Scheduling of Independent-Task Applications. In *19th IEEE International Parallel and Distributed Processing Symposium, IPDPS'2005*, pages 48a – 48a, 04-08 April 2005.

[14] C. Banino, O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Scheduling Strategies for Master-Slave Tasking on Heterogeneous Processor Platforms. *IEEE Transactions on Parallel and Distributed Systems*, 15(4):319–330, 2004.

[15] C. Banino-Rokkones. Domain Decomposition vs. Master-Slave in Apparently Homogeneous Systems. In *HCW '07: Proceedings of the 16th Heterogeneity in Computing Workshop*, page To appear, Long Beach, California, USA, 2007. IEEE Computer Society.

[16] C. Banino-Rokkones. Location-Aware Master-Slave Tasking on the Grid. *Journal of Future Generation Computing Systems*, 2007. To appear.

[17] C. Banino-Rokkones, J. Amundsen, and E. Smørgrav. Parallelizing Lattice Gauge Theory Models on Commodity Clusters. In *2006 IEEE International Conference on Cluster Computing (CLUSTER 2006), September 25-28 2006, Barcelona, Spain*. IEEE Computer Society, 2006.

[18] C. Banino-Rokkones, O. Beaumont, and L. Natvig. Master-Slave Tasking on Asymmetric Networks. In *Euro-Par*, Lecture Notes in Computer Science, pages 167–176. Springer, 2006.

[19] C. Banino-Rokkones, O. Beaumont, and L. Natvig. Master-Slave Tasking on Asymmetric Tree-Shaped Networks. Technical Report 02/06, Dept. of Computer and Info. Science, Norwegian University of Science and Technology, September 2006. URL: http://www.idi.ntnu.no/∼banino/research/research.html.

[20] A. Baratloo, M. Karaul, Z. M. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the web. In *Proc. of the 9th Int'l Conf. on Parallel and Distributed Computing Systems (PDCS-96)*, 1996.

[21] G. D. Barlas. Collection-Aware Optimum Sequencing of Operations and Closed-Form Solutions for the Distribution of a Divisible Load on Arbitrary Processor Trees. *IEEE Trans. Parallel Distrib. Syst.*, 9(5):429–441, 1998.

[22] J. Basney, R. Raman, and M. Livny. High Throughput Monte Carlo. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, TX, March 1999.

[23] H. Bast. On Scheduling Parallel Tasks at Twilight. *Theory of Computing Systems*, 33(5):489–563, November 2000.

[24] S. Bataineh, T.-Y. Hsiung, and T. G. Robertazzi. Closed Form Solutions for Bus and Tree Networks of Processors Load Sharing a Divisible Job. *IEEE Trans. Comput.*, 43(10):1184–1196, 1994.

[25] S. Bataineh and T. Robertazzi. Performance Limits for Processor Networks with Divisible Jobs. *IEEE Transactions on Aerospace and Electronic Systems*, 33:1189–1198, Octobre 1997.

[26] O. Beaumont, V. Boudet, and A. Petitet. A Proposal for a Heterogeneous Cluster ScaLAPACK (Dense Linear Solvers). *IEEE Trans. Comput.*, 50(10):1052–1070, 2001.

[27] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert. Matrix Multiplication on Heterogeneous Platforms. *IEEE Trans. Parallel Distrib. Syst.*, 12(10):1033–1051, 2001.

[28] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, L. Marchal, and Y. Robert. Centralized Versus Distributed Schedulers for Multiple Bag-of-Task Applications. In *IPDPS'2006: Proceedings of the 20th International Symposium on Parallel and Distributed Processing*. IEEE Computer Society Press, 2006.

[29] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Bandwidth-Centric Allocation of Independent Tasks on Heterogeneous Platforms. Technical Report RR-2001-25, LIP, ENS Lyon, France, June 2001. URL: http://www.ens-lyon/∼yrobert.

[30] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Bandwidth-Centric Allocation of Independent Tasks on Heterogeneous Platforms. In *International Parallel and Distributed Processing Symposium IPDPS'2002*, pages 67–72. IEEE Computer Society Press, 2002.

[31] O. Beaumont and A. Legrand. Pipelining Broadcasts on Heterogeneous Platforms. *IEEE Trans. Parallel Distrib. Syst.*, 16(4):300–313, 2005. Student Member-Loris Marchal and Senior Member-Yves Robert.

[32] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert. Steady-State Scheduling on Heterogeneous Clusters: Why and How? In *6th Workshop on Advances in Parallel and Distributed Computational Models, APDCM 2004*, page 171a (8 pages). IEEE Computer Society Press, 2004.

[33] O. Beaumont, A. Legrand, F. Rastello, and Y. Robert. Dense Linear Algebra Kernels on Heterogeneous Platforms: Redistribution Issues. *Parallel Comput.*, 28(2):155–185, 2002.

[34] O. Beaumont, A. Legrand, and Y. Robert. Optimal Algorithms for Scheduling Divisible Workloads on Heterogeneous Systems. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 98.2, Washington, DC, USA, 2003. IEEE Computer Society.

[35] O. Beaumont, A. Legrand, and Y. Robert. Scheduling Divisible Workloads on Heterogeneous platforms. *Parallel Computing*, 29:1121–1152, September 2003.

[36] O. Beaumont, A. Legrand, and Y. Robert. The Master-Slave Paradigm with Heterogeneous Processors. *IEEE Transactions on Parallel and Distributed Systems*, 14:897–908, 2003.

[37] O. Beaumont, A. Legrand, and Y. Yang. Scheduling Divisible Loads on Star and Tree Networks: Results and Open Problems. *IEEE Trans. Parallel Distrib. Syst.*, 16(3):207–218, 2005. Member-Henri Casanova and Senior Member-Yves Robert.

[38] O. Beaumont, L. Marchal, and Y. Robert. Scheduling Divisible Loads with Return Messages on Heterogeneous Master-Worker Platforms. In *International Conference on High Performance Computing HiPC'2005*, LNCS, pages 123–132. Springer Verlag, 2005.

[39] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov. Adaptive Computing on the Grid Using AppLeS. *IEEE Trans. Parallel Distrib. Syst.*, 14(4):369–382, 2003.

[40] D. Bertsimas and D. Gamarnik. Asymptotically optimal algorithm for job shop scheduling and packet routing. *Journal of Algorithms*, 33(2):296–318, 1999.

[41] V. Bharadwaj and G. Barlas. Scheduling Divisible Loads with Processor Release Times and Finite Size Buffer Capacity Constraints in Bus Networks. *Cluster Computing*, 6(1):63–74, 2003.

[42] V. Bharadwaj, D. Ghose, and V. Mani. Multi-Installment Load Distribution in Tree Networks with Delays. *IEEE Transactions on Aerospace and Electron. Systs*, 31:555–567, 1995.

[43] V. Bharadwaj, D. Ghose, V. Mani, and T. Robertazzi. *Scheduling Divisible Loads in Parallel and Distributed Systems*. IEEE Computer Society Press, Aug 1996.

[44] V. Bharadwaj, D. Ghose, V. Mani, and T. Robertazzi. *Scheduling Divisible Loads in Parallel and Distributed Systems*. IEEE Computer Society Press, 1996.

[45] V. Bharadwaj, D. Ghose, and T. G. Robertazzi. Divisible Load Theory: A New Paradigm for Load Scheduling in Distributed Systems. *Cluster Computing*, 6(1):7–17, 2003.

[46] J. Blazewicz and M. Drozdowski. The Performance Limits of a Two-dimensional Network of Load Sharing Processors. *Foundations of Computing and Decision Sciences*, 21:3–15, 1996.

[47] J. Blazewicz and M. Drozdowski. Distributed Processing of Divisible Jobs with Communication Startup Costs. *Discrete Appl. Math.*, 76(1-3):21–41, 1997.

[48] J. Blazewicz, M. Drozdowski, F. Guinand, and D. Trystram. Scheduling a Divisible Task in a Two-dimensional Toroidal Mesh. In *Proceedings of the third international conference on Graphs and optimization*, pages 35–50, Amsterdam, The Netherlands, 1999. Elsevier Science Publishers B. V.

[49] J. Blazewicz, M. Drozdowski, and M. Markiewicz. Divisible Task Scheduling – Concept and Verification. *Parallel Computing*, 25:87–98, January 1999.

[50] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, 2000.

[51] H. Casanova. SimGrid: A Toolkit for the Simulation of Application Scheduling. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, page 430. IEEE Computer Society, 2001.

[52] H. Casanova. Modeling Large-Scale Platforms for the Analysis and the Simulation of Scheduling Strategies. In *18th International Parallel and Distributed Processing Symposium*, page 170. IEEE Computer Society Press, Apr 26-30 2004.

[53] H. Casanova, M. Kim, J. S. Plank, and J. J. Dongarra. Adaptive Scheduling for Task Farming with Grid Middleware. *Int. J. High Perform. Comput. Appl.*, 13(3):231–240, 1999.

[54] T. L. Casavant and J. G. Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Trans. Softw. Eng.*, 14(2):141–154, 1988.

[55] A. J. Chakravarti, G. Baumgartner, and M. Lauria. The Organic Grid: Self-Organizing Computation on a Peer-to-Peer Network. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 35(3):373–384, 2005.

[56] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear Array Layouts for Hierarchical Memory Systems. In *International Conference on Supercomputing*, pages 444–453, 1999.

[57] S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottethodi. Recursive Array Layouts and Fast Parallel Matrix Multiplication. In *SPAA ’99:*

Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures, pages 222–231, New York, NY, USA, 1999. ACM Press.

[58] S. Chaudhuri, S. Chatterjee, N. Katz, M. Nelson, and M. Goldbaum. Detection of Blood Vessels in Retinal Images Using Two-Dimensional Matched Filters. *IEEE Transactions on Medical Imaging*, pages 263–269, 1989.

[59] T. Chen and C. Chang. Skewed Data Partition and Alignment Techniques for Compiling Programs on Distributed Memory Multicomputers. *J. Supercomput.*, 21(2):191–211, 2002.

[60] A. Chien, B. Calder, S. Elbert, and K. Bhatia. Entropia: Architecture and Performance of an Enterprise Desktop Grid System. *Journal of Parallel and Distributed Computing*, 63:597, May 2003.

[61] S. Choi, M. Baik, J. Gil, S. Jung, and C. Hwang. Adaptive Group Scheduling Mechanism Using Mobile Agents in Peer-to-Peer Grid Computing Environment. *Applied Intelligence*, 25(2):199–221, 2006.

[62] P. Chrétienne, E. G. J. Coffman, J. K. Lenstra, and Z. Liu, editors. *Scheduling Theory and its Applications*. John Wiley and Sons, 1995.

[63] R. L. Church and C. S. ReVelle. The Maximal Covering Location Problem. *Papers of the Regional Science Association*, 32:101–118, 1974.

[64] W. Cirne, D. P. da Silva, L. Costa, E. Santos-Neto, F. V. Brasileiro, J. P. Sauvé, F. A. B. Silva, C. O. Barros, and C. Silveira. Running Bag-of-Tasks Applications on Computational Grids: The MyGrid Approach. In *ICPP*, pages 407–. IEEE Computer Society, 2003.

[65] S. Coleman and K. S. McKinley. Tile Size Selection Using Cache Organization and Data Layout. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 279–290, New York, NY, USA, 1995. ACM Press.

[66] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken. LogP: A Practical Model of Parallel Computation. *Commun. ACM*, 39(11):78–85, 1996.

[67] J. Current, M. Daskin, and D. Schilling. Discrete Network Location Models. In Z. Drezner and H. Hamacher, editors, *Facility Location Theory: Applications and Methods*, chapter 3, pages 81–118. Springer-Verlag, Berlin, 2002.

[68] D. G. D and V. Mani. Distributed Computation with Communication Delays: Asymptotic Performance Analysis. *Journal of Parallel and Distributed Computing*, 23:293–305, December 1994.

[69] C. Dasgupta and B. I. Halperin. Phase Transition in a Lattice Model of Superconductivity. *Physical Review Letters*, 47:1556–1560, Nov. 1981.

[70] M. Daskin, L. V. Snyder, and R. T. Berter. Facility Location in Supply Chain Design. In A. Langevin and D. Riopel, editors, *Logistics Systems: Design and Optimization*, chapter 2, pages 39–66. Kluwer, 2005.

[71] J. Díaz, J. Petit, and M. Serna. A Survey of Graph Layout Problems. *ACM Comput. Surv.*, 34(3):313–356, 2002.

[72] C. Ding and Y. He. A Ghost Cell Expansion Method for Reducing Communications in Solving PDE Problems. In *Supercomputing ’01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 50–50, New York, NY, USA, 2001. ACM Press.

[73] J. Dongarra and A. Lastovetsky. An Overview of Heterogeneous High Performance and Grid Computing. In *Engineering The Grid: Status and Perspective*. American Scientific Publishers, 2006.

[74] J. J. Dongarra and D. W. Walker. Software Libraries for Linear Algebra Computations on High Performance Computers. *SIAM Rev.*, 37(2):151–180, 1995.

[75] M. Drozdowski and W. Gazek. Scheduling Divisible Loads in a Three-Dimensional Mesh of Processors. *Parallel Computing*, 25:381–404, April 1999.

[76] M. Drozdowski and P. Wolniewicz. Experiments with Scheduling Divisible Tasks in Clusters of Workstations. In *Proceedings of the 6th International Euro-Par Conference on Parallel Processing*, pages 311–319. Springer-Verlag, 2000.

[77] M. Drozdowski and P. Wolniewicz. Experiments with scheduling divisible tasks in clusters of workstations. In *Proceedings of Euro-Par 2000: Parallel Processing*, LNCS 1900, pages 311–319. Springer, 2000.

[78] M. Drozdowski and P. Wolniewicz. Divisible Load Scheduling in Systems with Limited Memory. *Cluster Computing*, 6(1):19–29, 2003.

[79] M. Drozdowski and P. Wolniewicz. Out-of-Core Divisible Load Processing. *IEEE Trans. Parallel Distributed Systems*, 14(10):1048–1056, 2003.

[80] P. F. Dutot. Master-slave Tasking on Heterogeneous Processors. In *International Parallel and Distributed Processing Symposium*, page 25b. IEEE Computer Society Press, April 2003.

[81] P. F. Dutot. Complexity of Master-slave Tasking on Heterogeneous Trees. *European Journal on Operationnal Research*, 164(3):690–695, August 2005.

[82] H. El-Rewini, H. H. Ali, and T. Lewis. Task Scheduling in Multiprocessing Systems. *Computer*, 28(12):27–37, 1995.

[83] H. El-Rewini and T. G. Lewis. Scheduling Parallel Program Tasks onto Arbitrary Target Machines. *J. Parallel Distrib. Comput.*, 9(2):138–153, 1990.

[84] W. Elwasif, J. S. Plank, and R. Wolski. Data Staging Effects in Wide Area Task Farming Applications. In *IEEE International Symposium on Cluster Computing and the Grid*, pages 122–129, Brisbane, Australia, May 2001.

[85] C. Engelmann and A. Geist. Super-Scalable Algorithms for Computing on 100, 000 Processors. In V. S. Sunderam, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, editors, *International Conference on Computational Science (1)*, volume 3514 of *Lecture Notes in Computer Science*, pages 313–321. Springer, 2005.

[86] A. Espinosa, T. Margalef, , and E. Luque. Automatic Performance Analysis of Master/Worker PVM Applications with Kpi. In *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 47–55, London, UK, 2000. Springer-Verlag.

[87] G. E. Fagg and J. J. Dongarra. Building and Using a Fault-Tolerant MPI Implementation. *Int. J. High Perform. Comput. Appl.*, 18(3):353–361, 2004.

[88] I. Foster and C. Kesselman. *The Grid, Blueprint for a New Computing Infrastructure*. Morgan Kaufman, San Francisco, 1999.

[89] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[90] A. Gerasoulis and T. Yang. A Comparison of Clustering Heuristics for Scheduling Directed Acyclic Graphs on Multiprocessors. *Journal of Parallel and Distributed Computing*, 16(4):276, 1992.

[91] N. E. Gibbs, W. G. Poole, and P. K. Stockmeyer. An Algorithm for Reducing the Bandwidth and Profile of a Sparse Matrix. *SIAM J Numer. Anal*, 13:236–250, 1976.

[92] P. Golle and I. Mironov. Uncheatable Distributed Computations. In *CT-RSA 2001: Proceedings of the 2001 Conference on Topics in Cryptology*, pages 425–440, London, UK, 2001. Springer-Verlag.

[93] J.-P. Goux, S. Kulkarni, M. Yoder, and J. Linderoth. Master-worker: An enabling framework for applications on the computational grid. *Cluster Computing*, 4(1):63–70, 2001.

[94] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, Cambridge, Massachusetts, USA, 1994.

[95] J. L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, 1988.

[96] D. Guster, A. Al-Hamamah, P. Safonov, and E. Bachman. Computing and Network Performance of a Distributed Parallel Processing Environment Using MPI and PVM Communication Methods. *J. Comput. Small Coll.*, 18(4):246–253, 2003.

[97] F. Gutbrod, N. Attig, and M. Weber. The SU(2)-lattice gauge theory simulation code on the Intel Paragon supercomputer. *Parallel Comput.*, 22(3):443–463, 1996.

[98] T. Hagerup. Allocating Independent Tasks to Parallel Processors: An Experimental Study. *J. Parallel Distrib. Comput.*, 47(2):185–197, 1997.

[99] C.-C. Han, K. G. Shin, and S. K. Yun. On Load Balancing in Multicomputer/Distributed Systems Equipped with Circuit or Cut-Through Switching Capability. *IEEE Trans. Comput.*, 49(9):947–957, 2000.

[100] D. W. Heermann and A. N. Burkitt. *Parallel Algorithms in Computational Science*. Springer-Verlag New York, Inc., New York, NY, USA, 1991.

[101] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[102] E. Heymann, M. A. Senar, E. Luque, and M. Livny. Adaptive Scheduling for Master-Worker Applications on the Computational Grid. In *GRID*

'00: Proceedings of the First IEEE/ACM International Workshop on Grid Computing, pages 214–227, London, UK, 2000. Springer-Verlag.

[103] R. W. Hockney. Performance Parameters and Benchmarking of Supercomputers. *Parallel Computing*, 17(10-11):1111–1130, 1991.

[104] B. Hong and V. K. Prasanna. Bandwidth-Aware Resource Allocation for Computing Independent Tasks in Heterogeneous Computing Systems. In *The 15th Annual International Conference on Parallel and Distributed Computing and Systems (PDCS 2003)*, page 539, November 2003.

[105] B. Hong and V. K. Prasanna. Bandwidth-Aware Resource Allocation for Heterogeneous Computing Systems to Maximize Throughput. In *ICPP*, pages 539–546. IEEE Computer Society, 2003.

[106] B. Hong and V. K. Prasanna. Distributed Adaptive Task Allocation in Heterogeneous Computing Environments to Maximize Throughput. In *International Parallel and Distributed Processing Symposium IPDPS'2004*, page 52b. IEEE Computer Society Press, 2004.

[107] B. Hong and V. K. Prasanna. Performance Optimization of a De-centralized Task Allocation Protocol via Bandwidth and Buffer Management. In *CLADE*, page 108, 2004.

[108] M. E. Houle, A. Symvonis, and D. R. Wood. Dimension-Exchange Algorithms for Token Distribution on Tree-Connected Architectures. *J. Parallel Distrib. Comput.*, 64(5):591–605, 2004.

[109] S. F. Hummel, J. Schmidt, R. N. Uma, and J. Wein. Load-Sharing in Heterogeneous Systems via Weighted Factoring. In *SPAA '96: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 318–328, New York, NY, USA, 1996. ACM Press.

[110] S. F. Hummel, E. Schonberg, and L. E. Flynn. Factoring: A Method for Scheduling Parallel Loops. *Commun. ACM*, 35(8):90–101, 1992.

[111] W. Janke and R. Villanova. Ising model on three-dimensional random lattices: A Monte Carlo study. *Physical Review B*, 66(13):134208–+, Oct. 2002.

[112] G. Jin, J. Mellor-Crummey, and R. Fowler. Increasing Temporal Locality with Skewing and Recursive Blocking. In *Proceedings of SuperComputing 2001*, Denver, CO, 2001.

[113] M. Kaddoura, S. Ranka, and A. Wang. Array Decompositions for Nonuniform Computational Environments. *J. Parallel Distrib. Comput.*, 36(2):91–105, 1996.

[114] A. Kalinov and A. Lastovetsky. Heterogeneous Distribution of Computations Solving Linear Algebra Problems on Networks of Heterogeneous Computers. *J. Parallel Distrib. Comput.*, 61(4):520–535, 2001.

[115] S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. Yelick. Impact of Modern Memory Subsystems on Cache Optimizations for Stencil Computations. In *MSP '05: Proceedings of the 2005 workshop on Memory system performance*, pages 36–43, New York, NY, USA, 2005. ACM Press.

[116] M. T. Kandemir, A. N. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam. A Linear Algebra Framework for Automatic Determination of Optimal Data Layouts. *IEEE Trans. Parallel Distrib. Syst.*, 10(2):115–135, 1999.

[117] N. Karmarkar. A New Polynomial-Time Algorithm for Linear Programming. *Combinatorica*, 4(4):373–395, 1984.

[118] L. G. Khachiyan. A Polynomial Algorithm in Linear Programming. *Doklady Akademia Nauk SSSR*, pages 1093–1096, 1979.

[119] H.-J. Kim. A Novel Optimal Load Distribution Algorithm for Divisible Loads. *Cluster Computing*, 6(1):41–46, 2003.

[120] T. H. Kim. Adaptive Divisible Load Scheduling Strategies for Workstation Clusters with Unknown Network Resources. *IEEE Trans. Parallel Distrib. Syst.*, 16(10):897–907, 2005. Member-Debasish Ghose and Member-Hyoung Joong Kim.

[121] T. Kindberg, A. Sahiner, and Y. Paker. Adaptive Parallelism under Equus. In *Proceedings of 2nd International Workshop on Configurable Distributed Systems*, pages 172–182. IEEE, March 1994.

[122] B. Ko and D. Rubenstein. Distributed Self-Stabilizing Placement of Replicated Resources in Emerging Networks. *IEEE/ACM Trans. Netw.*, 13(3):476–487, 2005.

[123] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric Multi-level Blocking. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 346–357, New York, NY, USA, 1997. ACM Press.

[124] M. Kowarschik and C. Weiß. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. In U. Meyer, P. Sanders, and J. Sibeyn, editors, *Algorithms for Memory Hierarchies — Advanced Lectures*, volume 2625 of *Lecture Notes in Computer Science (LNCS)*, pages 213–232. Springer, 2003.

[125] J. Krarup and P. Pruzan. The Simple Plant Location Problem: Survey and Synthesis. *European Journal of Operations Research*, 12:36–81, 1983.

[126] B. Kreaseck, L. Carter, H. Casanova, and J. Ferrante. Autonomous Protocols for Bandwidth-Centric Scheduling of Independent-Task Applications. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 26.1, Washington, DC, USA, 2003. IEEE Computer Society.

[127] C. P. Kruskal and A. Weiss. Allocating Independent Subtasks on Parallel Processors. *IEEE Trans. Softw. Eng.*, 11(10):1001–1016, 1985.

[128] M. D. Lam, E. E. Rothberg, and M. E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 63–74, New York, NY, USA, 1991. ACM Press.

[129] C. L. Lawson, R. J. Hanson, F. T. Krogh, and D. R. Kincaid. Algorithm 539: Basic Linear Algebra Subprograms for Fortran Usage [F1]. *ACM Trans. Math. Softw.*, 5(3):324–325, 1979.

[130] W. C. Lee. Topology Aggregation for Hierarchical Routing in ATM Networks. *SIGCOMM Comput. Commun. Rev.*, 25(2):82–92, 1995.

[131] A. Legrand, L. Marchal, and Y. Robert. Optimizing the Steady-state Throughput of Scatter and Reduce Operations on Heterogeneous Platforms. *J. Parallel Distrib. Comput.*, 65(12):1497–1514, 2005.

[132] C. Leopold. On Optimal Temporal Locality of Stencil Codes. In *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*, pages 948–952, New York, NY, USA, 2002. ACM Press.

[133] C. Leopold. Tight Bounds on Capacity Misses for 3D Stencil Codes. In *ICCS '02: Proceedings of the International Conference on Computational Science-Part I*, pages 843–852, London, UK, 2002. Springer-Verlag.

[134] K. Li. Improved Methods for Divisible Load Distribution on k-Dimensional Meshes Using Pipelined Communications. *IEEE Trans. Parallel Distributed Systems*, 14:1250–1261, December 2003.

[135] X. Li, B. Veeravalli, and C. C. Ko. Divisible Load Scheduling on a Hypercube Cluster with Finite-Size Buffers and Granularity Constraints. In *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, page 660, Washington, DC, USA, 2001. IEEE Computer Society.

[136] Z. Li and Y. Song. Automatic Tiling of Iterative Stencil Loops. *ACM Trans. Program. Lang. Syst.*, 26(6):975–1028, 2004.

[137] T. A. Lund. Porting a Monte Carlo Code from Shared Memory to Computational Clusters. Master's thesis, Dept. of Computer and Info. Science, Norwegian University of Science and Technology, 2004.

[138] M. Luscher. Solution of the Dirac equation in lattice QCD using a domain decomposition method. *Comput. Phys. Commun.*, 156:209–220, 2004.

[139] A. Mahanti and D. L. Eager. Adaptive Data Parallel Computing on Workstation Clusters. *J. Parallel Distrib. Comput.*, 64(11):1241–1255, 2004.

[140] M. Maheswaran, B. Maniymaran, P. Card, and F. Azzedin. Invisible Network: Concepts and Architecture. In *CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, page 464, Washington, DC, USA, 2002. IEEE Computer Society.

[141] M. Maheswaran, B. Maniymaran, P. Card, and F. Azzedin. MetaGrid: A Scalable Framework for Wide-Area Service Deployment and Management. In *HPCS '02: Proceedings of the 16th Annual International Symposium on High Performance Computing Systems and Applications*, page 61, Washington, DC, USA, 2002. IEEE Computer Society.

[142] V. Mani. An Equivalent Tree Network Methodology for Efficient Utilization of Front-Ends in Linear Network. *Cluster Computing*, 6(1):57–62, 2003.

[143] K. McKinley, S. Carr, and C.-W. Tseng. Improving Data Locality with Loop Transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, 1996.

[144] N. Megiddo, E. Zemel, and S. L. Hakimi. The Maximum Coverage Location Problem. *SIAM Journal on Algebraic and Discrete Methods,* 4(2):253–261, June 1983.

[145] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of State Calculations by Fast Computing Machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

[146] P. D. Michailidis and K. G. Margaritis. Performance Evaluation of Load Balancing Strategies for Approximate String Matching Application on an MPI Cluster of Heterogeneous Workstations. *Journal of Future Generation Computing Systems*, 19(7):1075–1104, 2003.

[147] D. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu. Peer-to-Peer Computing. Technical Report HPL-2002-57, Hewlett Packard Laboratories, Mar 2002.

[148] N. Mitchell, K. Högstedt, L. Carter, and J. Ferrante. Quantifying the multi-level nature of tiling interactions. *International Journal of Parallel Programming*, 26(6):641–670, 1998.

[149] A. Morajko, E. César, P. Caymes-Scutari, T. Margalef, J. Sorribes, and E. Luque. Automatic Tuning of Master/Worker Applications. In J. C. Cunha and P. D. Medeiros, editors, *Euro-Par*, volume 3648 of *Lecture Notes in Computer Science*, pages 95–103. Springer, 2005.

[150] S. Nandy, L. Carter, and J. Ferrante. A-FAST: Autonomous Flow Approach to Scheduling Tasks. In L. Bougé and V. K. Prasanna, editors, *HiPC*, volume 3296 of *Lecture Notes in Computer Science*, pages 363–374. Springer, 2004.

[151] M. A. Palis, J.-C. Liou, and D. S. L. Wei. Task Clustering and Scheduling for Distributed Memory Parallel Architectures. *IEEE Trans. Parallel Distrib. Syst.*, 7(1):46–55, 1996.

[152] N. Park, B. Hong, and V. K. Prasanna. Tiling, Block Data Layout, and Memory Hierarchy Performance. *IEEE Trans. Parallel Distrib. Syst.*, 14(7):640–654, 2003.

[153] C. D. Polychronopoulos and D. J. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Trans. Comput.*, 36(12):1425–1439, 1987.

[154] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 1992.

[155] M. Prieto, I. M. Llorente, and F. Tirado. Data Locality Exploitation in the Decomposition of Regular Domain Problems. *IEEE Trans. Parallel Distrib. Syst.*, 11(11):1141–1150, 2000.

[156] M. J. Quinn and P. J. Hatcher. On the Utility of Communication-Computation Overlap in Data-Parallel Programs. *J. Parallel Distrib. Comput*, 33(2):197–204, 1996.

[157] F. Rastello and Y. Robert. Automatic Partitioning of Parallel Loops With Parallelepiped-Shaped Tiles. *IEEE Trans. Parallel Distributed Systems*, 13(5), may 2002.

[158] G. Rivera and C. Tseng. Eliminating Conflict Misses for High Performance Architectures. In *International Conference on Supercomputing*, pages 353–360, 1998.

[159] G. Rivera and C. Tseng. Tiling Optimizations for 3D Scientific Computations. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 32, Washington, DC, USA, 2000. IEEE Computer Society.

[160] T. Robertazzi. Processor Equivalence for a Linear Daisy Chain of Load Sharing Processors. *IEEE Trans. Aerospace and Electronic Systems*, 29:1216–1221, 1993.

[161] A. L. Rosenberg. Sharing Partitionable Workloads in Heterogeneous NOWs: Greedier Is Not Better. In *CLUSTER '01: Proceedings of the 3rd IEEE International Conference on Cluster Computing*, page 124, Washington, DC, USA, 2001. IEEE Computer Society.

[162] E. Rosenvinge. Online Task Scheduling On Heterogeneous Clusters: An Experimental Study. Master's thesis, Dept. of Computer and Info. Science, Norwegian University of Science and Technology, 2004. URL:http://www.idi.ntnu.no/∼elster/students/ms-theses/rosenvinge-msthesis.pdf.

[163] E. M. R. Rosenvinge, A. C. Elster, and C. Banino. Online Task Scheduling on Heterogeneous Clusters: An Experimental Study. In J. Dongarra, K. Madsen, and J. Wasniewski, editors, *PARA*, volume 3732 of *Lecture Notes in Computer Science*, pages 1141–1150. Springer, 2004.

[164] E. E. Santos, S. Feng, and J. M. Rickman. Efficient Parallel Algorithms for 2-Dimensional Ising Spin Models. In *IPDPS '02: Proceedings of the 16th*

International Parallel and Distributed Processing Symposium, page 135, Washington, DC, USA, 2002. IEEE Computer Society.

[165] E. E. Santos and G. Muthukrishnan. Efficient Simulation Based on Sweep Selection for 2-D and 3-D Ising Spin Models on Hierarchical Clusters. In *IPDPS*, page 229b, 2004.

[166] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, MA, USA, 1989.

[167] L. F. G. Sarmenta. *Volunteer Computing*. PhD thesis, Dept. of Electrical Engineering and Computer Science, MIT, 2001.

[168] H. Senger, F. A. B. Silva, and W. M. Nascimento. Hierarchical Scheduling of Independent Tasks with Shared Files. In *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, page 51, Washington, DC, USA, 2006. IEEE Computer Society.

[169] G. Shao, F. Berman, and R. Wolski. Master/Slave Computing on the Grid. In *HCW '00: Proceedings of the 9th Heterogeneous Computing Workshop*, page 3, Washington, DC, USA, 2000. IEEE Computer Society.

[170] B. A. Shirazi, K. M. Kavi, and A. R. Hurson, editors. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1995.

[171] D. F. Sittig, D. Foulser, N. Carriero, G. McCorkle, and P. L. Miller. A Parallel Computing Approach to Genetic Sequence Comparison: The Master Worker Paradigm with Interworker Communication. *Computers and Biomedical Research*, 24:152–169, 1991.

[172] S. S. Skiena. *The algorithm design manual*. Springer-Verlag New York, Inc., New York, NY, USA, 1998.

[173] Y. Song and Z. Li. New Tiling Techniques to Improve Cache Temporal Locality. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 215–228, New York, NY, USA, 1999. ACM Press.

[174] D. Stainforth, J. Kettleborough, M. Allen, M. Collins, A. Heaps, and J. Murphy. Distributed Computing for Public-Interest Climate Modeling Research. *Computing in Science and Engg.*, 4(3):82–89, 2002.

[175] E. Strohmaier, J. J. Dongarra, H. W. Meuer, and H. D. Simon. The Marketplace of High-Performance Computing. *Parallel Computing*, 25(13–14):1517–1544, Dec. 1999.

[176] T. G. Robertazzi. Ten Reasons to Use Divisible Load Theory. *Computer*, 36(05):63–68, 2003.

[177] O. Temam, E. D. Granston, and W. Jalby. To Copy or Not to Copy: A Compile-Time Technique for Assessing When Data Copying Should be Used to Eliminate Cache Conflicts. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 410–419, New York, NY, USA, 1993. ACM Press.

[178] D. Thain, T. Tannenbaum, and M. Livny. Condor and the Grid. In F. Berman, G. Fox, and T. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc, December 2002.

[179] M. D. Theys, M. Tan, N. B. Beck, H. J. Siegel, and M. Jurczyk. A Mathematical Model and Scheduling Heuristics for Satisfying Prioritized Data Requests in an Oversubscribed Communication Network. *IEEE Trans. Parallel Distrib. Syst.*, 11(9):969–988, 2000.

[180] T. H. Tzen and L. M. Ni. Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers. *IEEE Trans. Parallel Distrib. Syst.*, 4(1):87–98, 1993.

[181] S. S. Vadhiyar, G. E. Fagg, and J. J. Dongarra. Towards an Accurate Model for Collective Communications. *Int. J. High Perform. Comput. Appl.*, 18(1):159–167, 2004.

[182] S. Vajracharya and D. Grunwald. Loop Re-Ordering and Pre-Fetching at Run-Time. In *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–13, New York, NY, USA, 1997. ACM Press.

[183] C. Weiß, W. Karl, M. Kowarschik, and U. Rüde. Memory Characteristics of Iterative Methods. In *Proc. of the ACM/IEEE Supercomputing Conf. (SC99)*, Portland, Oregon, USA, 1999.

[184] R. C. Whaley and J. J. Dongarra. Automatically Tuned Linear Algebra Software. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[185] M. E. Wolf and M. S. Lam. A Data Locality Optimizing Algorithm. *SIGPLAN Not.*, 39(4):442–459, 2004.

[186] M. Wolfe. *High-Performance Compilers for Parallel Computing*. Addison-Wesley, 1995. ISBN 0-8053-2730-4.

[187] R. Wolski. Experiences with Predicting Resource Performance On-line in Computational Grid Settings. *SIGMETRICS Perform. Eval. Rev.*, 30(4):41–49, 2003.

[188] R. Wolski, N. T. Spring, and J. Hayes. The Network Weather Service: a Distributed Resource Performance Forecasting Service for Metacomputing. *Journal of Future Generation Computing Systems*, 15(5-6):757–768, 1999.

[189] D. Wonnacott. Achieving Scalable Locality with Time Skewing. *International Journal of Parallel Programming*, 30(3):1–221, 2002.

[190] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, 1995.

[191] Y. Yang and H. Casanova. RUMR: Robust Scheduling for Divisible Workloads. In *HPDC '03: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, page 114, Washington, DC, USA, 2003. IEEE Computer Society.

[192] Y. Yang and K. van der Raadt. Multiround Algorithms for Scheduling Divisible Loads. *IEEE Trans. Parallel Distrib. Syst.*, 16(11):1092–1102, 2005. Member-Henri Casanova.

[193] Q. Yi, V. Adve, and K. Kennedy. Transforming Loops to Recursion for Multi-Level Memory Hierarchies. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 169–181, New York, NY, USA, 2000. ACM Press.

[194] D. Yu and T. Robertazzi. Divisible load scheduling for Grid computing. In *IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2003)*, November, 2003.

[195] Y. Zhao and K. Kennedy. Scalarization Using Loop Alignment and Loop Skewing. *Journal of Supercomputing*, Volume 31(1):5–46, 2005.

[196] T. Zhu, Y. Wu, and G. Yang. Scheduling divisible loads in the dynamic heterogeneous grid environment. In *InfoScale '06: Proceedings of the 1st international conference on Scalable information systems*, page 8, New York, NY, USA, 2006. ACM Press.

Part II

Contributions

Paper 1

Location-Aware Master-Slave Tasking on the Grid
Cyril Banino-Rokkones.
To appear in journal *Future Generation Computer Systems*.
Special Issue *”Application of Distributed and Grid Computing”*.

Location-Aware Master-Slave Tasking on the Grid

Cyril Banino-Rokkones

Department of Computer and Information Science
Norwegian University of Science and Technology
NO-7491 Trondheim, Norway

Abstract

This paper outlines the importance of resource placement decisions for Grid Computing. Decisions about where processing and storage facilities should be located on the Grid have a tremendous impact on system performance. We begin with a presentation of *Facility Location* and *Supply Chain Design*, that address challenges remarkably similar to some of the challenges encountered within Grid computing. Then we illustrate the impact of location-aware decisions with the classic Master-Slave tasking problem. We claim that deploying multiple masters is necessary to achieve good performance on large-scale platforms. The problem becomes to find the most profitable locations for hosting the master processes in order to maximize the platform task throughput. We show that this problem is NP-hard, but still introduce and evaluate a location-aware heuristic that achieves good performance on a wide range of simulations.

1.1 Introduction

Grid computing is a recent trend where computing platforms spanning over large networks are deployed in order to harness geographically dispersed computing resources [12]. The aim is often to provide computing power to applications at unprecedented scale. In this context, the efficient and effective movement of data from storage sites to processing facilities has a tremendous impact on system performance. Grid computing involves not only managing data movement, that is decided which data sould be processed on which machines, but also making decisions about where to produce and store the data.

Because the performance of resources that make up the Grid (computers, networks, storage systems) fluctuates dynamically due to contention between applications, Grid schedulers must choose the combination of resources from the available resource pool that is expected to maximize the performance of the applications [25]. Such scheduling decisions must often be changed in response to fluctuations in resource availability and performance. Resource placement decisions, on the other hand, are often fixed and difficult to change in the short term. The location of an expensive supercomputer cannot be changed as a result of changes in network performance or electricity prices. Inefficient locations for key components of the Grid infrastructure will result in poor system performance, no matter how well schedulers react in response to changing conditions.

This paper illustrates these issues with the classic Master-Slave tasking problem, which consists in the execution of a large number of independent tasks by a set of processors, called *slaves*, under the supervision of a particular processor, the *master*. The master holds all the tasks initially, and sends them out to the slaves over a network. Slaves are charged to compute the tasks and to send the computation results back to the master. This scheduling problem is well recognized, and several studies have recently revisited the Master-Slave paradigm for clusters and Grids [1, 5, 6, 13, 14, 17, 21, 22]. Applications implemented under this paradigm [3,8,23] are good candidates for Grid environments since the application tasks can be computed independently of each other and in any order. However, the centralization of the tasks in one single place limits the scalability of the application, as there is only one data access point in the system. This paper suggests a simple but mandatory departure from traditional implementations: The need to deploy several masters to efficiently utilize currently emerging large-scale platforms. Aida et al. [2] present a way to deploy multi-master master-slave applications in Grid environments. The idea is to rely on a hierarchical implementation where a supervisor controls multiple processor sets, each of which is composed of a master and several workers. The supervisor achieves load balancing by migrating tasks between masters. However, the work of Aida et al. does not take in consideration the topology of the platform, an essential aspect for large-scale Grids that we address in this paper. In particular, we address the problem of determining how many masters should be deployed, and where should these masters be located on the Grid in order to optimize system performance. As a consequence, a discrete network location problem arises. Thus, this paper provides an important step for efficient deployments of large Master-Slave applications on computational Grids.

The rest of this paper is organized as follows. Section 1.2 outlines the similarities between Grid computing and two classes of problem belonging to operation research, that are *Facility Location* and *Supply Chain Design*. Modeling Master-Slave tasking on a Grid is introduced in Section 1.3. The discrete network location problem considered in this paper is formally stated and shown to be NP-hard in Section 1.4. Greedy heuristics are presented in Section 1.5, and experimental results in Section 1.6. Finally, conclusions and future work are discussed in Section 1.7.

1.2 Grid computing, facility location and supply chain design

Grid systems provide storage and computing facilities at the disposal of user applications. When deploying applications on Grids, it is capital to have data stored *close* to processing facilities, in order to obtain efficient and effective data movements. Theys et al. [24] notice that the *data staging* problem in a distributed and heterogeneous network of resources, presents a high level of similarity with Facility Location problems. A classic Facility Location problem is a spatial resource allocation problem in which one or more service facilities have to be located to serve a geographically dispersed set of population demands according to some objective function [10, 16]. The term facility is used in its broadest sense, as it is meant to include entities such as factories, warehouses, schools, hospitals, subway stations, satellites, etc [10]. If we assimilate (1) facilities to storage locations, (2) the service to be provided to application data, and (3) the set of demands to the set of computing resources, then deploying Grid applications can be expressed as a Facility Location problem. Of particular interest because close in spirit to Grid computing settings, are the *Maximal Covering Location Problem* [9] that addresses planning situations which have an upper limit P on the number of facilities to be deployed, and the *Fixed Charge Location Problem* [10] that introduces capacities as well as economic cost constraints on the facilities to be deployed and operated.

Another class of problems that presents features very similar to Grid application scheduling is Supply Chain Design [11]. A supply chain can be defined as a network of facilities that manufactures finished products, and distributes these products to customers. Supply Chain Design involves deciding (1) where to produce, what to produce, and how much to produce at each site, and (2) where to locate plants and distribution centers [11]. Today, workflow applications compose one of the most popular class of Grid applications. A workflow application consists of a collection of interacting components that must be executed in a certain order for successful completion of the application. Hence, the application can be represented as a directed acyclic graph (DAG), where each node in the DAG represents an application component, and the edges denote control or data dependencies. If we now make an analogy between (1) the last task of the DAG and the manufactured product and (2) the computing resources and the production plants, then scheduling a workflow application on a Grid can be expressed as a Supply Chain Design problem.

Consequently, lessons can be drawn from the design of algorithms for different versions of Facility Location and Supply Chain Design problems. To the best of our knowledge, few studies have considered Facility Location theory within Grid computing settings. Maheswaran et al. [18, 19] present *MetaGrig*, an architecture for resource provisioning for WAN-enabled applications. MetaGrid introduces the notion of SubGrid, which spans a Grid and provides a “resource allocation class”. A SubGrid specifies a set of constraints that should be satisfied by the resources allocated for the SubGrid. Maheswaran et al. approach the SubGrid creation problem as a Fixed Charge Location Problem. Ko and Rubenstein [15] present a distributed protocol to place replicated resources in large-scale networks such that each vertex is “close” to some copy of any object. Closely related to our work is the paper of Shao et al. [22] that consider a resource selection problem within the Master-Slave scheduling framework. The aim is to select performance-efficient hosts for both the master and slave processes. For that end, an exhaustive search is performed, consisting in solving n network flow problems, where n is the number of processors composing the platform. Then the configuration that achieved the highest throughput is selected. However, this approach is not applicable when using several masters. There are indeed $\binom{n}{p}$ possible master locations sets, where p is the number of masters to be located on the platform. As an example, for $n = 50$ and $p = 10$, the resulting number of possibilities is $10, 272, 278, 170$. Clearly, even for moderate values of n and p, such enumeration is not realistic, and we need more advanced techniques.

1.3 Modeling master-slave tasking on the grid

Our model builds on a network flow model and is close to models presented by Hong and Prasanna [13] and by Shao et al. [22]. First, we assume that application tasks require input data files of size s_i, and produce output data files of size s_o. Then, the target architectural framework is represented by a platform graph $G = (V, E)$ as illustrated in Fig. 1.1. Each vertex $P_i \in V$ represents a computing resource of weight r_i, meaning that processor P_i can process r_i tasks per time units. Similarly, each link $(P_i, P_j) \in E$ has a capacity $F_{i,j}$ which limits the flow of data that can be transfered from P_i to P_j per time unit. We allow for bandwidth asymmetry between incoming and outgoing traffic such that $F_{i,j}$ may differ from $F_{j,i}$. Bandwidth asymmetry can appear in WAN environments when Internet routing for traffic between two machines uses different pathways in each direction.

All r_i are assumed to be positive rational numbers since they represent the processor computing rates, and we allow $r_i = 0$; then P_i has no computing power, but can still forward tasks to other processors (e.g. for modeling a hub or a switch). Similarly, we assume that all $F_{i,j}$ are positive rational numbers since they correspond to the peak bandwidths of the interconnection links. These platform parameters, can be obtained by Grid middleware services such as the Network Weather

Service (NWS) [25]. In addition, NWS provides accurate predictions of future performance measurements, as well as error quantifications associated to these predictions.

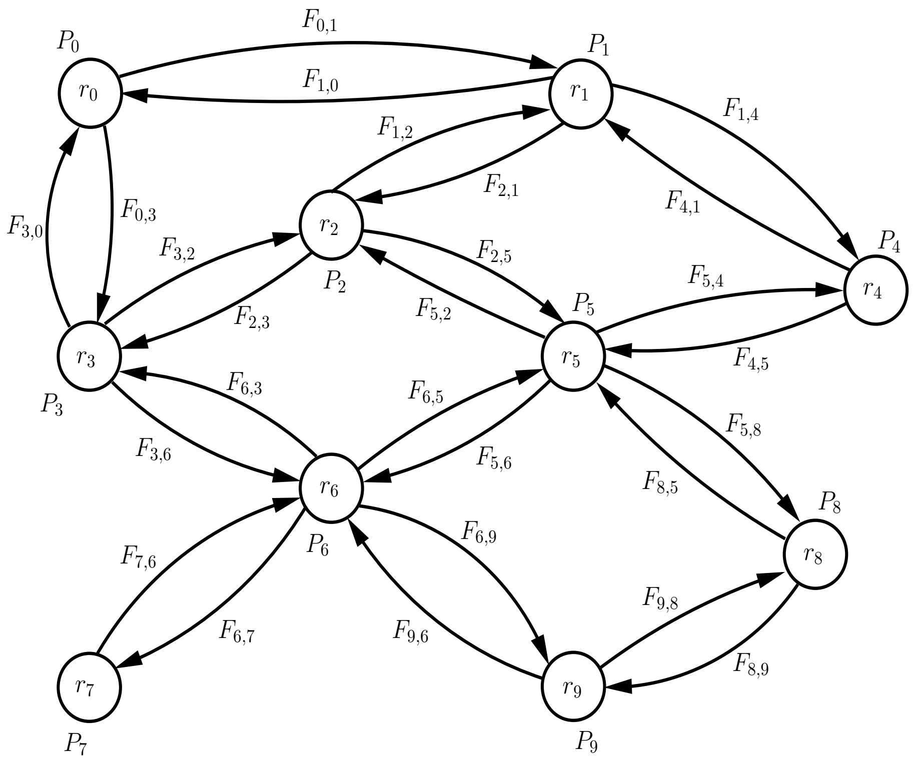


Figure 1.1: Example of a Grid platform.

Finally, we introduce a cost model to establish and operate masters on the platform graph. Let $J_m \subseteq V$ denote the index set of the processors susceptible to be chosen as a master. Then, for each processor $P_i \in J_m$, let c_i be the fixed cost of establishing a master at location P_i, and t_i be the per task cost for operating a master at location P_i. All c_i and t_i are assumed to be positive constants since, from a practical viewpoint, it is rather absurd to have negative costs for establishing or operating master locations. Given a general Grid platform, we consider the problem of selecting a set of master locations that optimizes the task throughput of the platform within a budget constraint B. We term such problem the *B-COVER* problem.

1.4 Optimizing master placement

1.4.1 Mathematical formulation of the B-COVER problem

To formally define the *B-COVER* problem, let $n(i)$ denote the index set of the neighbors of processor P_i. During one time unit, let α_i be the number of tasks computed by P_i, $f_{i,j}$ and $f'_{i,j}$ be the amount of input and output data respectively that flow from P_i to P_j. For each processor P_i, let $x_i \in \{0,1\}$ be the decision variable to place a master at location P_i, i.e. $x_i = 1$ if P_i is chosen as a master, and $x_i = 0$ otherwise. Further, let p_i be the rational number of input files produced per time unit by P_i if the latter is chosen as a master, and analogously, let p'_i be the number of output files collected per time unit by P_i, if the latter is chosen as a master. Defined on a platform graph G, a mathematical formulation of the *B-COVER* problem can be stated by the following mixed-integer linear program, whose objective function is to maximize the throughput $n_{task}(G)$ of the platform graph G.

Maximize

$$n_{task}(G) = \sum_i \alpha_i,$$

Subject to

$$\left\{\begin{array}{ll|ll}
 & & (5) & \forall i,\ 0 \le p_i \le x_i(r_i + \sum_{j\in n(i)} F_{i,j}) \\
(1) & \forall i,\ 0 \le \alpha_i \le r_i & (6) & \forall i,\ 0 \le p'_i \le x_i(\sum_{j\in n(i)} F_{j,i}) \\
(2) & \forall i \in J_m,\ x_i \in \{0,1\} & (7) & \forall i, \forall j \in n(i),\ f_{i,j}s_i + f'_{i,j}s_o \le F_{i,j} \\
(3) & \forall i \notin J_m,\ x_i = 0 & (8) & \forall i,\ p_i + \sum_{j\in n(i)} f_{j,i} = \alpha_i + \sum_{j\in n(i)} f_{i,j} \\
(4) & \sum_{i\in J_m} c_i x_i + t_i p_i \le B & (9) & \forall i,\ p'_i + \sum_{j\in n(i)} f'_{i,j} = \alpha_i + \sum_{j\in n(i)} f'_{j,i}
\end{array}\right.$$

• Equation set (1) specifies that computing resources are limited.

• Equation sets (2) and (3) identify candidate locations for establishing the masters.

• Equation (4) ensures that the cost generated by establishing and operating the master locations does not exceed the budget constraint B.

• Equation set (5) specifies that only the masters are allowed to produce computational tasks. The task production rate is limited by the number of tasks that P_i can process plus the number of input files that P_i can communicate to its neighbors per time unit.

• Equation set (6) specifies that only the masters are allowed to collect output files. The result collection rate is limited by the maximum number of output files that P_i can receive from its neighbors per time unit. The case where there is an overhead incurred for post-processing output files can be handled easily by adding a constraint on p'_i [22]. Note that our model does not guarantee that the masters will receive the output files corresponding to the input files that they produced locally. In some circumstances, it might be desirable to produce input files in one place and collect output files in some other place, in order to circumvent eventual network bottlenecks. However, it is often necessary that masters receive the output files associated to the input files that were generated locally (e.g. for fault tolerance issues). In that case, the model can be enhanced as follows: Each master will be given a color, and will be able to produce tasks and collect results only of its own color, while slaves can process tasks of any color.

• Equation set (7) specifies that communication resources are limited.

• And finally, Equation sets (8) and (9) stand for conservation laws. For every processor P_i, the number of input files produced, plus the number of outgoing input files, should be equal to the number of tasks processed locally, plus the number of incoming input files. Similarly, for every processor P_i, the number of output files collected, plus the number of outgoing output files, should be equal to the number of results produced locally, plus the number of incoming output files. Note that Equation sets (3), (5) and (6) prevent the slaves to produce input files or to collect output files. On the other hand, Equation sets (8) and (9) ensure that there are as many input files produced as output files consumed. In effect, by combining these two Equation sets, we can derive the following Equation: $\sum_i p_i = \sum_i p'_i$.

1.4.2 Complexity of *B-COVER*

Theorem 1.1. B-COVER *is NP-hard.*

Proof. We reduce the *MAXIMUM KNAPSACK* (MK) problem [4] to the *B-COVER* problem.

MAXIMUM KNAPSACK

INSTANCE:	Finite set U, for each $u \in U$ a size $s(u) \in \mathbf{Z}^+$ and a value $v(u) \in \mathbf{Z}^+$, a positive integer $B \in \mathbf{Z}^+$.
SOLUTION:	A subset $U' \subseteq U$ such that $\sum_{u \in U'} s(u) \leq B$.
MEASURE:	Total weight of the chosen elements, i.e. $\sum_{u \in U'} v(u)$.

Construct an instance of the *B-COVER* problem as follows: (1) Create a set V containing $|U|$ processors; (2) Create a bijective function $f : V \longmapsto U$; (3) $\forall P_i \in V$, let $r_i = v(f(P_i))$; (4) $\forall P_i \in V$, let $c_i = s(f(P_i))$; (5) $\forall P_i \in V$, let $t_i = 0$; and (6) let $E = \emptyset$ and $J_m = V$.

The graph of the *B-COVER* instance is edge-less ($E = \emptyset$), meaning that no tasks can be communicated among processors. Consequently, tasks can only be computed at the location where they are produced. A solution of the *B-COVER* instance consists in determining a subset $V' \subseteq V$ such that $\sum_{P_i \in V'} c_i \leq B$ in order to maximize the platform throughput, i.e. $\sum_{P_i \in V'} r_i$. Thus, a solution of the *B-COVER* problem instance provides a solution of the *MK* instance. This proves that *B-COVER* is at least as difficult as *MK*. Since

MK is NP-hard [4] and since the transformation is done in polynomial time, *B-COVER* is also NP-hard. □

1.5 Greedy heuristics

Typically, the first approach for finding the optimal solution of a mixed-integer linear program is to apply one of the well-known algorithms such as branch and bound or cutting plane [10]. Unfortunately, such methods are only useful on small-scale problems, and will quite often consume unacceptable computational resources when applied to realistic problem sizes.

In contrast, LP-relaxations (i.e. relaxing the integer constraints) have considerable interest since they provide the basis both for various heuristics and for the determination of bounds for the most successful integer linear programs [16]. If we replace Equation (2) by the following equation: $\forall i \in J_m,\ 0 \leq x_i \leq 1$, we obtain a linear program in rational numbers, that can be solved in polynomial time. Solving the relaxed linear program associated to *B-COVER*, gives the upper bound of the optimal platform throughput reachable without exceeding the budget constraint. However, this bound might not be achievable by any discrete solutions, and might not be tight.

In the rest of this paper, we let the fixed cost c_i to establish master locations be equal to 1, and the per task cost t_i for operating a master be equal to 0. In other words, there are no differences in term of cost among the different potential master locations, and the influent factors for choosing master locations become the platform topology and heterogeneity. We hence retrieve the model proposed by Shao et al. in [22]. As a consequence, the objective of the *B-COVER* problem becomes to maximize the platform throughput when using at most B masters. The problem becomes then similar to the Maximal Covering Location Problem [9] also known to be NP-hard.

Our first heuristic (**LP**) for solving the *B-COVER* problem consists in solving the relaxed linear program in the first place, and then select in a greedy fashion a set S of vertices P_i ($|S| = B$) that have the highest p_i values. Each master $P_i \in S$ will produce $\frac{p_i}{\sum_{P_i \in S} p_i}$ fraction of the total amount of tasks.

The second heuristic (**BW**) consists in selecting in a greedy fashion, B locations that maximize the quantity $q_i = r_i + \sum_{j \in n(i)} |\frac{F_{i,j}}{s_i} - \frac{F_{j,i}}{s_o}|$ and that do not have a neighboring master. Each master $P_i \in S$ will produce $\frac{q_i}{\sum_{P_i \in S} q_i}$ fraction of the total amount of tasks. This heuristic could be implemented in a distributed fashion, and seeks to maximize the bandwidth deliverable to the application.

Finally, we implemented a naive heuristic (**RD**), that selects in a greedy fashion B random master locations, still in preventing neighboring masters situations. Each master will produce $\frac{1}{B}$ fraction of the total amount of tasks. The aim is to demonstrate the impact of location-unaware decisions.

1.6 Simulations

1.6.1 Methodology

We evaluate and compare our heuristics through extensive simulations using the SimGrid toolkit [7]. We rely on simulations rather than direct experiments in order to make a fair comparison between the proposed heuristics. Indeed, simulation enables running of the different tests on computing platforms having exactly the same dynamic behavior. In the simulations, we used tree-shaped platforms as opposed to graph-shaped platforms, in order to remove data routing issues. Although this assumption considerably reduces the difficulty of the problem (Meggido et al. [20] proposed a polynomial time algorithm for solving the Maximal Covering Location Problem on tree-shaped networks), it nevertheless allows to simplifies the framework for exposing the main claim of this study, i.e. that location-aware decisions have a huge impact on application and system performance. Moreover, many organizations rely on tree-shaped networks to interconnect their computing resources [14].

Since a sub-tree can be reduced to a single super-node of equivalent processing power [5], it is not necessary to employ thousands of nodes to simulate large-scale systems [13]. In our simulations, the number of nodes in a tree was limited to 100, and each node could have at most 10 neighbors. A random tree is generated as follows. Each node is numbered with an ID number i between 0 and 99. Then, each node $P_i, i \in [1, 99]$ is connected randomly to a node $P_j, j \in [0, i-1]$. The links have peak performance values comprised between F_{min} and F_{max} and the nodes between r_{min} and r_{max}. All random distributions are uniform. The dynamic environments used in our simulations were generated as follows. Each resource R_i (node or link) has a cyclic behavior, i.e. its performance changes n_i times per cycle. The number of changes n_i per cycle is randomly taken within the interval $[5, 15]$. Resource performance fluctuations are relatively distant in time (every 50 treated tasks in average) in order to destabilize the system by creating a succession of contexts. We do not claim that these decisions correspond to realistic network conditions. We simply aim to compare the heuristics on different platform configurations.

In this paper, we report the simulation of an independent-task application composed of 5000 tasks on 100 trees where $F_{min} = 0.01$, $F_{max} = 0.02$, $r_{min} = 0.001$ and $r_{max} = 0.002$. In other words, a master can serve 10 slaves in average. For the sake of generality, we let $J_m = V$, i.e. every node can be chosen as a master. The aim of these arbitrary decisions is to keep the number of parameters as low as possible, while maintaining the problem complexity. Nevertheless, we expect our heuristics to perform better in presence of more constraints, e.g. with heterogeneous cost distributions, since the problem would become more specific.

Finally, inspired by Kreaseck et al. [17], we determine the throughput rate of the system by using a growing window. The total execution time is divided into 100 equal-sized time slots. Then, the window increases in size by step of 1 time slot, and the throughput rate delivered within the window time-frame is computed.

1.6.2 Scheduling issues

Once the set of master locations has been identified, the slave nodes are free to decide which master they are willing to serve. Communication patterns are simple and well-defined, requiring communication only between a master and a slave. When a slave requests a task to a master, it measures locally the time it takes the master to deliver the task. Based on this information, slaves will decide which master to address the next request. The slaves seek to identify at all times the master that is the “closest” to them. On the other hand, master nodes are forced to serve any incoming request, no matter how heavily loaded they are.

The slaves are allowed to buffer several tasks locally in order to avoid starvation. However, the number of local tasks is regulated by a threshold θ_i. If there are less than θ_i tasks buffered locally, then additional tasks will be requested. Initially, $\theta_i = 1$. During the execution, nodes are allowed to increment their local thresholds θ_i only when (1) they are starving and (2) if they recently succeeded to accumulate θ_i tasks locally (to ensure that the current threshold is not sufficient). This mechanism allows the slaves to collect enough tasks locally in order to avoid starvation.

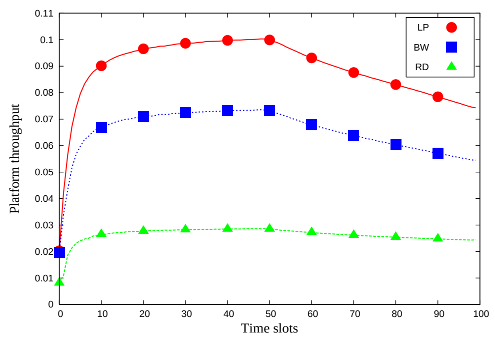
1.6.3 Results

The results of our simulations are depicted in Figure 1.2, which plots an average of the 100 throughput rates (associated to the 100 trees) over time, achieved by the three heuristics presented in Section 1.5. The x-axis reports the total execution time divided into 100 time slots, while the y-axis reports the platform throughput achieved within the time-frame window. Hence, the higher a curve gets, the better the corresponding heuristic performs.

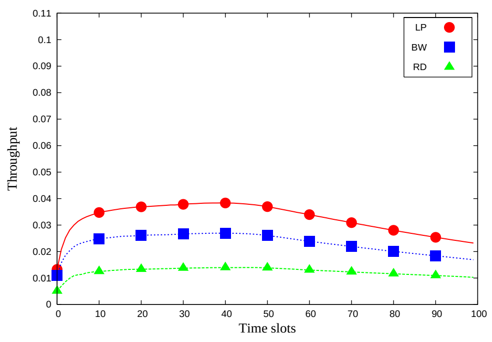
Figures 1.2 (a), (c), (e) and (g) correspond to static environments, i.e. with no system perturbation, whereas Figures 1.2 (b), (d), (f) and (h) correspond to highly dynamic environments, i.e. where resource performances can degrade down to 1% of the peak value.

The first observation to make is that the LP heuristic (which makes location decisions based on the LP-relaxation computed from an initial snapshot of the platform) outperforms the two other heuristics, even in dynamic environments. Then, introducing a minimal amount of knowledge in the location decision process, allows to obtain substantial performance improvements, as attests the superiority of the BW heuristic over the RD heuristic.

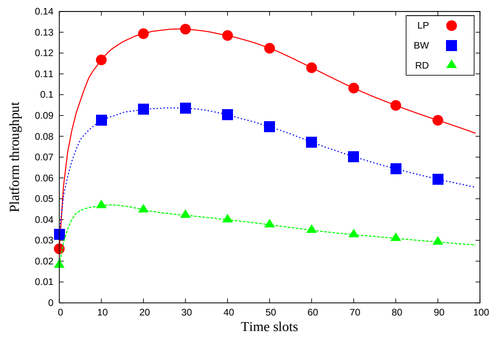
In most of the simulation sets, one can observe a steep initialization phase, followed by a short steady-state phase, in turn followed by a long wind-down phase. This “wave-shape” accentuates as one increases the number of masters. This phenomenon is due to the fact that more tasks can be handed out in the beginning of the execution, as one increases the number of data access points. Then, as the computation goes on, some masters will have handed out all their tasks before others, resulting in performance decrease. This point highlights the importance of task load-balancing between the masters.



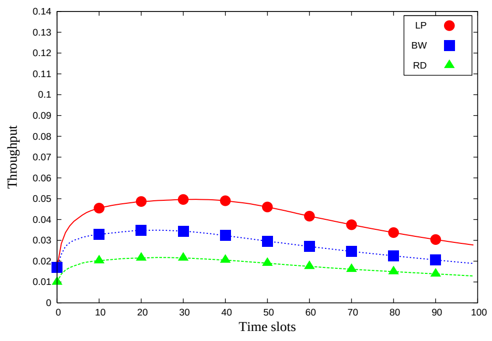
(a) static, B=1



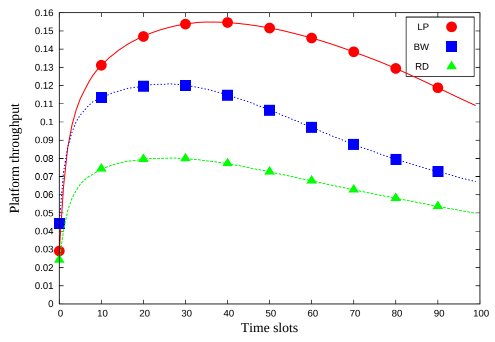
(b) dynamic, B=1



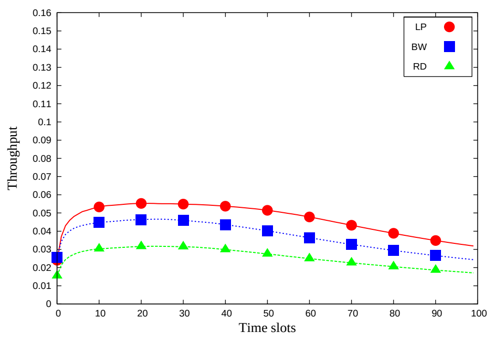
(c) static, B=2



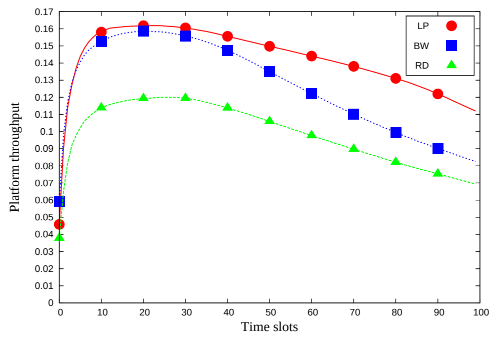
(d) dynamic, B=2



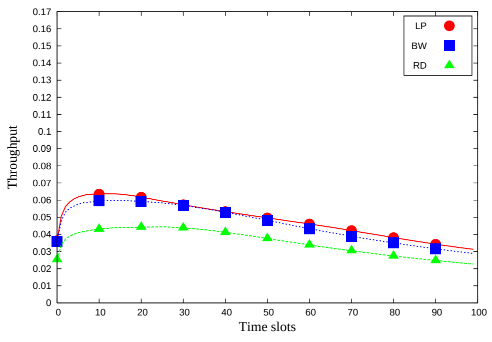
(e) static, B=4



(f) dynamic, B=4



(g) static, B=8



(h) dynamic, B=8

Figure 1.2: Average of the 100 throughput rates over time. In the dynamic environments, resource performances can degrade arbitrarily.

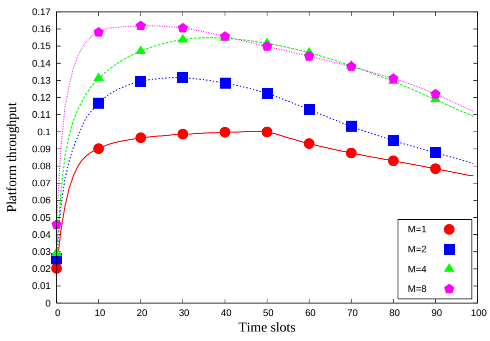
As expected, using a high number of masters closes the gap between the different heuristics. In effect, the more data access points there are, the less margin there is to make inefficient placement decisions. Nonetheless, the hierarchy established among the heuristics in static environments is respected within dynamic environments.

Finally, an important choice for the decision maker is deciding what level of expenditure (i.e. how many masters should be deployed) can be justified by the resultant throughput. Figure 1.3 re-plots the average throughput achieved by the LP heuristic, when increasing the number of masters deployed. One clearly sees that, deploying 4 masters would be appropriate, because there are not enough tasks to reach and sustain a worthy throughput with additional masters. Therefore, the deployment of a high number of masters should come along with a larger number of tasks to process.

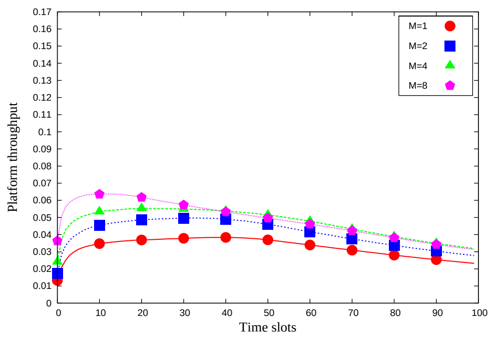
1.7 Conclusion and future work

This paper outlines the importance of resource placement decisions for Grid Computing. Decisions about where storage and computing facilities should be located on the Grid have a tremendous impact on application performance. The impact of location-aware decisions is illustrated with the classic Master-Slave tasking problem, which consists in allocating a large number of independent, equal-sized tasks to a Grid composed of a heterogeneous collection of computing and communication resources. This paper suggests a simple but mandatory departure from traditional implementations: The need to deploy several masters to efficiently utilize large-scale platforms. We provide a model for establishing and operating master locations, and show that the problem of finding the most profitable locations for hosting the masters is NP-hard. Still, we propose a location-aware heuristic that achieves very good performance on a wide range of simulations. This work can be extended in the following directions.

First, enabling the cooperation of several masters transparently is a challenging task, but mandatory in order to efficiently deploy multi-master Master-Slave applications. In



(a) static



(b) dynamic

Figure 1.3: Average throughputs achieved by the LP heuristic as one increases the number of masters.

particular, load-balancing techniques among masters should be designed, in order to cope with, and respond to fluctuations in resource performance and availability. Then, we showed in this paper that some Grid computing problems present tight connections with well-known problems from operation research. Facility Location theory and Supply Chain Design have been the subject of a wealth of research, and we believe that models and solutions to these problems can be adapted for Grid computing. Of particular interest are Facility Location models under uncertainty (i.e. under dynamic conditions), and Facility Location models with facility failures [11]. Effectively, failures and variations in resource availability are expected to be the rule rather than the exception within large-scale environments, especially when the overall processing time of applications keeps getting larger and larger.

Bibliography

[1] M. Adler, Y. Gong, and A. L. Rosenberg. Optimal Sharing of Bags of Tasks in Heterogeneous Clusters. In *15th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA'03)*, pages 1–10. ACM Press, 2003.

[2] K. Aida, W. Natsume, and Y. Futakata. Distributed Computing with Hierarchical Master-worker Paradigm for Parallel Branch and Bound Algorithm. In *3rd International Symposium on Cluster Computing and the Grid*, page 156, 2003.

[3] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti@home: An Experiment in Public-Resource Computing. *Communications of the ACM*, 45(11):56–61, 2002.

[4] G. Ausiello, M. Protasi, A. Marchetti-Spaccamela, G. Gambosi, P. Crescenzi, and V. Kann. *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer-Verlag New York, Inc., 1999.

[5] C. Banino, O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Scheduling Strategies for Master-Slave Tasking on Heterogeneous Processor Platforms. *IEEE Transactions on Parallel and Distributed Systems*, 15(4):319–330, 2004.

[6] O. Beaumont, A. Legrand, and Y. Robert. The Master-Slave Paradigm with Heterogeneous Processors. *IEEE Transactions on Parallel and Distributed Systems*, 14:897–908, 2003.

[7] H. Casanova. SimGrid: A Toolkit for the Simulation of Application Scheduling. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, page 430. IEEE Computer Society, 2001.

[8] A. Chien, B. Calder, S. Elbert, and K. Bhatia. Entropia: Architecture and Performance of an Enterprise Desktop Grid System. *Journal of Parallel and Distributed Computing*, 63:597, May 2003.

[9] R. L. Church and C. S. ReVelle. The Maximal Covering Location Problem. *Papers of the Regional Science Association*, 32:101–118, 1974.

[10] J. Current, M. Daskin, and D. Schilling. Discrete Network Location Models. In Z. Drezner and H. Hamacher, editors, *Facility Location Theory: Applications and Methods*, chapter 3, pages 81–118. Springer-Verlag, Berlin, 2002.

[11] M. Daskin, L. V. Snyder, and R. T. Berter. Facility Location in Supply Chain Design. In A. Langevin and D. Riopel, editors, *Logistics Systems: Design and Optimization*, chapter 2, pages 39–66. Kluwer, 2005.

[12] I. Foster and C. Kesselman. *The Grid, Blueprint for a New Computing Infrastructure*. Morgan Kaufman, San Francisco, 1999.

[13] B. Hong and V. K. Prasanna. Distributed Adaptive Task Allocation in Heterogeneous Computing Environments to Maximize Throughput. In *International Parallel and Distributed Processing Symposium IPDPS'2004*, page 52b. IEEE Computer Society Press, 2004.

[14] B. Hong and V. K. Prasanna. Performance Optimization of a De-centralized Task Allocation Protocol via Bandwidth and Buffer Management. In *CLADE*, page 108, 2004.

[15] B. Ko and D. Rubenstein. Distributed Self-Stabilizing Placement of Replicated Resources in Emerging Networks. *IEEE/ACM Trans. Netw.*, 13(3):476–487, 2005.

[16] J. Krarup and P. Pruzan. The Simple Plant Location Problem: Survey and Synthesis. *European Journal of Operations Research*, 12:36–81, 1983.

[17] B. Kreaseck, L. Carter, H. Casanova, and J. Ferrante. Autonomous Protocols for Bandwidth-Centric Scheduling of Independent-Task Applications. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 26.1, Washington, DC, USA, 2003. IEEE Computer Society.

[18] M. Maheswaran, B. Maniymaran, P. Card, and F. Azzedin. Invisible Network: Concepts and Architecture. In *CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, page 464, Washington, DC, USA, 2002. IEEE Computer Society.

[19] M. Maheswaran, B. Maniymaran, P. Card, and F. Azzedin. MetaGrid: A Scalable Framework for Wide-Area Service Deployment and Management. In *HPCS '02: Proceedings of the 16th Annual International Symposium on High Performance Computing Systems and Applications*, page 61, Washington, DC, USA, 2002. IEEE Computer Society.

[20] N. Megiddo, E. Zemel, and S. L. Hakimi. The Maximum Coverage Location Problem. *SIAM Journal on Algebraic and Discrete Methods*, 4(2):253–261, June 1983.

[21] A. Morajko, E. César, P. Caymes-Scutari, T. Margalef, J. Sorribes, and E. Luque. Automatic Tuning of Master/Worker Applications. In J. C. Cunha and P. D. Medeiros, editors, *Euro-Par*, volume 3648 of *Lecture Notes in Computer Science*, pages 95–103. Springer, 2005.

[22] G. Shao, F. Berman, and R. Wolski. Master/Slave Computing on the Grid. In *HCW '00: Proceedings of the 9th Heterogeneous Computing Workshop*, page 3, Washington, DC, USA, 2000. IEEE Computer Society.

[23] D. F. Sittig, D. Foulser, N. Carriero, G. McCorkle, and P. L. Miller. A Parallel Computing Approach to Genetic Sequence Comparison: The Master Worker Paradigm with Interworker Communication. *Computers and Biomedical Research*, 24:152–169, 1991.

[24] M. D. Theys, M. Tan, N. B. Beck, H. J. Siegel, and M. Jurczyk. A Mathematical Model and Scheduling Heuristics for Satisfying Prioritized Data Requests in an Oversubscribed Communication Network. *IEEE Trans. Parallel Distrib. Syst.*, 11(9):969–988, 2000.

[25] R. Wolski. Experiences with Predicting Resource Performance On-line in Computational Grid Settings. *SIGMETRICS Perform. Eval. Rev.*, 30(4):41–49, 2003.

Paper 2

A Distributed Procedure for Bandwidth-Centric Scheduling of Independent-Task Applications
Cyril Banino.
In *Proceedings of 19th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2005*,
April 04-08 2005, Page(s):48a - 48a, Denver, USA.

A Distributed Procedure for Bandwidth-Centric Scheduling of Independent-Task Applications

Cyril Banino

Department of Computer and Information Science
Norwegian University of Science and Technology
NO-7491 Trondheim, Norway

Abstract

The problem of scheduling independent tasks on heterogeneous trees is considered. The nodes of the tree may have different processing times, and links different communication times. The single-port, full overlap model is used for modeling the activities of the nodes. A distributed method for determining the maximum steady-state throughput of a tree is presented. Then, we show how each node can build up its own local schedule independently of the rest of the platform. In addition, the final schedule is asynchronous and event-driven, meaning that each node (except the root) acts without any time-related information. A local scheduling strategy which aims at minimizing the amount of tasks buffered at node locations during steady-state is introduced. As a consequence, the lengths of the start-up and wind-down phases are considerably reduced.

2.1 Introduction

A recent trend in high performance computing is to deploy computing platforms that span over large networks in order to harness geographically distributed computing resources. The aim is often to provide computing power to applications at unprecedented scale. Good candidates for such environments are *Master-Worker* applications, which are composed of a large number of computational tasks independent from each other, i.e. where no inter-task communications take place, and where the tasks can be computed in any order. Many applications have been and can be implemented under the Master-Worker paradigm. They include: The processing of large measurement data sets like the SETI@home project [1], biological sequence comparisons [14], or also distributed problems organized by companies like Entropia [9]. See [12] for more examples.

This paper is a follow on of recent work by Beaumont et al. [5] as well as Kreaseck et al. [12], who also considered the problem of scheduling Master-Worker applications onto heterogeneous tree-shaped computing platforms. The resources composing the platform may have different computation and communication speeds, as well as different overlap capabilities. The platform topology is modeled by a tree, where each node represents some computing resource capable of computing and/or communicating with its neighbors via message passing over interconnection links. The main advantage of using trees, as opposed to the more general graphs is that no choices need to be made about how to route the data [4]. The estimation of the different bandwidths of the platform links can be obtained using tools such as the Network Weather Service [15], or by local measurements. The application tasks are modeled as requiring some input data file, and producing some output data file. A special processor, called the *master*, generates the input files associated to the application tasks, decides which tasks to execute, and how many tasks to delegate to each of its children. In turn, each child decides which tasks to execute, and how many tasks to forward to its own children. In this paper, applications such as SETI@home, i.e. where the output files produced by the tasks are much smaller than the input files, are considered. Consequently, the return of the output files to the master node is negligible. Throughout the rest of the paper, we assimilate input files to tasks, and consider that tasks can be communicated and computed.

In this paper, we present a lightweight distributed communication procedure which allows each node to build up its local schedule autonomously in order to attain the maximum steady-state throughput of the tree, i.e. that maximizes the number of tasks computed per time unit. Our procedure is an efficient, practical and scalable implementation of the theoretical results presented in [5]. The rest of the paper is organized as follows: Related work is reviewed in Section 2.2. In Section 2.3 we formally state our model of computation and communication. Section 2.4 reassesses the bandwidth-centric principle that lays the foundations of our work. A distributed method for determining the optimal steady-state throughput of a tree is given in Section 2.5. We show in Section 2.6 how each node can build up its local schedule autonomously. An efficient start-up strategy is given in Section 2.7. Our main results are illustrated with an example in Section 2.8. Future work is discussed in Section 2.9, and our contributions are summarized in Section 2.10.

2.2 Related work

“The traditional objective of scheduling algorithms is makespan minimization: Given the application tasks and a set of resources, find a mapping of the tasks onto the set of processors and order the execution of the tasks so that (i) resource constraints are satisfied, and (ii) a minimum schedule length is provided” [6]. Recent studies have been conducted on makespan minimization under heterogeneous conditions. Beaumont et al. [7] revisited the Master-Worker paradigm with heterogeneous processors interconnected via a bus. Dutot extended this result to daisy-chains as well as “spider graphs” [10], and showed that the problem was NP-hard for heterogeneous trees [11]. We believe that the scheduling strategy presented in this paper is a good heuristic candidate to solve the problem studied by Dutot, since we are able to obtain the optimal platform throughput using quick start-up and wind-down phases.

"An idea to circumvent the difficulty of makespan minimization is to lower the ambition of the scheduling objective" [6]. The problem becomes to maximize the steady-state throughput of the platform, i.e. the number of tasks computed per time unit.

Shao et al. [13] considered general interconnection platform graphs, and solve the Master-Worker tasking problem in steady-state using a network-flow approach. The authors model the platform nodes using the *multiple-port, full overlap* model [4], where the number of simultaneous communications for a given node is not bounded. Banino et al. [2] showed how to solve this problem for general interconnection graphs whose nodes operate under the *single-port, full-overlap* model using a linear programming approach.

Heterogeneous trees were considered by Kreaseck et al. [12] who presented two *autonomous* bandwidth-centric scheduling protocols that address the practical problem of attaining the maximum steady-state rate after some start-up and maintaining that rate until wind-down. Two communication models are studied, the non-interruptible communication (which corresponds to our model), and the interruptible communication where a request from a higher priority child may interrupt a communication to a lower priority child. Although the work of Kreaseck et al. is a first step towards a practical implementation of bandwidth-centric scheduling algorithms, under the non-interruptible communication model, their autonomous protocol might take non-optimal decisions, generating hence long start-up phases as well as unnecessary large numbers of tasks buffered at node locations.

2.3 Our steady-state model

Our model builds on the model proposed by Beaumont et al. in [5] that we augment by introducing the task computing and task communicating rates of the processors.

The target architectural/application framework is represented by a node-weighted edge-weighted tree $T = (V, E, w, c)$ as depicted in Figure 2.1. Each node $P_i \in V$ represents a computing resource of weight w_i, meaning that node P_i requires w_i units of time to process one task. Each edge $e_{i,j} : P_i \rightarrow P_j$ corresponds to a communicating resource and is weighted by a value $c_{i,j}$ which represents the time needed by a parent node P_i to communicate one task to its child P_j.

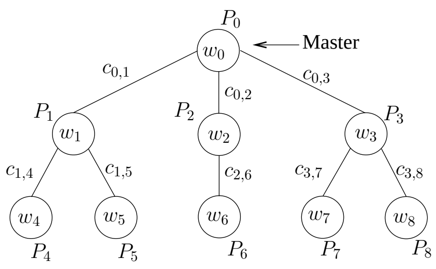


Figure 2.1: A tree labeled with node (computation) and edge (communication) weights.

All the w_i are assumed to be positive rational numbers since they represent the nodes processing times. We disallow $w_i = 0$ since it would permit node P_i to perform an infinite number of tasks, but we allow $w_i = +\infty$; then, P_i has no computing power, but can still forward tasks to other processors (e.g. to model a switch). Similarly, we assume that all $c_{i,j}$ are positive rational numbers since they correspond to the communication times between two processors.

We introduce the computing rate $r_i = \frac{1}{w_i}$ which is the number of tasks that processor P_i can process per time unit, and the communicating rate $b_{i,j} = \frac{1}{c_{i,j}}$ which is the number of tasks that processor P_i can send to child P_j per time unit. We let $\frac{1}{+\infty} = 0$, and $\mathcal{C}_i$ denotes the index set of the children of node P_i.

There are several scenarios for the operation mode of the processors, and we rely to the classification proposed in [4]. A processor can do three kinds of activity: (i) it can perform some computation, (ii) it can receive tasks from its parent, and (iii) it can send tasks to its children. The degree of simultaneity between these three activities indicates the level of performance of a processor and its networking device. It is important to point out that different processors of the platform may operate under different modes. In this paper, we concentrate on the *full overlap, single-port* model [2, 5], since it has been shown in [4] that the other models reduce to it. In the *full overlap, single-port* model, a processor can simultaneously receive tasks from its parent, perform some (independent) computation, and send tasks to one of its children. At any given time-step, a given processor may open only two connections, one in emission and one in reception. We state the communication model more precisely: If P_i sends one task to P_j at time-step t, then:

- P_j cannot start executing or sending this task before time-step $t + c_{i,j}$,
- P_j cannot initiate a new receive operation before time-step $t + c_{i,j}$ (but, it can perform a send operation and independent computation),
- P_i cannot initiate another send operation before time-step $t+c_{i,j}$ (but, it can perform a receive operation and independent computation).

2.4 The bandwidth-centric principle

An iterative method that determines the maximum steady-state rate of a heterogeneous tree has been presented by Beaumont et al. in [5]. Interestingly, it turns out that this strategy is *bandwidth-centric*: If enough bandwidth is available to the node, then all the children are kept busy. However, if bandwidth is limited, then tasks should be allocated only to the children which have sufficiently fast communication times regardless of their computing speeds. Nevertheless, the computing speeds of the children determine the frequency at which children will request tasks to their parent. So a faster processor will request tasks more often than a slower one. What the bandwidth-centric principle says is that if two children are in concurrence for obtaining a task, priority should be given to the child with fastest communication time, as this will optimize the communication resource of the parent.

Formally, let us recall the Proposition presented in [5] to solve the case for fork graphs.

A fork graph as shown in Figure 2.2 consists of a node P_0 and its k children $P_1, \dots, P_k$. P_0 needs c_i units of time to communicate a task to child P_i. Concurrently, P_0 can receive tasks from its own parent P_{-1}, requiring c_{-1} time units per task.

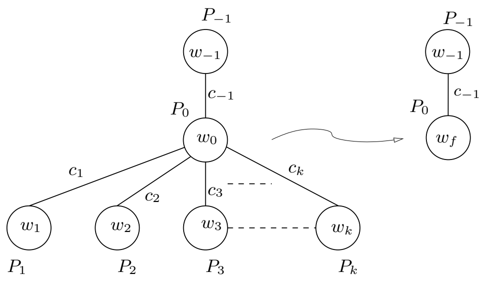


Figure 2.2: A fork graph and the reduced node of equivalent computing power.

Proposition 2.1 ([5]). *With the above notations, the minimal value of w_f for the fork graph is obtained as follows:*

1. *Sort the children by increasing communication times. Re-number them so that* $c_1 \leq c_2 \leq \cdots \leq c_k$.
2. *Let p be the largest index so that* $\sum_{i=1}^{p} \frac{c_i}{w_i} \leq 1$. *If* $p < k$ *let* $\varepsilon = 1 - \sum_{i=1}^{p} \frac{c_i}{w_i}$, *otherwise let* $\varepsilon = 0$.
3. *Then* $w_f = \max\left\{ c_{-1}, \frac{1}{\sum_{i=0}^{p} \frac{1}{w_i} + \frac{\varepsilon}{c_{p+1}}} \right\}$

Based on the bandwidth-centric principle, Beaumont et al. conceived a bottom-up method that iteratively determines the steady-state throughput of the tree: At each time-step, the leaves of the tree are reduced together with their parent into a single node of equivalent computing power determined by Proposition 2.1 (see Figure 2.2). The procedure ends when there remains a single node having a computing power equivalent to the entire tree. However, Beaumont et al. did not specify how to achieve this maximum steady-state throughput in practice.

2.5 Reversing the tree traversal

Although the bottom-up procedure based on the bandwidth-centric principle provides the optimal throughput of the tree, a large number of unnecessary operations are done for strongly bandwidth limited platforms (i.e. when there is a bottleneck somewhere high up in the hierarchy, causing that many nodes of the platform can not be fed with tasks). Indeed, in such cases, many fork graph reductions are performed unnecessarily since only few nodes of the platform will be actually used during the computation. Therefore, we propose to perform a depth-first traversal of the tree (according to the bandwidth-centric principle), allowing hence to visit only the nodes that will be used in the final schedule.

Beaumont et al. show in [5] that we can solve the problem within the time unit interval, and then build up a schedule where an integer number of tasks are sent and processed. Our procedure is based on this result and involves *transactions* between the nodes composing the platform.

Definition 2.1. *A* **transaction** *is defined as a two-phase protocol between a parent node P_p and a child node P_c. The first phase of the transaction consists in P_p sending a message to P_c containing a single number β that represents the number of tasks that P_p can supply to P_c per time unit. We term the first phase a* **proposal** *from P_p to P_c. The second phase of the transaction consists in P_c sending a message back to its parent P_p containing a single number θ that represents the number of tasks that P_c could not handle. We term the second phase an* **acknowledgment** *from P_c to P_p. Hence when the transaction is closed, P_p knows that its child P_c can consume $(\beta - \theta)$ tasks per time unit.*

We use the following notations:$P_x \xrightarrow{\sigma} P_y$ indicates that node P_x sends a number σ to node P_y, and $P_x \xleftarrow{\sigma} P_y$ indicates that node P_x receives a number σ from P_y.

Our procedure works as follows: The node P_0 currently visited during the traversal of the tree will receive a proposal from its parent P_{-1}. P_0 will then evaluate how many tasks it can process per time unit, and if there are some tasks left, try to propose them to its children. P_0 will then deal with its children one by one according to the bandwidth-centric principle, i.e. starting to deal with children that have the fastest communication times. P_0 will open a transaction with its first child P_1 by proposing the maximum number of tasks that it can supply to P_1 per time unit. P_1 in turn faces the same situation than P_0, and will keep a maximum of tasks for itself, and if there are some tasks left, try to delegate them to its children by negotiating new transactions. Hence, proposals will propagate down the tree, until either we reach a leave of the tree, either all the tasks have been allocated, or the current node has fully utilized its bandwidth and can not forward tasks further down the tree. Then the node P_c at which the proposal propagation stopped, will acknowledge its parent P_p with the amount of tasks that it could not process, and the transaction between P_p and P_c is closed. P_p will then take into consideration its transaction with P_c by reserving enough bandwidth to honor the transaction, and if it has some bandwidth left, as well as more tasks to delegate, will open a new transaction with another child. Hence, proposals travel down the tree opening transactions while acknowledgments travel up the tree closing the transactions in a recursive fashion.

Formally, let P_0 be the node currently visited during the tree traversal. Let P_{-1} be the parent of P_0, and $P_1, P_2, \ldots, P_k$ be the k children of P_0 with communication times $c_1, c_2, \ldots, c_k$ respectively. Further, let δ_0 be the number of virtual tasks owned by P_0 , τ_0 be the bandwidth time of P_0 (to send tasks to its children), and α_0 be the number of tasks computed by P_0 per time unit. At the beginning of the procedure we have $\delta_0 = 0$, $\tau_0 = 1$ since the time unit interval is considered, and $\alpha_0 = 0$.

Proposition 2.2. *With the above notations, the optimal throughput of a tree T_h of height h, is obtained via applying the* BW-First() *procedure.*

Algorithm 1: BW-First(P_0)

```
1  begin
2      $\delta_0 := 0, \tau_0 := 1, \alpha_0 := 0$;
3      $P_0 \xleftarrow{\lambda} P_{-1}$;
4      $\alpha_0 := \min\{r_0, \lambda\}$;
5      $\delta_0 := \lambda - \alpha_0$;
6      foreach child $P_i$ taken according to the bandwidth-centric principle do
7          if $\delta_0 = 0$ or $\tau_0 = 0$ then
8              goto instruction 14;
9          $\beta_i := \min\{\delta_0, \tau_0 \times b_i\}$;
10         $P_0 \xrightarrow{\beta_i} P_i$;
11         $P_0 \xleftarrow{\theta_i} P_i$;
12         $\delta_0 := \delta_0 - (\beta_i - \theta_i)$;
13         $\tau_0 := \tau_0 - (\beta_i - \theta_i) \times c_i$;
14     $P_0 \xrightarrow{\delta_0} P_{-1}$;
15 end
```

Proof. In order to apply procedure *BW-First()* on the root P_{root} of the tree, we merely create a link connecting the root to a virtual parent P_v with no computing power. Then, the maximum number of task t_{max} that the tree rooted in P_{root} can execute per time unit is evaluated. Under the *single-port, full overlap* model, we have $t_{max} = r_{root} + \max\{b_i \mid i \in \mathcal{C}_{root}\}$. We then make P_v propose t_{max} tasks to P_{root}, and call procedure *BW-First()* on P_{root}. At the end of the procedure, P_v will receive an acknowledgments of θ tasks from P_{root}, and the optimal throughput of the tree is equal to the quantity $(t_{max} - \theta)$.

The proof is done by recurrence over h, the height of the tree. For $h = 1$, the tree is actually a fork graph. We will prove that for fork graphs, the *BW-First()* procedure is equivalent to Proposition 2.1. Let r_f be the computing rate of the fork graph rooted in P_0. Let us show that $r_f = \sum_{i=0}^{p} r_i + \varepsilon \times b_{p+1}$, where p and ε are defined in Proposition 2.1.

First P_0 receives a proposal of λ tasks from its parent P_{-1}, and keeps as many tasks as possible for its own computation. Then, P_0 will propose the remaining tasks to its children according to the bandwidth-centric principle. For each child P_i, P_0 must determine whether all the remaining tasks can be communicated or not. Then a proposal is made to child P_i. It is now the turn of P_i to execute the *BW-First()* procedure. P_i receives a proposal of θ_i tasks from its parent P_0, keeps as many tasks as possible for its own computation and since P_i does not have any children, sends back to P_0 the number of tasks that it could not execute. At that point, either P_i is fully utilized ($\alpha_i = r_i$), or not ($\alpha_i < r_i$). In the first case scenario, P_0 will proceed to its next child, with previously adjusting δ_0 and τ_0, considering that $c_i \times r_i$ time units will be necessary to furnish r_i tasks to P_i. P_0 will then establish new transactions with its children until the second case scenario takes place (i.e. a child P_{q+1} is not fully utilized). In this case, either all the virtual tasks owned by P_0 have been processed, or P_0 utilized all its bandwidth time and can not send as many tasks as P_{q+1} can consume. In the first case, the limiting factor is the number of tasks λ received from P_{-1}. In the second case, the limiting factor is the bandwidth of P_0: P_0 fed fully its q first children with tasks, i.e. $\forall i \leq q, \alpha_i = r_i$. P_0 will hence spend $\sum_{i=1}^{i=q} c_i r_i$ time units to communicate with its q first children, and only $c_{q+1}\alpha_{q+1}$ time units with P_{q+1}. Since the bandwidth of P_0 is saturated, we have $\sum_{i=1}^{i=q} c_i r_i + \alpha_{q+1} r_{q+1} = 1$. Since P_{q+1} could consume all the tasks proposed by P_0 without being fully utilized, we have $\alpha_{q+1} < r_{q+1}$ which by scaling both sides with c_{q+1} gives $\alpha_{q+1}c_{q+1} < r_{q+1}c_{q+1}$. Consequently, we have $\sum_{i=1}^{i=q+1} c_i r_i > 1$, which gives $q = p$ and $\varepsilon = 1 - \sum_{i=1}^{i=q} c_i r_i$. If $q = k$, then all the children have been fully fed with tasks, and we have $\varepsilon = 0$.

We have hence established that

$$r_f = \min\left\{\lambda, \sum_{i=0}^{p} r_i + \varepsilon \times b_{p+1}\right\},$$

and, since in the fork graph case $\lambda = b_{-1}$, we have $r_f = \frac{1}{w_f}$, where w_f is given by Proposition 2.1. Consequently, procedure *BW-First()* applies Proposition 2.1 for the fork graphs case.

Assume now that Proposition 2.2 is true for rank h, i.e. for trees of height h. Let us now prove that Proposition 2.2 is also true for rank $h+1$. If a node P_i is not visited while applying procedure *BW-First()* (e.g. its parent has no time left to communicate, or no more tasks to delegate), then we can merely remove all the sub-tree rooted in P_i without influencing on the final throughput of the tree. Assume hence that applying procedure *BW-First()* to a tree T_{h+1} involves visiting a node P_i of depth h. If P_i does have some children, this implies that the sub-tree F_i rooted in P_i is a fork graph. The procedure *BW-First()* applied to the root P_i of the fork graph F_i will determine the throughput r_{F_i} of F_i. The fork graph F_i is then equivalent to a single node having a computing rate equal to r_{F_i}. Consequently, applying procedure *BW-First()* on a tree T_{h+1} of height $h+1$ is equivalent to applying procedure *BW-First()* on a tree T_h of height h. Then Proposition 2.2 holds for all $h \geq 1$. □

The *BW-First()* procedure is more *efficient* than the bottom-up method, since only the nodes that are effectively used in the final schedule are visited. Moreover, it is more *convenient* through a straightforward recursive implementation. Indeed, we merely traverse the tree in a depth-first manner, and are hence released from the burden of identifying for each step which set of leaves should be transformed. Particularly, the *BW-First()* procedure might be a useful tool for topological studies, which aim at determining the best tree *overlay network* that is built on top of the physical network topology [12]. A quick way to evaluate the throughput of a tree allows to consider a wider set of trees.

Moreover, the *BW-First()* procedure can be implemented as a lightweight communication protocol between the nodes of the platform. Indeed, the optimal throughput of the tree is obtained without access to any global information. Each node makes its decisions based on information that is directly measurable plus on additional information received from its parent and children. One could term such a distributed protocol *semi-autonomous*.

For dynamic adaptation concerns, one could imagine the following strategy: The root of the tree, receiving periodically the results of the computations, can measure if there has been a drop in throughput performance. Under a certain threshold, the root might initiate the *BW-First()* procedure once more in order to capture the actual state of the platform. Since the messages exchanged between two nodes during the *BW-First()* procedure are single numbers, we could argue that the running time of the *BW-First()* procedure is negligible as opposed to the time of communicating tasks. However, this last point needs more investigation, and we leave this issue for future work.

Finally, infinite networks have been studied by Bataineh and Robertazzi in [3]. The authors showed that a finite-size network tree load sharing a divisible job can perform almost as well as an infinite network tree. The *BW-First()* procedure allows to determine the throughput of infinite network trees, as opposed to the bottom-up method.

2.6 Reconstructing the schedule

When the *BW-First()* procedure has been executed, each node has all the rational values of its activity variables as its disposal. Hence, during one time unit, let $\eta_{-1} = \frac{\rho_{-1}}{\mu_{-1}} = (\lambda - \delta_0)$ be the number of tasks that node P_0 receives from its parent, $\eta_0 = \frac{\rho_0}{\mu_0} = \alpha_0$ be the number of tasks that P_0 computes locally, and $\eta_i = \frac{\rho_i}{\mu_i} = (\beta_i - \theta_i)$ be the number of tasks that P_0 sends to each child P_i. Note that all the numerators and denominators are positive integers, and η_{-1}, η_0 and η_i can be equal to zero. The steady-state regime is ensured by the fact that node P_0 receives as many tasks as it can consume. This *conservation law* translates into equation (2.1).

$$\eta_{-1} = \sum_{i=0}^{k} \eta_i \tag{2.1}$$

Our aim is now to build up a periodic schedule where an integer number of tasks are sent and/or executed. As mentioned in [6], we can obtain a period T by taking the least common multiple of all the denominators μ_i for each node. However, this approach has a major inconvenient: The period might be embarrassingly long, which makes it inconvenient to describe the activity of the nodes, and requires unnecessary large buffering spaces to store the tasks required from one period to another.

2.6.1 Asynchronous schedule

In order to obtain a more compact description of the schedule, we propose to desynchronize the activities of the *single-port, full overlap* model, i.e. receiving tasks, computing tasks and sending tasks. After all, this model allows to perform these three activities concurrently.

T_0^r is defined as the shortest period during which node P_0 receives an integer number φ_{-1} of tasks from its parent; T_0^x is defined as the shortest period during which node P_0 executes an integer number φ_0 of tasks; and T_0^s is defined as the shortest period during which node P_0 sends an integer number φ_i of tasks to each child P_i.

Lemma 2.1. *With the above notations, the minimal periods as well as the integer number of tasks treated are obtained as follows:*

$$\left\{\begin{array}{l|l} T_0^s = lcm\{\mu_i \mid i \in \mathcal{C}_0\} & \varphi_i = \eta_i \times T_0^s, \forall i \in \mathcal{C}_0 \\ T_0^x = \mu_0 & \varphi_0 = \eta_0 \times T_0^x \\ T_0^r = T_{-1}^s & \varphi_{-1} = \eta_{-1} \times T_0^r \end{array}\right. \tag{2.2}$$

Proof. Node P_0 must send $\eta_i = \frac{\rho_i}{\mu_i}$ tasks per time unit to each child P_i. In order to obtain a minimal period where an integer number of tasks is sent to each child, we have to take the least common multiple of all the denominators $\{\mu_i \mid i \in \mathcal{C}_0\}$.

Node P_0 must compute $\eta_0 = \frac{\rho_0}{\mu_0}$ tasks per time unit, which gives a minimal period of μ_0 time units during which ρ_0 tasks are computed.

Since any node P_0 receives tasks only from its parent, the receiving period T_0^r of P_0 should be equal to the sending period T_{-1}^s of its parent P_{-1}, which has been shown to be minimal. Obviously the root of the tree should not receive any tasks, and we can enforce $T_{root}^r = 0$.

□

Proposition 2.3. *Any node P_0 can desynchronize its activities according to Lemma 2.1 without violating the conservation law.*

Proof. By taking the least common multiple of the three asynchronous periods, we obtain a period T_0 during which all the received tasks are consumed. That is to say, every T_0 time units, P_0 receives an integer number χ_{-1} of tasks from its parent, computes an integer number χ_0 of tasks, and sends an integer number χ_i of tasks to each child P_i. This translates into equation set (2.3).

$$\left\{\begin{array}{l|l} T_0 = lcm\{T_0^s, T_0^x, T_0^r\} & \\ \chi_{-1} = \eta_{-1} \times T_0 & \chi_i = \eta_i \times T_0, \forall i \in \mathcal{C}_0 \\ \chi_0 = \eta_0 \times T_0 & \chi_{-1} = \sum_{i=0}^{k} \chi_i \end{array}\right. \tag{2.3}$$

The only requirement for ensuring steady-state with asynchronous activities is to dispose of enough tasks buffered at node locations. For now, assume that χ_{-1} tasks have been buffered during the start-up phase. Then, we have a steady number of tasks stored from one period T_0 to another, which ensures steady-state behavior.

□

2.6.2 Event-driven schedule

We now propose an event-driven schedule, where any time-related information has been removed (except for the root node). Consider first the case of any node P_0 different from the root. Lemma 2.1 gives the minimal period T_0^x and T_0^s for computing and sending tasks respectively. The minimal period T_0^c during which an integer number of tasks is consumed (either processed locally or delegated to a child) can be obtained by taking the least common multiple of T_0^x and T_0^s. Let ψ_0 be the number of tasks executed by P_0 every T_0^c time units, and ψ_i be the number of tasks delegated to child P_i every T_0^c time units. We have hence the following equation set:

$$\begin{cases} T_0^c = lcm\{T_0^x, T_0^s\} \\ \psi_0 = \eta_0 \times T_0^c \\ \psi_i = \eta_i \times T_0^c, \forall i \in \mathcal{C}_0 \end{cases} \tag{2.4}$$

P_0 does not need time-related information any longer. Instead, P_0 will handle incoming tasks by bunches of size $\Psi = \sum_{i=0}^{k} \psi_i$. Indeed, of all the tasks that P_0 will receive from its parent, $\frac{1}{\psi_0}$ fraction of them are intended for itself, and $\frac{1}{\psi_i}$ fraction of them are intended for each child P_i (provided that $\psi_i > 0$). Therefore, the only information which is necessary is to know how many tasks should be executed locally (i.e. ψ_0), and how many tasks should be delegated to each child P_i (i.e. ψ_i) every T_0^c time units. P_0 will then handle incoming tasks by bunches of size Ψ, without using any time-based information. The event-driven schedule for any node P_0 different from the root is summarized by procedure *Steady-State()*.

```
Procedure Steady-State($P_0$)
begin
    while true do
        foreach bunch of $\Psi$ tasks do
            Compute $\psi_0$ tasks locally ;
            foreach child $P_i$ do
                Send $\psi_i$ tasks to $P_i$;
end
```

Since the computing platform is a tree, nodes receive tasks only from their parent. One can then let a receiving thread blocked in reception, waiting for tasks to arrive from the parent, and storing them locally upon reception. Or one can use non-blocking receive calls. Consequently, we do not need to describe further the receiving activity.

The case of the root P_{root} is slightly different, since P_{root} has all its tasks as its disposal. Hence we can let the root compute all the time, and only consider the sending activity. P_{root} will consequently use the φ_i instead of the ψ_i. However, if P_{root} has enough bandwidth to feed all the nodes of the platform, then P_{root} must use its sending period T_{root}^s in order to ensure that the conservation laws are respected, and hence to maintain the steady-state behavior of the system. In this way, the root will regulate the arrival rates of tasks to the other nodes of the tree, by feeding every child P_i with φ_i tasks every T_{root}^s time units. The associated schedule of the root is described by procedure *Steady-State-Root()*.

Procedure `Steady-State-Root(`P_{root}`)`

```
begin
    if $\tau_{root} > 0$ then
        foreach time period $T^s_{root}$ do
            foreach child $P_i$ do
                Send $\varphi_i$ tasks to $P_i$;
    else
        while true do
            foreach child $P_i$ do
                Send $\varphi_i$ tasks to $P_i$;
end
```

2.6.3 Local scheduling

Although the event-driven schedule ensures the steady-state regime, some scheduling decisions remain to be taken. Indeed, P_0 will treat tasks received from its parent by bunches of size Ψ, but in which order should it delegate tasks to its children? And which tasks should be kept for itself? All the schedules are equivalent in terms of steady-state throughput. However, some schedules might be more advantageous than others with respect to memory limitations. The one we are proposing has been designed with the objective of minimizing the number of tasks that will be buffered at steady-state. Obviously, minimizing the number of tasks buffered at steady-state, is of interest since it reduces memory usage during the computation. In addition, the low number of tasks required to ensure the steady-state regime, will lead to fast start-up and wind-down phases (see Section 2.7).

Our local schedule strategy interleaves the incoming tasks proportionally to the ψ quantities (φ for the root). Let us start with node P_0 itself which should compute ψ_0 tasks. We merely split the unity domain into $(\psi_0 + 1)$ parts, each of size $\Delta_0 = \frac{1}{\psi_0}$. The same operation is repeated for each child P_i, and the unity domain is split into $(\psi_i + 1)$ parts of size $\Delta_i = \frac{1}{\psi_i}$ ($\forall \psi_i > 0$). When this is done, we obtain an order among the incoming tasks that corresponds to our allocation of tasks to nodes. Let us take an example to illustrate this strategy. Consider a node P_0 having two children P_1 and P_2. We let $\psi_0 = 1, \psi_1 = 2$ and $\psi_2 = 4$. The allocation for our example is depicted in Figure 2.3. The first task is sent to P_2, the second to P_1, the third to P_2, etc.

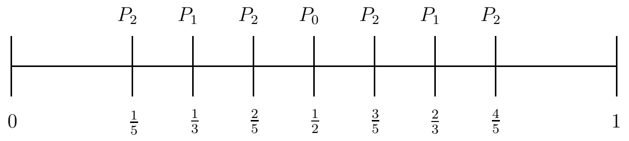


Figure 2.3: Scheduling the incoming tasks

If two processors are contesting for one task, we arbitrarily prioritize the processor with smallest ψ. If both processors have an equal ψ, we prioritize the one with smallest index. The rationale behind this strategy, is to space out tasks intended to one node as much as possible. Instead of giving the nodes all their tasks at once, we disseminate them along the period, making it possible for the nodes to consume tasks almost as fast as they receive them. Furthermore, due to symmetrical reasons, the description of the local schedules can be divided by two.

2.7 Efficient start-up phase

Usually, the start-up phase is considered as just a way to enter the steady-state regime [2, 5, 6]. The traditional answer to this problem is to send tasks down the tree without doing any useful computation, until each node gets the number of tasks required to enter the steady-state regime. This takes T times the maximum depth of the tree, where T is the steady-state period of the tree [5].

Alternatively, Kreaseck et al. [12] propose a demand-driven start-up strategy, where nodes request tasks to their parent, which in turn will forward the demands up the hierarchy. However, they observed that in practice, their protocol experienced long start-up phases.

It is of prior importance to be able to reach steady-state as fast as possible. Indeed, under dynamic conditions, recomputing the optimal schedule might be necessary in order to efficiently utilize the platform. Under these conditions, not being able to execute any tasks during the start-up phase becomes no longer acceptable. For these reasons, the importance of the start-up and wind-down phases should not be minimized. We propose a start-up phase where computations are allowed. In fact, every node will act according to its event-driven schedule from the beginning of the computation.

Proposition 2.4. *Applying the event-driven schedule, from the beginning of the computation, leads every node P_0 to its steady-state regime, in at most $\sum T_i^s \mid i \in \mathcal{A}_0$ time units, where $\mathcal{A}_0$ is the set of ancestors of node P_0.*

Proof. Intuitively, during the start-up phase, nodes will receive tasks at full rate, but since they do not have buffered tasks at their disposal, will starve waiting for them to arrive. Nodes will store locally incoming tasks and schedule them immediately. Hence, node buffers will be filled up with tasks, much like the way a pipeline is getting full, until the task consuming rate catches up with the task receiving rate.

Formally, consider any node P_0 different from the root. From equation set (2.3) at steady-sate we have: P_0 receives χ_{-1} tasks every T_{-1}^s time units from its parent P_{-1}. Assume that at time step t, P_{-1} is in steady-state. Assume also that during the time period $[t, t+T_{-1}^s]$, P_0 can consume only λ of these χ_{-1} tasks (since P_0 does not have any tasks buffered yet, some time will be spent waiting for them to arrive). The $(\chi_{-1} - \lambda)$ tasks left are hence stored by P_0 for the next period of time T_{-1}^s. During the time period $[t + T_{-1}^s, t + 2T_{-1}^s]$, P_0 receives χ_{-1} new tasks. We know that P_0 can consume λ of the new tasks. Since P_0 owns $(\chi_{-1} - \lambda)$ tasks from the previous period, P_0 is no longer idle while waiting for tasks. Hence, P_0 consumes $\lambda + \chi_{-1} - \lambda = \chi_{-1}$ tasks, and is in steady-state at time-step $t + T^s_{-1}$. Since the root of the tree is already in steady-state from the beginning of the computation (at $t = 0$), applying this reasoning from the root down the hierarchy gives $t = \sum T^s_i \mid i \in \mathcal{A}_i$.

$\square$

It is important to point out that, due to the continuity between one time period to another, P_0 will enter into its steady-state regime earlier than time step $t = \sum T^s_i \mid i \in \mathcal{A}_0$. While the entire tree enters the steady-state regime as soon as all the nodes entered in steady-state. Such behavior can be observed in the example of Section 2.8.

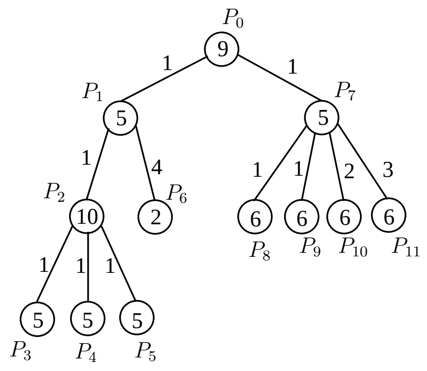
2.8 Example

Let us illustrate our results with an example taken from [4]. Consider the tree T depicted in Figure 2.4 (a). Procedure *BW-First()* obtained a throughput of 10 tasks every 9 time units, which corresponds to the result obtained by the bottom-up method of Beaumont et al. [4]. The successive transactions established during the *BW-First()* procedure are depicted in Figure 2.4 (b). Note that nodes P_8, P_9, P_{10} and P_{11} were not visited, meaning that they will not be used in the final schedule. The number of tasks that each node receives (η_{-1}) and computes (η_0) per time unit are depicted in Figure 2.4 (c). The final description of the local schedules is very compact and is depicted in Figure 2.4 (d). The final computation, with start-up and wind-down phases is depicted with a Gantt diagram in Figure 2.5.

We would like to point out few interesting observations. The tree has a steady-state period T of 360 time units, while the rootless tree has a throughput of 40 tasks every 40 time units. The start-up phase lasts for 40 time units, which is equal to one steady-state period of the rootless tree. During the start-up phase, the rootless tree executes 32 tasks, that is to say 80% of its optimal throughput. At an arbitrary point in steady-state (time step 115), we stopped delegating tasks to the tree, and observed that the wind-down phase lasts for only 10 time units (4 times shorter than the steady-state period of the rootless tree). This very short wind-down phase is the result of our local schedule strategy, which aim at minimizing the number of tasks buffered during steady-state.

2.9 Future work

Handling the return of the results back to the master should be considered for future work. The bandwidth-centric principle does not hold when the return of the results are considered, despite the claim of Beaumont et al. [5] and Kreaseck et al. [12]. In their work, the aforementioned authors model the communication times between two processors P_i and P_j as being the time needed by a parent P_i to communicate the data for one task to a child P_j plus the time for the child to return the result when it is finished. They



(a) The platform tree.

$P_v \xrightarrow{\frac{10}{9}} P_0,$	$P_0 \xrightarrow{1} P_1,$	$P_1 \xrightarrow{\frac{4}{5}} P_2,$	$P_2 \xrightarrow{\frac{7}{10}} P_3,$
$P_2 \xleftarrow{\frac{1}{2}} P_3,$	$P_2 \xrightarrow{\frac{1}{2}} P_4,$	$P_2 \xleftarrow{\frac{3}{10}} P_4,$	$P_2 \xrightarrow{\frac{3}{10}} P_5,$
$P_2 \xleftarrow{\frac{1}{10}} P_5,$	$P_1 \xleftarrow{\frac{1}{10}} P_2,$	$P_1 \xrightarrow{\frac{3}{40}} P_6,$	$P_1 \xleftarrow{0} P_6,$
$P_0 \xleftarrow{\frac{1}{40}} P_1,$	$P_0 \xrightarrow{\frac{1}{40}} P_7,$	$P_0 \xleftarrow{0} P_7,$	$P_v \xleftarrow{0} P_0.$

(b) Transactions established.

	P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
η_{-1}	0	$\frac{39}{40}$	$\frac{7}{10}$	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{3}{40}$	$\frac{1}{40}$
η_0	$\frac{1}{9}$	$\frac{1}{5}$	$\frac{1}{10}$	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{1}{5}$	$\frac{3}{40}$	$\frac{1}{40}$

(c) Number of tasks received and computed per time unit.

Nodes	Ψ	Local Schedules
P_0	40	$19P_1, P_7, 20P_1$
P_1	39	$3P_2, P_1, 3P_2, P_1, P_2, P_6, 2P_2, P_1,$ $3P_2, P_1, 2P_2, P_6, 2P_2, P_1, 3P_2, P_1,$ $2P_2, P_6, P_2, P_1, 3P_2, P_1, 3P_2$
P_2	7	$P_3, P_4, P_5, P_2, P_3, P_4, P_5$
P_3	1	P_3
P_4	1	P_4
P_5	1	P_5
P_6	1	P_6
P_7	1	P_7

(d) Local Schedules.

Figure 2.4: Building-up local schedules.

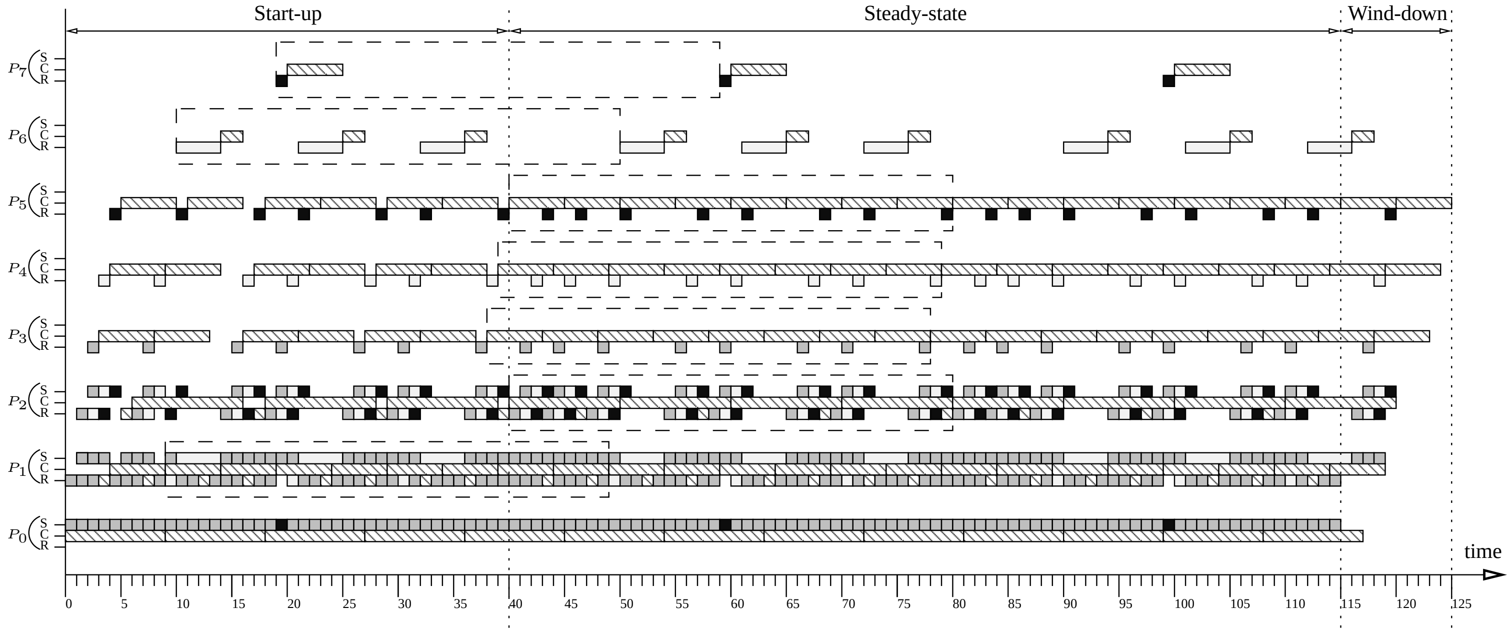


Figure 2.5: Final computation: S = Send, C = Compute, R = Receive.

argue that "for the purpose of computing steady-state behavior, it does not matter what fraction of the communication time is spent sending a problem and what fraction is spent receiving the results" [5]. We will show that this simplification is erroneous. Although the simplification holds for the traffic of messages on the communication links, it neglects the receiving port resource. Let us illustrate this with a small platform example composed of three nodes. The master has only two children. Each child can process 1 task per time unit. It takes 0.5 time units to send one task from the master to its children, and 0.5 time units to return the results of one task from the children to the master. The optimal throughput of the platform is then 2 tasks per time units. If we join the time sending the input data with the time for receiving the results (as suggested in [5, 12]), we obtain a platform throughput of 1 task per time unit. Hence, the simplification does not work for returning the results back to the master, and consequently this problem is still open.

It would be interesting to evaluate the *BW-First()* procedure using simulations (for example with the SimGrid toolkit [8]), and compare it to the autonomous protocol proposed by Kreaseck et al. [12]. Especially, measuring the overhead incurred by the global synchronization phase would give some insight on how frequently the *BW-First()* procedure might be initiated by the root. Finally, trying different local schedules might be interesting with respect to start-up and wind-down phases as well as memory limitations.

2.10 Conclusion

The problem of allocating a large number of independent, equal-sized tasks to heterogeneous trees was considered. We assumed that a specific node, the master initially, holds the data associated to the tasks, and that returning the results of the computations to the master is negligible. This paper made the following contributions to this problem:

• We proposed a distributed method, the *BW-First()* procedure which is an efficient, practical and scalable implementation of the theoretical results presented in [5].

• Based on the results of the *BW-First()* procedure, each node can then build up its own local schedule independently of the rest of the platform. The result is a loosely synchronized schedule, where nodes are synchronized only with their children, as opposed to the traditional approach where all the nodes of the platform are synchronized together.

• The resulting local schedules are event driven, meaning that every node (except the root) acts without any time-related information, and consequently, their description is very compact.

• We proposed a local schedule strategy that makes use of a small amount of tasks buffered at steady-state. Not only this approach requires less memory, but it also considerably reduces the length of the start-up and wind-down phases.

• We presented a start-up phase strategy which allows useful computation as opposed to the traditional approach.

The goal of this paper was to close the gap between theory and practice by embedding theoretical knowledge into a practical and scalable implementation. We believe that the techniques presented in this paper are valuable for conceiving scheduling strategies that tackle the platform dynamics, i.e. where resources exhibit dynamic performance characteristics and availability.

Acknowledgments

The author would like to thank Anne C. Elster as well as Olivier Beaumont for their comments and suggestions, which greatly improved the final version of the paper. This research was supported by the Department of Computer and Information Science (IDI), at the Norwegian University of Science and Technology (NTNU).

Bibliography

[1] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti@home: An Experiment in Public-Resource Computing. *Communications of the ACM*, 45(11):56–61, 2002.

[2] C. Banino, O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Scheduling Strategies for Master-Slave Tasking on Heterogeneous Processor Platforms. *IEEE Transactions on Parallel and Distributed Systems*, 15(4):319–330, 2004.

[3] S. Bataineh and T. Robertazzi. Performance Limits for Processor Networks with Divisible Jobs. *IEEE Transactions on Aerospace and Electronic Systems*, 33:1189–1198, Octobre 1997.

[4] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Bandwidth-Centric Allocation of Independent Tasks on Heterogeneous Platforms. Technical Report RR-2001-25, LIP, ENS Lyon, France, June 2001. URL: http://www.ens-lyon/∼yrobert.

[5] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Bandwidth-Centric Allocation of Independent Tasks on Heterogeneous Platforms. In *International Parallel and Distributed Processing Symposium IPDPS'2002*, pages 67–72. IEEE Computer Society Press, 2002.

[6] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert. Steady-State Scheduling on Heterogeneous Clusters: Why and How? In *6th Workshop on Advances in Parallel and Distributed Computational Models, APDCM 2004*, page 171a (8 pages). IEEE Computer Society Press, 2004.

[7] O. Beaumont, A. Legrand, and Y. Robert. The Master-Slave Paradigm with Heterogeneous Processors. *IEEE Transactions on Parallel and Distributed Systems*, 14:897–908, 2003.

[8] H. Casanova. SimGrid: A Toolkit for the Simulation of Application Scheduling. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, page 430. IEEE Computer Society, 2001.

[9] A. Chien, B. Calder, S. Elbert, and K. Bhatia. Entropia: Architecture and Performance of an Enterprise Desktop Grid System. *Journal of Parallel and Distributed Computing*, 63:597, May 2003.

[10] P. F. Dutot. Master-slave Tasking on Heterogeneous Processors. In *International Parallel and Distributed Processing Symposium*, page 25b. IEEE Computer Society Press, April 2003.

[11] P. F. Dutot. Complexity of Master-slave Tasking on Heterogeneous Trees. *European Journal on Operationnal Research*, 164(3):690–695, August 2005.

[12] B. Kreaseck, L. Carter, H. Casanova, and J. Ferrante. Autonomous Protocols for Bandwidth-Centric Scheduling of Independent-Task Applications. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 26.1, Washington, DC, USA, 2003. IEEE Computer Society.

[13] G. Shao, F. Berman, and R. Wolski. Master/Slave Computing on the Grid. In *HCW '00: Proceedings of the 9th Heterogeneous Computing Workshop*, page 3, Washington, DC, USA, 2000. IEEE Computer Society.

[14] D. F. Sittig, D. Foulser, N. Carriero, G. McCorkle, and P. L. Miller. A Parallel Computing Approach to Genetic Sequence Comparison: The Master Worker Paradigm with Interworker Communication. *Computers and Biomedical Research*, 24:152–169, 1991.

[15] R. Wolski, N. T. Spring, and J. Hayes. The Network Weather Service: a Distributed Resource Performance Forecasting Service for Metacomputing. *Journal of Future Generation Computing Systems*, 15(5-6):757–768, 1999.

Paper 3

Master-Slave Tasking on Asymmetric Networks
Cyril Banino-Rokkones, Olivier Beaumont and Lasse Natvig.
In *Proceedings of 12th International Euro-Par Conference, Euro-Par 2006*,
August 29 - September 1, Pages 167–176, Dresden, Germany.

Master-Slave Tasking on Asymmetric Networks

Cyril Banino-Rokkones[1], Olivier Beaumont[2] and Lasse Natvig[1]

[1]Norwegian University of Science and Technology, NO-7491 Trondheim, Norway.
[2]LaBRI, UMR CNRS 5800, Domaine Universitaire, 33405 Talence Cedex, France

Abstract

This paper presents new techniques for master-slave tasking on tree-shaped networks with fully heterogeneous communication and processing resources. A large number of independent, equal-sized tasks are distributed from the master node to the slave nodes for processing and return of result files. The network links present bandwidth asymmetry, i.e. the send and receive bandwidths of a link may be different. The nodes can overlap computation with at most one send and one receive operation. A centralized algorithm that maximizes the platform throughput under static conditions is presented. Thereafter, we propose several distributed heuristics making scheduling decisions based on information estimated locally. Extensive simulations demonstrate that distributed heuristics are better suited to cope with dynamic environments, but also compete well with centralized heuristics in static environments.

3.1 Introduction

In this paper, we consider the allocation of a large number of independent equal-sized tasks onto a tree platform. We concentrate on tree-shaped platforms since they represent a natural framework for master slave tasking. More importantly, administrative organizations often rely on tree-shaped networks to interconnect computing resources [14]. Initially, the root of the tree (master node) holds a large bunch of tasks. Those tasks will be either processed by the master node or transmitted to its child nodes (also called slave nodes). Then, in turn, the child nodes face the same allocation problem (either processing the tasks locally or forwarding them to their child nodes). We consider the case where slave processors need to send back a file of results after processing each task. Even if this is the most natural situation, it is worth noting that most of the papers on independent tasks scheduling or Divisible Load Theory (DLT) do not consider those return communications. Targeted platforms are fully heterogeneous, i.e. both the processing resources and the communication resources have different capacities in terms of processing power and bandwidth. Moreover, the network links present bandwidth asymmetry in the sense that the bandwidth for sending tasks down the tree may be different from the bandwidth for returning results up the tree.

We concentrate on the influence of dynamic resource characteristics on the allocation scheme. In shared and unstable environments such as grids and peer to peer systems, the performance of the resources may well change during the execution of the whole process. In this context, it is not realistic to assume that one of the nodes knows at any time step the exact performance of all resources and is able to make optimal scheduling decisions [14]. Therefore, the main question consists in determining whether the allocation scheme can make use of some static knowledge about the platform (for instance, the optimal solution computed from an initial snapshot of the platform), or whether we need to rely on fully dynamic scheduling schemes. In order to answer this question, we first derive optimal scheduling algorithms (with respect to throughput maximization). Then we present several heuristics. Some of them make their scheduling decisions using the optimal scheduling policy, computed using a snapshot of resource performance characteristics. Those heuristics may lead to optimal scheduling decisions in static environments. On the other hand, we propose a set of fully dynamic allocation heuristics that make their scheduling decisions only according to information measurable locally. Those heuristics may give poor results in static environments, but their performances are expected to be more robust in dynamic environments. We compare all those heuristics through extensive simulations using the SimGrid toolkit [9]. We rely on simulations rather than direct experiments in order to make a fair comparison between proposed heuristics. Indeed, simulation enables running of the different tests on computing platforms having exactly the same dynamic behavior. Moreover, SimGrid enables to define the trace of performance data over time for each processing or communication resource. Therefore, it is possible to compute (offline) the optimal solution at any given time step and it is therefore possible to compare the performances of the different heuristics between them and against the optimal ideal solution.

The rest of the paper is organized as follows. Section 3.2 is devoted to a survey of related work, both DLT studies, independent tasks scheduling and on dynamic scheduling. Then, we present our platform model in Section 3.3 and how to find the optimal solution, in presence of return messages, in Section 3.4. Section 3.5 states the main Theorem of this paper, which provides a mean to optimize the nodes bandwidth utilization. Section 3.6 presents a task-flow control mechanism that regulates the amount of tasks and results buffered by the nodes throughout the execution. The set of centralized and distributed heuristics are described in Section 3.7. The methodology and results of the simulations are discussed in Section 3.8. Finally, we give some remarks and conclusions in Section 3.9. Due to space limitation, many of the technical details have been omitted, but can be found in the extended version of this paper [3].

3.2 Related work

The problem of master-slave tasking on heterogeneous tree platforms has already been widely studied, both in the context of Divisible Load Theory (DLT) and independent tasks scheduling. A divisible load is a perfect parallel task that can be arbitrarily split and allocated to slave processors, without processing overhead. The overall load is first split at the master node in order to minimize the total execution time. Tasks are distributed in one round to the slaves, so that the master node makes the decisions about the set of slaves to be used, the amount of data to be sent to each slave, and the communication ordering [7, 10, 16]. When return messages are taken into account, two permutations must then be determined (one for tasks distribution and one for results collection) [4, 8]. Although the complexity of this problem is still open, Rosenberg et al. [1] proved that in the case of a homogeneous single-level tree, the optimal schedule for both outgoing and incoming messages can be determined, and the optimal LIFO and FIFO orderings are given in [5] for heterogeneous single-level trees.

On the other hand, when considering independent tasks scheduling, the master node faces the allocation problem for each task and the communications with its child nodes may well be split into several rounds [11, 15, 17]. Recently research studies have focused on steady-state scheduling, i.e. throughput maximization [2, 13, 15]. The steady-state scheduling approach has been pioneered by Bertsimas and Gamarnik [6] who considered packet routing and proposed to concentrate first on resource occupation rather than scheduling. The optimal solution for resource occupation, given link capacities, is obtained via a linear program. Then, an algorithm based on super-steps is proposed for building the actual schedule of packets. This idea has been adapted in [2] to the distribution of independent tasks on static platforms. Results collection was not considered in [2], but the linear program presented in Section 3.4 is a direct adaptation of the solution proposed in [2].

Dynamic scheduling of independent tasks has not been widely studied. Recently, Hong et al. [12–14] proposed a very nice algorithm, based on decentralized versions of flow algorithms. It is worth noting that this algorithm assumes a strongly different communication model than the one presented in this paper, and consequently cannot be easily adapted to our model. Here again, the results collection has not been considered.

3.3 Platform model

The model considered in this paper is based on the model proposed in [2] that we augment by introducing communication weights for returning computation results back to the master. Processing nodes are assumed to be connected via a node-weighted edge-weighted tree $T = (V, E, w, c, c')$ as depicted in Figure 3.1.

Each node $P_i \in V$ represents a computing resource of weight w_i, meaning that node P_i requires w_i units of time to process one task. Each edge corresponds to a communicating resource and is weighted by two values: c_i which represents the time needed by a parent node to send one task to its child P_i, and c'_i which represents the time needed by the child P_i to send one result back to its parent. All the w_i's are assumed to be positive rational numbers since they represent node processing times. We disallow $w_i = 0$ since it would permit node P_i to perform an infinite number of tasks. Similarly, we assume that all c_i's and c'_i's are positive rational numbers since they correspond to the communication

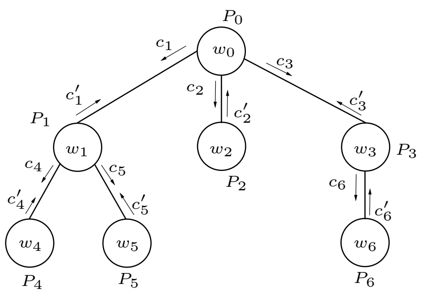


Figure 3.1: Tree-shape platform.

times between two processors. A node can perform three kinds of activity simultaneously: (i) it can process a task, (ii) it can receive a task file from its parent or a result file from one of its children, and (iii) it can send a result file to its parent or a task file to one of its children. This model is known under the name *full overlap, bidirectional-single-port* model [2, 15]. At any given time-step, a node may overlap computation with only two connections, one for incoming communications and one for outgoing communications. Computation and communication are assumed to be atomic operations, i.e. once initiated they cannot be preempted. Finally the communication model works in a store-and-forward fashion.

3.4 Maximizing the throughput

Given the resources of a weighted tree T operating under the *full overlap, bidirectional-single-port* model, we aim at maximizing the number of tasks processed per time unit. Let $\mathcal{C}_i$ denote the set of P_i's children. During one time unit, let α_i be the fractional number of tasks processed by P_i, and β_i be the fractional number of tasks received by P_i from its parent. Equivalently, α_i and β_i correspond respectively to the fractional number of results produced by P_i, and to the fractional number of results sent by P_i to its parent. The optimal throughput is obtained by solving the following linear programming problem (LPP), whose objective function is to maximize the number of tasks processed per time unit.

Maximize $n_{task}(T) = \sum_i \alpha_i$
subject to

$$\begin{cases} \forall i, & 0 \leq \alpha_i \leq \frac{1}{w_i} \\ \forall i \neq m, & 0 \leq \beta_i \\ \forall i \neq m, & \beta_i = \alpha_i + \sum_{j \in \mathcal{C}_i} \beta_j \\ \forall i, & \sum_{j \in \mathcal{C}_i} c_j \beta_j + c_i' \beta_i \leq 1 \\ \forall i, & \sum_{j \in \mathcal{C}_i} c_j' \beta_j + c_i \beta_i \leq 1 \end{cases}$$

The first set of constraints states that computation resources are limited. The second set of constraints confines the variables β_i within non-negative values. Note that the master P_m does not have a parent, so that we let $\beta_m = 0$. The third set of constraints deals with *conservation laws*. For each node P_i (except the master), the number of tasks received by P_i, should be equal to the number of tasks that P_i processes locally, plus the number of tasks forwarded to the children of P_i. Equivalently, the number of results sent by P_i to its parent, should be equal to the number of results produced locally by P_i, plus the number of results received from its children. The last constraints account for the single-port model. The send and receive operations performed by the nodes are assumed to be sequential.

Since we are looking for a solution of the LPP into rational numbers, optimal rational values for all variables can be obtained in polynomial time. However, the solution of the above LPP is in general not unique and some solutions might be more interesting than others in our context. In particular, *compact* solutions, i.e. that utilize nodes close to the root in priority, are more preferable than *stretched* solutions (that utilize nodes far away from the root). Indeed, start-up time (required to enter the steady-state) and wind-down time (required to gather the last results to the root) will be longer for stretched solutions than for compact ones. In order to obtain compact solutions, we first need to solve the initial LPP to derive the optimal throughput $n_{task}(T)$ of the tree. The objective function of the second LPP becomes the minimization of all the communications, under the aforementioned constraints plus an additional one that states the conservation of the optimal throughput obtained by the former LPP. Minimizing the amount of communications while maintaining the optimal number of tasks processed implicitly enforces compact solutions. We hence add the following constraint: $\sum \alpha_i = n_{task}(T)$. And the objective function of the second LPP becomes: **Minimize** $\sum_i \beta_i$. Once a solution has been obtained, one needs to construct a schedule that (i) ensures that the optimal throughput is achieved and (ii) exhibits a correct orchestration of communication events, i.e. where simultaneous communications involve disjoint pairs of senders and receivers. We can obtain a time period Γ by taking the least common multiple (lcm) of all the denominators of the variables α_i. Then, the integer number of tasks γ_i that must be communicated to P_i during each time period Γ is obtained by $\gamma_i = \beta_i \Gamma$.

Proposition 3.1. *Sending and receiving files by bunches of γ_i in a round robin fashion generates an optimal steady-state schedule where single-port constraints are satisfied.*

Proof. The proof is done by induction over h, the height of the tree T [3]. □

Initially, nodes do not dispose of tasks nor results buffered locally to comply with Proposition 3.1. Therefore an initialization phase must take place before entering steady-state. During start-up, nodes will act as if they were in steady-state, at the difference that fake results will be sent to the parents if not enough results are available. Thus, tasks will be propagated down the tree, while fake results will be propagated up the tree. The fake results received by parents nodes are simply discarded. Once the first bunch of results processed by all the deepest nodes used in the schedule have been transmitted to the root node, then steady-state has been reached.

3.5 Bandwidth optimization

A simple scheduling principle is presented in [2] when returning results is neglected. This scheduling algorithm was termed *bandwidth-centric* because priorities do not depend on the children processing capabilities, but only on their communication capabilities. The bandwidth-centric principle is extended to our problem as follows. First, observe that for each task that a node P_i delegates to a child P_j, P_i must first receive the task from its parent, then forward it to P_j, receive the associated result back, and finally send the result to its parent. Consequently, P_i will spend $x_j = c_j + c'_i$ time units sending data, and $y_j = c'_j + c_i$ time units receiving data. Since the master P_m does not have a parent, we let $x_m = c_m$ and $y_m = c'_m$. The bandwidth utilization of a node P_i can be sketched within the Cartesian plane, where the X and Y axes represent the time spent in emission and reception respectively. Hence, allocating a task to child P_j corresponds to a displacement in the Cartesian plane along vector $\vec{v_j}$ of components (x_j, y_j).

Theorem 3.1. *In steady-state, the bandwidth utilization of a parent node is optimized when using at most 2 children (if processing capabilities are not taken into account).*

Proof. The proof is done by induction over n, the number of children that are utilized by a parent in addition to the two nodes mentioned in Theorem 3.1. Consider the case where $n = 1$, i.e. when a parent delegates α_1, α_2 and α_3 tasks per time unit to three children P_1, P_2 and P_3 respectively (see Figure 3.2). Displacements OA_1, A_1A_2 and A_2A_3 stand for delegating α_1, α_2 and α_3 tasks to the children P_1, P_2 and P_3 respectively.

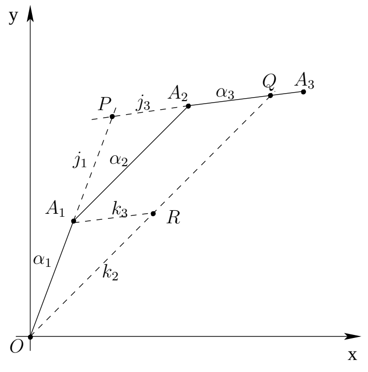


Figure 3.2:

Consider the triangle A_1A_2P where the displacements A_1P and PA_2 amount to allocate j_1 and j_3 tasks to P_1 and P_3 respectively. Consider now both quantities (j_1+j_3) and α_2. If $(j_1+j_3) \geq \alpha_2$, it means that it is more profitable to spend the bandwidth time assigned to P_2 by allocating more tasks to P_1 and P_3. As a consequence, P_2 should not be used. But if $(j_1 + j_3) < \alpha_2$, then consider the triangle ORA_1, where the displacements OR and A_1R amount to allocate k_2 and k_3 tasks to P_2 and P_3 respectively. Since both triangles A_1A_2P and ORA_1 are equal (since their internal angles are equal), if $(j_1 + j_3) < \alpha_2$ then $(\alpha_1 + k_3) < k_2$. In that case, it becomes more profitable to assign k_2 tasks to P_2 instead of α_1 tasks to P_1 and j_3 tasks to P_3, and P_1 should not be used. Assume now that Theorem 3.1 is true for rank n, and let us prove that it holds also for rank $n+1$. Consider a parent utilizing $n+3$ children. Extract 3 of the $n+3$ children and apply the aforementioned geometric transformation. One then utilizes only $n+2$ children without degrading the initial throughput. □

Theorem 3.1 assumes that nodes can provide as much computing power as necessary which contravenes the fact that computing resources are limited. Nonetheless, it allows identifying the way to optimize the bandwidth of any node P_i in using at most two children. Furthermore, we show in [3] that if such a pair of children exists, then the emission and reception bandwidth of P_i are equally utilized.

3.6 Task-flow control

In order to regulate the number of tasks and results that nodes are allowed to buffer locally throughout the execution, a threshold value θ_i is introduced for each node $P_i, i \neq m$. On the one hand, if the number of tasks buffered locally by P_i is beneath the threshold, then P_i will request more tasks in order to prevent starvation. On the other hand, if the number of results buffered locally by P_i is larger than the threshold, then P_i will not request additional tasks in order to hinder a monotonic accumulation of results. Initially, $\theta_i = 1, \forall i \neq m$. Since we search for compact solutions, parent nodes will try to process as many tasks as possible. If additional tasks arrive while a node is busy processing, then the task will be forwarded down the tree. During the execution, nodes are allowed to increase their local thresholds θ_i only when (i) they are starving and (ii) if they recently succeeded to accumulate θ_i tasks locally (to ensure that the current threshold is not sufficient) and (iii) if the number of results buffered locally is strictly lower than θ_i. This mechanism allows nodes to collect enough tasks locally to feed their sub-trees, while ensuring that results do not accumulate monotonically locally. On the other hand, nodes must decrease their local thresholds whenever the number of results buffered locally exceeds the threshold. This threshold growth mechanism provides a mean to adapt to the platform dynamics.

3.7 Scheduling heuristics

Round Robin (RR). This heuristic implements Proposition 3.1. Once all the α_i are known, the period Γ is estimated as follows. Let us set $x = \lfloor \log_{10}(\max_i \alpha_i) \rfloor$. If $x \leq 0$ then $\Gamma = 10^{|x|+1}$, $\Gamma = 10^x$ otherwise. The aim is to obtain a compromise between a short time period, and an approximation close to the optimal solution. Then we get the number of tasks computed by each node P_i by rounding $\Gamma \alpha_i$ to the nearest integer.

On the Fly (OTF). This heuristic makes use of the centralized knowledge. Once all the β_i's are known, each node maintains a table $tasks\_given[j]$, which records the number of tasks delegated to child P_j so far. The child node that has the lowest $\frac{tasks\_given[j]}{\beta_j}$ ratio is served in priority.

FIFO. Tasks are delegated in a first-come first-served basis.

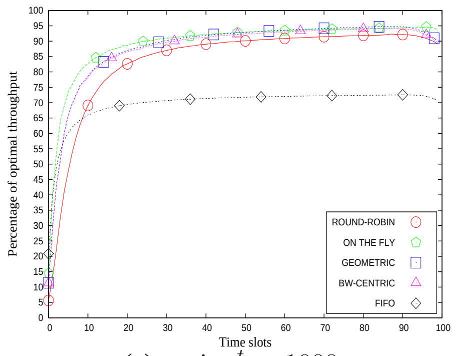
Bandwidth-Centric (BC). Let $r_j = \min\{\frac{1}{x_j}, \frac{1}{y_j}\}$ denote the maximum amount of tasks that P_i can delegate to child P_j per time unit. The child which has the highest r_j is served in priority.

Geometric (Geo). This heuristic makes use of Theorem 3.1, but starts by applying the bandwidth-centric heuristic, in order to determine which child obtains the highest r_j. Then, it inspects if a pair of children can improve that rate. If such a pair of children exists, one must decide which child should be served. In order to make the right decision, we use a variable Δ which works much like a pair of scales. At start, $\Delta = 0$. Each time a child node P_j is served, we put x_j in one scale, and y_j in the other, which amounts to $\Delta = \Delta + x_j - y_j$. When a pair of children nodes is elected, then the child which brings Δ closest to 0 is serve. The aim is to utilize equally the emission and reception bandwidths of the parent nodes. Such strategy will optimize the bandwidth utilization of the nodes, while naturally adapting to the platform dynamics.

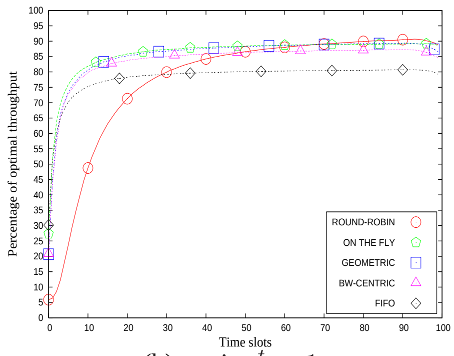
3.8 Simulations results

To evaluate our heuristics, we simulate the execution of an application on different random trees. Since a sub-tree can be reduced to a single super-node of equivalent processing power [2], it is not necessary to employ thousands of nodes to simulate large-scale systems [13]. We arbitrarily limited the number of nodes in a tree to 100. Each node was arbitrarily restricted to have at most 10 child nodes. A random tree is generated as follows. Each node is numbered with an ID number between 0 and 99. Then, each node $P_i, i \in [1, 99]$ is connected randomly to a node $P_j, j \in [0, i-1]$. The links have static performance values comprised between c_{min} and c_{max} and the nodes between w_{min} and w_{max}. All random distributions are uniform. The dynamic environments used in our simulations were generated as follows. Each resource R_i (node or link) has a cyclic behavior, i.e. its performance changes n_i times per cycle. The number of changes n_i per cycle is randomly taken within the interval $[5, 15]$. Resource performance changes will occur every 25 treated tasks in average. We do not claim that these arbitrary decisions correspond to realistic network conditions. Our aim is to compare our heuristics on a set of different tree configurations. Inspired by Kreaseck et al. [15], we determine the throughput rate by using a growing window. The execution time is divided into 100 equal-sized time slots. Then, the window increases in size by step of one time slot, and the throughput rate delivered within the window time-frame is computed. The throughput rates delivered by the trees have been normalized to the maximum steady-state rates obtained with the LPP in static environments. However, throughput rates obtained in dynamic environments have been scaled up by a *dynamic factor* that accounts for the performance loss incurred by the platform dynamics. The dynamic factors have been obtained by successively solving LPPs of static platforms and comparing them to their homologous LPPs where some dynamism have been introduced (i.e. with the same platform topologies but with scaling down resource performances). More details about our methodology as well as a broader set of simulation can be found in [3].

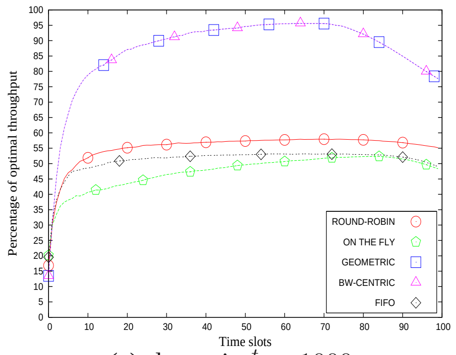
In this paper, we report the simulation of an independent-task application of 2500 tasks on 50 trees where $c_{min} = 1$, $c_{max} = 10$, $w_{min} = 20$ and $w_{max} = 200$. Two scenarios for the data volume associated to the tasks and results were considered: (i) task data are 1000 times larger than result ones ($\frac{t}{r} = 1000$), and (ii) task and result data have the same size ($\frac{t}{r} = 1$). Figure 3.3 plots an average of the 50 throughput rates (associated to the 50 trees) over time. Figure 3.3 (a) and (b) correspond to static environments, while Figure 3.3 (c) and (d) correspond to fully dynamic environments, i.e. where resource performances can degrade down to 1% of the static value. The RR heuristic has



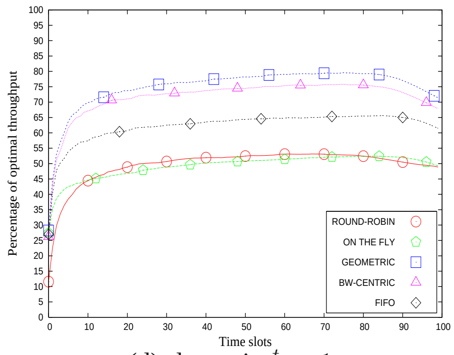
(a) static, $\frac{t}{r} = 1000$



(b) static, $\frac{t}{r} = 1$



(c) dynamic $\frac{t}{r} = 1000$



(d) dynamic $\frac{t}{r} = 1$

Figure 3.3: Average of the 50 throughput rates (associated to the 50 trees) over time, with the computation to communication ratio $\frac{w_i}{c_i} = 20$. In the dynamic environments, resource performances can degrade arbitrarily without failing, i.e. down to 1% of the static performance value.

been simulated with more than 2500 tasks in order to overcome the long start-up time required to enter steady-state. Still, RR does not outperform the other heuristics in static environments, certainly due to the truncating and rounding operations that occurred when computing Γ and the γ_i's. Not only the integer number of tasks intended to each node may be sub-optimal, but also the schedule of communications gets disturbed. The centralized heuristics (RR and OTF) are the highest performers in static environments, but the lowest ones in dynamic environments. Indeed, the information on which they rely throughout the execution becomes misleading in dynamic settings. As expected, the BC heuristic works very well when result data are small, while Geo only departs from BC when result data become significant.

Interestingly, when result data become significant, the performance of the best heuristics decrease, whereas the performance of FIFO increases. On the one hand, the decline of the best heuristics can be explained by the scheduling problem becoming more complicated. Returning results up the tree taking as long as sending tasks down the tree, parent nodes may sometimes have to stall a long time, waiting for a child to become available in reception. On the other hand, the performance increase of FIFO is a direct consequence of the task-flow control mechanism. When returning results takes a long time, local accumulations of results will arise, hindering the ineffective nodes to request for additional tasks. In contrast, when returning results is quick, no local results accumulations take place, increasing the margin to make wrong scheduling decisions.

Finally, it is worth noticing that BC and Geo compete well with the centralized heuristics even in static environments. See [3] for further details and interpretations.

3.9 Conclusion and future work

The problem of distributing a large number of independent tasks onto heterogeneous tree-shaped platforms with bandwidth asymmetry was considered. In contrast with most previous studies, the cost of returning results to the master node was represented in the problem formulation. We provided theoretical results that were embedded into autonomous heuristics. Simulations results showed that the autonomous heuristics put together with the task-flow control mechanism not only behaved very well in dynamic environments, but also compete well with centralized heuristics in static environments.

The scope of this paper was restricted to tree-shaped networks. However, at the backbone level, various geographically organizations are connected via the Internet resulting in a graph topology. Adapting the theoretical results presented in this paper to graph-shape platforms is a natural continuation of this work, albeit graph topology introduces routing problems. Another direction is to consider master-slave tasking in the presence of multiple masters. This situation arises naturally when several applications share the same platform, or when multiple masters collaborate on a single application.

Bibliography

[1] M. Adler, Y. Gong, and A. L. Rosenberg. Optimal Sharing of Bags of Tasks in Heterogeneous Clusters. In *15th ACM Symp. on Parallelism in Algorithms and Architectures (SPAA'03)*, pages 1–10. ACM Press, 2003.

[2] C. Banino, O. Beaumont, L. Carter, J. Ferrante, A. Legrand, and Y. Robert. Scheduling Strategies for Master-Slave Tasking on Heterogeneous Processor Platforms. *IEEE Transactions on Parallel and Distributed Systems*, 15(4):319–330, 2004.

[3] C. Banino-Rokkones, O. Beaumont, and L. Natvig. Master-Slave Tasking on Asymmetric Tree-Shaped Networks. Technical Report 02/06, Dept. of Computer and Info. Science, Norwegian University of Science and Technology, September 2006. URL: http://www.idi.ntnu.no/∼banino/research/research.html.

[4] G. D. Barlas. Collection-Aware Optimum Sequencing of Operations and Closed-Form Solutions for the Distribution of a Divisible Load on Arbitrary Processor Trees. *IEEE Trans. Parallel Distrib. Syst.*, 9(5):429–441, 1998.

[5] O. Beaumont, L. Marchal, and Y. Robert. Scheduling Divisible Loads with Return Messages on Heterogeneous Master-Worker Platforms. In *International Conference on High Performance Computing HiPC'2005*, LNCS, pages 123–132. Springer Verlag, 2005.

[6] D. Bertsimas and D. Gamarnik. Asymptotically optimal algorithm for job shop scheduling and packet routing. *Journal of Algorithms*, 33(2):296–318, 1999.

[7] V. Bharadwaj, D. Ghose, V. Mani, and T. Robertazzi. *Scheduling Divisible Loads in Parallel and Distributed Systems*. IEEE Computer Society Press, 1996.

[8] J. Blazewicz, M. Drozdowski, F. Guinand, and D. Trystram. Scheduling a Divisible Task in a Two-dimensional Toroidal Mesh. In *Proceedings of the third international conference on Graphs and optimization*, pages 35–50, Amsterdam, The Netherlands, 1999. Elsevier Science Publishers B. V.

[9] H. Casanova. SimGrid: A Toolkit for the Simulation of Application Scheduling. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, page 430. IEEE Computer Society, 2001.

[10] M. Drozdowski and P. Wolniewicz. Experiments with scheduling divisible tasks in clusters of workstations. In *Proceedings of Euro-Par 2000: Parallel Processing*, LNCS 1900, pages 311–319. Springer, 2000.

[11] P. F. Dutot. Complexity of Master-slave Tasking on Heterogeneous Trees. *European Journal on Operationnal Research*, 164(3):690–695, August 2005.

[12] B. Hong and V. K. Prasanna. Bandwidth-Aware Resource Allocation for Heterogeneous Computing Systems to Maximize Throughput. In *ICPP*, pages 539–546. IEEE Computer Society, 2003.

[13] B. Hong and V. K. Prasanna. Distributed Adaptive Task Allocation in Heterogeneous Computing Environments to Maximize Throughput. In *International Parallel and Distributed Processing Symposium IPDPS'2004*, page 52b. IEEE Computer Society Press, 2004.

[14] B. Hong and V. K. Prasanna. Performance Optimization of a De-centralized Task Allocation Protocol via Bandwidth and Buffer Management. In *CLADE*, page 108, 2004.

[15] B. Kreaseck, L. Carter, H. Casanova, and J. Ferrante. Autonomous Protocols for Bandwidth-Centric Scheduling of Independent-Task Applications. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 26.1, Washington, DC, USA, 2003. IEEE Computer Society.

[16] T. Robertazzi. Processor Equivalence for a Linear Daisy Chain of Load Sharing Processors. *IEEE Trans. Aerospace and Electronic Systems*, 29:1216–1221, 1993.

[17] A. L. Rosenberg. Sharing Partitionable Workloads in Heterogeneous NOWs: Greedier Is Not Better. In *CLUSTER '01: Proceedings of the 3rd IEEE International Conference on Cluster Computing*, page 124, Washington, DC, USA, 2001. IEEE Computer Society.

Paper 4

Online Task Scheduling on Heterogeneous Clusters:
An Experimental Study
Einar M.R. Rosenvinge, Anne C. Elster, Cyril Banino.
In *Proceedings of PARA 2004: 7th International Conference on Applied Parallel Computing, LNCS 3732/2006,*
June 20-23 2004, Pages 1141-1150, Lyngby, Denmark.

Online Task Scheduling on Heterogeneous Clusters: An Experimental Study

Einar M.R. Rosenvinge, Anne C. Elster, Cyril Banino

Department of Computer and Information Science
Norwegian University of Science and Technology
NO-7491 Trondheim, Norway

Abstract

This paper considers the problem of scheduling efficiently applications composed of a large number of independent tasks on heterogeneous clusters. The Master/Worker paradigm is used, where tasks are maintained by a master node which hands out batches of a variable amount of tasks to requesting worker nodes. The **Monitor** strategy is introduced and compared to other strategies suggested in the literature. Our online strategy is especially suitable for heterogeneous clusters with dynamic loads.

4.1 Introduction

In today's international high-performance computing arena, there is a clear trend from traditional supercomputers towards cluster and Grid computing solutions. This is mainly motivated by the fact that clusters typically can be constructed at a cost that is modest compared to the cost for traditional supercomputers that have equivalent computing power. The operation, use and performance characteristics of such clusters are, however, significantly different from those of traditional supercomputers. For instance, clusters typically have a much slower communication medium between nodes (e.g. a high-latency, low-bandwidth interface such as Ethernet or the faster Myrinet.)

Clusters give rise to some challenges not typically found on traditional supercomputers. A cluster may be a *heterogeneous* environment, meaning that its nodes may have different performance characteristics. Also, if the nodes composing a cluster are not dedicated, the cluster will be a *dynamic* environment, because its nodes may have a non-negligible background processing load. These challenges imply that one needs an adequate scheduling strategy to get good performance on a cluster.

Our work aims at developing effective scheduling strategies for clusters for the class of applications that fall into the Master/Worker paradigm. These applications can be divided into a large number of independent *work units*, or *tasks*. There is no inter-task communication, so the tasks can be computed in any order. Finally, the tasks are atomic, i.e. their computation cannot be preempted. Many applications can be parallelized in such a way, including matrix multiplication, Gaussian elimination, image processing applications such as ray-tracing [6] and Monte Carlo simulations [1].

The rest of this paper is organized as follows: The test-case application and the test-bed cluster are described in Section 4.2. In Section 4.3, previous scheduling strategies, as well as our scheduling strategy, are presented. Section 4.4 exposes implementation-specific issues, and Section 4.5 discusses empirical results from our work. Finally, conclusions and suggestions for future work are provided in Section 4.6.

4.2 Framework

4.2.1 Test-case application: matched filtering

The test-case application used in this study is an image filtering application, known as *matched filtering* [3]. This application has been developed by O. C. Eidheim, a PhD student at IDI/NTNU, who was looking for speed improvements, giving us great access to a developer of the application.

The matched filtering application is used in medical imaging in order to detect blood vessels in Computer Tomography (CT) images. A CT image is a cross-section of the human body. By using this application to detect blood vessels in multiple adjacent CT images, one is able to construct a 3D representation of the blood vessels in the human body.

The input to the application is a grayscale image, see Fig. 4.1 (a), which is filtered through an image correlation step. The correlation kernel that is used is a Gaussian hill, which is rotated in all directions and scaled to several sizes. For more detailed information about this filtering technique, see [3]. The noise in the input image, makes the blood vessel identification quite challenging. After filtering, see Fig. 4.1 (b), the noise has been removed and blood vessels are now identifiable.

Since the input image can be divided into tasks corresponding to different parts (lines, columns, blocks), each node can process one or more tasks, and thus produce the corresponding parts of the output image, this application parallelizes easily in a homogeneous and static environment. However, in a heterogeneous and/or dynamic environment provided by most of today's clusters, parallelizing this application efficiently is more complicated.

4.2.2 Test-bed platform: athlon-based cluster

ClustIS is a fairly homogeneous cluster composed of 38 nodes, with AMD Athlon XP/MP CPUs at clock frequencies of 1.4 to 1.66 GHz with 0.5 to 2 GB of RAM. A few of the nodes are dual-CPU nodes. The nodes are connected through 100Mbit switched Ethernet. The operating system is Linux, the MPI implementation is MPICH 1.2.5.2, and the queuing system is OpenPBS.

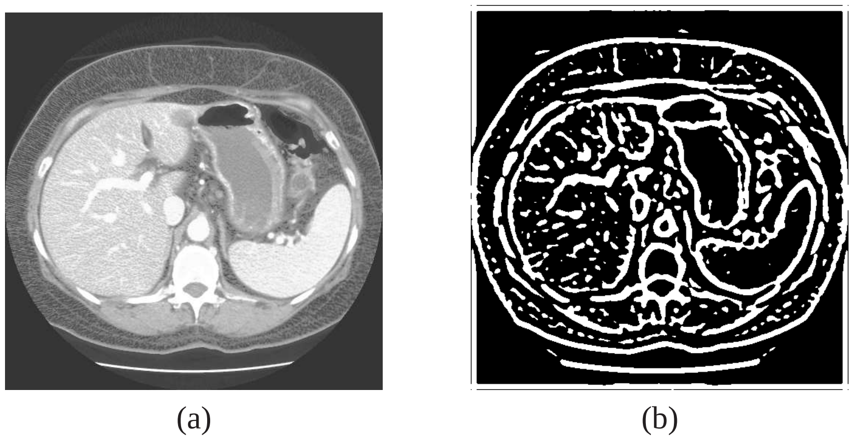


Figure 4.1: Image before filtering (a), and after filtering (b). CT image courtesy of Interventional Center, Rikshospitalet, Oslo, Norway.

On ClustIS, data storage is provided by one node, *Story*, which provides home directories through NFS. Consequently, all disk I/O from the nodes will go through this slow Ethernet interface. One solution could be to use local disk I/O instead. However, the scattering of input data and gathering of output data would add to the total application execution time, so regardless, input and output data would have to travel through the network.

Nevertheless, we were able to demonstrate some I/O parallelism on this cluster. In fact, we got more or less linear speedup when reading data concurrently from up to 8 processes. This indicates that having the worker nodes read their part of the data themselves will be faster than having the master scatter and gather data to/from workers. Writing data in parallel also gave a significant speedup compared to centralized writing, but the speedup was not quite as linear. See [9] for details.

4.3 Scheduling master/worker applications

On a cluster, each processor might have very different performance characteristics (heterogeneity), as well as varying background workloads (dynamism). To a certain degree, heterogeneity can be handled through the job scheduler, by requesting processors with a certain CPU frequency. Such functionality, however, is not available with many job scheduling systems.

Dynamism, however, cannot be handled through a job scheduler. The background processing load of the processors is unknown before the computation starts, and might vary throughout the computation. This must therefore be handled by the scheduling strategy used.

4.3.1 Previous scheduling strategies

All popular scheduling strategies give out *batches* of tasks to workers, but since the workers might have different and possibly varying processing speeds, giving only one batch to each worker might lead to non-equal finish times for the workers. To compensate for this, some strategies give batches to workers in several *rounds*. In the following, N denotes the total number of tasks, p denotes the number of workers (processors), and R denotes the number of remaining unassigned tasks on the master at a given time.

The *Static Chunking (SC)* strategy [6] assigns one batch of N/p tasks to each worker. At the other end of the spectrum is the *Self Scheduling (SS)* strategy [6], where tasks are handed out one by one. The *Fixed-Size Chunking (FSC)* strategy uses batches of tasks of one fixed size, and it is possible to approximate the optimal batch size [7]. The *Guided Self Scheduling (GSS)* strategy [8] gives each worker batches of size R/p. GSS thus uses exponentially decreasing batch sizes. The *Trapezoid Self-Scheduling (TSS)* strategy [10] also uses decreasing batch sizes, but the batch sizes decrease linearly from a first size f to a last size l. They advocate the use of $f = N/(2p)$ and $l = 1$. The *Factoring (Fac.)* and *Weighted Factoring (WF)* strategies also use decreasing batch sizes. At each time step, half of the remaining tasks are given out. The WF strategy works by assigning a weight to each processor corresponding to the computing speed of the processor before the computation starts, and allocates tasks based on these weights in every round [5, 6].

4.3.2 The monitor strategy

The Monitor strategy is fairly similar to Weighted Factoring (WF), which assigns tasks to workers in a weighted fashion for each round, where each worker has a static weight. For WF, this weight has to be computed in advance, before the actual computations start, which is a disadvantage in a dynamic environment. The Monitor strategy, however, performs such benchmarking *online* throughout the computation and thus uses dynamic weights, which also allows for good performance in a truly dynamic environment. The strategy uses an initialization phase and several batch computation phases.

During the initialization phase, workers request tasks from the master which are handed out one by one. Workers measure the time it takes to compute their respective task, and report these timings to the master when they request another task. When all workers have reported their task computation times, the initialization phase is done.

Formally, let x_i be the number of tasks that worker w_i will be given in the current batch computation phase, and y_i be the number of uncomputed tasks queued by worker w_i. Let t_i denote the time taken to process one task, and T_i denote the time taken for worker w_i to finish the current phase. Recall that R denotes the number of unassigned tasks held by the master, and p denotes the number of workers. In a batch computation phase, the master starts by solving the following system of equations:

$$\begin{cases} (1) \quad \forall i \in [0,p\rangle, \quad T_i = (y_i + x_i) \times t_i \\ (2) \quad \forall i \in [1,p\rangle, \quad T_i = T_{i-1} \\ (3) \qquad\qquad\qquad \sum\limits_{i=0}^{p-1} x_i = R/2 \end{cases}$$

In a given phase, worker w_i receive x_i tasks that are added to the y_i uncomputed tasks already stored in its local task queue. It will hence finish its execution of the current phase at time $T_i = (y_i + x_i) \times t_i$ (equation 1). For the total execution time to be minimized, all workers must finish their computations simultaneously, hence $\forall i \in [1,p\rangle, T_i = T_{i-1}$ (equation 2). This condition has been proved in the context of Divisible Load Theory [2]. The sum of all x_i is equal to $R/2$, meaning that $R/2$ tasks are allocated during the current phase (equation 3). It has been found experimentally [5] that handing out half of the remaining tasks in each round gives good performance.

Throughout the computation, workers periodically (i.e. every r computed tasks) report their task computation times t_i and the number of uncomputed tasks y_i waiting in their local task queues to the master. Hence the master is continuously monitoring the worker states. Consequently, after the first batch computation phase, there is no need for another initialization phase, since the master has up-to-date knowledge of the performance of the workers. Note that the parameter r must be tuned for the application and cluster in use. As soon as a worker is done with its local tasks, a request is sent to the master, which then enters the next computation phase. A new system of equations is solved with the last up-to-date values of T_i and y_i.

Throughout the computation, y_i has a great significance. Suppose that at phase k, worker w_i has no external load, and can thus supply a large amount of computing power to our application. The master will then delegate a large number of tasks to w_i. Suppose now that during phase k, w_i receives a large external load, slowing down its task execution rate. At the end of phase k worker w_i will still have a lot of uncomputed tasks. The master has up-to-date knowledge of this, and allocates only a few (or no) new tasks to w_i in phase $k+1$.

Note that if some workers are slowed down *drastically* the above system of equations may yield negative x_i values. Since the Monitor strategy does not consider withdrawing tasks from workers, the corresponding equations are removed, and the linear system is solved once more, distributing hence tasks among the remaining workers. This process is repeated until the solution yields no negative x_i values.

The task computation time t_i reported by worker w_i will typically be the mean value of its η last computation times. Having $\eta = 1$ might give a non-optimal allocation, since the timing can vary a lot in a dynamic environment. At the other end of the spectrum, a too high value for η conceals changes in processing speeds, which is also non-optimal. The parameter η needs to be adjusted for the individual application and/or environment.

Fig. 4.2 shows the allocated batch sizes for the scheduling strategies described in Section 4.3.1 as well as the Monitor strategy, when the processors report the task computation times shown in Fig. 4.3. Note that for the Monitor strategy, we assume $y_i = 0$ at the beginning of every phase, meaning that all the processors have computed all their assigned tasks from the previous phase.

Strategy	**Batch sizes**
SC	**128 128 128 128**
SS	**1 1 1 1** 1 1 1 1 **1 1 1 1** 1 1 1 1 **1 1 1 1** ...
GSS	**128 96 72 54** 40 30 23 17 **13 9 7 5** 4 3 2 2 **1 1 1 1** 1 1 1
TSS	**64 60 56 52** 48 44 40 36 **32 28 24 20** 8
Fac.	**64 64 64 64** 32 32 32 32 **16 16 16 16** 8 8 8 8 **4 4 4 4** 2 2 2 2 **1 1 1 1** 1 1 1 1
WF	**180 32 20 24** 90 16 10 12 **45 8 5 6** 22 4 3 3 **11 2 1 2** 6 1 0 1 **3 1 0 0** 1 1 0 0 **1 0 0 0** 1 0 0 0
Monitor	1 1 1 1 1 1 1 1 **177 32 20 23** 73 27 12 14 **11 23 13 16** 4 7 14 6 **6 3 3 4** 4 1 1 1 **0 2 1 1** 1 2 1 1

Figure 4.2: Batch sizes for various sched. strat. with $N = 512$ tasks and $p = 4$ workers.

Proc	**Task computation times at time steps**								
	1	2	3	4	5	6	7	8	9
1	0.10	0.15	1.01	0.90	0.28	0.29	0.99	0.90	0.89
2	0.56	0.40	0.50	0.48	0.52	0.53	0.47	0.49	0.50
3	0.89	0.90	0.89	0.24	0.67	0.88	0.60	0.66	0.63
4	0.75	0.76	0.74	0.50	0.45	0.70	0.69	0.63	0.62

Figure 4.3: Examples of task computation times for 4 processors at 9 time steps throughout a computation. Note that for WF, the times at step 1 are used as weights.

4.4 Implementation

4.4.1 Data staging

In order to improve I/O parallelism, the worker nodes read the necessary input data from the NFS disk themselves, much like the data staging technique presented in [4]. The master receives short requests from workers, and answer these requests with a short message containing only a pointer to the input data to be fetched, thus circumventing the master bottleneck.

However, because our test-case application has such a high computation to I/O ratio, our experiments showed that data staging did not have a great performance impact for this application [9]. Data staging is nevertheless a valuable technique for applications whose computation to I/O ratio is lower.

4.4.2 Multithreaded processes

In order to avoid processor idleness, we decided to implement a multithreaded approach. Every worker process is composed of three threads: a main thread for communicating with the master, a thread for reading input data from disk, and a thread for computing output data. The main thread requests tasks and buffers them in a task queue until the number of tasks buffered is above a user defined threshold ϕ. It then goes to sleep and wakes up when the number of tasks in the queue is below ϕ. The goal of the main thread is to keep ϕ tasks queued at all times. The input reader thread will fetch tasks from the task queue, and, if the queue is empty, sleep while waiting for a task. Once the task queue is non-empty, the input reader thread will read and store input data, then add pointers to the data locations in the input data queue. And this, until the number of input data units in the queue is above ϕ. It then goes to sleep, and wakes up when the number of input data units in the queue is below ϕ. The procedure is then repeated. The computer thread works in much the same way as the input reader thread. See [9] for details.

The threshold ϕ regulates how soon the workers will request tasks from the master. Intuitively, $\phi = 1$ might be non-optimal, since the task computer thread might become idle while the main thread is waiting for the master to allocate a task. Note that since each worker has two queues of size ϕ, it buffers 2ϕ tasks.

The master process is also multi-threaded, with one thread continuously probing for, receiving and sending messages to/from workers using MPI, one thread executing the scheduling strategy in use, and one worker thread computing tasks on the master processor. The MPI thread terminates as soon as all tasks have been allocated, but until that point, it consumes quite a lot of CPU time that could have been used by the worker thread. This means that for the worker thread on the master, a high ϕ value is optimal, since the workers will request tasks quickly and the MPI thread will terminate early. This is a side effect of our non-optimal master architecture, since the MPI thread consumes unnecessary CPU power. One possible optimization would be to merge the thread for communicating through MPI and the thread for executing the scheduling strategy, but this would lead to non-modular code. Another possible optimization would be to use two threads calling MPI, one for receiving and one for sending, but this is impossible with the non-thread-safe MPICH library we had at our disposal. For more on this side effect, see [9].

4.5 Empirical results and analysis

The implemented scheduling strategies were compared for different values of ϕ on our dedicated cluster which is a static environment [9]. Our goal was to find the best scheduling strategy combined with the optimal parameters ϕ, η and r for the test-case application running on the test-bed cluster. Note that for the Monitor strategy, we experimentally found $r = 4$ and $\eta = 20$ to be optimal values for our application and cluster [9], and these values have been used in the following experiments. The results from our experiments are shown in Fig. 4.4; for more experiments, see [9].

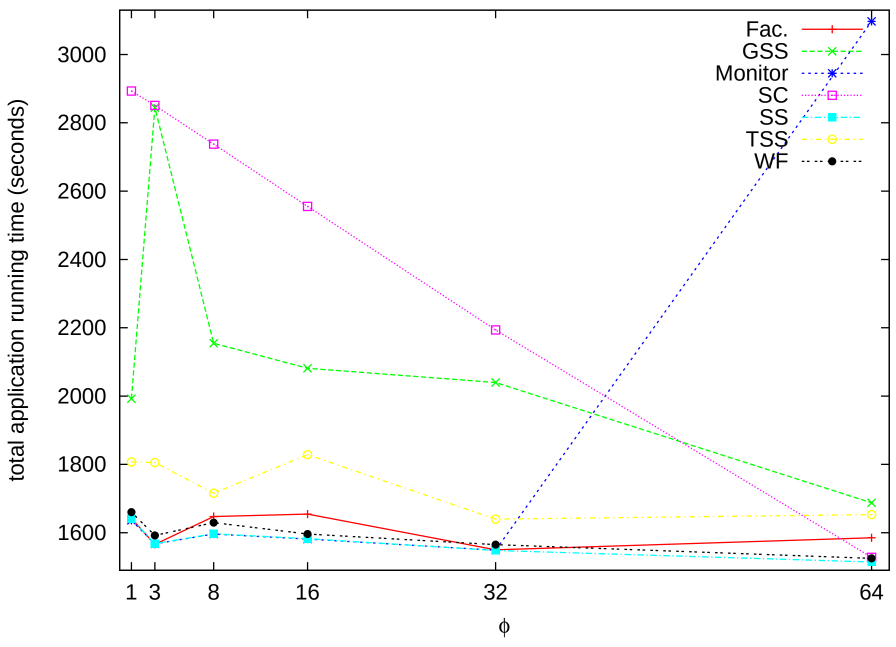


Figure 4.4: Comparison of sched. strategies with increasing ϕ values, static environment.

These experiments were conducted using 8 nodes, and an image of size 2048×2048 pixels decomposed into 1024 blocks, each block corresponding to a task. One interesting finding is that the Static Chunking strategy performs linearly better when using a larger ϕ value. When ϕ increases, the workers request new tasks earlier, hence causing the termination of the master MPI thread earlier. This frees resources for the worker thread on the master, and thus makes it process tasks faster. One might argue that the MPI thread on the master should have been terminated early regardless of ϕ since SC only uses one round of allocation, but in order to be fair, we kept the same master implementation for all the scheduling strategies. Consequently, all scheduling strategies must take into account the worker thread on the master which is very slow compared to the dedicated workers. Therefore, SC is a bad choice in a heterogeneous environment, while it is good in a homogeneous environment, as shown when $\phi = 64$.

The SS, Fac., WF and Monitor strategies are all quite similar. The reason why we get better results with $\phi = 32$ than with $\phi = 3$ is the same as for the SC case. The master MPI thread is stopped earlier, and we have one faster worker for the rest of the computation. With $\phi > 32$, the Monitor strategy performs very badly. One possible explanation for this is that during the initialization phase, the master computing thread is very slow, and will be given a small amount of tasks, less than ϕ. Consequently, the master computing thread will constantly request more tasks. As a result, the scheduling thread will solve a lot of unnecessary systems of equations further slowing down the computing thread.

Nevertheless, it should be noted that using very high ϕ values prevents good load balancing, since in order to allocate tasks when the workers need them (or slightly before, to avoid idle time), ϕ must be kept relatively low.

It is quite surprising that the Self-Scheduling strategy, which has the highest amount of communication of all strategies, is among the very fastest scheduling strategies. A possible explanation is that our multi-threaded implementation is able to hide the communication delays, and because our application has a high computation to I/O ratio. However, our environment is relatively homogeneous and static, and we expect the Monitor strategy to outperform SS in a strongly heterogeneous and dynamic environment.

Fig. 4.5 shows speedup results. Note that using e.g. 2 processors means using the master with its separate worker thread and 1 dedicated worker. With a 512$\times$512-pixel image, the speedup drops significantly when using more than 4 processors. This is due to using a suboptimal task size for this relatively small image size [9]. For the larger image sizes, the speedup increases when adding more processors. Intuitively, this comes from the fact that the relatively slow worker thread of the master processor plays a smaller role when adding more processors. With a sufficiently large image and 8 or more processors, we have a close to linear speedup.

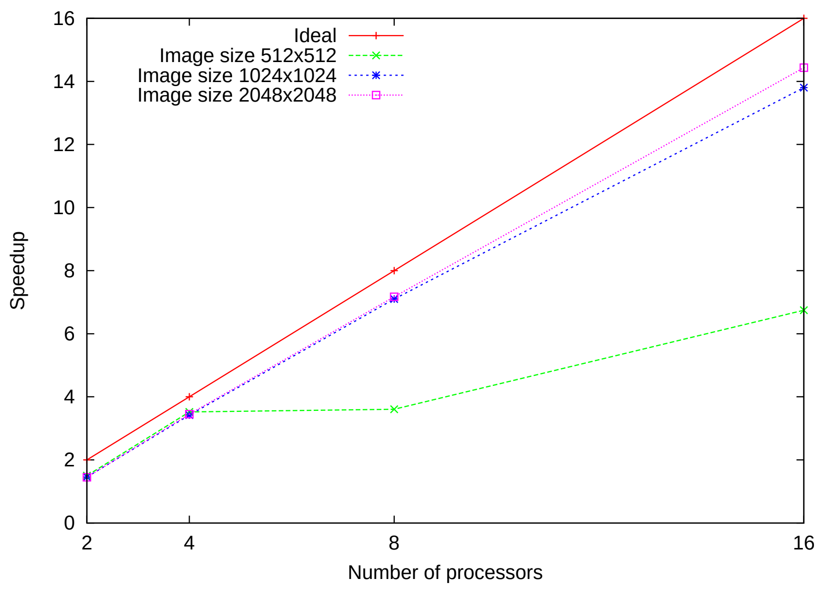


Figure 4.5: The speedup of the application.

4.6 Conclusions and future work

A novel online scheduling strategy has been designed and implemented. The Monitor strategy was experimentally compared to implementations of six other popular scheduling strategies found in the literature. Experiments show that the Monitor strategy performs excellently compared to these strategies, and should be especially well suited for dynamic environments.

The monitor strategy implementation involved multi-threading and data staging, two techniques that decrease processor idle time and increase master utilization. Our test-case application, the matched filtering algorithm, has a very high computation to I/O ratio, and consequently data staging is probably unnecessary for this application. Experimental tests on our cluster show, however, that data staging is a valuable technique for applications whose computation to I/O ratio is lower. The multi-threaded implementation of the worker processes, using task queuing mechanisms, is able to hide communication delays and keep the workers processing data continuously. The excellent performance of both the Self-Scheduling and the Monitor strategy substantiate this.

This work could be extended in the following directions. First, running more experiments on other clusters which provide more heterogeneity and more dynamism would enable to measure the potential of the Monitor strategy.

Second, the Monitor strategy has three user-specifiable parameters, r, η and ϕ. A way to determine the optimal values of these parameters online would be desirable. It might be that an optimal solution in heterogeneous environment necessitate different values of these parameters for different workers.

Then, the optimal task size used in this study has been found experimentally, but it would be desirable to determine it online. Two procedures for doing this are discussed in [9].

Finally, a thread-safe MPI library would enable us to implement the master process differently [9], which would increase performance.

Bibliography

[1] J. Basney, R. Raman, and M. Livny. High Throughput Monte Carlo. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, TX, March 1999.

[2] V. Bharadwaj, D. Ghose, V. Mani, and T. Robertazzi. *Scheduling Divisible Loads in Parallel and Distributed Systems*. IEEE Computer Society Press, Aug 1996.

[3] S. Chaudhuri, S. Chatterjee, N. Katz, M. Nelson, and M. Goldbaum. Detection of Blood Vessels in Retinal Images Using Two-Dimensional Matched Filters. *IEEE Transactions on Medical Imaging*, pages 263–269, 1989.

[4] W. Elwasif, J. S. Plank, and R. Wolski. Data Staging Effects in Wide Area Task Farming Applications. In *IEEE International Symposium on Cluster Computing and the Grid*, pages 122–129, Brisbane, Australia, May 2001.

[5] S. F. Hummel, J. Schmidt, R. N. Uma, and J. Wein. Load-Sharing in Heterogeneous Systems via Weighted Factoring. In *SPAA '96: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 318–328, New York, NY, USA, 1996. ACM Press.

[6] S. F. Hummel, E. Schonberg, and L. E. Flynn. Factoring: A Method for Scheduling Parallel Loops. *Commun. ACM*, 35(8):90–101, 1992.

[7] C. P. Kruskal and A. Weiss. Allocating Independent Subtasks on Parallel Processors. *IEEE Trans. Softw. Eng.*, 11(10):1001–1016, 1985.

[8] C. D. Polychronopoulos and D. J. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Trans. Comput.*, 36(12):1425–1439, 1987.

[9] E. Rosenvinge. Online Task Scheduling On Heterogeneous Clusters: An Experimental Study. Master's thesis, Dept. of Computer and Info. Science, Norwegian University of Science and Technology, 2004. URL:http://www.idi.ntnu.no/∼elster/students/ms-theses/rosenvinge-msthesis.pdf.

[10] T. H. Tzen and L. M. Ni. Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers. *IEEE Trans. Parallel Distrib. Syst.*, 4(1):87–98, 1993.

Paper 5

Parallelizing Lattice Gauge Theory Models on Commodity Clusters
Cyril Banino-Rokkones, Jørn Amundsen and Eivind Smørgrav.
In *Proceedings of IEEE International Conference on Cluster Computing*
September 25-28, 2006. Barcelona, Spain. To appear.

Parallelizing Lattice Gauge Theory Models on Commodity Clusters

Cyril Banino-Rokkones[1], Jørn Amundsen[1] and Eivind Smørgrav[2]

[1]Department of Computer and Information Science
[2]Department of Physics
Norwegian University of Science and Technology
NO-7491 Trondheim, Norway

Abstract

This paper addresses fundamental parallel computing issues for efficiently parallelizing 3D Lattice Gauge Theory models (LGT) on distributed memory systems. The long-range application stencil of LGT models put together with the impossibility of updating neighboring lattice sites simultaneously greatly complicates the parallelizing of such simulations. Our algorithms decompose the domain among the processors, and settle a staggered execution with the help of virtual tokens that circulate among the processors, allowing the token holders to update their boundaries. Experimental results show that these algorithms are scalable, and that simple communication trajectories prevail over low surface-to-volume ratios. Rigorous theoretical results are provided under the LogGP model to demonstrate the superiority of our approach over other methods found in the literature.

5.1 Introduction

Lattice spin and gauge theories are studied extensively in many areas of physics, especially in particle and condensed matter physics. The spin and gauge field variables are defined on every site of a multi- dimensional lattice, and the thermodynamic properties of the system can be deduced from the partition function, which is a sum over all possible configurations of the fields. Exact solutions to these multi- dimensional sums are rare and in general one must resort to some numerical approximation. The largest and most important class of numerical methods used for this problem is the Monte Carlo (MC) method, which in stead of doing the sum over all configurations, utilizes random numbers to mimic the random thermal fluctuations of the system from one configuration to the other. A considerable proportion of the computational resources used by physicists around the world is spent on MC simulations.

The excellent price-performance ratio of PC clusters, unmatched by any other high performance computing platform, promotes the emergence of distributed computing. The availability of inexpensive high performance microprocessors and high speed networks allows organizations to purchase a powerful cluster at reasonable price. Existing applications implemented under the shared-memory paradigm must be ported to distributed memory systems, requiring new distributed algorithms. The key for efficient parallelizations on distributed memory systems is to keep the communication overhead to a minimum. This is especially true for PC clusters, which are typically composed of powerful nodes interconnected by a low-cost communication network.

When using Metropolis MC dynamics, the *detailed balance condition* implies that adjacent sites cannot be updated simultaneously [8]. Therefore, when parallelizing such applications, one must ensure at all times that processors do not update remote but adjacent sites simultaneously. Various studies have focused on parallelizing MC simulations on distributed memory systems [7, 10, 11, 15]. Most of these studies rely on domain decomposition methods where all the processors are synchronized in order to respect the detailed balance condition. The main drawback of these methods resides in the communication phases requiring several message transfers between neighboring processors per iteration, albeit conventional wisdom argues that data should be grouped for communication [7, 14].

Past efforts to overlap communication with computation for regular domain problems converge towards an approach which divides the local domain into an inner region and an outer region [2, 13, 14]. The inner region is updated while waiting for the boundaries from neighboring processors, and thereafter the outer region is in turn updated. Although this technique aims at overlapping computation with communication, it also restricts the number of messages to the minimum, i.e. one per neighboring processor per iteration. This technique can suit well the parallelization of LGT models on distributed memory systems, because processors are working simultaneously on different parts of their domains which reduces the starvation imposed by the detailed balance condition. However, Prieto et al. [13] showed that this computing paradigm may degrade the performance, due to the large distance between the memory locations of the exterior sites, leading to poor performance when updating the outer region. This limitation has been incorporated in our theoretical models, and confirmed by our experiments.

The rest of this paper is organized as follows. Section 5.2 introduces the London superconductor model, the test-case application used in this study. The parallelizations of the different domain decompositions are exposed in Section 5.3. Theoretical results derived under the LogGP model are given in Section 5.4. Experimental results are presented and discussed in Section 5.6. The superiority of our approach over other methods found in the literature is exposed in Section 5.7. Finally concluding remarks are given in Section 5.8.

The experiments reported in this study have been performed on a SMP cluster, called Snowstorm located at the university of Tromsø (UIT). Snowstorm is a computational cluster composed of 100 HP Integrity rx4640 server nodes, each with 4 itanium2 processors clocked at 1.3 GHz, 4 GB of memory and 144 GB of internal disk and interconnected with the Infiniband network.

5.2 Test-case application

The 3-dimensional London superconductor model, see Equation (5.1), is a typical gauge theory in which a real valued scalar field θ is coupled to a real valued vector field $\mathbf{A}$.

$$\begin{aligned} Z &= \int \mathcal{D}\theta \int \mathcal{D}\mathbf{A} e^{-\frac{1}{T}E[\theta,\mathbf{A}]} \\ E[\theta,\mathbf{A}] &= -\sum_{i,\mu}\left[\cos\left(\Delta\theta_i - e\mathbf{A}_i\right)_\mu - \frac{1}{2}(\Delta\times\mathbf{A}_i)^2_\mu\right], \end{aligned} \tag{5.1}$$

where e is the charge of the system which couples the scalar field θ and the vector field $\mathbf{A}$, and Δ is the lattice difference operator. A thorough description of the London superconductor model can be found in [4]. The MC algorithm used in this work is the celebrated Metropolis algorithm [12] which can be described the following way.

1. Pick one site in the lattice and suggest new values for the fields at that site.
2. Calculate the difference in energy $\Delta E = E_{new} - E_{old}$ for the move, or update.
3. Draw a random number r$\in [0, 1\rangle$ and accept the new values if $min\{1, e^{-\Delta E/T}\} > r$.
4. Repeat step 1 to 3 until enough statistics is gathered.

The London application domain is a 3-dimensional lattice of size (S_x, S_y, S_z) with periodic boundary conditions. The local energy E_s at one site s in the lattice is dependent on $\Delta\theta$ and $\Delta\times\mathbf{A}$. That is the nearest neighbor of θ, the nearest neighbors and half of the next nearest neighbors of $\mathbf{A}$. More formally, all the sites *adjacent* to s are involved in the computation of E_s.

Definition 5.1. *Two lattice sites* $s_1 = (x, y, z)$ *and* $s_2 = (t, u, v)$ *are said to be* **adjacent** *if and only if* $(t, u, v) \in \{(x, y, z-1), (x, y+1, z-1), (x+1, y, z-1), (x-1, y, z), (x-1, y+1, z), (x, y-1, z), (x, y+1, z), (x+1, y-1, z), (x+1, y, z), (x-1, y, z+1), (x, y-1, z+1), (x, y, z+1)\}$*, as depicted in Figure 5.1.*

5.3 Domain decomposition and detailed balance condition

Domain decomposition methods divide the global lattice into P local lattices assigned to each processor. Assigning equally sized portions of the lattice to each processor ensures that the computational load is well balanced on a homogeneous system. Three different

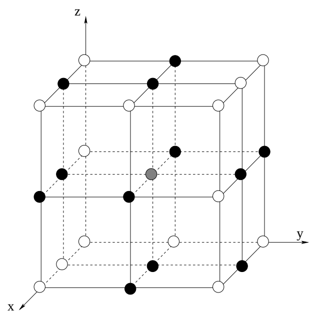
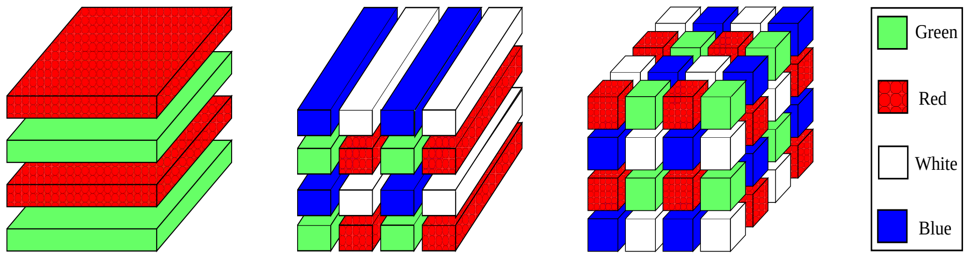


Figure 5.1: London's application stencil. Black sites are adjacent to the grey site.

decompositions are considered in this work. The 1D decomposition divides the global domain into local lattices of size (S_x, S_y, B_z), where $\frac{S_z}{B_z} = P$ (see Figure 5.2 b). Each processor is identified by an integer $i_z \mid i_z \in [0, P\rangle$. The 2D decomposition assigns local lattices of size (S_x, B_y, B_z), where $(\frac{S_y}{B_y}) \times (\frac{S_z}{B_z}) = P$ (see Figure 5.2 c). Each processor is identified by a pair of integers: $(i_y, i_z) \mid i_y \in [0, \frac{S_y}{B_y}\rangle,\ i_z \in [0, \frac{S_z}{B_z}\rangle$. At last, the 3D decomposition divides the global domain into P local lattices of size (B_x, B_y, B_z), where $P = (\frac{S_x}{B_x}) \times (\frac{S_y}{B_y}) \times (\frac{S_z}{B_z})$ (see Figure 5.2 d). Each processor is then identified by a triplet of integers: $(i_x, i_y, i_z) \mid i_x \in [0, \frac{S_x}{B_x}\rangle,\ i_y \in [0, \frac{S_y}{B_y}\rangle,\ i_z \in [0, \frac{S_z}{B_z}\rangle$.



(a) 1D. (b) 2D. (c) 3D.

Figure 5.2: The different domain decompositions.

In order to respect the detailed balanced condition, the processors are sorted into different color sets, such that processors of the same color can update their exterior sites simultaneously. For the 1D case, two colors are necessary and sufficient, whereas four colors are necessary for the 2D and 3D decompositions. Then, an ordering is established among the colors to orchestrate the updates of the outer regions. For the 1D case, green nodes start ahead of the red nodes, while the color ordering of 2D and 3D decompositions is 1) green, 2) red, 3) white and 4) blue.

The message passing paradigm provides a natural synchronization mechanism for the boundary updates. Nodes are allowed to update their outer regions only when they have received the boundaries from all their neighbors. At start, an initialization takes place in order to settle the color ordering mechanism. During this initialization, nodes send their domain boundaries only to their neighbors preceding in the color ordering. In this way, the color nodes in first position will be ready to update their outer regions, while the nodes in second position will lack boundaries only from the nodes in first position, the nodes in third position will lack boundaries from the nodes in first and second position, and so on. To alleviate the exposition, we introduce the concept of *virtual tokens*. The reception of all the boundaries is liken to the reception of a virtual token, that allows updating the outer region. Thereafter processors hand on the token by sending their updated local boundaries to their neighbors.

Different boundary-exchange patterns occur depending on the chosen domain decomposition. For the 1D case, only vertical boundary-exchanges take place. The 2D decomposition imposes also vertical boundary-exchanges, but in addition, horizontal (along the y-axis) and diagonal (along the yz-axis) boundary-exchanges are necessary. The horizontal exchange happens like the vertical one, by exchanging messages between the two nodes concerned. The diagonal exchange however (where only edges must be communicated), can be performed indirectly via other nodes, by using the *diagonal communication elimination* technique [5, 7], which consists of including ghost cells in messages. To perform correct diagonal exchanges, green and white nodes include ghost cells in their horizontal exchanges, while red and blue nodes include ghost cells in their vertical exchanges. The 3D decomposition imposes the same boundary-exchanges than the 2D decomposition. In addition, in-depth (along the x-axis) and diagonal (along the xz- and the xy-axes) boundary-exchanges must also take place. The in-depth exchange happens like the vertical and horizontal ones, by exchanging messages between the two nodes concerned. Most of the diagonal exchanges can be performed indirectly using the aforementioned diagonal elimination technique. Green and white nodes include ghost cells in the horizontal exchange, while red and blue nodes include ghost cells in the in-depth exchange, handling thus 4 diagonal transfers. However, there remains two diagonal transfers that cannot be handled indirectly and that require additional messages. These are the edges needed by the nodes which are following the running color in the token round-trip ordering.

When passing the token to the next color, all the nodes of the same color use the same communication trajectories (i.e. communicate with their neighbors in the same order), and thus obtain only independent communications, i.e. with disjoint pairs of senders and receivers.

5.4 Theoretical models

Santos et al. [15] represent the run-time cost of updating one site of the lattice with a single variable C_s. Then updating N sites simply lasts for NC_s time units. The problem with such a linear model, is that it does not account for *data locality*, a critical aspect with respect to the cache-based memory hierarchy of modern systems. There are two types of *locality* that can be exploited to improve performance when implementing stencil codes. There is *spatial* locality when accessing neighboring points (in address space), and there is *temporal* locality when array elements are reused several times before being evicted from the cache. Typically, parallel implementations that must deal with the detailed balance condition, decompose the local lattices into a few number of regions that are updated one after another, in alternation with communication events [7, 10, 11, 15]. This has a direct negative impact on data locality. Indeed, spatial locality is not fully exploited as neighboring sites may not belong to the same region, which by induction degrades temporal locality, as these sites will be updated at different time steps. In these conditions, lattice sites will be brought into cache multiple times per iteration. We propose a coarser-grained computational model, by modeling the run-time costs for updating each region composing the local domain. Thus, we let T_i and T_o be the run-time costs for updating the inner and the outer regions of the domain respectively.

On the other hand, communication modeling has more or less converged towards an extension of the Hockney's model [9] which characterizes the time t for sending a message of m bytes as follows: $t = \tau + \gamma m$, where τ is the *latency*, and γ the inverse of the *asymptotic bandwidth* of the network. Culler et al. [3] have proposed the *LogP* model for modeling sequences of point-to-point communications of short messages. LogP defines the *Latency* L incurred in sending a message, the *overhead* o which is the time a processor spends sending or receiving a message, the *gap* g between two messages defined as the minimum time interval between consecutive message transmissions or consecutive message receptions, and the number of processors P. Alexandrov et al. [1] have extended the LogP model in order to capture both short and long messages. The resulting model, called LogGP, presents an additional parameter G which is the *Gap per byte* for long messages. The cost of sending a k-bytes message in the LogGP model is simply $L + 2o + (k-1)G$, and the cost of sending two messages in a row (of lengths k_1 and k_2 respectively) is $L + 2o + g + (k_1 + k_2 - 2)G$. LogGP clearly models the important characteristics of a homogeneous cluster of machines.

5.5 Theoretical results

Although our algorithms support overlapping computation with communication, no overlap at all is assumed in this analysis. This assumption seems to match the underlying hardware of low-cost PC clusters, and provides a fair comparison with other methods. The local domains being decomposed into an inner and an outer regions, a sweep does not terminate before both regions have been updated. Because, outer regions cannot be updated before the token has arrived, the total run-time cost of a sweep is given by:

$$T = T_o + \max\{T_{com} + T_i\ ,\ T_r\} \tag{5.2}$$

where:

- T_{com} is the local communication overhead.
- T_r is the token round-trip run-time cost.

In the remaining analysis, for the sake of simplicity, we assume a global lattice of size $(S \times S \times S)$ with $B = S/P$ for 1D, $B = S/\sqrt{P}$ for 2D and $B = S/\sqrt[3]{P}$ for 3D decomposition.

5.5.1 1D decomposition

Passing the token to the opposite color involves sending 2 consecutive messages composed of S^2 sites. This operation costs $L + 2o + g + 2(8S^2 - 1)G$ (in double precision). Since there are only 2 colors involved in the computation, we have $T_r = T_o + 2L + 4o + 2g + 4(8S^2 - 1)G$. Only 2 messages are sent and received by each nodes which gives $T_{com} = 4o$.

5.5.2 2D decomposition

The token round-trip goes over four colors, and the boundary-exchanges are composed of 4 messages. However, only the 2 first messages intervene in the token round-trip run-time cost, as they complete the token transmission to the next color. The two last messages are intended to a color that is two steps away of getting the token, and therefore do not delay the token round-trip (assuming they do not congest the network). Starting from the green nodes, here is what the token round-trip looks like: 2 horizontal communications with red, 1 surface update (red), 2 vertical communications with white, 1 surface update (white), 2 horizontal communications with blue, 1 surface update (blue) and finally 2 vertical communications with green. By symmetry over the colors, the run-time cost of the token round-trip is identical for all the color nodes. Sending 2 consecutive messages composed of $S(B+1)$ sites (including ghost cells) costs $L+2o+g+2(8S(B+1)-1)G$. Since the token goes over three colors before coming back, the run-time cost of the token round-trip is then: $T_r = 4L + 8o + 4g + 8(8S(B + 1) - 1)G + 3T_o$. Each node sends and receives 4 messages which gives $T_{com} = 8o$.

5.5.3 3D decomposition

In the 3D case, all the colors must communicate with each other, and the boundary-exchanges comprise 8 messages. Assume that green nodes are holding the token. After updating the outer region, each green node sends 4 messages to its red neighbors, which can from then on start updating their own outer regions. Thereafter 2 messages are sent by each green node to its white neighbors. In the best case scenario, updating the red outer region takes more time than the communication between the green and white nodes. Hence the red-white communication is not delayed by the green-white communication. In the worst case scenario, the update of the outer region takes less time than the green-white communication, which delays the red-white communication. Similarly, the white-blue communication can possibly be delayed by the red-blue communication, the blue-green communication by the white-blue communication, and the green-red communication can be delayed by the blue-red communication. Starting from the green nodes, here is what the token round-trip looks like: 1 eventual delay, 2 horizontal and 2 diagonal communications with red, 1 surface update (red), 1 eventual delay, 2 horizontal and 2 diagonal communications with white, 1 surface update (white), 1 eventual delay, 2 horizontal and 2 diagonal communications with blue, 1 surface update (blue) and finally 1 eventual delay, 2 horizontal and 2 diagonal communications with green.

For all the colors the eventual delay is caused by 2 consecutive vertical communications, each of size $8B^2$ bytes, which together cost $T_v = L + 2o + g + (8B^2 - 1)G$. The run-time cost of the delay is then $T_d = \max\{T_v - T_o, 0\}$. Except the vertical communications that can provoke a delay in the token transmission, the only messages that intervene in the run-time cost of the token round-trip are the 4 first messages sent by each processor, as they complete the token transmission to the next color set. For each color, 2 messages of $8(B + 1)^2$ bytes are followed by 2 diagonal messages of $8B$ bytes which altogether costs $L + 4o + 3g + (8(B + 1)^2 + 8B - 2)G$. The run-time cost of the token round-trip is hence: $T_r = 3T_o + 4L + 16o + 12g + 8(8B^2 + 8B + 3)G + 4T_d$. Each node sends and receives 8 messages which gives $T_{com} = 16o$.

Note that for the 3D case, increasing P to a maximum, would amount to allocate one single lattice site per processing unit. In fact, the inherent level of parallelism of the London model would be reached at that point, since the 3D color code depicted in Figure 5.2 (d) can also be extended to the lattice sites. At all times, only $\frac{1}{4}$ of the sites can be updated simultaneously.

5.5.4 Speed-up and efficiency tradeoff

Usually, when the problem size is fixed, an increase in the number of processors can begin to have a negative impact on the speed-up S_p. At some point, adding more resources causes performance to decrease and speed-down is observed. Interestingly, increasing the number of processors used by our algorithms will always incur a speed-up improvement. Off-course, this yields only for reasonably large problem sizes (i.e. that cannot be tackled by a single processor), and if the network can handle the communications generated by additional processors. However, improving the speed-up at all costs may come at the expense of a poor efficiency, defined as $E_p = S_p/P$.

According to Equation set (5.2), the run-time cost of a sweep is $T = T_o + \max\{T_{com} + T_i\ ,\ T_r\}$. When minimizing T, the interesting component is the maximum quantity. For fixed problem size and fixed P, consider the quantities $(T_{com} + T_i)$ and T_r.

If $(T_{com} + T_i) > T_r$, the token comes back before the inner region has been updated. In that case, it will be beneficial to increase P since it will increase the surface to volume ratio of the local lattice. In other words, T_i will decrease faster than T_o, which by induction will tighten up the gap between $(T_{com} + T_i)$ and T_r, albeit the overhead incurred by the communications involved in the token round-trip decreases too.

But if $(T_{com} + T_i) < T_r$, the processors have already finished to update the inner region, and are starving while waiting for the token. Increasing further the number of processors still improves the speed-up, albeit processor starvation may increase. Indeed, although T_r would still dominate the maximum quantity of Equation (5.2), increasing P generates a reduction of the data volumes that need to be processed and exchanged. In other words, when P increases, T_o, T_i and T_r decrease. Consequently, T is minimal when T_i and T_o are minimal. The parallel efficiency, however, may degrade as starvation increases.

Finally, there is a threshold value for which $(T_{com} + T_i) = T_r$, that may be a good compromise with respect to speed-up and efficiency. Indeed, such value would accelerate the run-time of a sweep without introducing processor starvation. Going beyond this threshold, may admittedly improve the speed-up, but would definitely degrade the efficiency by bringing about starvation. On dedicated systems, this effect is highly undesirable. On the contrary, staying under this threshold will certainly optimize efficiency, but will hold back the speed-up.

5.6 Experimental results

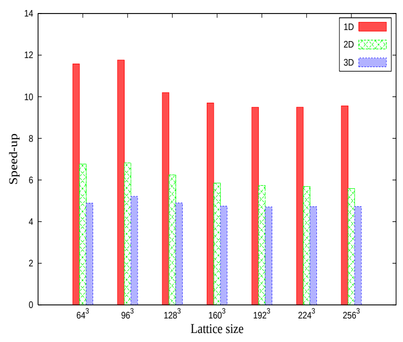
The Scali implementation of the Message Passing Interface (MPI) [6] has been used in this study. MPI features like persistent requests and derived datatypes have been used for implementing the successive boundary-exchanges. Special care has been taken when posting and completing the communication requests such that the MPI *ready* communication mode could be used. All these decisions contribute to keep the communication overhead to a minimum. For the sake of portability, non-blocking requests have been used in order to exploit the inherent computation-communication overlap of the computing paradigm, even though many implementations cannot overlap without extra hardware in the form of a communication coprocessor.

In our experiments, the number of sweeps was arbitrarily fixed to 500 in order to highlight differences between the different decompositions while keeping measurement times relatively low. The processors were exclusively dedicated to our application, which reduces external interferences to system fluctuations. Finally, the performance curves presented in this paper correspond to the average values over 3 runs.

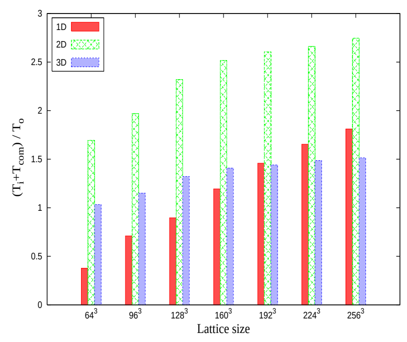
Figure 5.3 (a) which is representative of our experiments, reveals that the 1D decomposition clearly outperforms the other decompositions, although the 2D and 3D decompositions present better surface-to-volume ratios. Figure 5.3 (b) depicts the $\frac{(T_i+T_{com})}{T_o}$ ratio for the different decompositions. The 2D decomposition clearly presents the best ratio, while the 3D decomposition only is beneficial for relatively small problem sizes, albeit these ratios are not as large as expected. This observation is in line with the study of Prieto et al. [13]: Updating separately the inner and outer regions of the local domain degrades the performance of the outer region update (especially for the 3D decomposition). This comes from the relatively lower density, or sparsity, of the outer regions, that incurs non contiguous data access patterns. Moreover, the amount of data required by LGT simulations is very large, such that even for moderate problem sizes, data do not fit into processor caches. Indeed, each site composing the lattice represents one scalar field and three vector fields, i.e. amounts to 4 real numbers. In these conditions, a $16 \times 16 \times 16$ lattice does not fit in the 9MB L3 cache of the itanium2 processor. This contributes to lower the performance of the outer region update.

But more importantly, the supremacy of the 1D decomposition over the 2D and 3D decompositions is certainly due to the complicated token round-trip trajectories of the latter decompositions as opposed to the much simpler trajectory of the 1D case. Indeed, the 1D token round-trip imposes only 1 outer region update and 4 messages, as opposed to 3 outer region updates and 8 messages for the 2D case, and 3 outer region updates and 12 messages for the 3D case. For all the experiments, the token round-trip dominates the total sweep run-time cost, which means that processors are starving, waiting for the token to arrive. Figure 5.3 (c) illustrates this observation, as the ratio $\frac{T_r}{T_i+T_{com}}$ is much smaller for the 1D decomposition than for the other ones. The run-time costs for updating the inner regions were roughly equivalent for the 3 domain decompositions, which means that processors are starving longer under the 2D and 3D decompositions than for the 1D decomposition.

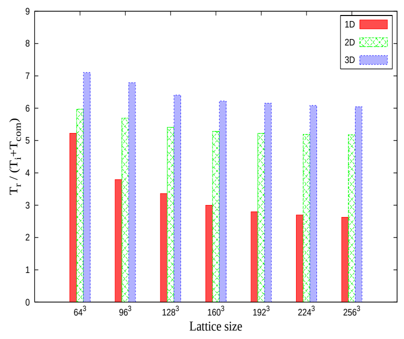
Nonetheless, the token-passing algorithms expose both *strong* and *weak* scalability (see Figure 5.3 (d)). Strong scalability means that, for fixed problem size, the speed-up is roughly proportional to the number of processors used. On the other hand, weak scalability means the ability of maintaining a fixed efficiency when the problem size and the number of processors increase. The 1D token-passing algorithm presents an efficiency greater than 0.25 for all the experiment conducted on the Snowstorm cluster. For problem sizes of interest for the physicists at NTNU (up to 96^3), the efficiency of the 1D algorithm is above 0.33.



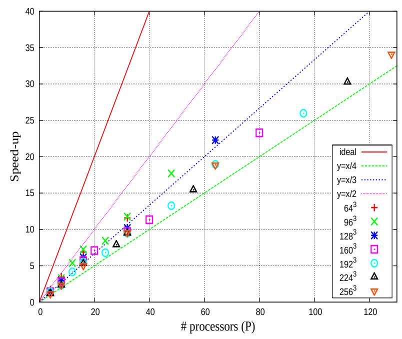
(a) Speed-up with 32 processors.



(b) $(T_i + T_{com})/T_o$ with 32 processors.



(c) $T_r/(T_i + T_{com})$ with 32 processors.



(d) 1D Speed-up

Figure 5.3: Experimental results.

5.7 Related work

MC simulations for the Ising model, which uses only nearest-neighbors interactions, have been successfully implemented on shared-memory systems with checker-board algorithms. The lattice sites are sorted into a red set (where sum of coordinates is even) and a black set (where sum of coordinates is odd) in a checker-board fashion. Thus, all the red sites can be updated simultaneously, and so it is for the black sites. Checker-board algorithms have been ported onto distributed memory systems by numerous studies [7, 8, 11]. For each sweep, all the processors start by updating one color set, say the red one. Thereafter the nodes exchange the red sites located on the boundaries, and do the same with the black set. This approach performs boundary-exchanges with two messages per boundary. For longer-range interactions models, such as the one presented in this study, new updating schemes that fit with the stencil application must be applied. In these conditions, the checker-board is likely to be composed of four colors, leading to boundary-exchange with four messages per boundary. For the London superconductor model, the color mask that could be utilized by a checker-board algorithm is the one depicted in Figure 5.2 (d), where each cube corresponds to a lattice site.

Recently, Santos et al. [15] conducted research on MC simulations for 2D and 3D Ising models in another direction. Each local domain is divided into different regions, that are updated one after another, in alternation with communication events. For each sweep, all the processors update the same region of their local domain, in order to avoid situations where remote but adjacent site updates would enter in conflict. Then some boundary-exchanges take place, allowing the parallel computation to proceed with the next region. The number of regions composing the local domain is dependent on the chosen decomposition (2 for 1D, 3 for 2D, and 4 for 3D). The result is an increase of the number of messages required for the boundary-exchanges, namely 2 for 1D, 8 for 2D, and 24 for the 3D decomposition.

The two aforementioned methods handle the detailed balance condition by increasing the number of messages per boundary-exchange, which can considerably increase the communication phase run-time cost. This results from the synchronization imposed to the processors. In effect, at all times, the processors update the same region of their local domains in concert. Table 5.1 reports the sweep run-time costs of the methods found in the literature when applied to a simplified version of the London superconductor model, and compares them to the algorithms proposed in this paper. T_{col} stands for the run-time cost for updating the sites of a given color when using a checker-board algorithm, and T_{comp} stands for the run-time cost for updating all the regions composing the local domain when using the Santos methods.

When deriving these theoretical results, first we assumed that processors can send and receive messages simultaneously without additional cost. This situation is ideal for the checker-board and Santos methods, but does not favor the token-passing algorithms. Then, we assumed the token round-trip time to dominate the total sweep run-time. This is conform to what has been observed in our experiments, and corresponds to the worst case scenario.

CB	1D	$4T_{col} + 4L + 8o + 4g + (16S^2 - 8)G$
	2D	$4T_{col} + 4L + 8o + 12g + (32SB - 16)G$
	3D	$4T_{col} + 4L + 8o + 20g + (48B^2 - 24)G$
[15]	1D	$T_{comp} + 2L + 4o + (16S^2 - 2)G$
	2D	$T_{comp} + 3L + 16o + 5g + (32SB - 8)G$
	3D	$T_{comp} + 4L + 48o + 20g + (48B^2 + 18)G$
TP	1D	$2T_o + 2L + 4o + 2g + (32S^2 - 4)G$
	2D	$4T_o + 4L + 8o + 4g + (64SB - 8)G$
	3D	$4T_o + 4L + 16o + 12g + (64B^2 - 16)G$

Table 5.1: Comparison with other methods found in the literature (CB checker-board, Santos et al. [15] and TP token-passing).

Even under the aforementioned assumptions, favoring the two other methods, the token-passing algorithms seem to be the most appropriate. For small computation-to-communication ratios (the difficult case), the start-up and latency costs will dominate the communication phase. In that case, our algorithms will most likely perform better, because exchanging fewer messages per iteration. On the other hand, when computation dominates the sweep run-time cost, the token-passing algorithms that decompose the domain into fewer regions, will much likely better utilize the memory hierarchy of modern processors.

Finally, there are several situations where the token-passing algorithms clearly outperform the other methods. For instance, many MPI implementations do not support simultaneous send and receive operations. A typical example is the implementation of the MPI_Sendrecv routine, which often resorts on odd-even ordering of communications resulting in a serialization of the messages. Thus, the amount of bytes transferred would be multiplied by 2 for the checker-board and Santos methods. In contrast, the run-time cost of the token-passing algorithms, which separate the message transmissions from the receptions, would remain unchanged. Then, for applications that present a high enough computation-communication ratio, computation would become the dominating factor of the total sweep run-time cost. In that case, the token would come back before the inner region update terminates, restricting thus the communication overhead to the minimum.

5.8 Conclusion

This paper addresses the difficult task of parallelizing LGT models on distributed memory platforms. The token-passing algorithms presented in this paper provide a mean to effectively orchestrate boundary-updates in order to cope with the detailed balance condition. These parallel algorithms combine efficient techniques for domain decompositions methods, such as diagonal elimination and can take advantage of computation-communication overlap if this feature is supported by the system, as opposed to previous studies presented in the literature. The main departure from previous LGT studies consists in minimizing the number of messages exchanged during the computation. This has been possible by staggering the parallel execution, i.e. by making the processors work simultaneously on different parts of their local domains. However, decomposing the domain into an outer and an inner regions comes at the expense of a reduced performance when updating the outer region, because of its lower density.

Nonetheless, we observed that increasing the number of processors still generates a speed-up improvement. Although the 2D and 3D decompositions present lower surface-to-volume ratios, the 1D decomposition achieves the best performance due to its simpler token round-trip communication patterns. This signifies that the London LGT model exhibits a too low computation-to-communication ratio on the cluster used in this study, to take advantage of the 2D and 3D decompositions. Finally, we provide rigorous theoretical results for all the domain decompositions under the LogGP model, and show the superiority of our approach over other methods found in the literature, for different system and problem configurations.

The algorithms presented in this study assume a homogeneous system dedicated to the application. However, emerging clusters are usually multi-users systems composed of heterogeneous resources, allowing several independent applications to run concurrently. Developing new techniques and algorithms that cope with and respond to heterogeneity, instability and system fluctuations is the direction for future work.

Bibliography

[1] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model – One step closer towards a realistic model for parallel computation. In *SPAA '95: Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 95–105, New York, NY, USA, 1995. ACM Press.

[2] S. B. Baden and S. J. Fink. Communication overlap in multi-tier parallel algorithms. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–20, Washington, DC, USA, 1998. IEEE Computer Society.

[3] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken. LogP: A Practical Model of Parallel Computation. *Commun. ACM*, 39(11):78–85, 1996.

[4] C. Dasgupta and B. I. Halperin. Phase Transition in a Lattice Model of Superconductivity. *Physical Review Letters*, 47:1556–1560, Nov. 1981.

[5] C. Ding and Y. He. A Ghost Cell Expansion Method for Reducing Communications in Solving PDE Problems. In *Supercomputing ’01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 50–50, New York, NY, USA, 2001. ACM Press.

[6] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, Cambridge, Massachusetts, USA, 1994.

[7] F. Gutbrod, N. Attig, and M. Weber. The SU(2)-lattice gauge theory simulation code on the Intel Paragon supercomputer. *Parallel Comput.*, 22(3):443–463, 1996.

[8] D. W. Heermann and A. N. Burkitt. *Parallel Algorithms in Computational Science*. Springer-Verlag New York, Inc., New York, NY, USA, 1991.

[9] R. W. Hockney. Performance Parameters and Benchmarking of Supercomputers. *Parallel Computing*, 17(10-11):1111–1130, 1991.

[10] W. Janke and R. Villanova. Ising model on three-dimensional random lattices: A Monte Carlo study. *Physical Review B*, 66(13):134208–+, Oct. 2002.

[11] M. Luscher. Solution of the Dirac equation in lattice QCD using a domain decomposition method. *Comput. Phys. Commun.*, 156:209–220, 2004.

[12] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of State Calculations by Fast Computing Machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

[13] M. Prieto, I. M. Llorente, and F. Tirado. Data Locality Exploitation in the Decomposition of Regular Domain Problems. *IEEE Trans. Parallel Distrib. Syst.*, 11(11):1141–1150, 2000.

[14] M. J. Quinn and P. J. Hatcher. On the Utility of Communication-Computation Overlap in Data-Parallel Programs. *J. Parallel Distrib. Comput*, 33(2):197–204, 1996.

[15] E. E. Santos and G. Muthukrishnan. Efficient Simulation Based on Sweep Selection for 2-D and 3-D Ising Spin Models on Hierarchical Clusters. In *IPDPS*, page 229b, 2004.

Paper 6

Domain Decomposition vs. Master-Slave in Apparently Homogeneous Systems
Cyril Banino-Rokkones
To appear in HCW'2007, the 16th Heterogeneous Computing Workshop
March 26 2007, Long Beach, California, USA.

Domain Decomposition vs. Master-Slave in Apparently Homogeneous Systems

Cyril Banino-Rokkones

Department of Computer and Information Science
Norwegian University of Science and Technology
NO-7491 Trondheim, Norway

Abstract

This paper investigates the utilization of the master-slave (MS) paradigm as an alternative to domain decomposition (DD) methods for parallelizing lattice gauge theory (LGT) models within distributed memory environments. The motivations for this investigation are twofold. First, LGT models are inherently difficult to parallelize efficiently with DD methods. Second, DD methods have proven useful for homogeneous environments, but are impractical for heterogeneous and dynamic environments. Besides, many modern supercomputer architectures that look homogeneous (such as multi-core or SMP), are in fact heterogeneous and dynamic environments. We highlight this issue by comparing a traditional first-come first-served MS implementation to a simple but yet efficient selective MS scheduling strategy that automatically accounts for system heterogeneity and variability. Our experimental results with the parallelization of our LGT model, reveal that the selective MS implementation achieves good efficiency, but lacks of scalability. In contrast, the DD method is highly scalable, but at the expense of a poor efficiency. These results open up for a hybrid approach, where the MS and the DD methods would be combined for achieving scalable high performance.

6.1 Introduction

Domain decomposition (DD) methods have been studied extensively because of their utility in a wide range of application areas such as, physics, chemistry, solid and fluid mechanics, or climate modeling. Domain decomposition on parallel computers consists in splitting the computational domain into smaller sub-domains, each of which is assigned to one processor. Then, during the execution, computation and communication phases alternate, as neighboring processors (in the topological decomposition) need to periodically exchange data located on the boundaries of their local domains.

On the one hand, the efficiency of DD methods is strongly affected by the heterogeneity and variability present in the underlying computing system. Indeed, DD methods are efficient only when the computational load is well balanced among the processors. The processors being tightly coupled by the communication phases, the execution proceeds at the pace of the slowest processor. For homogeneous and stable systems, the computational domain needs simply to be decomposed into p equally-sized sub-domains. For heterogeneous environments on the other hand, things get complicated as the domain must be decomposed into p sub-domains whose size must be proportional to the processor computational speeds. In dynamic environments, where resources exhibit unforeseeable performance fluctuations, things get even worse, as it becomes necessary to frequently redistribute the computational domain among the processors.
In addition, many modern supercomputer architectures (such as multi-core or SMP clusters) that look homogeneous, hide in fact an heterogeneous and dynamic environment. For instance, processors located within the same node are actually competing for shared resources (e.g. caches), and intra-node communication is typically much faster than inter-node communication. The impact of the heterogeneity and variability hidden in the system on the performance of DD methods is difficult to evaluate, but undoubtedly degrades the performance.
Last but not least, an important issue concerns fault tolerance. Currently, the most common technique for handling fault tolerance within DD methods is checkpoint/restart. That is, checkpoints are saved to disk periodically, and if a processor fails, the computation halts and restarts from the last consistent checkpoint. For lengthy applications that make use of a large number of processors, failures are more likely to be the rule rather than the exception. In these conditions, the checkpoint/restart technique could take longer than the time to the next failure. Hence, there is a need to survive failures without relying on global recovery operations.

On the other hand, the efficiency of DD methods is directly subject to the characteristics of the scientific problem to be solved. Some problems are more suited to DD methods than others. In this paper, we are interested in lattice gauge theory (LGT) models, a class of Monte Carlo (MC) simulations particularly difficult to parallelize efficiently with DD decompositions in distributed memory environment (i.e. when message passing is unavoidable).
LGT models belong to the wide class of *stencil computations*, where typically, each site in a multi-dimensional lattice is updated with contributions from a subset of its neighbors (see Figure 6.1). For each iteration, the stencil kernel is applied to all the lattice sites - usually the boundaries receive a special treatment.
When parallelizing LGT models with DD methods, one must ensure at all times that processors owning neighboring sub-domains do not update adjacent sites simultaneously. Although neighboring lattice sites may be updated in any order, physical properties impose these updates to happen sequentially, creating thus constraining data dependencies. The message passing paradigm provides a simple and natural way to orchestrate the lattice updates without violating these data dependencies. Communication events can be used as *tokens*, such that incoming messages from neighboring processors trigger the up-

date of the corresponding sub-domain boundary. However, in the case of LGT models, this technique introduces a significant amount of idle time on the processors, degrading significantly the parallel efficiency.

Thus, there are two main reasons for considering an alternative way to DD methods: The inadequacy of DD methods for dealing with heterogeneous and dynamic environments; and the lack of efficiency of DD methods for parallelizing LGT models. In this paper, we study the suitability of the master-slave paradigm (MS) as an alternative to DD methods for implementing LGT models within distributed memory environments. The MS paradigm admittedly comes along with some limitations, but presents most of the features required for dealing not only with LGT models, but also with heterogeneous and dynamic environments.
In its simplest form, the MS paradigm works as follows. The master initially distributes one task to every slave. The slaves compute their tasks and send the results back to the master, which triggers the latter to send additional tasks. The main assets of the MS paradigm are *flexibility* and *robustness*. As slaves execute tasks at their own paces, they will automatically request tasks proportionally to their computing speeds. This is popularly known as *self-scheduling*, *demand-driven* or *first-come first-served* (FCFS). By construction, FCFS adapts well to the performance fluctuations of the computational resources. If a slave suddenly gets some external load, it will process tasks less rapidly, and hence request tasks less frequently. When the conditions get back to normal, the slave will request tasks at its maximal pace. However, FCFS is not efficient when point-to-point communication times are heterogeneous. In that case, resource selection strategies become necessary in order to efficiently utilize the available computing and communication resources. In this paper, we show that a simple, yet effective, *selective* scheduling scheme is more appropriate for dealing with heterogeneous and dynamic environments than the traditional FCFS strategy.
Finally, the loosely coupled structure of the MS paradigm presents only one *single point of failure* in the form of the master process. This means that one only needs to backup the master node to achieve reliability. If some slave processes die, the computation can still carry on with the remaining slaves.

The rest of this paper is organized as follows. Section 6.2 reviews previous work related to LGT model parallelizations, DD methods and the MS paradigm. Section 6.3 introduces the LGT model considered in this study. The DD and MS parallelizations of the LGT model are presented respectively in Section 6.4 and Section 6.5. In addition, our MS selective scheduling strategy is exposed and compared to the FCFS strategy in Section 6.5. Section 6.6 reports an experimental comparison between the MS and the DD implementations. Future work is discussed in Section 6.7. Finally concluding remarks are given in Section 6.8.
The experiments reported in this study have been performed on a SMP cluster composed of 100 HP Integrity rx4640 server nodes. Each SMP node comports 4 itanium2 processors clocked at 1.3 GHz sharing 4 GB of memory. The 100 SMP nodes are interconnected with the Infiniband network.
In all the experiments reported in this paper, the number of iterations was arbitrarily fixed

to 500 in order to highlight differences between the different implementations while keeping measurement times relatively low. The experiments were performed on a dedicated set of computing nodes, which reduces external interferences. Finally, all the performance curves reported in this study correspond to the average values over 3 runs.

6.2 Related work

Several studies have considered parallelizing MC simulations using DD methods [3, 13, 17, 21, 29]. MC simulations for the Ising model, which uses only nearest-neighbors interactions (6-point stencil in 3 dimensions), have been successfully implemented on shared-memory systems with checker-board algorithms. The lattice sites are sorted into a red set (where sum of coordinates is even) and a black set (where sum of coordinates is odd) in a checker-board fashion. Thus, all the red sites can be updated simultaneously, and so it is for the black sites. Checker-board algorithms have been ported onto distributed memory systems by several studies [13, 15, 21]. For each iteration, all the processors start by updating one color set, say the red one. Thereafter the nodes exchange the red sites located on the boundaries, and do the same with the black set. This approach performs boundary-exchanges with two messages per boundary. For longer-range or more complex interactions models, such as the one presented in this study, new updating schemes that fit with the stencil kernel must be applied. In these conditions, the checker-board is likely to be composed of at least four colors, leading to boundary-exchanges with four messages per boundary.

Santos et al. [28, 29] conducted research on MC simulations for 2D and 3D Ising models in another direction. Each local domain is partitioned into different sets, that are updated one after another, in alternation with communication events. For each iteration, all the processors update the same set of their local domain, in order to avoid situations where remote but adjacent site updates would enter in conflict. Then some boundary-exchanges take place, allowing the parallel computation to proceed with the next set. The number of sets composing the local domain is dependent on the chosen decomposition (2 for 1D, 3 for 2D, and 4 for 3D). The result is an increase of the number of messages required for the boundary-exchanges (namely 2 for 1D, 8 for 2D, and 24 for the 3D decomposition).

The two aforementioned methods handle the data dependencies between neighboring sites by increasing the number of messages per boundary-exchange, which considerably increases the communication run-time cost. Recently, we provided token-passing algorithms based on DD methods that minimize the number of messages exchanged between neighboring processors [3]. Our algorithms are presented and explained in great details in [3], but we provide a brief summary in Section 6.4.

Although DD methods are relatively easy to deploy efficiently on homogeneous environments, dealing with heterogeneous and dynamic environments is a much more complicated task. Several studies have been conducted on deploying DD methods within heterogeneous environments [4, 5, 18, 19, 22]. In most cases, the problem is reduced to the problem of partitioning some mathematical objects, such as matrices, sets or graphs [9]. The

main difficulty resides in the combinatorial nature of the problem which typically turns out to be NP-complete. Even though efficient (i.e. polynomial) heuristics are derived, the dynamic nature of the underlying platform makes static strategies not well suited to these environments. In dynamic environments, the processor speeds and network contention will fluctuate during the execution requiring online load redistribution mechanisms. Online redistribution is difficult to handle, as it poses the question of when should one redistribute the load? And how to measure the quality of a load distribution? Beaumont et. al. [6] consider the matrix multiplication problem in heterogeneous and dynamic environments, and propose to redistribute the load only between large static-phases. Still, one must find a good load redistribution frequency, since a too conservative approach may not result in significant improvements, wheras being too aggressive may incur too much overhead. An important point stressed by Beaumont et. al. is the necessity to minimize the amount of communication when redistributing the load. The amount and location of the data should be taken into account in order to maintain the relative position of the processors, otherwise the cost of the redistribution may be prohibitive. Similarly Mahanti and Eager [22] find that data migration costs should be minimized for efficient redistribution, and propose redistribution policies that try to leave the relative position of the nodes unaltered. In their work, Mahanti and Eager consider data redistribution following addition/removal of processors.
Although these studies on DD methods within heterogeneous environments present interesting results that give insights on the problem difficulties, these different strategies typically rely on a centralized algorithm to (re)distribute the work among the heterogeneous processors. This clearly poses the question of the scalability of the approach. On the other hand, the problem of online load redistribution frequency is difficult to address without disposing of some form of centralized information about the platform state.

Similarly to DD methods, the MS paradigm is well known and has been the subject of a wealth of studies both in the context of Cluster computing [10, 24, 25] and of Grid computing [7, 12, 16]. Usually the applications implemented under the MS paradigm are composed of a large number of independent tasks. All the popular scheduling strategies designed for minimizing the total execution time, hand out tasks by chunks of decreasing size, in order to reduce the scheduling overhead while achieving a good load balance at the end of the execution [14]. However, this kind of MS strategies cannot be utilized in our study, because the tasks composing our target applications are not fully independent of each other (more on this in Section 6.5).

6.3 Our lattice gauge theory model

Lattice spin and gauge theories are studied extensively in many areas of physics, especially in particle and condensed matter physics. The spin and gauge field variables are defined on every site of a multi-dimensional lattice, and the thermodynamic properties of the system can be deduced from the partition function, which is a sum over all possible configurations of the fields. Exact solutions to these multi-dimensional sums are rare and in general one must resort to some numerical approximation. The largest and most important class of numerical methods used for this problem is the Monte Carlo (MC) method, which in stead of doing the sum over all configurations, utilizes random numbers to mimic the random thermal fluctuations of the system from one configuration to the other. A considerable proportion of the computational resources used by physicists around the world is spent on MC simulations.

The LGT model studied in this paper is a superconductor model in which a real valued scalar field is coupled to a real valued vector field. This model is a simplified version of the one presented in [1]. The MC algorithm used for the simulations is the celebrated Metropolis algorithm [23] which can be described the following way.

1. Pick one site in the lattice and suggest new values for the fields at that site.
2. Calculate the difference in energy $\Delta E = E_{new} - E_{old}$ for the move, or update.
3. Draw a random number r$\in$ $[0, 1\rangle$ and accept the new values if $\min\{1, e^{-\Delta E/T}\} > r$.
4. Repeat step 1 to 3 until enough statistics are gathered.

The computational domain is a 3-dimensional lattice with periodic boundary conditions. The charge of the system (reparted among all the lattice sites) couples a scalar field and a three-dimensional vector field. Hence, to each lattice site are associated 4 double precisions real numbers. The local energy E_s at one site s in the lattice is dependent on the nearest neighbor of s, and half of the next nearest neighbors of s. More formally, all the sites *adjacent* to s are involved in the computation of E_s.

Definition 6.1. *Two lattice sites* $s_1 = (x, y, z)$ *and* $s_2 = (t, u, v)$ *are said to be* **adjacent** *if and only if* $(t, u, v) \in \{(x, y, z-1), (x, y+1, z-1), (x+1, y, z-1), (x-1, y, z), (x-1, y+1, z), (x, y-1, z), (x, y+1, z), (x+1, y-1, z), (x+1, y, z), (x-1, y, z+1), (x, y-1, z+1), (x, y, z+1)\}$*, as depicted in Figure 6.1.*

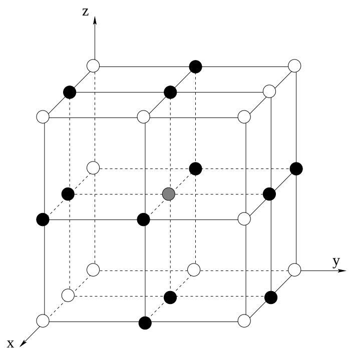


Figure 6.1: Stencil of the LGT model. Black sites are used to update the grey site, they are *adjacent* to the grey site.

6.4 Domain decomposition implementation

In [3], we proposed token-passing algorithms based on DD methods that minimize the amount of communication, i.e. one message per neighboring processor per iteration. Our token-passing algorithms are built upon a classic technique for allowing communication overlap with computation in DD computations. The idea is to partition each local domain into an inner set and an outer set [2, 26, 27]. The inner set is updated while waiting for the boundaries from neighboring processors, and thereafter the outer set is in turn updated. The reception of all the boundaries is liken to the reception of a virtual token, that allows updating the outer set. Thereafter processors hand on the token by sending their updated local boundaries to their neighbors.

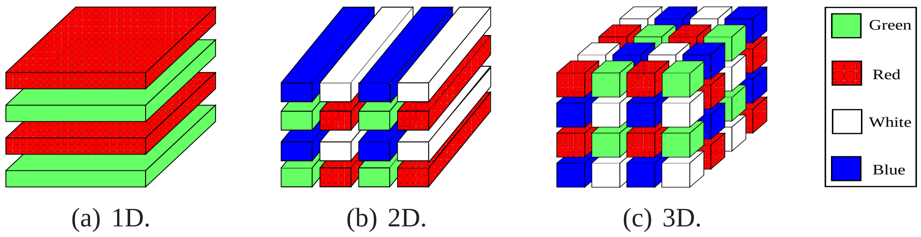


Figure 6.2: The different domain decompositions.

In order to respect the data dependencies imposed by the LGT model (sequential updates of adjacent lattice sites), the processors are sorted into different color sets (see Figure 6.2), such that processors of the same color can update their exterior sites simultaneously. For the 1D case, two colors are necessary and sufficient, whereas four colors are required for the 2D and 3D decompositions. Then, an ordering is established among the colors to orchestrate the updates of the outer sets. For the 1D case, green processors start ahead of the red processors, while the color ordering of 2D and 3D decompositions is 1) green, 2) red, 3) white and 4) blue. Figure 6.3 sketches the parallel execution of the token-passing algorithm based on a 2D decomposition.

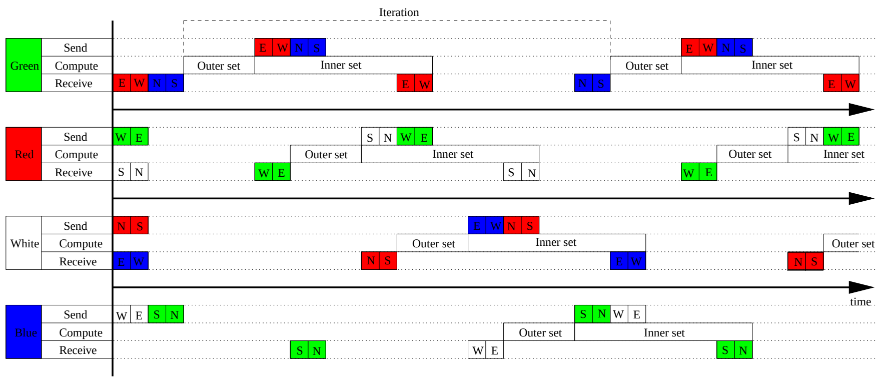
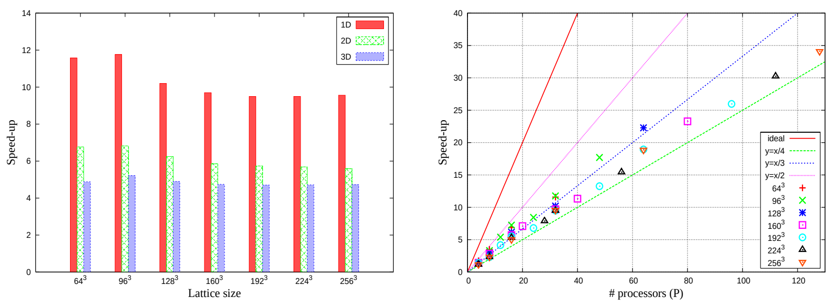


Figure 6.3: Sketch of the token-passing algorithm with a 2D decomposition.

MPI features like persistent requests and derived datatypes have been used for implementing the successive boundary-exchanges. Special care has been taken when posting and completing the communication requests such that the MPI *ready* communication mode could be used. All these decisions contribute to keep the communication overhead to a minimum. Also, we used the *diagonal communication elimination* technique [8, 13], which consists of including ghost cells within messages in order to avoid diagonal communications for exchanging lattice sites located on the edges of the sub-domains.. At last, for the sake of portability, non-blocking requests have been used in order to exploit the inherent computation-communication overlap of the partitioning method, even though many implementations cannot overlap without extra hardware in the form of a communication co-processor.



(a) Speed-up with 32 processors. (b) 1D Speed-up

Figure 6.4: Speed-up of our token-passing algorithms based on DD methods. Reproduced from [3].

Figure 6.4 (a) depicts the respective speed-up of the three different token-passing algorithms when using 32 processors. The better performance of the 1D decomposition over the 2D and 3D decompositions is certainly due to the complicated token round-trip trajectories of the latter decompositions as opposed to the much simpler trajectory for the 1D case. Indeed, the 1D token round-trip imposes only 1 outer set update and 4 messages, as opposed to 3 outer set updates and 8 messages for the 2D case, and 3 outer set updates and 12 messages for the 3D case (see [3] for a thorougher performance analysis). For all the experiments, the token round-trip dominates the total iteration run-time cost, meaning that processors are starving, waiting for the token to arrive. The run-time costs for updating the inner sets were roughly equivalent for the 3 domain decompositions, which means that processors are starving longer under the 2D and 3D decompositions than for the 1D decomposition.

Although, Prieto et al. [26] showed that the separation of the inner and outer set updates may degrade the performance due to the large distance between the memory locations of the exterior sites (causing a poor cache memory exploitation when updating the outer set), we found that our token-passing algorithms were scalable with an efficiency comprised between 0.25 and 0.5 depending on the problem size and number of processors utilized (see Figure 6.4 (b)).

6.5 Master-slave implementation

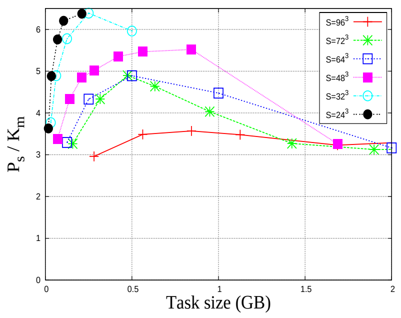
6.5.1 Task partitioning

The sites of the 3-dimensional lattice must be partitioned into disjoint sets to allow for parallel execution. The goal is to enable the processors to work on different parts of the lattice simultaneously. We rely on the same domain decomposition and the same color code than the ones used for the DD implementation of the LGT model (see Figure 6.2), such that blocks of the same color can be processed simultaneously. In our context, each block represents a task to be scheduled by the master.
Depending on the chosen decomposition and the LGT model stencil, different dependencies take place between neighboring blocks of different colors. For the 1D decomposition, each block is dependent on 2 blocks of the opposite color (above and beneath). For the 2D decomposition, each block is dependent on 2 blocks of each of the other colors. At last, for the 3D decomposition, each block is dependent on 4 blocks of each of the other colors.

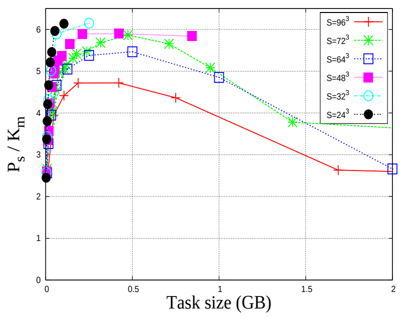
In order to respect the site update dependencies, the master deals with one color at a time. Thus the scheduling overhead on the master node is alleviated by only keeping track of block dependencies from one color to another. To detect block eligibility, the master maintains for each block a dependency variable (integer), as well as pointers to the dependency variables of the adjacent dependent blocks. Initially all the dependency variables are set to the number of dependencies generated by the task partition. Upon reception of a computed block, the variables of all the adjacent blocks are decremented, and if some of them become equal to zero, the corresponding blocks become eligible for computation. In that case, the block pointers are inserted into a FIFO queue holding all the blocks eligible for computation. This mechanisms relieves the master from waiting for the termination of a given color to switch over the next color. Instead, the color transition happens smoothly by delegating blocks as soon as they become eligible for computation.

The master must decompose the global lattice in such a way that there are enough blocks available to the slaves. On the one hand, the number of blocks should be large enough in order to dispose of enough eligible tasks at all times to keep the slaves busy. On the other hand, the master should determine an appropriate task size in order to reduce the overhead incurred by the total amount of communication combined with the post processing of the blocks (copy operations due to the periodic conditions of the 3-dimensional lattice).

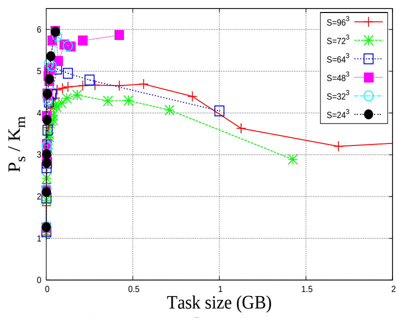
The first thing to determine is which task partitioning scheme gives the best performance. This includes finding the best domain decomposition and the optimal block size. A simple way to compare the different task partitioning consists in estimating the ratio α between the time it takes a slave to process a task, and the time it takes the master to send the task, receive the associated results and post-process the task. The ratio α gives an indication on how scalable is the MS implementation. The larger this ratio is, the better will perform the MS implementation, as it would be able to use more processors. Actually, this ratio gives an indication on the number of slaves that the master can handle, assuming that the slaves have homogeneous computing and communication characteristics.



(a) 1D task partition.



(b) 2D task partition.



(c) 3D task partition.

Figure 6.5: Estimating the α ratio for different problem and task sizes. Note that the task decomposition schemes must deal with lattice size constraints that dictate the sizes and shapes of the blocks. Hence, the more dimensions are used by the task partitioning scheme, the smaller the tasks can be.

Figure 6.5 reports the results of an experiment with 2 processors (a master and a slave), for different problem and task sizes. The experimental values for the different α ratios obtained during this experiment indicate that the master would not be able to handle more than 6 slaves on the test-bed machine, which might seem a low number at first sight. In addition, when problem size increases, the α ratio decreases. For every decomposition, a good task size seems to be situated between 250 and 500 MB.

The 2D decomposition performs slightly better than the other ones, most likely due to the shape of the blocks and consequently to the derived datatypes involved in the communications. Indeed, when delegating a task, the master must extract a block from the global lattice, whose shape depends on the chosen decomposition. The 2D decomposition is a good compromise between few large blocks (1D) and many small blocks (3D).

6.5.2 Selective scheduling

Because our computational domain is decomposed into relatively few tasks that become eligible for computation alternatively throughout the computation, we aim at task throughput maximization instead of total run-time minimization. Our scheduling strategy consists in handing out the tasks one-by-one, in a demand-driven fashion. If several slaves are competing for a task, then the master must decide which one to serve according to a priority scheme.

Since all the tasks are computationally identical, we let $P_s(t)$ denote the time it takes to slave s to process a task at time-step t. Further, it takes $C_s(t)$ time units for the master to send a task to slave s at time-step t, and $C'_s(t)$ time units for the slave s to return the results to the master at time-step t.

Our MPI parallel implementation involves advanced programming techniques such as derived datatypes, non-blocking communications and persistent requests. This complicates the online monitoring of the different communication events. Therefore, for each slave s, we define the *task round-trip* at time step t, noted $R_s(t)$, as follows: $R_s(t) = C_s(t) + P_s(t) + C'_s(t)$, that corresponds to the time it takes for sending a task to slave s plus the time it takes slave s to compute the task plus the time it takes to send the results back to the master. Throughout the computation, the master can monitor the value R_s of each slave in order to make efficient scheduling decisions. Thus, we account for the possible performance fluctuations of both computation and communication resources throughout the computation. Monitoring R_s simply consists in starting a timer right before sending a task to a slave, and stopping the timer when the results have returned.

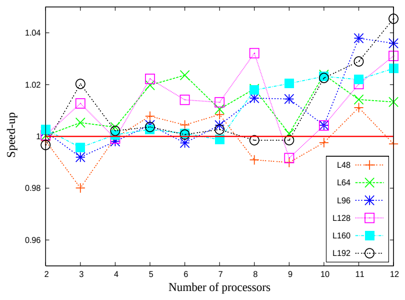
When several slaves are in competition for receiving a task, the master will choose the one with the smallest R_s value. Indeed, no distinction is made between the computation and communication run-time costs relative to a slave, since the tasks are not really independent of each other. It is indeed, important that tasks come back as soon as possible in order to allow other tasks to become eligible for computation.

At the beginning of the execution, all the slaves are given a task, which allows to initialize all the R_s values. Then, as the computation proceeds, the R_s values are updated with the newest value measured by the master. More advanced methods based on averages over the last n values or on performance predictions exist [30], but our simple method gave satisfactorily results.

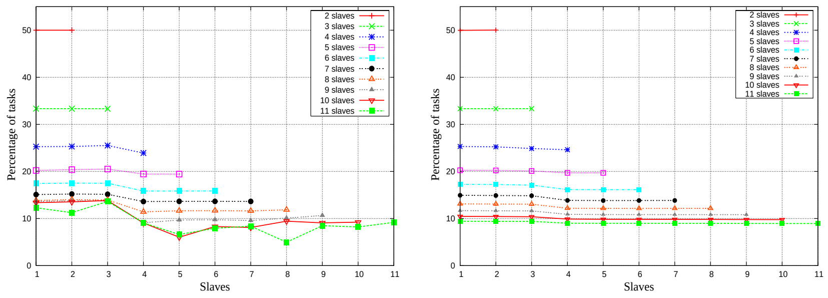
6.5.3 FCFS vs. selective scheduling

To demonstrate the need for priority schemes, we compared our selective scheduling strategy to the FCFS scheme, which works without priorities. Figure 6.6 (a) depicts the ratio of the task throughput of the selective scheme over the task throughput of the FCFS scheme. Clearly, selective scheduling achieves a higher throughput than FCFS in most situations. This phenomenon strengthens as the number of slaves increases, which corroborates our hypothesis that intra-node interferences as well as inter-node communications introduce heterogeneity and variability in the computing environment.

Figures 6.6 (b) and (c) report the task share among the slaves for the selective and FCFS schemes respectively. We observe that for the selective scheme, 3 slaves get a bigger share of the tasks than the others. This is not surprising since each SMP node is composed of 4 processors, meaning that 3 slaves are located on the same node as the master.



(a) Task throughput ratio $\frac{selective}{FCFS}$.



(b) Task share for the selective scheme. (c) Task share for the FCFS scheme

Figure 6.6: Comparison of the selective and FCFS schemes for $L = 128^3$.

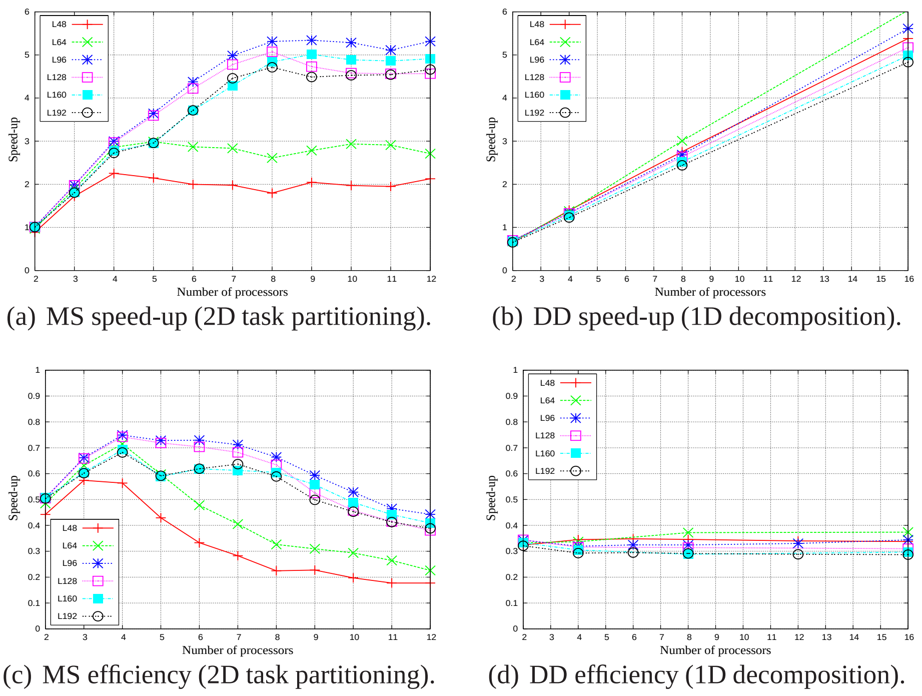
The intra-node communication (shared-memory) being faster than inter-node communication (message passing), the slaves located on the same SMP node as the master will be prioritized if they are in concurrency with other slaves located on a different SMP node because exhibiting a smaller R_s value. This phenomenon is less visible for the FCFS strategy.

Finally, note that the cluster was used in dedicated mode, meaning that no external load other than operating system calls or network contention fluctuations interfered with our application. Consequently, all the nodes have roughly the same computing power which explains the linearity of the curves. Nonetheless, when using a high number of slaves, the selective scheme seems to adapt to some interferences that take place, while the FCFS scheme maintains a fair share of the tasks. Thus, although the system *looks homogeneous*, there are still a certain amount of heterogeneity and variability in the system that degrade the overall performance.

6.6 Domain decomposition vs. master-slave

Figure 6.7 reports the speed-up and efficiency obtained with the selective MS implementation using a 2D task partitioning scheme (Figures (a) and (c)), and with the token-passing algorithm using a 1D decomposition (Figures (b) and (d)). One can observe that the MS implementation scales well up to 6 slaves, and thereafter begins to saturate. This result

conforms with the experiment conducted for determining the appropriate task partitioning schemes (see Section 6.5.1) predicting that the master could not handle more than 6 slaves efficiently.
Note that the MS implementation achieves a perfect speed-up up to 3 slaves (if the master is not accounted). When using more than 3 slaves, the inter-node communications begin to drive the performance away from optimality.
The poor performance of the MS implementation for small problem sizes ($L = 48^3$ and $L = 64^3$) comes from our task partitioning scheme that utilizes blocks of size greater than 250 MB. Thus, for small problem sizes, there were simply not enough independent tasks to feed all the slaves. In such situation, one should use a finer grained task partition scheme, albeit there is a limit on how small a task can be.



(a) MS speed-up (2D task partitioning). (b) DD speed-up (1D decomposition).
(c) MS efficiency (2D task partitioning). (d) DD efficiency (1D decomposition).

Figure 6.7: Comparison of the selective MS and the DD implementations.

As opposed to the MS implementation, the DD implementation is highly scalable, albeit this comes at the expense of a poor efficiency. For a small number of processors, the DD implementation is less efficient than the MS implementation. Hence, it seems that the MS approach is better suited for dealing with our LGT model than parallel algorithms based on DD methods. However, the lack of scalability of the MS implementation makes it useless for large-scale simulations.

6.7 Future work

The natural solution to tackle the lack of scalability of the MS paradigm, is to deploy several masters [20]. A direction for future work would be to design a hybrid approach where the MS paradigm and DD methods would be used in concert. The computational domain would be decomposed among few processors (the masters), but each master would update its sub-domain using the MS paradigm. The number of masters to deploy depends on the problem to be solved as well as on the underlying computing system. For our LGT model and our SMP cluster, a master could manage a SMP node (or span over two SMP nodes). Such hybrid approach combines the benefits of the two paradigms: The MS flexibility with the DD scalability. Interestingly, inter-master load balancing could be tackled at two levels. First, the computational load can be redistributed between masters. This approach makes it possible to use existing load redistribution strategies [4–6, 18, 19, 22]. But a more promising approach would be to handle the load redistribution as a slave redistribution. If a master experiences a lack of computing power from its slaves, it could request additional slaves from other masters. Hence, slaves could be traded between masters on demand. This "computing-power" balancing mechanism is more flexible and practical than traditional load-balancing algorithms, as data would not need to be migrated throughout the computation.

Fault tolerance still becomes easier to handle as one needs only to back-up the master processes. For that matter, note that any slave can act as a master whenever needed. Hence, masters can periodically back-up their data, by sending a copy to one or few slaves that would replace them in case of failure. For such implementations, one could use the Fault Tolerant MPI library (FT_MPI) [11], which offers a range of recovery options other than just returning to some previous check-pointed state. This is especially useful in the case of slave failure since the computation can in principle proceed seamlessly.

6.8 Conclusion

High performance computing systems are no longer stable and fully homogeneous. This greatly complicates the efficient deployment of traditional DD methods, since applications must deal with system heterogeneity, resource performance fluctuations and resource failures. In addition to that, there are certain classes of problems for which DD methods are inappropriate, such as the LGT model presented in this paper. Hence, there are two good reasons for considering an alternative way to DD methods.

In this paper, we study the suitability of the MS paradigm as an alternative to DD methods for implementing LGT models within distributed memory environments. We provide three different MS implementations based on three task partitioning schemes. More importantly we demonstrate, via a comparison between a selective and the FCFS scheduling strategies, that apparently homogeneous systems used in dedicated mode are actually heterogeneous environments subjects to unforeseeable resource performance fluctuations.

Overall, our experimental results reveal that the MS implementation achieves very good efficiency on few processors, but lacks of scalability. In contrast, the DD method is highly scalable, but at the expense of a poor efficiency. The peculiarity of the LGT model, namely the constraining data dependencies, is better handled with a MS implementation than with DD methods. Hence, the MS paradigm is a good candidate for small-scale LGT models with high computation-to-communication ratios. Finally, we discussed a promising future work direction by sketching an hybrid approach that combines the MS paradigm and DD methods for achieving scalable high performance.

[1] E. Babaev, A. Sudbø, and N. W. Ashcroft. A Superconductor to Superfluid Phase Transition in Liquid Metallic Hydrogen. *Nature*, 431:666, 2004.

[2] S. B. Baden and S. J. Fink. Communication overlap in multi-tier parallel algorithms. In *Supercomputing ’98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–20, Washington, DC, USA, 1998. IEEE Computer Society.

[3] C. Banino-Rokkones, J. Amundsen, and E. Smørgrav. Parallelizing Lattice Gauge Theory Models on Commodity Clusters. In *2006 IEEE International Conference on Cluster Computing (CLUSTER 2006), September 25-28 2006, Barcelona, Spain*. IEEE Computer Society, 2006.

[4] O. Beaumont, V. Boudet, and A. Petitet. A Proposal for a Heterogeneous Cluster ScaLAPACK (Dense Linear Solvers). *IEEE Trans. Comput.*, 50(10):1052–1070, 2001.

[5] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert. Matrix Multiplication on Heterogeneous Platforms. *IEEE Trans. Parallel Distrib. Syst.*, 12(10):1033–1051, 2001.

[6] O. Beaumont, A. Legrand, F. Rastello, and Y. Robert. Dense Linear Algebra Kernels on Heterogeneous Platforms: Redistribution Issues. *Parallel Comput.*, 28(2):155–185, 2002.

[7] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov. Adaptive Computing on the Grid Using AppLeS. *IEEE Trans. Parallel Distrib. Syst.*, 14(4):369–382, 2003.

[8] C. Ding and Y. He. A Ghost Cell Expansion Method for Reducing Communications in Solving PDE Problems. In *Supercomputing ’01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 50–50, New York, NY, USA, 2001. ACM Press.

[9] J. Dongarra and A. Lastovetsky. An Overview of Heterogeneous High Performance and Grid Computing. In *Engineering The Grid: Status and Perspective*. American Scientific Publishers, 2006.

[10] A. Espinosa, T. Margalef, , and E. Luque. Automatic Performance Analysis of Master/Worker PVM Applications with Kpi. In *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 47–55, London, UK, 2000. Springer-Verlag.

[11] G. E. Fagg and J. J. Dongarra. Building and Using a Fault-Tolerant MPI Implementation. *Int. J. High Perform. Comput. Appl.*, 18(3):353–361, 2004.

[12] J.-P. Goux, S. Kulkarni, M. Yoder, and J. Linderoth. Master-worker: An enabling framework for applications on the computational grid. *Cluster Computing*, 4(1):63–70, 2001.

[13] F. Gutbrod, N. Attig, and M. Weber. The SU(2)-lattice gauge theory simulation code on the Intel Paragon supercomputer. *Parallel Comput.*, 22(3):443–463, 1996.

[14] T. Hagerup. Allocating Independent Tasks to Parallel Processors: An Experimental Study. *J. Parallel Distrib. Comput.*, 47(2):185–197, 1997.

[15] D. W. Heermann and A. N. Burkitt. *Parallel Algorithms in Computational Science*. Springer-Verlag New York, Inc., New York, NY, USA, 1991.

[16] E. Heymann, M. A. Senar, E. Luque, and M. Livny. Adaptive Scheduling for Master-Worker Applications on the Computational Grid. In *GRID '00: Proceedings of the First IEEE/ACM International Workshop on Grid Computing*, pages 214–227, London, UK, 2000. Springer-Verlag.

[17] W. Janke and R. Villanova. Ising model on three-dimensional random lattices: A Monte Carlo study. *Physical Review B*, 66(13):134208–+, Oct. 2002.

[18] M. Kaddoura, S. Ranka, and A. Wang. Array Decompositions for Nonuniform Computational Environments. *J. Parallel Distrib. Comput.*, 36(2):91–105, 1996.

[19] A. Kalinov and A. Lastovetsky. Heterogeneous Distribution of Computations Solving Linear Algebra Problems on Networks of Heterogeneous Computers. *J. Parallel Distrib. Comput.*, 61(4):520–535, 2001.

[20] T. Kindberg, A. Sahiner, and Y. Paker. Adaptive Parallelism under Equus. In *Proceedings of 2nd International Workshop on Configurable Distributed Systems*, pages 172–182. IEEE, March 1994.

[21] M. Luscher. Solution of the Dirac equation in lattice QCD using a domain decomposition method. *Comput. Phys. Commun.*, 156:209–220, 2004.

[22] A. Mahanti and D. L. Eager. Adaptive Data Parallel Computing on Workstation Clusters. *J. Parallel Distrib. Comput.*, 64(11):1241–1255, 2004.

[23] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of State Calculations by Fast Computing Machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

[24] P. D. Michailidis and K. G. Margaritis. Performance Evaluation of Load Balancing Strategies for Approximate String Matching Application on an MPI Cluster of Heterogeneous Workstations. *Journal of Future Generation Computing Systems*, 19(7):1075–1104, 2003.

[25] A. Morajko, E. César, P. Caymes-Scutari, T. Margalef, J. Sorribes, and E. Luque. Automatic Tuning of Master/Worker Applications. In J. C. Cunha and P. D. Medeiros, editors, *Euro-Par*, volume 3648 of *Lecture Notes in Computer Science*, pages 95–103. Springer, 2005.

[26] M. Prieto, I. M. Llorente, and F. Tirado. Data Locality Exploitation in the Decomposition of Regular Domain Problems. *IEEE Trans. Parallel Distrib. Syst.*, 11(11):1141–1150, 2000.

[27] M. J. Quinn and P. J. Hatcher. On the Utility of Communication-Computation Overlap in Data-Parallel Programs. *J. Parallel Distrib. Comput*, 33(2):197–204, 1996.

[28] E. E. Santos, S. Feng, and J. M. Rickman. Efficient Parallel Algorithms for 2-Dimensional Ising Spin Models. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 135, Washington, DC, USA, 2002. IEEE Computer Society.

[29] E. E. Santos and G. Muthukrishnan. Efficient Simulation Based on Sweep Selection for 2-D and 3-D Ising Spin Models on Hierarchical Clusters. In *IPDPS*, page 229b, 2004.

[30] R. Wolski. Experiences with Predicting Resource Performance On-line in Computational Grid Settings. *SIGMETRICS Perform. Eval. Rev.*, 30(4):41–49, 2003.

Paper 7

Data Layout and Access Transformations for Efficient Stencil Computations
Cyril Banino-Rokkones.
Submitted to *21st ACM International Conference on Supercomputing*,
July 2007, Seattle, USA.

Data Layout and Access Transformations for Cache-Efficient Stencil Computations

Cyril Banino-Rokkones

Department of Computer and Information Science
Norwegian University of Science and Technology
NO-7491 Trondheim, Norway

Abstract

Stencil codes form the basis for a wide range of scientific applications, but unfortunately exhibit a particularly poor memory behavior with respect to processor caches. In this paper, we present transformation techniques that improve the performance of 2D and 3D stencil codes on modern computer architectures. On the one hand, we present theoretical and experimental results that demonstrate how spatial locality is improved by using skewed data layouts. Although skewing the multi-dimensional array seems a fairly traditional approach of altering the memory layout to boost performance, we are not aware of any paper that reports theoretical or practical results on the matter. On the other hand, we present a new technique that improves temporal locality by exploiting the symmetrical property of the stencil kernel. Performance analysis using the PAPI interface show that the techniques presented in this paper considerably reduce the number of L1, L2 and TLB data cache misses, and enable to increase the level of parallelism exposed to the compiler. Overall, our experimental results on 3 modern processors (Intel Pentium 4, AMD Opteron and IBM Power5+) confirm that these techniques yield to substantial performance improvements over the traditional tiling optimization technique.

7.1 Introduction

In the last decade, processor performance has been steadily improving at a much more higher rate (55%) than memory performance (7%) [8]. Most modern CPUs are so fast that memory transfers constitute the practical limitation on processing speed: The CPU spends much of its time stalling, waiting for memory transfers to complete. Modern computer architectures rely on a hierarchical arrangement of memory (or caches) to help bridging that widening gap. However, effectively using caches for numerical applications is a challenging task.

Stencil codes form the basis for a wide range of scientific applications: Iterative solvers, Monte Carlo simulations and image filtering applications all rely on some form of stencil computation. These applications are called *stencil codes* because each element in a multidimensional array is updated with contributions from a subset of its neighbors. Then, for each iteration, the stencil kernel is applied to each element of the array. Stencil codes are among the most time-consuming routines of the aforementioned applications, and that is why it makes sense to aspire for ultimate performance.

Unfortunately, stencil codes exhibit a particularly poor memory behavior with respect to processor caches. This poor behavior is imputed to the fact that each array element is accessed a small, constant number of times per iteration, which simply amounts to the number of points in the stencil kernel. For large problem sizes, array elements must be brought into cache several times per iteration, dramatically degrading the overall performance. Reorganizing these computations in order to efficiently utilize the memory hierarchy has been the subject of a wealth of research.

Cache blocking or *tiling* is the standard transformation technique which improves locality by moving reuses to the same data closer in time [5, 11–13, 15, 16, 18, 21, 23, 26]. However, the evolution of memory system features (e.g. large on-chip caches combined with automatic prefetch) seems to reduce the effectiveness of traditional cache blocking optimizations [9]. In the worst case scenario, cache blocking transformations may even interfere with prefetch policies, resulting in performance degradation. Prefetching (both in hardware and software) improves the performance of long stride-1 accesses, while discontinuities in access patterns (exhibited by transformations like tiling) may counter the benefits of prefetching. In contrast, the transformations presented in this paper improve performance while performing contiguous data accesses.

The first contribution of this paper is the theoretical and experimental analysis of skewed data layouts for improving the spatial locality of stencil codes. To the best of our knowledge, this is the first paper that presents theoretical as well as experimental results on this memory alteration technique. We first define the stencil *footprint memory distance*, that represents the longest distance in memory between two array elements used by the stencil kernel. Then we demonstrate - and quantify - that 2D and 3D skewed data layouts exhibit a much lower average stencil *footprint memory distance* than traditional row-major or column-major storage orders. Having the array elements used by the stencil close to each other in address space allows to better utilize the cache capacities and hence to reduce the number of cache misses.

The second main contribution of this paper is a new data access transformation, called *stencil decomposition*, that improves the temporal locality of stencil codes. This loop transformation technique, based on loop fission and loop fusion, decomposes the stencil kernel into two micro-stencils, such that the updates are now performed in two passes. The partial results obtained from the first pass are added to the partial results obtained from the second pass to produce the final results. The two micro-stencil updates are fused within the innermost loop, but applied to different data in order to exploit the symmetrical properties of the original stencil, by computing the mutual contributions of two elements that do not have spatial locality.

The Jacobi and Gauss-Seidel iterative methods are used to evaluate and compare our techniques with the ones found in the literature. Although these methods have been replaced by more efficient schemes such as multigrid, they nevertheless play an important role because they are building blocks of the advanced methods, and because they have similar computational properties [13]. Our transformation techniques are evaluated and compared on three modern processors which are Intel Pentium 4, AMD Opteron and IBM Power5+. Wall-time measurements combined to hardware performance counter analysis are used to understand and interpret our results. We used the PAPI interface to hardware performance counters [1] to collect these data.

The rest of this paper is arranged as follows. Section 7.2 presents the challenges imposed by traditional stencil codes as well as the iterative methods implemented in this study. Section 7.3 reviews previous work and presents how our contributions fit in with previous studies. Sections 7.4 and 7.5 present respectively the data layout and data access transformations proposed for 2D and 3D stencil codes. Section 7.6 evaluates our transformations through actual performance measurements and discusses our results with the help of information collected with PAPI. Finally, Section 7.7 summarizes our contributions, states our conclusions and discusses possible future work directions.

7.2 Stencil computations

An important class of scientific applications rely on solving partial differential equations (PDEs) using finite differencing techniques [17]. Consider for instance the Laplace equation:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \tag{7.1}$$

To solve this iteratively, u is discretized with N points in the x and y directions, and for each new iteration $n+1$, the approximate value of the solution $u_{i,j}^{n+1}$ is computed as:

$$u_{i,j}^{n+1} = \frac{1}{4}(u_{i-1,j}^{n} + u_{i+1,j}^{n} + u_{i,j-1}^{n} + u_{i,j+1}^{n}) \tag{7.2}$$

Equation 7.2 represents the *Jacobi* iteration kernel which consists of a 4-point stencil in two dimensions. Another classical method is the *Gauss-Seidel* method, which makes use of updated values of u on the right hand side of Equation 7.2 as soon as they become available. Thus, the averaging is done in place instead of being copied from an earlier iteration to a later one, as depicted below.

$$u_{i,j}^{n+1} = \frac{1}{4}(u_{i-1,j}^{n+1} + u_{i+1,j}^{n} + u_{i,j-1}^{n+1} + u_{i,j+1}^{n}) \tag{7.3}$$

Conventional wisdom argues that the innermost loop of a loop nest should step through the array sequentially in memory. This is often called *stride-1 indexing* [26]. Most previous studies assuming row-major storage order present the code for the 2D Gauss-Seidel method as depicted in Figure 7.1.

```
  int i,j;
  double A[N][N];

  for(i=1;i<N-1;i++)
    for(j=1;j<N-1;j++)
      A[i][j]=0.25*(A[i-1][j]+A[i][j-1]+

A[i][j+1]+A[i+1][j]);
```

Figure 7.1: 2D Gauss-Seidel stride-1 indexing with row-major storage order.

Although the Gauss-Seidel scheme requires less data storage than the Jacobi scheme, it imposes however more constraints on the execution order, and exhibits less opportunities for optimization through reordering [13]. To sidestep this limitation, a Red-Black checkerboard algorithm has been implemented, which accesses all the “red” elements (where sum of coordinates is even) to compute values for the “black” elements (where sum of coordinates is odd), then it does the other way around using black elements to update red elements [17, 19, 23].

Historically, PDE solvers have focused on 2D domains. But as more computing power became available, scientists became interested in 3D domains as well. The 3D Jacobi scheme reads thus:

$$u^{n+1}_{i,j,k} = \frac{1}{6}(u^n_{i-1,j,k} + u^n_{i+1,j,k} + u^n_{i,j-1,k} + u^n_{i,j+1,k} + u^n_{i,j,k-1} + u^n_{i,j,k+1}) \tag{7.4}$$

For large problem sizes, array elements must be brought into cache multiple times per iteration, dramatically degrading the overall performance. This problem arises more often in 3D codes than in 2D codes because more data need to be held in cache to fully exploit group reuse.

There are two types of *locality* that can be exploited to improve performance when implementing stencil codes [25]. There is *spatial* locality when accessing neighboring points (in address space), and there is *temporal* locality when array elements are reused several times before being evicted from the cache. Roughly speaking, spatial locality deals with the data layout, i.e. how the multidimensional array is mapped into address space, while temporal locality deals with the ordering of the updates.

7.3 Previous work

A wealth of optimization techniques have been proposed to improve memory hierarchy performance. Kowarschik and Weiß [11] and Wolfe [26] provide overviews of such optimization techniques. Loop transformation techniques target temporal locality by modifying the program's iteration space. Loop interchange (or more generally loop permutation) modifies the order of selected loops in a loop nest [22]. Loop fusion fuses two loops that have the same iteration space traversal into a single loop, while loop fission does the opposite [14]. Loop alignment and loop skewing are proposed by Zhao and Kennedy [27] to reduce the memory storage required by stencil codes similar to the Jacobi method. The authors show that these transformations can achieve the asymptotically minimal memory allocation for this kind of stencil computations.

Tiling is the standard loop transformation technique for improving temporal reuse in cache. Tiling reduces the working sets by grouping the updates into rectangular blocks that are processed one after another, in order to reduce capacity misses. However, modern caches have limited set-associativity, and tiling can suffer from considerable conflict misses. To reduce conflict misses, copying and padding techniques have been proposed [5, 12, 18, 21]. Weiß et al. [23] apply tiling and padding transformations for 2D and 3D stencil codes, while Rivera and Tseng [19] follow on this work by developing compiler optimizations that automate the search of pads and tile sizes. The authors propose a *partial blocking* scheme for 3D stencil codes in order to reduce non contiguous data accesses. This results in a stack of 2D slices in the unblocked dimension.

Several studies have emphasized the need to take into account the Translation Look-Aside Buffer (TLB) when optimizing for performance [5, 15, 16]. As problem size increases, TLB thrashing occurs and can considerably degrade performance. Mitchell et al. [15] derive multi-level cost functions that pays attention simultaneously to cache and TLB performance for guiding the optimal choice of tile size and shape. Coleman and McKinley [5] suggest that tile sizes need to be constrained such that the number of non consecutive elements accesses is smaller than the number of page table entries in the TLB.

Although tiling is a well understood technique that proved to significantly improve reuse, a recent study of Kamil et al. [9] show that the evolution of memory system features (e.g. large on-chip caches combined with automatic prefetch) seems to reduce the effectiveness of traditional cache blocking optimizations. The authors stress the issue that non contiguous data access may interfere with prefetch policies, resulting in a performance decrease.

A way to reduce non contiguous data accesses and to avoid conflict misses in tiled code is to change the data layout in order to match the data access pattern. Chatterjee et al. [2, 3] use recursive array layouts and different space-filling curves for fast matrix multiplication. Kandemir et al [10] make use of hyperplane theory for memory layout representation. The array references in a given loop nest are modeled by a coefficient matrix and offset vector [26] in order to detect suitable memory layouts expressed by hyperplanes. The ATLAS project [24] uses block data layout with tiling to exploit temporal and spatial locality. To promote portability, the selection of the optimal tile size is done empirically by running several off-line tests. In contrast, Park et al. [16] analyze the intrinsic TLB and cache performance using tiling and block data layout, in order to derive a block size selection algorithm.

Bandwidth and profile reduction problems [6, 20] are closely related to the problem studied in this paper. Applied to matrices, the bandwidth minimization problem consists in finding a permutation of the rows and columns of a sparse matrix so as to minimize the distance b of any nonzero entry from the center diagonal. This problem is very close to the profile minimization problem that consists in minimizing the sum of the maximum distances from the diagonal. Bandwidth minimization is important in solving linear systems, because direct methods such as Gaussian elimination can be performed in $O(nb^2)$ on matrices of bandwidth b, which is very valuable when $b << n$ [20]. On the other hand, profile minimization leads to a reduction of the amount of space needed for storing the sparse matrix [6]. The bandwidth and profile minimization problems are known to be NP-complete [6], and one of the most popular heuristics for bandwidth and profile minimizations is the one of Gibbs et al. [7]. At last, layout problems arise also within parallel processing settings where the amount of communication should be minimized in order to obtain good speed-up. The graph partitioning problem within parallel processing settings consists in distributing the nodes of a graph onto a set of processors in such a way that the number of edges stretching over two processors is minimal. Chen and Chang [4] propose to utilize a 2D skewed data layout for minimizing data communication over processors for distributed memory multicomputers.

The contributions of this paper fit in with previous work as follows. We adhere to the idea of changing memory data layouts in order to match data access patterns that improve memory hierarchy performance [3, 4, 10, 16, 24]. We consider the use of skewed data layouts for improving spatial locality. Although skewing the multi-dimensional array seems a fairly traditional approach of altering the memory layout to boost performance, we are not aware of any paper that reports theoretical nor practical results on the matter. Skewed data layouts reduce the average distance between simultaneously referenced memory locations, which intrinsically applies to multiple levels of the memory hierarchy such as cache and TLB performance [15, 16]. While, bandwidth and profile minimization heuristics were originally designed for improving the performance of direct methods and for reducing storage requirements, this paper demonstrates that similar concepts allow to improve the memory traffic performance of iterative methods as well. Analogously to Chen and Chang [4], we exploit the properties of skewed data layouts, but for different purposes. While Chen and Chang target the minimization of data communication over processors for distributed memory multicomputers, we attempt to optimize stencil codes for cache efficient computations. Similarly to McKinley et al. [14], we present a new data access scheme that combines loop fission and loop fusion in order to improve temporal locality. The stencil kernel is split in two halves, and the updates are now performed in two passes. We term this approach *stencil decomposition*. Finally, it is important to understand that previous studies, such as the one of Kandemir et al. [10], that target data layout optimization by utilizing a coefficient matrix and an offset vector to represent the data access pattern of a loop nest, will not be able to detect the benefits of skewed data layouts for the iterative methods studied in this paper. Indeed, in the Gauss-Seidel scheme, the order in which the grid points are updated in the sequential algorithm is not fundamental for the solution, since it is an approximate method that iterates until convergence. Hence, a different ordering of the grid points, will not affect the quality of the solution, as long as the grid points are equally visited and updated. For this reason, representing these iterative methods by a loop nest may be misleading because it intrinsically implies an ordering scheme.

7.4 Data layout transformations

The transformations presented in this Section aim at reducing the average distance in memory between the array elements involved in a stencil update. We term such distance the stencil *footprint memory* distance. Let us define a *group* $G(u)$ as the set of the array elements contributing to the update of element u. For instance, we get for 2D Gauss-Seidel, $G(u_{i,j}) = \{u_{i-1,j}, u_{i+1,j}, u_{i,j-1}, u_{i,j+1}\}$. Let $m(u)$ be the memory location of element u. The aim of using skewed data layouts is to reduce the average stencil footprint memory distance:

$$\sum_u (\max\{m(v) | v \in G(u)\} - \min\{m(v) | v \in G(u)\})$$

. In plain words, one aims to group together in memory the elements belonging to the same group. For notational convenience, the transformation techniques presented throughout the rest of this paper, are reported for symmetrical domains, i.e $N \times N$ in 2D and $N \times N \times N$ in 3D, albeit these techniques extend to non symmetrical domains as well.

7.4.1 2D skewed data layout

The average footprint memory distance of the row-major - or column-major - storage order is equal to $2N$. This means that the cache only needs to be able to hold two rows of the array to fully exploit group reuse. The evolution of memory system features such as larger on chip caches and prefetch policies, leaves small place for tiling improvements of 2D codes [9, 19]. The 2D skewed data layout studied in this paper pushes the limits of tiling improvements even further.

To ease the presentation, we adopt the approach of Kandemir et al. [10] who express memory data layout with hyperplanes. Briefly, in a *m*-dimensional space, a hyperplane can be defined as a set of tuples $\{(a_1, a_2, \ldots, a_m) \mid g_1 a_1 + g_2 a_2 + \cdots + g_m a_m = c\}$, where the row vector $\bar{g}^T = (g_1, g_2, \ldots, g_m)$ is composed of rational numbers g_i called hyperplane coefficients and c is a rational number called hyperplane constant. Two array elements represented by coordinate vectors $\bar{c_1}$ and $\bar{c_2}$ are said to belong to the same hyperplane if $\bar{g}^T \bar{c_1} = \bar{g}^T \bar{c_2}$. For example the hyperplane vector $(1, 0)$ indicates that two array elements belong to the same hyperplane as long as they have the same value for the row index, which amounts to row-major storage order. The 2D skewed data layout is expressed by the hyperplane vector $(1, 1)$, i.e. two array elements belong to the same hyperplane if the sums of their coordinates are equal. As an example, elements with coordinates (2,4) and (3,3) belong to the same hyperplane. Hyperplanes are linearly mapped into address space in increasing order of their hyperplane constants. Hence, the array elements belonging to hyperplane H_p with constant $c = p$ are stored before the array elements belonging to the hyperplane H_{p+1}. Within a hyperplane, we arbitrarily store the elements by increasing row index as depicted in Figure 7.2. The data access pattern when updating the array elements must now follow the dashed line depicted in Figure 7.2 in order to perform stride-1 access. This issue will be discussed in Section 7.5. For now, let $\Delta(L, N)$ be the average footprint memory distance of a given data layout L for a $N \times N$ data array.

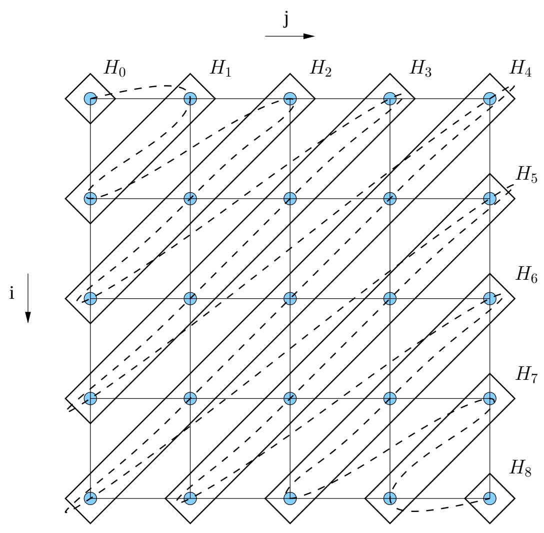


Figure 7.2: 2D Skewed data layouts for $N = 5$.

Property 7.1. *The average footprint memory distance of the 2D skewed data layout is*

$$\lim_{N\to+\infty} \Delta(SK, N) = \frac{4}{3}N$$

.

Proof. Let us decompose the set $\mathcal{P}$ of all the hyperplanes into three disjoint subsets $\mathcal{P}_1, \mathcal{P}_2$ and $\mathcal{P}_3$ such that $\mathcal{P}_1 = \{H_p \mid p \in [0, N-2]\}$, $\mathcal{P}_2 = H_{N-1}$, and $\mathcal{P}_3 = \{H_p \mid p \in [N, 2N-2]\}$. In other words, $\mathcal{P}_1$ comprises all the array elements located in the upper triangular region of the array, $\mathcal{P}_2$ comprises the elements located on the main anti-diagonal of the array, and $\mathcal{P}_3$ comprises the elements located on the lower triangular region of the array.

Consider a pair (H_p, H_{p+1}) of consecutive hyperplanes belonging to $\mathcal{P}_1$. The number of elements of H_{p+1} is one more than the number of elements of H_p. The cardinal numbers of the hyperplanes belonging to $\mathcal{P}_1$ can therefore be represented by the sequence of consecutive integers $1, 2, 3, 4, \ldots$, i.e. $card(H_p) = p + 1, \forall p \in [0, N-2]$. Assume that the stencil kernel is applied to element $u_{i,j} \in H_p \subset \mathcal{P}_1$. The trailing reference $u_{i-1,j}$ of the stencil kernel is distant from element $u_{i,j}$ by $card(H_p)$ locations, while the leading reference $u_{i+1,j}$ of the stencil kernel is distant by $card(H_{p+1})$ locations. Since no computation is performed on the array boundaries, the sum of the footprint memory distances of the updated elements belonging to H_p is equal to $(p-1)(2p+3), p \in [2, N-2]$. Consequently, the sum s_1 of the footprint memory distances of the updated elements belonging to $\mathcal{P}_1$ is given by $s_1 = \sum_{p=2}^{N-2}(p-1)(2p+3)$. Assume now that the stencil kernel is applied to element $u_{i,j} \in \mathcal{P}_2$. In that case, both the trailing and leading references are distant by $card(H_{N-1}) = N$ locations. Since there are $(N-2)$ elements to be updated on the main anti-diagonal, the sum s_2 of the footprint memory distances of the updated elements belonging to $\mathcal{P}_2$ is given by $s_2 = 2N(N-2)$. Finally, by symmetry we have $s_3 = s_1$. Since there are $(N-2)^2$ elements to be updated, we get:

$$\Delta(SK, N) = \frac{2N(N-2) + 2\sum_{p=2}^{N-2}(p-1)(2p+3)}{(N-2)^2} = \frac{\frac{4}{3}N^3 - 3N^2 - \frac{13}{3}N + 10}{N^2 - 4N + 4}$$

□

7.4.2 3D skewed data layout

When moving to 3D codes, and assuming row-major storage order, the cache needs now to be able to hold two planes of the array to fully exploit group reuse. Hence, for row-major and column-major storage orders, the average footprint memory distance becomes $\Delta(RM, N) = 2N^2$. The 3D skewed data layout is expressed by the couple of hyperplane vectors $(1,1,1)$ and $(1,0,0)$, i.e. two array elements have spatial locality if the sum of their coordinates are equal *and* if their first coordinate (plane index) are equal. To avoid confusion, we speak of macro $(1,1,1)$ and micro $(1,1,1) \cup (1,0,0)$ hyperplanes. As an example elements $(3,2,4)$ and $(3,1,5)$ belong to the same micro hyperplane, while the elements $(3,2,4)$ and $(2,2,5)$ belong to the same macro hyperplane, but to different micro hyperplanes. Macro hyperplanes are linearly mapped into address space in increasing order of their hyperplane constants as depicted in Figure 7.2. Within a macro hyperplane, micro hyperplanes are linearly mapped into address space in increasing order of their hyperplane constants (i.e. by increasing plane index). And finally, within micro hyperplanes, array elements are arbitrarily mapped into address space by increasing row index. For instance, element $(2,3,4)$ is stored before element $(2,4,3)$ but after element $(5,1,2)$.

Property 7.2. *The average footprint memory distance of the 3D skewed data layout is*

$$\lim_{N\to+\infty} \Delta(SK, N) = \frac{11}{10}N^2$$

.

Proof. For even values of N, let us decompose the set S of all the macro hyperplanes into four disjoint subsets $\mathcal{P}_1, \ldots, \mathcal{P}_4$ such that $\mathcal{P}_1 = \{H_p \mid p \in [0, N-1]\}$, $\mathcal{P}_2 = \{H_p \mid p \in [N, \frac{3}{2}N-2]\}$, $\mathcal{P}_3 = \{H_p \mid p \in [\frac{3}{2}N-1, 2N-3]\}$ and $\mathcal{P}_4 = \{H_p \mid p \in [2N-2, 3N-3]\}$. For odd values of N, symmetrical reasons require to divide the 3D array using five subsets, the one in the middle containing only one macro hyperplane. Due to space limitation reasons, we do not expose the case where N is odd, as in both cases $\Delta(SK, N)$ tends towards the same limit.

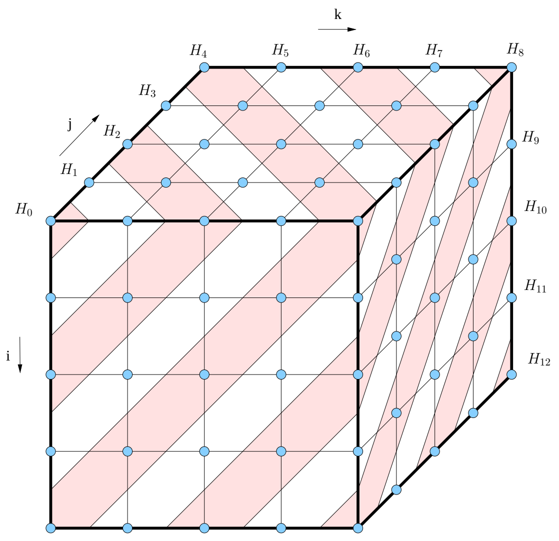


Figure 7.3: 3D Skewed data layouts for $N = 5$.

Consider a pair (H_p, H_{p+1}) of consecutive hyperplanes such that $p \in [0, N-1]$. The number of elements in each dimension of H_{p+1} is one unit longer than the number of elements in each dimension of H_p. Therefore the cardinal numbers of such hyperplanes can be represented by the sequence of triangular numbers $1, 3, 6, 10, 15, 21, 28, 36, 45, 55, \ldots$, i.e. $card(H_p) = \binom{p+2}{2}, p \in [0, N-1]$. Due to the 3D cube topology, the cardinal of the hyperplanes $H_p, p \in [N, \frac{3}{2}N - 2]$ cannot be represented by triangular numbers. Indeed, the cardinal of such hyperplanes corresponds to a triangular number that has been truncated by $(p + 1 - N)$ elements in each dimension. That is to say, each hyperplane $H_p, p \in [N, \frac{3}{2}N - 2]$ comprises $\binom{p+2}{2} - 3\binom{p+2-N}{2}$ elements. Hence, the trailing reference $T(p)$ of the stencil kernel applied to elements belonging to the hyperplane H_p is given by:

$$T(p) = \begin{cases} \binom{p+2}{2}, \text{ if } p \in [3, N-1] \\ \binom{p+2}{2} - 3\binom{p+2-N}{2} + (p + 1 - N), \text{ if } p \in [N, \frac{3}{2}N - 2] \end{cases}$$

Note that the quantity $(p + 1 - N)$ is added for hyperplanes $H_p, p \in [N, \frac{3}{2}N - 2]$ because stepping to the trailing reference in address space involves crossing only 2 of the 3 dimensions of the hyperplane. Similarly, the leading reference $L(p)$ of the stencil kernel applied to elements belonging to the hyperplane H_p is equal to:

$$L(p) = \begin{cases} \binom{p+3}{2}, \text{ if } p \in [3, N-2] \\ \binom{p+3}{2} - 3\binom{p+3-N}{2} + (p + 2 - N), \text{ if } p \in [N-1, \frac{3}{2}N - 2] \end{cases}$$

Since no computation is performed on the array boundaries, the sum s_1 of the footprint memory distances of the updated elements belonging to $\mathcal{P}_1$ is given by:

$$s_1 = \sum_{p=3}^{N} \binom{p-2}{2} (T(p) + L(p))$$

And the sum s_2 of the footprint memory distances of the updated elements belonging to $\mathcal{P}_2$ is given by:

$$s_2 = \sum_{p=N+1}^{\frac{3}{2}N-2} \left[\binom{p-1}{2} - 3\binom{p+1-N}{2} \right] (T(p) + L(p))$$

Due to symmetrical reasons we have $s_3 = s_2$ and $s_4 = s_1$. Since there are $(N-2)^3$ elements to be updated, we get:

$$\Delta(SK, N) = \frac{2(s_1 + s_2)}{(N-2)^3} = \frac{\frac{11}{10}N^5 - \frac{55}{12}N^4 + 2N^3 + \frac{103}{12}N^2 + \frac{9}{10}N - 14}{N^3 - 6N^2 + 12N - 8}$$

$\square$

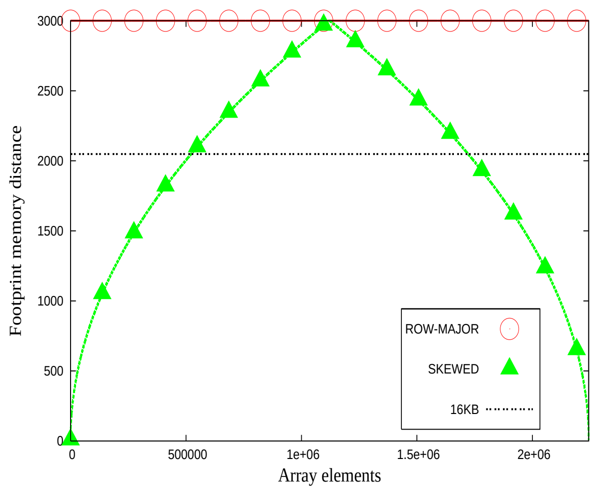
Properties 7.1 and 7.2 show that skewed data layouts improve spatial locality by a factor of 1.5 for 2D and $1.8\bar{1}$ for 3D codes against row-major storage orders. For illustration, Figure 7.4 plots the footprint memory distance for each updated array element (as one steps sequentially in address space) for the skewed and row-major data layouts. For problem sizes where two array lines do not fit in cache, the 2D skewed data layout will better utilize the cache, resulting in less capacity and cache conflict misses, while the 3D skewed data layout may even be able to fully exploit group reuse as opposed to the row-major storage order. This comes from the shape of the macro hyperplanes involved in the 3D skewed data layout, which are composed of hexagons framed by triangles as depicted in Figure 7.3.

7.5 Data access transformations

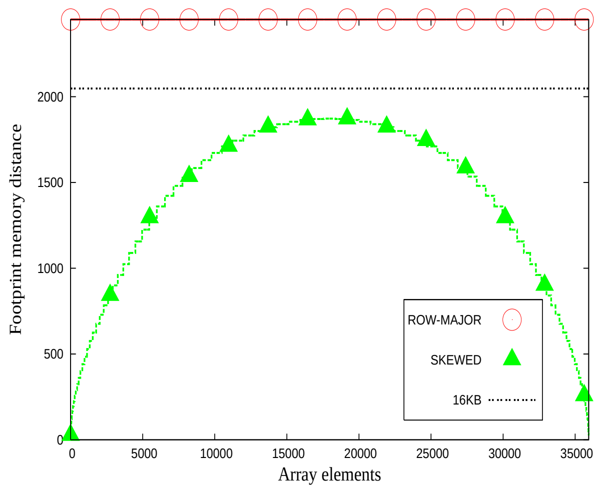
The data access transformations presented in this section will be illustrated with 2D examples only, albeit these techniques extend to multidimensional arrays as well. The reason for this is that the 3D skewed data layout requires a more complicated index manipulation, that we do not report in order to alleviate the presentation.

7.5.1 Skewed stride-1 indexing

When considering skewed data layouts, one must perform index manipulation by hand in order to hit the correct memory locations. A direct implementation would be to provide an address computation function that maps the array locations to memory locations. Such function should be bijective (different array elements map to different memory locations), its image should be dense (there are no holes in the memory footprint of the array), and it



(a) 2D, N=1500



(b) 3D, N=35

Figure 7.4: Footprint memory distance for each updated array element as one steps sequentially in memory.

should be easily computable [2]. For the 2D skewed data layout such function may look like this:

$$f(i,j) = \begin{cases} \binom{i+j+1}{2} + i, & \text{if } (i+j) < N \\ N^2 - f(N-i-1, N-j-1), & \text{otherwise} \end{cases}$$

However, computing the address function for every array element referenced incurs a significant overhead [2]. Instead, one can improve the performance by exploiting some knowledge about the 2D skewed data layout structure. The 2D skewed stride-1 indexing scheme for the Gauss-Seidel method is depicted in Figure 7.5. Note that the code is decomposed into 3 main blocks, which correspond to the 3 disjoint hyperplane sets $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$ presented in Section 7.4.1. For 3D skewed data layouts, the code is decomposed into 5 main blocks.

```
int i,j,base;
double A[N*N];
base= 1;
for(i=2;i<N-1;i++){
  base+=i;
  for(j=base+1;j<base+i;j++)
    A[j]=0.25*(A[j-i-1]+A[j-i]+
               A[j+i+1]+A[j+i+2]);
}

base+=i;
for(j=base+1;j<base+i;j++)
  A[j]=0.25*(A[j-i-1]+A[j-i]+
             A[j+i]+A[j+i+1]);
for(;i>1;i--){
  base+=i+1;
  for(j=base+1;j<base+i-1;j++)
    A[j]=0.25*(A[j-i-1]+A[j-i]+
               A[j+i-1]+A[j+i]);
}
```

Figure 7.5: 2D skewed stride-1 indexing for the Gauss-Seidel iterative method.

7.5.2 Stencil decomposition

Temporal locality can be improved by exploiting the symmetrical properties of the stencil kernel. Consider two array elements which do not have spatial locality, but which belong to the same group G (i.e. that contribute in each other updates). When updating the first element, both elements must be loaded into cache. In turn, when updating the other element, both elements must be loaded into cache once more. Based on this observation, temporal locality can be improved by computing the mutual contributions of the two elements when both of them reside in cache, requiring thus the two elements to be simultaneously present in cache only once. The original stencil kernel S is then *decomposed* in two micro-stencils S_1 and S_2, and the updates will now be performed in two passes. The partial results obtained from the first pass using S_1 must be stored while waiting for summation with the partial results issued from the second pass using S_2. For Jacobi-like iterative methods, i.e. that make use of additional storage, partial results are simply stored in the second array. But for iterative methods such as Gauss-Seidel that

```
int i,j,base;
double A[N*N];

A[4]=0.25*(A[1]+A[2]);
base = 1;
for(i=2;i<N-1;i++){
  base+=i;
  for(j=base+1;j<base+i;j++){
    A[j]+=0.25*(A[j+i+1]+A[j+i+2]);
    A[j+i+1]=0.25*(A[j-1]+A[j]);
  }
  A[j+i+1]=0.25*(A[j-1]+A[j]);
}

base+=i;
j=base+1;
A[j]+=0.25*(A[j+i]+A[j+i+1]);
for(j=base+2; j<base+i;j++){
  A[j]+=0.25*(A[j+i]+A[j+i+1]);
  A[j+i]=0.25*(A[j-1]+A[j]);
}

for(;i>2;i--){
  base+=i+1;
  j=base+1;
  A[j]+=0.25*(A[j+i-1]+A[j+i]);
  for(j=base+2;j<base+i-1;j++){
    A[j]+=0.25*(A[j+i-1]+A[j+i]);
    A[j+i-1]=0.25*(A[j-1]+A[j]);
  }
}
```

Figure 7.6: Stencil Decomposition for 2D skewed data layout.

work only with a single array, we would like to avoid resorting to additional storage, such that the contributions obtained when applying the first micro-stencil would be stored in-place. However, the data dependencies exhibited by such methods impose a careful stencil decomposition, as well as a careful micro-stencil ordering. Depending on the data layout, different decomposition schemes will take place. One must ensure that temporary results (obtained with S_1) are not referenced by neighboring array elements when updating. Our implementations of the stencil decomposition schemes for skewed and row-major data layouts are depicted in Figures 7.6 and 7.7. However, although stencil decomposition improves temporal locality, it nevertheless comes at the expense of additional memory references and floating point operations.

```
int i,j;
double A[N][N];

for(j=1;j<N-1;j++)
  A[1][j]=0.25*(A[0][j]+A[1][j+1]);

for(i=1;i<N-1;i++)
  for(j=1;j<N-1;j++){
    A[i][j]+=0.25*(A[i][j-1]+A[i+1][j]);
    A[i+1][j]=0.25*(A[i][j]+A[i+1][j+1]);
  }

for(j=1;j<N-1;j++)
  A[N-2][j]+=0.25*(A[N-2][j-1]+A[N-1][j]);
```

Figure 7.7: Stencil Decomposition for 2D row-major data layout.

7.6 Experimental results

7.6.1 Methodology

The performance of the transformations considered in this study (see Table 7.1) is evaluated for the Jacobi and Gauss-Seidel methods. All the transformations have been implemented in C. The problem size varied over a range of values, such that the L2 data cache would be able to preserve some group reuse for small problem sizes, but not for large problem sizes. For 2D problems, we let $N \in [100, 5000]$ in steps of 100, and for 3D problems, we let $N \in [5, 350]$ in steps of 5. The number of iterations was arbitrarily fixed to 200 in order to highlight differences between the different transformations while keeping measurement times relatively low. The processors were exclusively dedicated to our application, which reduces external interferences to operating system fluctuations. The performance curves presented in this paper correspond to the average values over 3 runs. Finally, the processor characteristics of interest for the scope of this study are depicted in Table 7.2.

The performance of the different loop transformations is evaluated with wall-time measurements. However, hardware performance counters are used in order to understand and interpret our results. These data are collected using the PAPI portable interface to hardware performance counters [1]. We are interested in 5 quantities which are: Execution time, number of L1, L2 and TLB data cache misses, and finally the number of cycles stalled on any resources. The four last quantities correspond to the PAPI_L1_DCM, PAPI_L2_DCM, PAPI_TLB_DM and PAPI_RES_STL native events respectively. The performance measurements have been normalized with respect to the *RM Stride-1* implementation, and correspond hence to speed-up improvements over the latter method.

7.6.2 Results and interpretation

The experimental results of this study are reported in Figures 7.9, 7.10 and 7.11. Unfortunately, PAPI information was available only for the Opteron processor, but the performance trends depicted in Figure 7.9 are quite similar for the three processors. Due to

Transformations	J	G-S
RM Stride-1 : Row-major stride-1 indexing (see Figure 7.1)	X	X
RM Decomposed: Row-major decomposed stride-1 indexing (see Figure 7.7).	X	X
RM Tiled 32: Row-major partial blocking by Rivera and Tseng [19]. An exhaustive search revealed that tile size 32 for the innermost loop gives the best results.	X	X
SK Stride-1: Skewed stride-1 indexing (see Figure 7.5).	X	X
SK Decomposed: Skewed decomposed stride-1 indexing (see Figure 7.7).	X	X
RM Temp Buff: Inspired by Zhao and Kennedy [27], we restrict the use of additional storage to a minimum (i.e. to a single row for 2D codes and to a single plane for 3D codes). The aim is to better exploit cache capacity. The 2D implementation is depicted in Figure 7.8.	X	
RM Temp Buff tiled 32: Partial blocking applied to the *RM Temp Buff* transformation.	X	
RM Skewed: Loop skewing with row-major storage order to expose parallelism to the compiler. Note that the data access is identical to the one adopted by *SK Stride-1*, but applied to a row-major storage order.		X
RM Red-Black: Row-major Red-Black ordering with row-major storage order to expose parallelism to the compiler. In order to avoid bringing data into cache multiple times (when the array size exceeds the cache size), black points in each row (or plane for 3D) are updated immediately after the red points in the next row (plane) [19,23].		X
RM Red-Black Tiled 32: Partial blocking applied to *RM Red-Black*		X

Table 7.1: Code transformations applied to the Jacobi (J) and Gauss-Seidel (G-S) methods. Note that for the Jacobi iterative method, the updated values of each iteration are stored in an auxiliary array. The array pointers are then swapped at the end of each iteration.

.

```
int i,j;
double A[N][N], B[N];
double tmp;

for(j=1;j<N-1;j++)
  B[j]=0.25*(A[0][j]+A[1][j-1]+
             A[1][j+1]+A[2][j]);

for(i=2;i<N-1;i++)
  for(j=1;j<N-1;j++){
    tmp = B[j];
    B[j]=0.25*(A[i-1][j]+A[i][j-1]+
               A[i][j+1]+A[i+1][j]);
    A[i-1][j] = tmp;
  }

for(j=1;j<N-1;j++)
  A[i][j] = B[j];
```

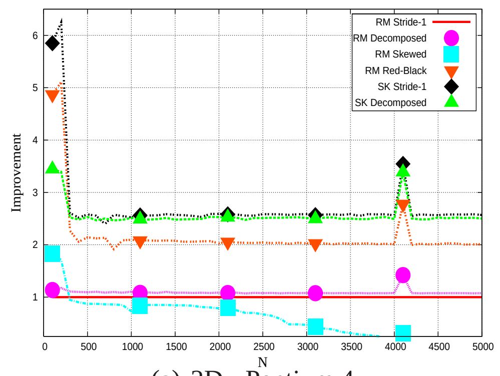
Figure 7.8: Minimizing temporary storage for 2D Jacobi assuming row-major storage order.

	Pentium 4	Opteron	Power5+
Clock rate	3.4 GHz	2.6 GHz	1.9 GHz
L1 Dcache	16 KB	64 KB	64 KB
L2 Dcache	512 KB	1 MB	1.9 MB
L3 Dcache	-	-	32 MB
TLB Size	128	512	2048
Compiler	Intel 9.0	PGI 6.0	xlc V8.0
Opt. Flags	-O3 -align -xWK -prefetch -fno-alias -fno-fnalias -rcd -mcpu=pentium4	-O3 -fastsse -Mipa=fast -Mpfi -Minline -Msafeptr	-O4 -qhot -qnostrict -qalias=allptrs

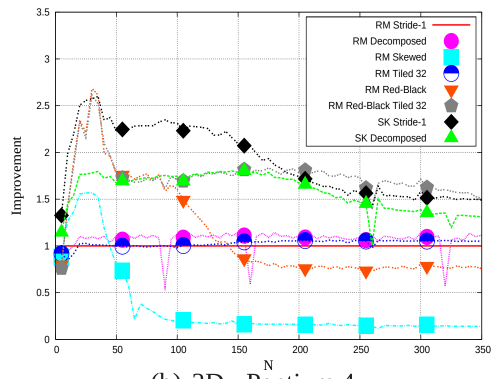
Table 7.2: Processor characteristics of interest.

space limitations, only the Gauss-Seidel experiments as well as limited PAPI information are reported (see Figures 7.10 and 7.11).

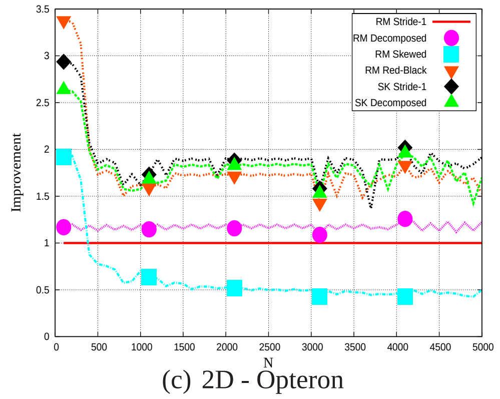
Figures 7.9 depicts the speed-up for the Gauss-Seidel method on the three processors considered in this study. For the Pentium and Opteron processors, we clearly observe a performance drop when the problem size does not fit into the L2 cache ($N > 256$ (2D) and $N > 40$ (3D) for the Pentium, and $N > 362$ (2D) and $N > 50$ (3D) for the Opteron). This phenomenon is less visible for the Power5+ processor, most likely due to its large L3



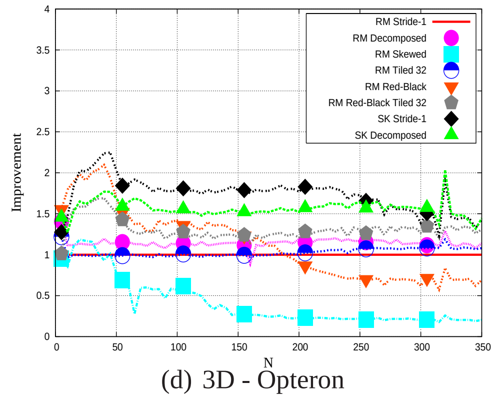
(a) 2D - Pentium 4



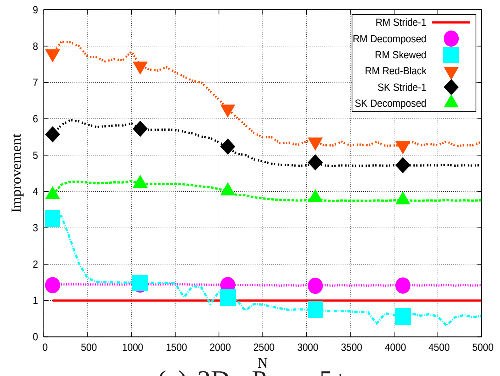
(b) 3D - Pentium 4



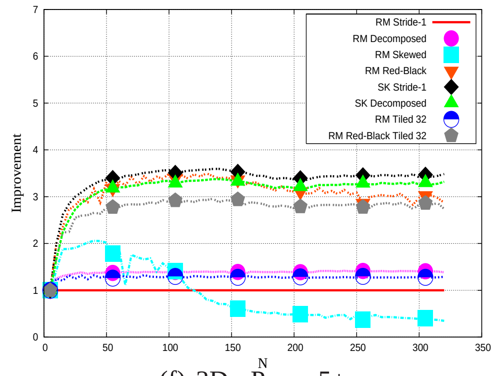
(c) 2D - Opteron



(d) 3D - Opteron



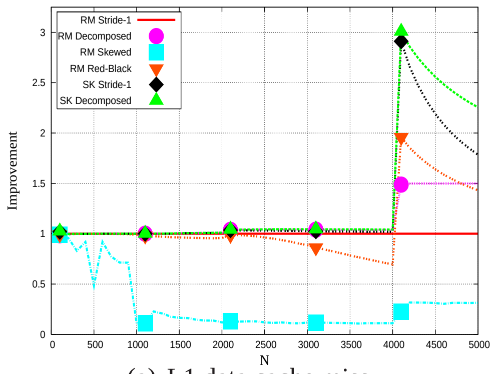
(e) 2D - Power5+



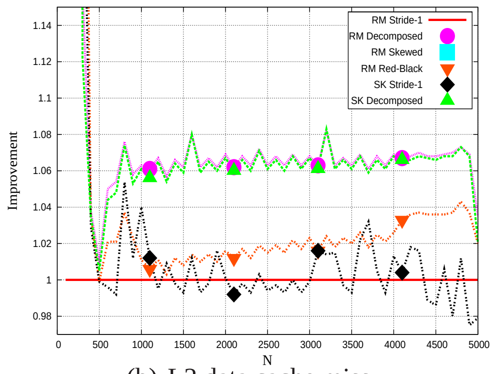
(f) 3D - Power5+

Figure 7.9: Speed-up for the Gauss-Seidel stencil kernel.

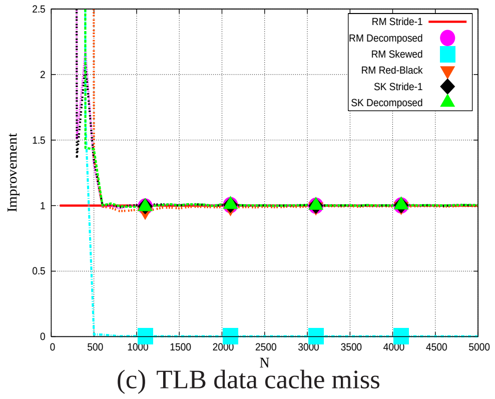
cache (32MB) that attenuates L2 cache misses. However, a careful look at the behavior of the *RM Skewed* transformation reveals the same trends observed for the Pentium 4 and Opteron processors. Remember that this transformation implements loop skewing upon a row-major storage order, and is hence very sensitive to problem size not fitting into cache. We clearly observe a performance drop of the *RM Skewed* transformation when



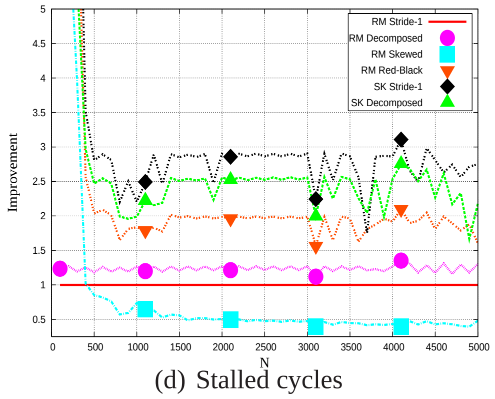
(a) L1 data cache miss



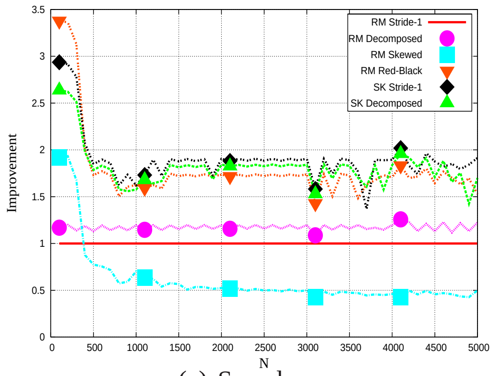
(b) L2 data cache miss



(c) TLB data cache miss



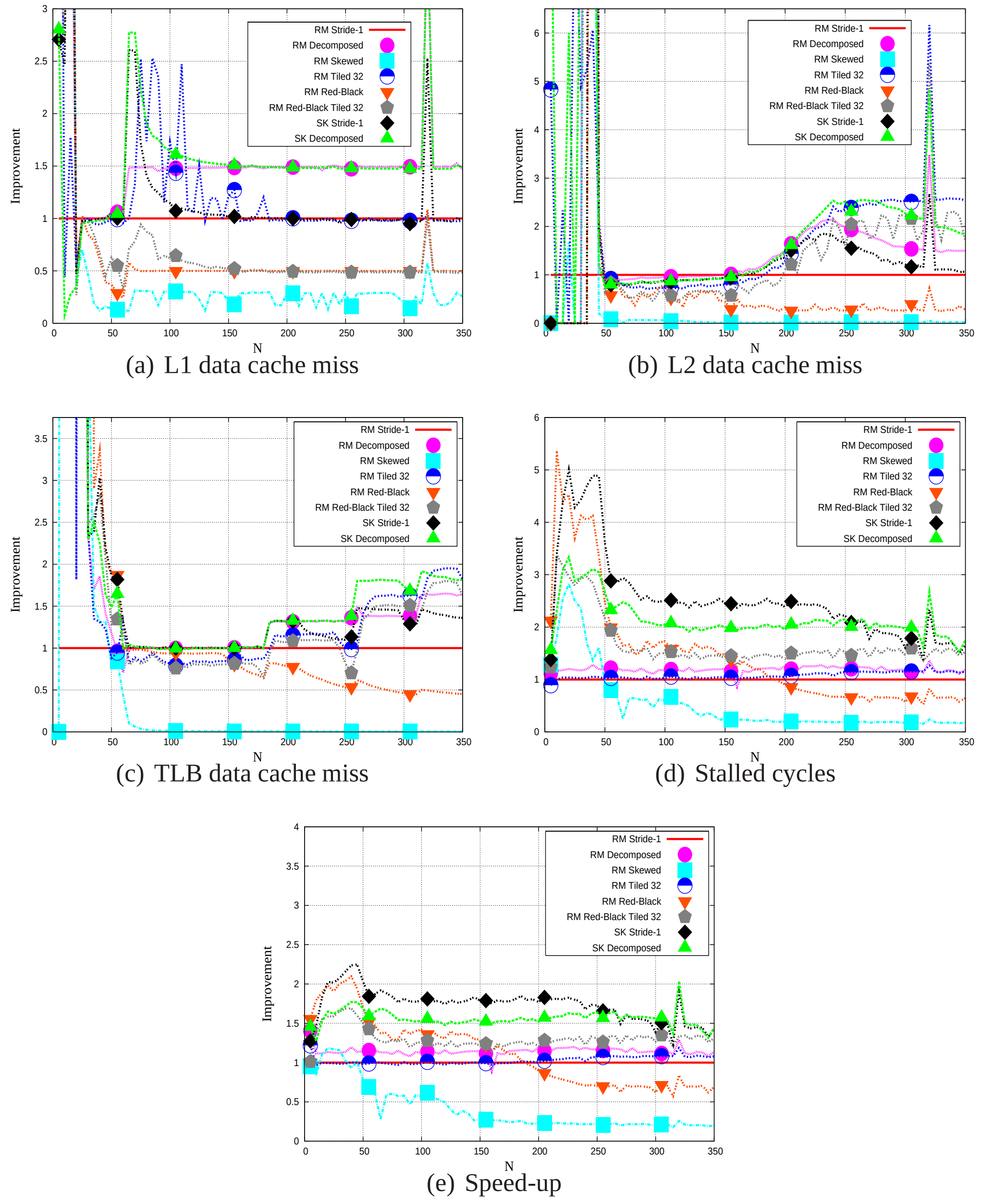
(d) Stalled cycles



(e) Speed-up

Figure 7.10: PAPI information on the Opteron for the 2D Gauss-Seidel stencil kernel.

the problem size does not fit into the L2 cache ($N > 500$ (2D) and $N > 62$ (3D)), and this yields also for the L3 cache ($N > 2048$ (2D) and $N > 161$ (3D)). We now need to look at PAPI information (Figures 7.10 and 7.11) to find out what makes the difference between the performance achieved by the different transformations.



(a) L1 data cache miss

(b) L2 data cache miss

(c) TLB data cache miss

(d) Stalled cycles

(e) Speed-up

Figure 7.11: PAPI information on the Opteron for the 3DGauss-Seidel stencil kernel.

For small problem sizes, one can observe erratic performances for the quantities monitored with PAPI. Indeed, very few cache misses occur when the arrays fit entirely into cache. Then, normalizing the results (i.e. dividing them by the quantity of reference) gives very high and unstable ratios, creating these erratic curves. However, this phenomenon disappears for larger problem sizes.

A correlation exists between execution time (speed-up curves) and the number of cycles stalled on any resource. Indeed, the two sets of curves (see Figures 7.10 (d) and 7.10 (e) on one hand, and Figures 7.11 (d) and 7.11 (e) on the other hand) are remarkably similar such that we can confidently claim that the latter have influence on the former. In other words a high numbers of stalled cycles degrades performance, as expected. Processor stalls can arise when data is not available in cache and need to be fetched from memory, or when one instruction depends on another. For the Jacobi method, processor stalls are provoked by L2 and TLB data cache misses. The transformations that achieve best performance expose a much lower number of L2 and TLB data cache misses. In contrast, for the Gauss-Seidel method, a correlation between stall cycles and data cache misses cannot be established directly from our experiments. Processor stalls are most likely provoked by data dependencies since the transformations that achieve best performance are those who expose parallelism to the compiler. Indeed, the 3 processors adopt a dynamic, hardware-intensive approach allowing out-of-order execution [8]. The key concept of out-of-order execution is to allow the processor to avoid a class of stalls that occur when the data needed to perform an operation is not available. The processor fills these ”slots” in time with other instructions that are ready, then re-orders the results at the end to make it appear that the instructions were processed in the sequential order. The benefit of out-of-order execution grows as the instruction pipeline deepens and the speed difference between main memory (or cache memory) and the processor widens.

For 2D Gauss-Seidel, transformations improve the L1 data cache miss rates as soon as group reuse is lost (i.e. when $N > 4096$). As expected, decomposed schemes achieve better L2 utilization, when the problem size does not fit into cache ($N > 362$). The *RM Skewed* transformation achieves remarkably poor performance with respect to L1, L2 and TLB data cache misses, suffering from its non contiguous data access pattern. The transformations that achieve best performance, are those that expose parallelism to the compiler. The superiority of *SK Stride-1* over *SK Decomposed* cannot be explained by the L1, L2 nor TLB data cache performances, as the performance curves speak for the decomposed scheme. This superiority is most likely due to the higher data dependencies exhibited by the decomposed scheme, as *SK Stride-1* suffers from less stalled cycles. Indeed, the micro-stencils being applied after each other, there are data dependencies between the micro-stencil updates, whereas applying the original stencil involve only independent memory locations. Moreover, the decomposed transformation involves more memory references as well as floating points operations. The superiority of the *SK Stride-1* and *SK Decomposed* schemes over *RM Red-Black* can be explained by their stride-1 data access patterns, while the Red-Black scheme utilizes only half of each cache line for each color. Finally, the marginal performance improvement of L2 data cache misses (7%) achieved by *RM Decomposed* seems to be fruitful, translated into an overall 20% performance increase.

For 3D Gauss-Seidel, erratic performance stops as soon as the problem size does not fit into cache, i.e. when $N > 20$ (L1) and $N > 50$ (L2). Most transformations improve the L1 data cache miss rates as soon as group reuse is lost ($N > 64$). For the L2 cache on the other hand, the data cache miss rate improvement saturates or even decreases when group reuse is lost ($N > 256$). As expected, decomposed schemes achieve better L1, and to some extent L2 and TLB utilization, while the *RM Skewed* transformation achieves remarkably poor performance, still suffering from its non contiguous data access pattern. Again, the transformations that achieve best performance, are those that expose parallelism to the compiler.

7.7 Conclusion

In this paper, we present transformation techniques for cache-efficient stencil computations. We formally demonstrate - and quantify - how spatial locality can be improved by using skewed data layouts as opposed to the traditional row-major and column-major storage orders. To the best of our knowledge, this is the first paper that presents theoretical and experimental results on this memory alteration technique. The other main contribution of this paper is the *stencil decomposition* transformation, that improves temporal locality. This technique exploits the symmetrical properties of the stencil kernel by computing the mutual contributions of two array elements, thus requiring the two elements to be simultaneously present in cache only once.

Overall, our experiments confirm previous results reported in the literature, and reveal other aspects of interests. In line with previous studies [9, 19], we found that tiling is not beneficial for 2D stencil codes, and that it improves performance only for sufficiently large problem sizes. However, we believe that as cache sizes of modern processors keep increasing, tiling will become less and less beneficial, as testify our experiments on the Power5+ processor which confirm the claim of Kamil et al. [9].

Experimental results using the PAPI interface showed that the techniques presented in this paper reduce significantly the number of L1, L2 as well as TLB data cache misses. We find tiling schemes work better than skewed transformations for the Jacobi method on the Pentium 4 and Opteron processors, but are outperformed for the Gauss-Seidel method. The former method is known to converge slower than the latter one, motivating thus the seek for new transformation techniques, such as the ones presented in this paper.

For iterative methods using in-place averaging, transformations that exhibit parallelism to the compiler are required to achieve high performance. Data access patterns exposing a high level of parallelism to the compiler work well provided that data can be moved quickly to the processor. The tiled Red-Black scheme achieves this goal to some extent, albeit limited for large problem sizes by its non contiguous data access pattern. However, data access patterns alone are not sufficient to improve performance as problem size increases. Changing the data layout in order to match the data access pattern proves to be fruitful.

The scope of this paper was restricted to 5-point and 7-point stencil kernels. However, longer range interaction stencil kernels (such as 9-point and 27-point) are broadly used within scientific computations, and should be studied as a future work direction. Another direction would be to derive theoretical lower bounds for the average footprint memory distance of 2D and 3D memory layouts. We conjecture that the problem of finding the memory layout that yields to the minimum average footprint memory distance is NP-complete, since similar problems like bandwidth and profile matrix minimizations are NP-complete. While a formal proof of the NP-completeness of our problem is still to be provided, lower bound results would give insights on how good are the skewed data layouts studied in this paper.

[1] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, 2000.

[2] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear Array Layouts for Hierarchical Memory Systems. In *International Conference on Supercomputing*, pages 444–453, 1999.

[3] S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottethodi. Recursive Array Layouts and Fast Parallel Matrix Multiplication. In *SPAA '99: Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures*, pages 222–231, New York, NY, USA, 1999. ACM Press.

[4] T. Chen and C. Chang. Skewed Data Partition and Alignment Techniques for Compiling Programs on Distributed Memory Multicomputers. *J. Supercomput.*, 21(2):191–211, 2002.

[5] S. Coleman and K. S. McKinley. Tile Size Selection Using Cache Organization and Data Layout. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 279–290, New York, NY, USA, 1995. ACM Press.

[6] J. Díaz, J. Petit, and M. Serna. A Survey of Graph Layout Problems. *ACM Comput. Surv.*, 34(3):313–356, 2002.

[7] N. E. Gibbs, W. G. Poole, and P. K. Stockmeyer. An Algorithm for Reducing the Bandwidth and Profile of a Sparse Matrix. *SIAM J Numer. Anal*, 13:236–250, 1976.

[8] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[9] S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. Yelick. Impact of Modern Memory Subsystems on Cache Optimizations for Stencil Computations. In *MSP '05: Proceedings of the 2005 workshop on Memory system performance*, pages 36–43, New York, NY, USA, 2005. ACM Press.

[10] M. T. Kandemir, A. N. Choudhary, N. Shenoy, P. Banerjee, and J. Ramanujam. A Linear Algebra Framework for Automatic Determination of Optimal Data Layouts. *IEEE Trans. Parallel Distrib. Syst.*, 10(2):115–135, 1999.

[11] M. Kowarschik and C. Weiß. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. In U. Meyer, P. Sanders, and J. Sibeyn, editors, *Algorithms for Memory Hierarchies — Advanced Lectures*, volume 2625 of *Lecture Notes in Computer Science (LNCS)*, pages 213–232. Springer, 2003.

[12] M. D. Lam, E. E. Rothberg, and M. E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 63–74, New York, NY, USA, 1991. ACM Press.

[13] C. Leopold. On Optimal Temporal Locality of Stencil Codes. In *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*, pages 948–952, New York, NY, USA, 2002. ACM Press.

[14] K. McKinley, S. Carr, and C.-W. Tseng. Improving Data Locality with Loop Transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, 1996.

[15] N. Mitchell, K. Högstedt, L. Carter, and J. Ferrante. Quantifying the multi-level nature of tiling interactions. *International Journal of Parallel Programming*, 26(6):641–670, 1998.

[16] N. Park, B. Hong, and V. K. Prasanna. Tiling, Block Data Layout, and Memory Hierarchy Performance. *IEEE Trans. Parallel Distrib. Syst.*, 14(7):640–654, 2003.

[17] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 1992.

[18] G. Rivera and C. Tseng. Eliminating Conflict Misses for High Performance Architectures. In *International Conference on Supercomputing*, pages 353–360, 1998.

[19] G. Rivera and C. Tseng. Tiling Optimizations for 3D Scientific Computations. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 32, Washington, DC, USA, 2000. IEEE Computer Society.

[20] S. S. Skiena. *The algorithm design manual*. Springer-Verlag New York, Inc., New York, NY, USA, 1998.

[21] O. Temam, E. D. Granston, and W. Jalby. To Copy or Not to Copy: A Compile-Time Technique for Assessing When Data Copying Should be Used to Eliminate Cache Conflicts. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 410–419, New York, NY, USA, 1993. ACM Press.

[22] S. Vajracharya and D. Grunwald. Loop Re-Ordering and Pre-Fetching at Run-Time. In *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–13, New York, NY, USA, 1997. ACM Press.

[23] C. Weiß, W. Karl, M. Kowarschik, and U. Rüde. Memory Characteristics of Iterative Methods. In *Proc. of the ACM/IEEE Supercomputing Conf. (SC99)*, Portland, Oregon, USA, 1999.

[24] R. C. Whaley and J. J. Dongarra. Automatically Tuned Linear Algebra Software. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[25] M. E. Wolf and M. S. Lam. A Data Locality Optimizing Algorithm. *SIGPLAN Not.*, 39(4):442–459, 2004.

[26] M. Wolfe. *High-Performance Compilers for Parallel Computing*. Addison-Wesley, 1995. ISBN 0-8053-2730-4.

[27] Y. Zhao and K. Kennedy. Scalarization Using Loop Alignment and Loop Skewing. *Journal of Supercomputing*, Volume 31(1):5–46, 2005.