Memory-aware algorithms : from multicores to large scale platforms

Mathias Jacquelin

N° d'ordre : 633 *N° attribué par la bibliothèque : ENSL633*

- ÉCOLE NORMALE SUPÉRIEURE DE LYON -
Laboratoire de l'Informatique du Parallélisme - UMR5668 - LIP

THÈSE

en vue d'obtenir le grade de

Docteur de l'Université de Lyon - École Normale Supérieure de Lyon

Spécialité : Informatique

au titre de l'École Doctorale Informatique et Mathématiques

présentée et soutenue publiquement le 20 juillet 2011 par

Mathias JACQUELIN

Memory-aware algorithms : From multicores to large scale platforms

Directeur de thèse :	Yves	ROBERT	
Co-encadrant de thèse :	Loris	MARCHAL	
Après avis de :	Michel	DAYDÉ	*Rapporteur*
	Jean	ROMAN	*Rapporteur*
	Oliver	SINNEN	*Rapporteur*

Devant la commission d'examen formée de :

Michel	DAYDÉ	*Rapporteur*
Loris	MARCHAL	*Membre*
Yves	ROBERT	*Membre*
Jean	ROMAN	*Rapporteur*
Eric	SANLAVILLE	*Membre*

Acknowledgements

First of all, I would like to thank my three reviewers for their work. Thanks to Michel Daydé, Jean Roman and Oliver Sinnen for their numerous advices and insightful remarks, which all have improved the clarity, the readability and the accuracy of this thesis.

I would also like to thank Eric Sanlaville for joining my examining committee, and for clearing his schedule to attend the defense in Lyon.

A special thank you to my advisors, Loris and Yves, for having agreed to accompany me through these three years. Your remarks, always professional, your explanations and your ideas have always been a great help and benefited me a lot. I will always keep good memories of these three years by your side.

Last but not least, I would like to thank my family, my friends and my girlfriend for their support all through my PhD, for trying to understand my misleading explanations on what I was doing, and for allowing me to go through these three years without a hitch. Thank you all for your support, and especially to the one who was brave enough to deal with me everyday.

Remerciements

En premier lieu, je tiens à remercier l'ensemble des rapporteurs pour leur travail. Un grand merci à Michel Daydé, Jean Roman et Oliver Sinnen pour leurs nombreux conseils et remarques qui ont permis d'améliorer grandement la clarté, la lisibilité et la précision de ce manuscrit.

Je tiens en outre à remercier Eric Sanlaville d'avoir accepté de participer à mon jury, et de se libérer en cette fin de mois de juillet afin de venir à Lyon.

Merci aussi à mes deux encadrants, Loris et Yves, pour avoir accepté de m'encadrer pendant ces trois années de thèse. Vos remarques, toujours professionnelles, vos explications et vos idées ont à chaque fois été une aide précieuse et m'ont grandement apporté. Je garderai toujours en mémoire les trois années qui viennent de s'écouler à vos côtés.

Enfin, je souhaite remercier ma famille, mes amis et ma compagne pour m'avoir soutenu tout au long de ma thèse, avoir tenté de comprendre les explications, rarement claires, sur l'objet de mes travaux, et permis de traverser ces trois années sans encombre. Merci à tous pour votre soutien, et plus particulièrement à celle qui a eu le courage de me supporter au quotidien.

Contents

Introduction

Many domains have greatly benefited from the processing power of computers for decades. As new levels of computing power were unleashed, scientific progress gained momentum. However, a processor needs memory to be able to compute: since their invention, processor and memory were deeply tangled. Back in the early 1940's, pioneer computers, like the *ENIAC*, were limited to a few bytes of memory stored in vacuum tube accumulators. The next generation of memory, *Delay line memory*, significantly increased the capacity, but remained volatile: it required power to maintain the information. This deeply impacted the reliability of such systems. Non-volatile memory were discovered in the late 1940's with *Magnetic core memory*, thus allowing information recovery after a power loss. *Hard disk drives* were introduced by IBM in 1956 and still dominates the storage market nowadays. During almost twenty years, non-volatile memory replaced volatile memories and satisfied processors' needs, until semiconductor-based memories supplanted them.

Though the power of a single processor was steadily increasing, it did not meet tremendous requirements of scientific applications. Multiple processors were embedded within a single machine, the *Burroughs D825* being the first true multiprocessor system. However, semiconductors memories were scarce and expensive. To remedy this issue, complex hierarchies of memory were introduced, ranging from semi-conductor memories, magnetic cores, to hard disk drives. For commodity reasons, all these memories were mapped into a single flat space called *Virtual memory*. This space was shared among all processors. The processing power was significantly improved. Though an upper bound on the achieved speed-up has been derived for parallel systems [8], it does not take the underlying memory layout into account. Furthermore, dealing with multiple parallel accesses to a single shared memory was in itself a challenging task and led to numerous algorithmic problems.

Because of hardware constraints, adding processors within a single machine comes at a high price. In order to push processing power further, those multi-processor machines were interconnected: *computer clusters* were born. Sadly, memory layouts became even more intricate: some processors, residing in the same machine, were sharing the same memory while some other processors were working within distributed memories. Modeling such platforms became very complex, related algorithmic problems bewildering. Furthermore, before the 1980's, memory access was slightly slower than internal register access. However the gap between processor and memory has been growing since. Memory became a serious bottleneck, processors facing the well-known *memory wall*. This led to the introduction of *cache memories*: a large inexpensive low-speed memory associated with a small high-speed memory called *cache*. This allowed to alleviate the penalty of memory accesses at a reasonable cost.

Following the advances in semiconductors lithography, each new generation of processors were twice as faster as the previous one, leading to the famous "Moore's law". During the previous century, clock speed and processing power were tightly coupled, and evolved fast. The pace has regrettably slowed down at the beginning of the twenty-first century. Mainly because of heat dissipation and transistor leakages due to miniaturization, it was very hard to increase clock speed further. To cope with these constraints, several processing units were embedded within a single *multicore processor*. The

last decade has seen the advent of multicore processors in every computer and electronic device, from personal laptops to supercomputers. Multicore architectures deeply impacted cache hierarchies by introducing the notion of shared/distributed cache even within a single processor. Furthermore, as opposed to what has been done for decades, being oblivious of cache hierarchy began to impact performance.

Nowadays, heterogeneous multicore processors are emerging in order to cope with electronic constraints while offering ever increasing performance. This heterogeneity also resides in memory architectures: new processors embed always deeper cache hierarchies. From now on, cache hierarchies need to be finely taken into consideration to unleash their full capabilities. Storage has gone through similar changes: the cost of non-volatile semiconductor memories has decreased enough so as to allow its use for storage devices. While these *solid state disks* are nowadays available, they are often mixed with hard disk drives and tape drives. This hierarchy of non-volatile memories resembles that of volatile memories found within processors. In terms of performance, this trend has definitively moved the focus from the hardware to the software. Deploying an application on such platforms becomes a challenging task due to the increasing heterogeneity at every level. In addition, the crucial need to squeeze the most out of every resource leads to revisit many problems and to introduce new algorithms.

In this thesis, we focus on designing memory-aware algorithms and schedules tailored for hierarchical memory architectures. Indeed, these memory layouts can be found from within multicore processors to the storage architectures of supercomputers. Various platforms have thus been studied in order to assess the impact of such memory architectures on both performance and peak memory usage. Our main contributions are sketched in the following paragraphs.

Complexity analysis and performance evaluation of matrix product on multicore architectures

In Chapter 1, we study both the complexity and the performance of matrix product on multicore architectures. Indeed, dense linear algebra kernels are the key to performance for many scientific applications. Some kernels, like matrix multiplication, have extensively been studied on parallel 2D processor grid architectures. However, they are not well suited for multicore architectures, as memory is shared and data accesses are performed through a hierarchy of caches. Taking further advantage of data locality is crucial to increase performance. To this end, we introduce a realistic but still tractable model of a multicore processor with two levels of cache, and derive multiple lower bounds on the communication volume, based on the work of [95]. We extend the Maximum Reuse Algorithm, introduced in [77], to multicore architectures, by taking into account both cache levels. We thus present the Multicore Maximum Reuse Algorithm (MMRA). Therefore, the focus is set on minimizing communications by adapting the data allocation scheme. We assess both model relevance and algorithms performance through an extensive set of experiments, ranging from simulation to real implementation on a GPU.

Tiled QR factorization

In Chapter 2, we revisit existing algorithms for the QR factorization of rectangular matrices composed of $p \times q$ tiles, where $p \geq q$. This dense linear algebra kernel lies at the foundation of many scientific applications. It is more complex than the matrix product kernel studied in Chapter 1: parallelism needs to be fully exploited on multicore processors in order to reach high levels of performance.

Within this framework, we study the critical paths and performance of algorithms such as SAMEH-KUCK, FIBONACCI, GREEDY, and those found within PLASMA, a state-of-the-art dense linear algebra library. Although neither FIBONACCI nor GREEDY is optimal, both are shown to be asymptotically optimal for all matrices of size $p = q^2 f(q)$, where f is any function such that $\lim_{+\infty} f = 0$. This

novel and important complexity result applies to all matrices where p and q are proportional, $p = \lambda q$, with $\lambda \geq 1$, thereby encompassing many important situations in practice (least squares). We provide an extensive set of critical path lengths obtained through simulation, as well as real experiments that show the superiority of the new algorithms for tall matrices both in terms of theoretical and practical performance.

Well known algorithms like matrix product, where memory behavior could be precisely handled, could benefit from having total control over caches. Having such a control over caches is equivalent to consider them as fast local memories. Such memories are often encountered on today's accelerator architectures like the Cell processor [61] or GPUs. Moreover, on more complex kernels like QR factorization, multicore processors bring a new level of parallelism that needs to be exploited. Processor computing power increases much faster than memory bandwidth. As this well-known memory wall is closing in, the need of such algorithms is getting more important. Both previous studies prove that good results can be obtained using a high level, analytical and architecture-independent approach.

Scheduling streaming applications on a complex multicore platform

Streaming applications represent another example of widely used applications that could benefit from multicore processors. In Chapter 3, we consider the problem of scheduling such streaming applications on a heterogeneous multicore platform, the IBM QS 22 platform, embedding two STI Cell BE processors, so as to optimize the throughput of the application.

A stream is a sequence of *instances*, such as images in the case of a video stream. We show that deploying a complex streaming application on a heterogeneous multicore architecture (the QS 22 platform) under the objective of throughput maximization, that is the number of instances processed per time-unit, is NP-complete in the strong sense. We thus present a mixed integer programming (MIP) approach that allows to compute a mapping with optimal throughput. This extends the study introduced in [39] which rather focuses on computing Grids, although new issues have to be solved due to the completely different granularity of the tasks. We also propose simpler scheduling heuristics to compute mapping of the application task-graph on the platform.

In order to assess the performance of our MIP approach and heuristics on the QS 22, a complex software framework is needed: it has to map tasks on different types of processing elements and to handle all communications. Although there already exist some frameworks dedicated to streaming applications, none of them is able to deal with complex task graphs while allowing to statically select the mapping. Thus, we have decided to develop our own framework. Experiments were conducted on 25 random task graphs, having various depths, widths, and branching factors. The smallest graph has 20 tasks while the largest has 135 tasks. For all graphs, we generated 10 variants with different Communication-to-Computation Ratio (CCR), resulting in 250 different random applications and 230 hours of computation.

In the experiments, our MIP strategy is usually close to the throughput predicted by the linear program, while displaying a good and scalable speed-up when using up to 16 SPEs. Some of the proposed heuristics are also able to reach very good performance.

We have shown that considering load-balancing among processing elements, and carefully estimating communications between the different components of this bi-Cell platform, is the key to performance. Overall, this demonstrates that scheduling a complex application on a heterogeneous multicore processor is a challenging task, but that scheduling tools can help to achieve good performance.

On optimal tree traversals for sparse matrix factorization

As already said, communication and memory architectures must be considered in order to offer a high level of performance, especially for scientific applications. Moreover, on most hardware platforms, the available memory or storage space determines the size of the problems that could be solved by such applications. Therefore, for a given application, it is crucial to minimize the amount of required space. This work is presented in Chapter 4.

Applications are often modeled by complex DAGs. However in a first step, we will study the complexity of traversing tree-shaped workflows whose tasks require large I/O files. Such workflows typically arise in the multifrontal method of sparse matrix factorization. Therefore, they impact a significant range of scientific applications. We target a classical two-level memory system, where the main memory is faster but smaller than the secondary memory. In this context, I/O represent transfers from a memory to the other.

The traversal of a tree plays a key role in determining which amount of main memory and I/O volume are needed for a successful execution of the whole tree. We propose a new exact algorithm called $MinMem$ to determine the minimum amount of main memory that is required to execute the tree without any access to secondary memory, based upon a novel approach that systematically explores the tree with a given amount of memory. Although the worst-case complexity of the proposed $MinMem$ algorithm is the same as that of Liu's [69], it is much more efficient in practice, as demonstrated by our experiments with elimination trees arising in sparse matrix factorization. We also show that there exist trees where commonly used postorder based traversals require arbitrarily larger amounts of main memory than the optimal one.

As for the determination of the minimum I/O volume to execute the tree with a given memory of size M, we show that the problem is NP-hard, both for postorder based and for arbitrary traversals. We provide a set of heuristics to solve this problem. Our heuristics use various greedy criteria to select the next node to be scheduled, and those files to be temporarily written to secondary memory. All these heuristics are evaluated using assembly trees arising in sparse matrix factorization methods.

Comparing archival policies for BLUE WATERS

Hierarchical memory architectures can be found from within multicore processors, to the storage system of cutting-edge supercomputers where parallel hard disk drives are connected to archival devices.

In Chapter 5, we focus on BLUE WATERS, a supercomputer developed by the University of Illinois at Urbana-Champaign, the NCSA, IBM, and the Great Lakes Consortium for Petascale Computation [19]. Indeed, BLUE WATERS implements a hierarchical storage system heavily relying upon a novel tape storage system. The main objective of this study is to write data to tapes efficiently and resiliently. We thus introduce two archival policies tailored for the tape storage system of BLUE WATERS and adapt the well known RAIT strategy. We provide an analytical model of the tape storage platform of BLUE WATERS, and use it to assess and analyze the performance of the three policies through simulations. We use random workloads whose characteristics model various realistic scenarios. The average (weighted) response time for each user is the main objective.

The experiments were performed using a discrete event based simulator: SimGrid [90, 67], on a platform embedding 500 tape drives, matching the scale of BLUE WATERS. We show that PARALLEL strategy outperforms the state-of-the-art RAIT policy when using the same policy for every request. We introduce the HETERO policy and show that a significant benefit can be gained from using multiple policies depending on the request's characteristics.

Altogether these research topics led to several contributions, and allowed us to assess the impact of

hierarchical memory architectures on application performance. Admittedly, this work represents a first step toward the design of high-level, architecture-independent, algorithms, capable of efficiently taking memory layout into consideration.

Chapter 1

Impact of cache misses on multicore matrix product algorithms

1.1 Introduction

In this chapter, we study the matrix-matrix product on multicore architectures, and assess the impact of cache misses on the performance of these algorithms. Indeed, dense linear algebra kernels are the key to performance for many scientific applications.

Some of these kernels, like matrix multiplication, have extensively been studied on parallel architectures. Two well-known parallel versions are Cannon's algorithm [23] and the ScaLAPACK outer product algorithm [17]. Typically, parallel implementations work well on 2D processor grids: input matrices are sliced horizontally and vertically into square blocks; there is a one-to-one mapping of blocks onto physical resources; several communications can take place in parallel, both horizontally and vertically. Even better, most of these communications can be overlapped with (independent) computations. All these characteristics render the matrix product kernel quite amenable to an efficient parallel implementation on 2D processor grids.

However, algorithms based on a 2D grid (virtual) topology are not well suited for multicore architectures. In particular, in a multicore architecture, memory is shared, and data accesses are performed through a hierarchy of caches, from shared caches to distributed caches. To increase performance, we need to take further advantage of data locality, in order to minimize data movement. This hierarchical framework resembles that of out-of-core algorithms [95] (the shared cache being the disk) and that of master-slave implementations with limited memory [77] (the shared cache being the master's memory). The latter paper [77] presents the Maximum Reuse Algorithm which aims at minimizing the communication volume from the master to the slaves. Here, we extend this study to multicore architectures, by taking both the shared and distributed cache levels into account. We analytically show how to achieve the best possible tradeoff between shared and distributed caches.

We implement and evaluate several algorithms on two multicore platforms, one equipped with one Xeon quad-core, and the second one enriched with a GPU. It turns out that the impact of cache misses is very different across both platforms, and we identify what are the main design parameters that lead to peak performance for each target hardware configuration. For the sake of reusability, the source code for the implementation and comparison of all algorithms is publicly available [1].

The rest of this chapter is organized as follows. Section 1.2 introduces the model for multicore platforms, and derives new bounds on the number of shared and distributed cache misses of any matrix product algorithm. These bounds derive from a refined analysis of the CCR (Communication-to-

1. http://graal.ens-lyon.fr/~mjacquel/mmre_cpu_gpu.html

Computation Ratio) imposed by the underlying architecture. Section 1.3 presents the new algorithms designed to optimize shared cache misses, distributed cache misses, or a combination of both. In section 1.4 we evaluate the behavior of all algorithms on various cache configurations using a dedicated simulator. In Section 1.5 we proceed to an experimental evaluation of these algorithms, together with a bunch of reference algorithms and with the vendor library routines, on a CPU platform. Section 1.6 is the counterpart for a GPU platform. Finally in Section 1.7, we provide final remarks, and directions for future work.

1.2 Problem statement

1.2.1 Multicore architectures

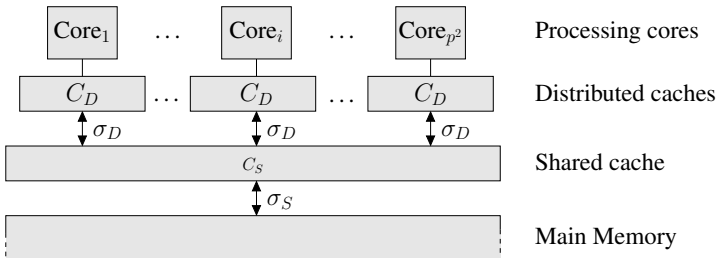


Figure 1.1: Multicore architecture model.

A major difficulty of this study is to come up with a realistic but still tractable model of a multicore processor. We assume that such a processor is composed of p^2 cores, and that each core has the same computing speed. The processor is connected to a memory, which is supposed to be large enough to contain all necessary data (we do not deal with out-of-core execution here). The data path from the memory to a computing core goes through two levels of caches. The first level of cache is shared among all cores, and has size C_S, while the second level of cache is distributed: each core has its own private cache, of size C_D. Caches are supposed to be *inclusive*, which means that the shared cache contains *at least* all the data stored in every distributed cache. Therefore, this cache must be larger than the union of all distributed caches: $C_S \geq p^2 \times C_D$. Our caches are also "fully associative", and can therefore store any data from main memory. Figure 1.1 depicts the multicore architecture model.

The hierarchy of caches is used as follows. When a data is needed in a computing core, it is first sought in the distributed cache of this core. If the data is not present in this cache, a *distributed-cache miss* occurs, and the data is then sought in the shared cache. If it is not present in the shared cache either, then a *shared-cache miss* occurs, and the data is loaded from the memory in the shared cache and afterward in the distributed cache. When a core tries to write to an address that is not in the caches, the same mechanism applies. Rather than trying to model this complex behavior, we assume in the following an *ideal cache model* [38]: we suppose that we are able to totally control the behavior of each cache, and that we can load any data into any cache (shared of distributed), with the constraint that a data has to be first loaded in the shared cache before it could be loaded in the distributed cache. Although somewhat unrealistic, this simplified model has been proven not too far from reality: it is shown in [38] that an algorithm causing N cache misses with an *ideal* cache of size L will not cause more than $2N$ cache misses with a cache of size $2L$ implementing a classical *LRU* replacement policy.

In the following, our objective is twofold: (i) minimize the number of cache misses during the computation of matrix product, and (ii) minimize the predicted data access time of the algorithm. To this end, we need to model the time needed for a data to be loaded in both caches. To get a simple and

yet tractable model, we consider that cache speed is characterized by its bandwidth. The shared cache has bandwidth σ_S, thus a block of size S needs S/σ_S time-unit to be loaded from the memory in the shared cache, while each distributed cache has bandwidth σ_D. Moreover, we assume that concurrent loads to several distributed caches are possible without contention. Since we assume an ideal cache model with total control on the cache behavior, we also suppose that coherency mechanisms' impact on performance can be neglected.

Finally, the purpose of the algorithms described below is to compute the classical matrix product $C = A \times B$. In the following, we assume that A has size $m \times z$, B has size $z \times n$, and C has size $m \times n$. We use a block-oriented approach, to harness the power of BLAS routines [17]. Thus, the atomic elements that we manipulate are not matrix coefficients but rather square blocks of coefficients of size $q \times q$. Typically, q ranges from 32 to 100 on most platforms.

1.2.2 Communication volume

The key point to performance in a multicore architecture is efficient data reuse. A simple way to assess data locality is to count and minimize the number of cache misses, that is the number of times each data has to be loaded in a cache. Since we have two types of caches in our model, we try to minimize both the number of misses in the shared cache and the number of misses in the distributed caches. We denote by M_S the number of cache misses in the shared cache. As for distributed caches, since accesses from different caches are concurrent, we denote by M_D the maximum of all distributed caches misses: if $M_D^{(c)}$ is the number of cache misses for the distributed cache of core c, $M_D = \max_c M_D^{(c)}$.

In a second step, since the former two objectives are conflicting, we aim at minimizing the overall time T_{data} required for data movement. With the previously introduced bandwidth, and assuming a fully balanced load, it can be expressed as $T_{\text{data}} = \frac{M_S}{\sigma_S} + \frac{M_D}{\sigma_D}$. Depending on the ratio between cache speeds, this objective provides a tradeoff between both cache miss quantities.

1.2.3 Lower bound on communications

In [95], Irony, Toledo and Tiskin propose a lower bound on the number of communications needed to perform a matrix product. We have extended this study to the hierarchical cache architecture. In what follows, we consider a computing system (which consists of one or several computing cores) using a cache of size Z. We estimate the number of computations that can be performed owing to Z consecutive cache misses, that is owing to Z consecutive load operations. We recall that each matrix element is in fact a matrix block of size $q \times q$. We use the following notations :

- Let η_{old}, ν_{old}, and ξ_{old} be the number of blocks in the cache used by blocks of A, B and C just before the Z cache misses.
- Let η_{read}, ν_{read}, and ξ_{read} be the number of blocks of A, B and C read in the main memory when these Z cache misses occurs.
- Let $comp(c)$ be the amount of computation done by core c

Before the Z cache misses, the cache holds at most Z blocks of data, therefore, after Z cache misses, we have:

$$\begin{cases} \eta_{old} + \nu_{old} + \xi_{old} \leq Z \\ \eta_{read} + \nu_{read} + \xi_{read} = Z \end{cases} \tag{1.1}$$

Loomis-Whitney's inequality

The following lemma, given in [59] and based on Loomis-Whitney inequality, is valid for any conventional matrix multiplication algorithm $C = AB$, where A is $m \times z$, B is $z \times n$ and C is $m \times n$.

A processor that contributes to N_C elements of C and accesses N_A elements of A and N_B elements of B can perform at most $\sqrt{N_A N_B N_C}$ elementary multiplications. According to this lemma, if we let K denote the number of elementary multiplications performed by the computing system, we have:

$$K \leq \sqrt{N_A N_B N_C}$$

No more than $(\eta_{old} + \eta_{read})q^2$ elements of A are accessed, hence $N_A = (\eta_{old} + \eta_{read})q^2$. The same holds for B and C: $N_B = (\nu_{old} + \nu_{read})q^2$ and $N_C = (\xi_{old} + \xi_{read})q^2$. Let us simplify the notations using the following variables:

$$\begin{cases} \eta_{old} + \eta_{read} = \eta \times Z \\ \nu_{old} + \nu_{read} = \nu \times Z \\ \xi_{old} + \xi_{read} = \xi \times Z \end{cases} \tag{1.2}$$

Then we derive $K \leq \sqrt{\eta\nu\xi} \times Z\sqrt{Z} \times q^3$. Writing $K = kZ\sqrt{Z}q^3$, we obtain the following system of equations:

$$\text{MAXIMIZE } k \text{ SUCH THAT}$$
$$\begin{cases} k \leq \sqrt{\eta\nu\xi} \\ \eta + \nu + \xi \leq 2 \end{cases}$$

Note that the second inequality comes from Equation (1.1). This system admits a solution which is $\eta = \nu = \xi = \frac{2}{3}$ and $k = \sqrt{\frac{8}{27}}$. This gives us a lower bound on the Communication-to-Computation Ratio (in terms of blocks) for any matrix multiplication algorithm:

$$CCR \geq \frac{Z}{k \times Z\sqrt{Z}} = \sqrt{\frac{27}{8Z}}$$

Bound on shared-cache misses

We will first use the previously obtained lower bound to study shared-cache misses, considering everything above this cache level as a single processor and the main memory as a master which sends and receives data. Therefore, with $Z = C_S$ and $K = \sum_c comp(c)$, we have a lower bound on the Communication-to-Computation Ratio for shared-cache misses:

$$CCR_S = \frac{M_S}{K} = \frac{M_S}{\sum_c comp(c)} \geq \sqrt{\frac{27}{8C_S}}$$

Bound on distributed-caches misses

In the case of the distributed caches, we first apply the previous result, on a single core c, with cache size C_D. We thus have

$$CCR^{(c)} \geq \sqrt{\frac{27}{8C_D}}$$

We define the overall distributed CCR as the average of all $CCR^{(c)}$, so this result also holds for the CCR_D:

$$CCR_D = \frac{1}{p^2}\sum_{c=1}^{p^2}\left(\frac{{M_D}^{(c)}}{comp(c)}\right) \geq \sqrt{\frac{27}{8C_D}}$$

Indeed, we could even have a stronger result, on the minimum of all $CCR^{(c)}$.

Bound on overall data access time

The previous bound on the CCR can be extended to the data access time, as it is defined as a linear combination of both M_S and M_D:

$$T_{\text{data}} = \frac{M_S}{\sigma_S} + \frac{M_D}{\sigma_D}$$

We can bound M_S using the bound on the CCR:

$$M_S = CCR_S \times mnz \geq mnz \times \sqrt{\frac{27}{8C_S}}$$

As for distributed-cache misses, it is more complex, since M_D is the maximum of all misses on distributed caches, and CCR_D is the average CCR among all cores. In order to get a tight bound, we consider only algorithms where both computation and cache misses are equally distributed among all cores (this applies to all algorithms developed in this chapter). In this case, we have

$$M_D = \max_c M_D^{(c)} = M_D^{(0)} = \frac{mnz}{p^2} \times CCR^{(0)} \geq \frac{mnz}{p^2} \times \sqrt{\frac{27}{8C_D}}$$

Thus, we get the following bound on the overall data access time:

$$T_{\text{data}} \geq mnz \times \left(\frac{1}{\sigma_S} \times \sqrt{\frac{27}{8C_S}} + \frac{1}{p^2\sigma_D} \times \sqrt{\frac{27}{8C_D}} \right).$$

1.3 Algorithms

In the out-of-core algorithm of [95], the three matrices A, B and C are equally accessed throughout time. This naturally leads to allocating one third of the available memory to each matrix. This algorithm has a Communication-to-Computation Ratio of $O\left(\frac{mnz}{\sqrt{M}}\right)$ for a memory of size M but it does not use the memory optimally. The Maximum Reuse Algorithm [77] proposes a more efficient memory allocation: it splits the available memory into $1 + \mu + \mu^2$ blocks, storing a square block $C_{i_1 \ldots i_2, j_1 \ldots j_2}$ of size μ^2 of matrix C, a row $B_{i_1 \ldots i_2, j}$ of size μ of matrix B and one element $A_{i,j}$ of matrix A (with $i_1 \leq i \leq i_2$ and $j_1 \leq j \leq j_2$). This allows to compute $C_{i_1 \ldots i_2, j_1 \ldots j_2} += A_{i,j} \times B_{i_1 \ldots i_2, j}$. Then, with the same block of C, other computations can be accumulated by considering other elements of A and B. The block of C is stored back only when it has been processed entirely, thus avoiding any future need of reading this block to accumulate other contributions. Using this framework, the Communication-to-Computation Ratio is $\frac{2}{\sqrt{M}}$ for large matrices.

To extend the Maximum Reuse Algorithm to multicore architectures, we must take into account both cache levels. Depending on the objective, we modify the previous data allocation scheme so as to fit with the shared cache, with the distributed caches, or with both. This will lead to three versions of a new multicore Maximum Reuse Algorithm, or MMRA: SHAREDMMRA, DISTRIBUTEDMMRA and TRADEOFFMMRA. In all cases, the main idea is to design a "data-thrifty" algorithm that reuses matrix elements as much as possible and loads each required data only once in a given loop. Since the outermost loop is prevalent, we load the largest possible square block of data in this loop, and adjust the size of the other blocks for the inner loops, according to the objective (shared-cache, distributed-cache, tradeoff) of the algorithm. We define two parameters that will prove helpful to compute the size of the block of C that should be loaded in the shared cache or in a distributed cache:

• λ is the largest integer with $1 + \lambda + \lambda^2 \leq C_S$;

• μ is the largest integer with $1 + \mu + \mu^2 \leq C_D$.
In the following, we assume that λ is a multiple of μ, so that a block of size λ^2 that fits in the shared cache can be easily divided in blocks of size μ^2 that fit in the distributed caches.

1.3.1 Minimizing shared-cache misses

Algorithm 1: SHAREDMMRA.

for Step $= 1$ *to* $\frac{m \times n}{\lambda^2}$ **do**
  Load a new block $C[i, \ldots, i+\lambda;\ j, \ldots, j+\lambda]$ of C in the shared cache
  for $k = 1$ *to* z **do**
    Load a row $B[k;\ j, \ldots, j+\lambda]$ of B in the shared cache
    for $i' = i$ *to* $i + \lambda$ **do**
      Load the element $a = A[k;\ i']$ in the shared cache
      foreach *core* $c = 1 \ldots p^2$ *in parallel* **do**
        Load the element $a = A[k;\ i']$ in the distributed cache of core c
        for $j' = j + (c-1) \times \frac{\lambda}{p^2}$ *to* $j + c \times \frac{\lambda}{p^2}$ **do**
          Load $B_c = B\,[k;\ j']$ in the distributed cache of core c
          Load $C_c = C\,[i';\ j']$ in the distributed cache of core c
          Compute the new contribution: $C_c \leftarrow C_c + a \times B_c$
          Update block C_c in the shared cache
  Write back the block of C to the main memory

To minimize the number of shared-cache misses M_S, we extend the Maximum Reuse Algorithm with the new parameter λ. A square block C_{block} of size λ^2 of C is allocated in the shared cache, together with a row of λ elements of B and one element of A. Then, each row of C_{block} is divided into sub-rows of $\frac{\lambda}{p^2}$ elements, which are then distributed to the computing cores together with corresponding element of B and one element of A, updated by the different cores, and written back in shared cache. As soon as C_{block} is completely updated, it is written back in main memory and a new C_{block} is loaded. This is described in Algorithm 1, and the memory layout is depicted in Figure 1.2.

Note that the space required in each distributed-cache to process one block of q^2 elements of C is $1 + 1 + 1$ blocks. For now, we have not made any assumption on the size of distributed caches. Let S_D be the size of each distributed cache, expressed in the number of matrix coefficients (remember that C_D is expressed in blocks): the constraint $3 \leq C_D$ simply translates into $3q^2 \leq S_D$. We compute the number of cache misses of SHAREDMMRA as follows:

– *Shared-cache misses*
  In the algorithm, the whole matrix C is loaded in the shared cache, thus resulting in mn cache misses. For the computation of each block of size λ^2, z rows of size λ are loaded from B, and $z \times \lambda$ elements of A are accessed. Since there are mn/λ^2 steps, this amounts to a total number of shared cache misses of:

$$M_S = \frac{mn}{\lambda^2} \times \left(z \times (\lambda + \lambda) + \lambda^2\right) = mn + \frac{2mnz}{\lambda}$$

  The Communication-to-Computation Ratio is therefore:

$$CCR_S = \frac{mn + \frac{2mnz}{\lambda}}{mnz} = \frac{1}{z} + \frac{2}{\lambda}$$

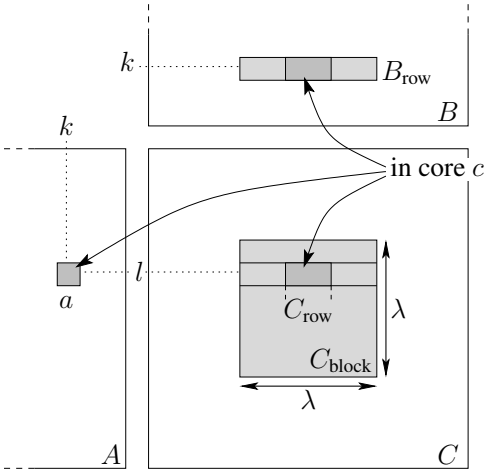


Figure 1.2: Data layout for Algorithm 1.

For large matrices, this leads to a shared-cache CCR of $2/\lambda$, which is close to the lower bound derived earlier.

– *Distributed-caches misses*
In the algorithm, each block of C is sequentially updated z times by rows of λ elements, each row is distributed element per element, thus requiring $\frac{\lambda}{p^2}$ steps. Therefore, each distributed cache holds one element of C. This results in $\frac{mnz}{p^2}$ cache misses. For the computation of each block of size λ^2, $z \times \frac{\lambda}{p^2}$ elements are loaded from B in each distributed cache, and $z \times \lambda$ elements of A are accessed. Since there are mn/λ^2 steps, this amounts to a total number of:

$$M_D = \frac{\frac{mn}{\lambda^2} \times z \times \lambda \times p^2 \times \left(1 + \frac{\lambda}{p^2} + \frac{\lambda}{p^2}\right)}{p^2} = \frac{2mnz}{p^2} + \frac{mnz}{\lambda}$$

The Communication-to-Computation Ratio is therefore:

$$CCR_D = \frac{\frac{2mnz}{p^2} + \frac{mnz}{\lambda}}{\frac{mnz}{p^2}} = 2 + \frac{p^2}{\lambda}$$

This Communication-to-Computation Ratio does not depend upon the dimensions of the matrices, and is the same for large matrices. Moreover, it is far from the lower bound on distributed cache misses.

1.3.2 Minimizing distributed-cache misses

The next objective is to minimize the number of distributed-cache misses. To this end, we use the parameter μ defined earlier to store in each distributed cache a square block of size μ^2 of C, a fraction of row (of size μ) of B and one element of A. Contrarily to the previous algorithm, the block of C will be totally computed before being written back to the shared cache. All p^2 cores work on different blocks of C. Thanks to the constraint $p^2 \times C_D \leq C_S$, we know that the shared cache has the capacity to store all necessary data.

The $\mu \times \mu$ blocks of C are distributed among the distributed caches in a 2-D cyclic way, because it helps reduce (and balance between A and B) the number of shared-cache misses: in this case, we load a

Algorithm 2: DISTRIBUTEDMMRA.

$offset_i = (\text{My\_Core\_Num}() - 1) \pmod p$
$offset_j = \lfloor \frac{\text{My\_Core\_Num}()-1}{p} \rfloor$
for Step $= 1$ *to* $\frac{m \times n}{p^2 \mu^2}$ **do**
  Load a new block $C[i, \ldots, i + p\mu;\ j, \ldots, j + p\mu]$ of C in the shared cache
  foreach *core* $c = 1 \ldots p^2$ *in parallel* **do**
    Load
    $C_c = C[i + offset_i \times \mu, \ldots, i + (offset_i + 1) \times \mu;\ j + offset_j \times \mu, \ldots, j + (offset_j + 1) \times \mu]$
    in the distributed cache of core c
  for $k = 1$ *to* z **do**
    Load a row $B[k;\ j, \ldots, j + p\mu]$ of B in the shared cache
    foreach *core* $c = 1 \ldots p^2$ *in parallel* **do**
      Load $B_c = B[k;\ j + offset_j \times \mu, \ldots, j + (offset_j + 1) \times \mu]$ in the distributed cache of core c
      for $i' = i + offset_i \times \mu$ *to* $i + (offset_i + 1) \times \mu$ **do**
        Load the element $a = A[k;\ i']$ in the shared cache
        Load the element $a = A[k;\ i']$ in the distributed cache of core c
        Compute the new contribution: $C_c \leftarrow C_c + a \times B_c$
  foreach *core* $c = 1 \ldots p^2$ *in parallel* **do**
    Update block C_c in the shared cache
  Write back the block of C to the main memory

$p\mu \times p\mu$ block of C in shared cache, together with a row of $p\mu$ elements of B. Then, $p \times \mu$ elements from a column of A are sequentially loaded in the shared cache (p non contiguous elements are loaded at each step), then distributed among distributed caches (cores in the same "row" (resp. "column") accumulate the contribution of the same (resp. different) element of A but of different (resp. the same) $p \times \mu$ fraction of row from B). We compute the number of cache misses of DISTRIBUTEDMMRA (see Algorithm 2) as follows:

– *Shared-cache misses*
  The number of shared-cache misses is:

$$M_S = \frac{mn}{p^2\mu^2} \times \left(p^2\mu^2 + z \times 2p\mu\right) = mn + \frac{2mnz}{\mu p}$$

  Hence, the Communication-to-Computation Ratio is:

$$CCR_S = \frac{mn + \frac{2mnz}{\mu p}}{mnz} = \frac{1}{z} + \frac{2}{\mu p}$$

  For large matrices, the Communication-to-Computation Ratio is $\frac{2}{\mu p} = \sqrt{\frac{32}{8 \times pC_D}}$. This is far from the lower bound since $pC_D \leq p^2 C_D \leq C_S$.
– *Distributed-caches misses*
  The number of distributed-cache misses is:

$$M_D = \frac{\frac{mn}{p^2\mu^2} \times p^2 \times \left(\mu^2 + z \times 2\mu\right)}{p^2} = \frac{mn}{p^2} + \frac{2mnz}{\mu \times p^2}$$

Therefore, the Communication-to-Computation Ratio is:

$$CCR_D = \frac{\frac{mn}{p^2} + \frac{2mnz}{\mu \times p^2}}{\frac{mnz}{p^2}} = \frac{1}{z} + \frac{2}{\mu}$$

For large matrices, the Communication-to-Computation Ratio is asymptotically close to the value $\frac{2}{\mu} = \sqrt{\frac{32}{8C_D}}$, which is close to the lower bound $\sqrt{\frac{27}{8C_D}}$.

1.3.3 Minimizing data access time

The two previous objectives are antagonistic: for both previous algorithms, optimizing the number of cache misses of one type leads to a large number of cache misses of the other type. Indeed, minimizing M_S ends up with a number of distributed-cache misses proportional to the common dimension of matrices A and B, and in the case of large matrices this is clearly problematic. On the other hand, focusing only on M_D is not efficient since we dramatically under-use the shared cache: a large part of it is not utilized.

This motivates us to look for a tradeoff between the latter two solutions. However, both kinds of cache misses have different costs, since bandwidths between each level of the memory architecture are different. Hence we introduce the overall time for data movement, defined as:

$$T_{\text{data}} = \frac{M_S}{\sigma_S} + \frac{M_D}{\sigma_D}$$

Depending on the ratio between cache speeds, this objective provides a tradeoff between both cache miss numbers. To derive an algorithm optimizing this tradeoff, we start from the algorithm presented for optimizing the shared-cache misses. Looking closer to the downside of this algorithm, which is the fact that the part of M_D due to the elements of C is proportional to the common dimension z of matrices A and B, we see that we can reduce this amount by loading blocks of β columns (resp. of rows) of A (resp. B). This way, square blocks of C will be processed longer by the cores before being unloaded and written back in shared-cache, instead of being unloaded after that every element of the column of A residing in shared-cache has been used. However, blocks of C must be smaller than before, and instead of being λ^2 blocks, they are now of size α^2 where α and β are defined under the constraint $2\alpha \times \beta + \alpha^2 \leq C_D$. The sketch of the TRADEOFFMMRA (see Algorithm 3) is the following:

1. A block of size $\alpha \times \alpha$ of C is loaded in the shared cache. Its size satisfies $p^2 \times \mu^2 \leq \alpha^2 \leq \lambda^2$. Both extreme cases are obtained when one of σ_D and σ_S is negligible in front of the other.
2. In the shared cache, we also load a block from B, of size $\beta \times \alpha$, and a block from A of size $\alpha \times \beta$. Thus, we have $2\alpha \times \beta + \alpha^2 \leq C_D$.
3. The $\alpha \times \alpha$ block of C is split into sub-blocks of size $\mu \times \mu$ which are processed by the different cores. These sub-blocks of C are cyclically distributed among every distributed-caches. The same holds for the block-row of B which is split into $\beta \times \mu$ block-rows and cyclically distributed, row by row (i.e. by blocks of size $1 \times \mu$), among every distributed-caches.
4. The contribution of the corresponding β (fractions of) columns of A and β (fractions of) lines of B is added to the block of C. Then, another $\mu \times \mu$ block of C residing in the shared cache is distributed among every distributed-caches, going back to step 3.
5. As soon as all elements of A and B have contributed to the $\alpha \times \alpha$ block of C, another β columns/lines from A/B are loaded in shared cache, going back to step 2.
6. Once the $\alpha \times \alpha$ block of C in shared cache is totally computed, a new one is loaded, going back to step 1.

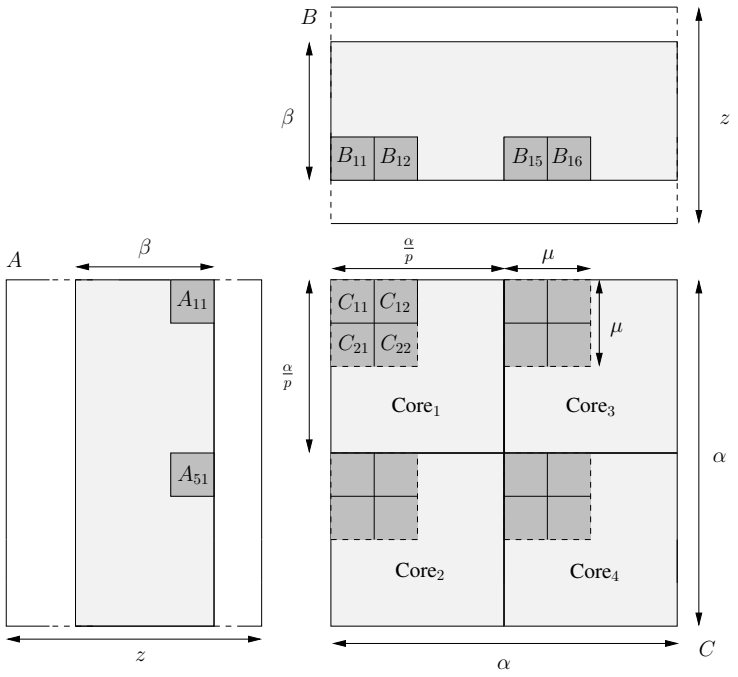


Figure 1.3: Data distribution of matrices A, B and C: light gray blocks reside in shared-cache, dark gray blocks are distributed among distributed-caches ($\alpha = 8, \mu = 2,\ p^2 = 4$).

We compute the number of cache misses as follows:

– *Shared-cache misses*
The number of shared-cache misses is given by:

$$M_S = \frac{mn}{\alpha^2}\left(\alpha^2 + \frac{z}{\beta} \times 2\alpha\beta\right) = mn + \frac{2mnz}{\alpha}$$

The Communication-to-Computation Ratio is therefore:

$$CCR_S = \frac{1}{z} + \frac{2}{\alpha}$$

For large matrices, the Communication-to-Computation Ratio is asymptotically close to $\frac{1}{\alpha}\sqrt{\frac{32}{8}}$ which is farther from the lower bound than the shared-cache optimized version since $\alpha \leq \lambda \approx \sqrt{C_S}$.

– *Distributed-caches misses*
In the general case (i.e. $\alpha > p\mu$), the number of distributed-cache misses achieved by TRADEOFFMMRA is:

$$M_D = \frac{1}{p^2 2} \times \frac{mn}{\alpha^2} \times \left[\frac{\alpha^2}{\mu^2} \times \left(\mu^2 \times \frac{z}{\beta} + \frac{z}{\beta} \times 2\beta\mu\right)\right] = \frac{mn}{p^2} \times \frac{z}{\beta} + \frac{2mnz}{p^2\mu}$$

The Communication-to-Computation Ratio is therefore $\frac{1}{\beta} + \frac{2}{\mu}$, which for large matrices is asymptotically close to $\frac{1}{\beta} + \sqrt{\frac{32}{8C_D}}$. This is farther from the lower bound $\sqrt{\frac{27}{8C_D}}$ derived earlier than with the dedicated algorithm. To optimize this CCR, we could try to increase the value of β. However, increasing the parameter β implies a lower value of α, resulting in more shared-cache misses.

Algorithm 3: TRADEOFFMMRA

$offset_i = (\text{My\_Core\_Num}() - 1) \pmod p$
$offset_j = \lfloor \frac{\text{My\_Core\_Num}()-1}{p} \rfloor$
for Step $= 1$ *to* $\frac{m \times n}{\alpha^2}$ **do**
  Load a new block $C[i, \ldots, i+\alpha;\ j, \ldots, j+\alpha]$ of C in the shared cache
  for Substep $= 1$ *to* $\frac{z}{\beta}$ **do**
    $k = 1 + (Substep - 1) \times \beta$
    Load a new block row $B[k, \ldots,\ k+\beta;\ j, \ldots, j+\alpha]$ of B in the shared cache
    Load a new block column $A[i, \ldots, i+\alpha;\ 1+(k-1)\times\beta, \ldots,\ 1+k\times\beta]$ of A in the shared cache
    foreach *core* $c = 1 \ldots p^2$ *in parallel* **do**
      for $subi = 1$ *to* $subi = \frac{\alpha}{p\mu}$ **do**
        for $subj = 1$ *to* $subj = \frac{\alpha}{p\mu}$ **do**
          Load
          $C_{\mu c} = C[i + offset_i \times \frac{\alpha}{p} + (subi-1)\times\mu, \ldots, i + offset_i \times \frac{\alpha}{p} + (subi)\times\mu;\ j + offset_j \times \frac{\alpha}{p} + (subj-1)\times\mu, \ldots, j + offset_j) \times \frac{\alpha}{p} + (subj)\times\mu]$ in the distributed cache of core c
          for $k' = k$ *to* $k' = k + \beta$ **do**
            Load
            $B_c = B[k';\ j + offset_j \times \frac{\alpha}{p} + (subj-1)\times\mu, \ldots, j + offset_j \times \frac{\alpha}{p} + (subj)\times\mu]$
            in the distributed cache of core c
            for $i' = i + offset_i \times \frac{\alpha}{p} + (subi-1)\times\mu$ *to* $i + offset_i \times \frac{\alpha}{p} + (subi)\times\mu$ **do**
              Load the element $a = A[i',\ k']$ in the distributed cache of core c
              Compute the new contribution: $C_c \leftarrow C_c + a \times B_c$
          Update block $C_{\mu c}$ in the shared cache
  Write back the block of C to the main memory

Remark In the special case $\alpha = p\mu$, we only need to load each $\mu \times \mu$ sub-block of C once, since a core is only in charge of one sub-block of C, therefore the number of distributed-caches misses becomes:

$$M_D = \frac{1}{p^2} \times \frac{mn}{\alpha^2} \times \left[\frac{\alpha^2}{\mu^2} \times \left(\mu^2 \times 1 + \frac{z}{\beta} \times 2\beta\mu\right)\right] = \frac{mn}{p^2} + \frac{2mnz}{p^2\mu}$$

In this case, we come back to the distributed-cache optimized case, and the distributed CCR is close to the bound.

– *Data access time*
With this algorithm, we get an overall data access time of:

$$T_{\text{data}} = \frac{M_S}{\sigma_S} + \frac{M_D}{\sigma_D} = \frac{mn + \frac{2mnz}{\alpha}}{\sigma_S} + \frac{\frac{mnz}{p^2\beta} + \frac{2mnz}{p^2\mu}}{\sigma_D}$$

Together with the constraint $2\alpha \times \beta + \alpha^2 \leq C_D$, this expression allows us to compute the best value for parameters α and β, depending on the ratio σ_S/σ_D. Since we work under the assumption

of large matrices, the first term in mn can be neglected in front of the other terms: in essence the problem reduces to minimizing the following expression:

$$\frac{2}{\sigma_S \alpha} + \frac{1}{p^2 \sigma_D \beta} + \frac{2}{p^2 \sigma_D \mu}$$

The constraint $2\beta\alpha + \alpha^2 \leq C_S$ enables us to express β as a function of α and C_S. As a matter of a fact, we have:

$$\beta \leq \frac{C_S - \alpha^2}{2\alpha}$$

Hence, the objective function becomes:

$$F(\alpha) = \frac{2}{\sigma_S \alpha} + \frac{2\alpha}{p^2 \sigma_D (C_S - \alpha^2)}$$

Note that we have removed the term $\frac{2}{p^2 \sigma_D \mu}$ because it only depends on μ and therefore is minimal when $\mu = \lfloor \sqrt{C_S - 3/4} - 1/2 \rfloor$, i.e. its largest possible value.
The derivative $F'(\alpha)$ is:

$$F'(\alpha) = \frac{2(C_S + \alpha^2)}{p^2 \sigma_D (C_S - \alpha^2)^2} - \frac{2}{\sigma_S \alpha^2}$$

And therefore, the root is

$$\alpha_{\text{num}} = \sqrt{C_S \frac{1 + 2\frac{p^2 \sigma_D}{\sigma_S} - \sqrt{1 + 8\frac{p^2 \sigma_D}{\sigma_S}}}{2\left(\frac{p^2 \sigma_D}{\sigma_S} - 1\right)}}$$

Altogether, the best parameters values in order to minimize the total data access time in the case of square blocks are:

$$\begin{cases} \alpha = \min\left(\alpha_{\text{max}}, \max\left(p\mu, \alpha_{\text{num}}\right)\right) \\ \\ \beta = \max\left(\left\lfloor \frac{C_S - \alpha^2}{2\alpha} \right\rfloor, 1\right) \end{cases}$$

where:

$$\alpha_{\text{max}} = \sqrt{C_S + 1} - 1$$

Parameter α depends on the values of bandwidths σ_S and σ_D. In both extreme cases, it will take a particular value indicating that tradeoff algorithm will follow the sketch of either shared-cache optimized version or distributed-caches one:

– When bandwidth σ_D is significantly higher than σ_S, the parameter α becomes:

$$\alpha_{\text{num}} \approx \sqrt{C_S} \quad \Longrightarrow \quad \alpha = \alpha_{\text{max}}, \beta = 1$$

which means that the tradeoff algorithm chooses the shared-cache optimized version whenever distributed caches are significantly faster than the shared cache.

– On the contrary, when the bandwidth σ_S is significantly higher than σ_D, α becomes:

$$\alpha_{\text{num}} \approx \sqrt{C_S \frac{2p^2 \sigma_D}{-\sigma_S}} \quad \Longrightarrow \quad \alpha = p\mu, \beta = 1$$

which means that the tradeoff algorithm chooses the distributed optimized version whenever distributed caches are significantly slower than the shared cache (although this situation does not seem realistic in practice).

1.4 Performance evaluation through simulation

We have presented three algorithms minimizing different objectives (shared cache misses, distributed cache misses and overall time spent in data movement) and provided a theoretical analysis of their performance. However, our simplified multicore model makes some assumptions that are not realistic on a real hardware platform. In particular it uses an *ideal* and omniscient data replacement policy instead of a classical *LRU* policy. This led us to design a multicore cache simulator and implement all our algorithms, as well as the outer-product [17] and Toledo [95] algorithms, using different cache policies. The goal is to experimentally assess the impact of the policies on the actual performance of the algorithms, and to measure the gap between the theoretical prediction and the observed behavior. The main motivation behind the choice of a simulator prior to a real hardware platform resides in commodity reasons: simulation enables to obtain desired results faster and allows to easily modify multicore processor parameters (cache sizes, number of cores, bandwidths, ...).

1.4.1 Simulation setting and cache management policies

The driving feature of our simulator was simplicity. It implements the cache hierarchy of our model, and basically counts the number of cache misses in each cache level. It offers two data replacement policies, *LRU* (Least Recently Used) and *ideal*. In the *LRU* mode, read and write operations are made at the distributed cache level (top of hierarchy); if a miss occurs, operations are propagated throughout the hierarchy until a cache hit happens. In the *ideal* mode, the user manually decides which data needs to be loaded/unloaded in a given cache; I/O operations are not propagated throughout the hierarchy in case of a cache miss: it is the user responsibility to guarantee that a given data is present in every caches below the target cache.

Algorithms We have implemented two reference algorithms: (i) ScaLaPack outer-product, the algorithm in [17], for which we organize cores as a (virtual) processor torus and distribute square blocks of data elements to be updated among them; and (ii) the equal-layout algorithm, inspired by [95], which uses a simple equal-size memory scheme: one third of distributed caches is equally allocated to each loaded matrix sub-block. In fact, the algorithm in [95] deals with a single cache level, hence we divide it in two main versions, *Shared Equal* for shared cache optimization, and *Distributed Equal* for distributed cache optimization. This led to four versions of the equal-layout algorithm: shared/distributed version, and *ideal*/*LRU* cache. We have also implemented the three versions of the Multicore Maximum Reuse Algorithm : SHAREDMMRA, DISTRIBUTEDMMRA and TRADEOFFMMRA for each management policy.

Altogether, this leads to 11 versions, which are labeled as follows:

- OUTERPRODUCT is the outer product algorithm
- SHAREDOPT is SHAREDMMRA using the entire C_S
- SHAREDOPT-LRU is SHAREDMMRA using half of C_S, the other half being used as a buffer for *LRU* policy
- DISTRIBUTEDOPT is DISTRIBUTEDMMRA using the entire C_D
- DISTRIBUTEDOPT-LRU is DISTRIBUTEDMMRA using half of C_D (other half for *LRU*)
- TRADEOFF is TRADEOFFMMRA using the entire C_S and C_D
- TRADEOFF-LRU is the TRADEOFFMMRA using half of C_S and C_D (other half for *LRU*)
- DISTRIBUTEDEQUAL is the equal-layout algorithm [95] using the entire C_D
- DISTRIBUTEDEQUAL-LRU is the equal-layout algorithm [95] using half of C_D (other half for *LRU*)

– SHAREDEQUAL is the equal-layout algorithm [95] using the entire C_S
– SHAREDEQUAL-LRU is the equal-layout algorithm [95] using half of C_S (other half for *LRU*)

Simulated platform In the experiments, we simulated a "realistic" quad-core processor with 8MB of shared cache and four distributed caches of size 256KB dedicated to both data and instruction. We assume that two-thirds of the distributed caches are dedicated to data, and one-third for the instructions. Results using a more pessimistic repartition of one half for data and one-half for instructions are also given. Recall that square blocks of matrix coefficients have size $q \times q$. Depending on the chosen unit block size, caches sizes communicated to our algorithms will thus be the following:

– For $q = 32$: we derive $C_S = 977$ and $C_D = 21$ (or $C_D = 16$ for the pessimistic assumption)
– For $q = 64$: $C_S = 245$ & $C_D = 6$ (or $C_D = 4$)
– Finally, for $q = 80$: $C_S = 157$ & $C_D = 4$ (or $C_D = 3$).

1.4.2 Cache management policies: *LRU* vs. *ideal*

Here we assess the impact of the data replacement policy on the number of shared cache misses and on the performance achieved by the algorithm. Figure 1.4 shows the total number of shared cache misses for SHAREDOPT, in function of the matrix dimension. Figure 1.5 is the counterpart of Figure 1.4 but for DISTRIBUTEDOPT. The same holds for Figure 1.6 and TRADEOFF.

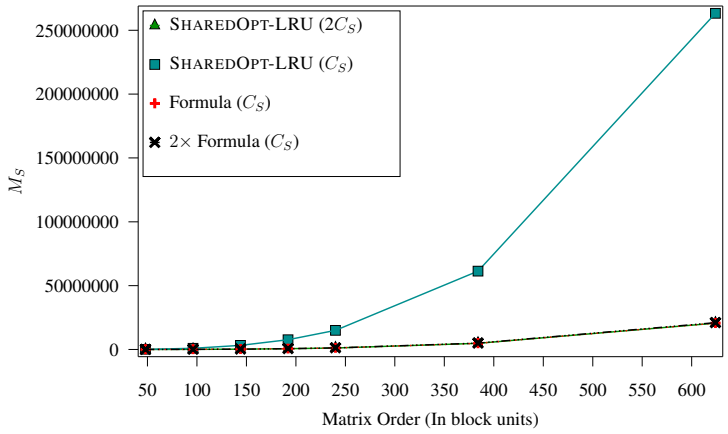


Figure 1.4: Impact of *LRU* policy on the number of shared cache misses M_S of SHAREDOPT with $C_S = 977$

While *LRU* (C_S) (the *LRU* policy with a cache of size C_S) achieves significantly more cache-misses than predicted by the theoretical formula, *LRU* ($2C_S$) is quite close and always achieve less than twice the number of cache misses predicted by the theoretical formula, thereby experimentally validating the prediction of [38]. Furthermore, similar results are obtained for SHAREDEQUAL and DISTRIBUTED-EQUAL. Note that OUTERPRODUCT is insensitive to cache policies, since it is not focusing on cache usage.

This leads us to run our tests using the following two simulation settings:

– The IDEAL setting, which corresponds to the use of the omniscient *ideal* data replacement policy assumed in the theoretical model. It relies on the *ideal* mode of the simulator and declares entire cache sizes (C_S and/or C_D) to the algorithms

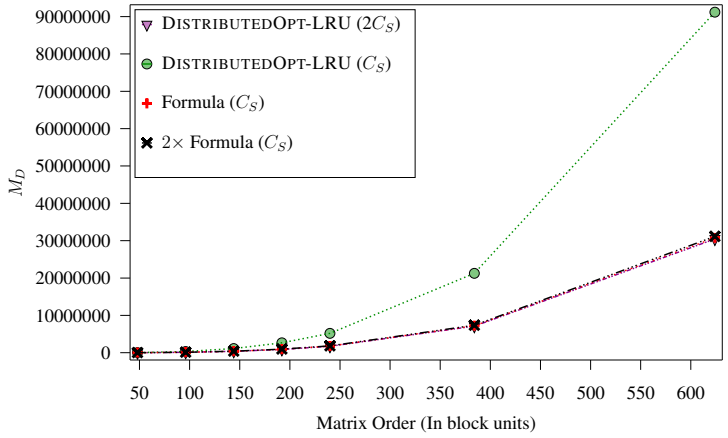


Figure 1.5: Impact of *LRU* policy on the number of distributed cache misses M_D of DISTRIBUTEDOPT with $C_D = 21$

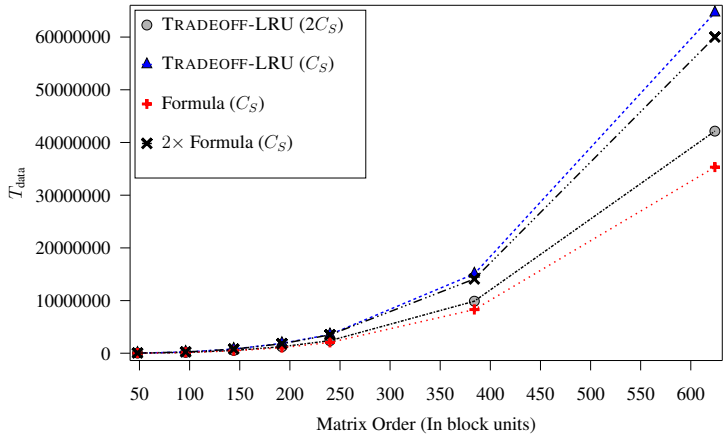


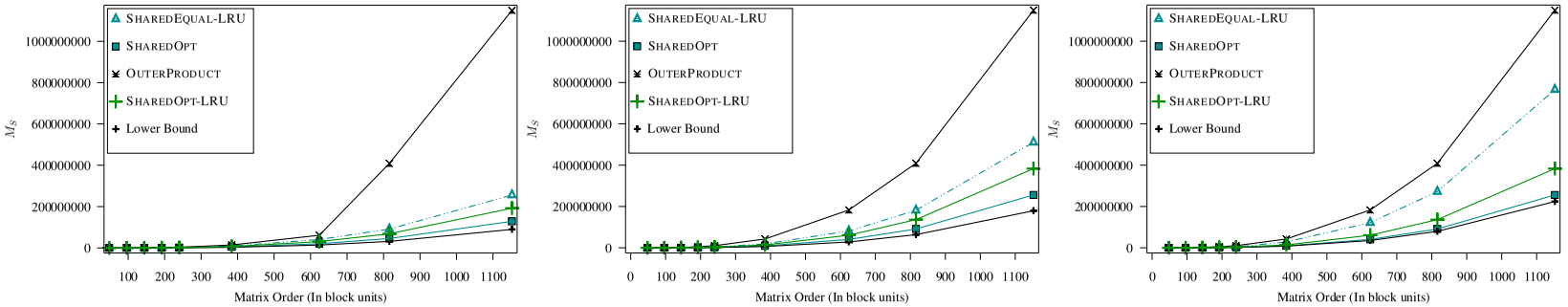
Figure 1.6: Impact of *LRU* policy on T_{data} of TRADEOFF with $C_S = 977$ and $C_D = 21$

– The LRU setting, which relies on a *LRU* cache data replacement policy, but declares only one half of cache sizes to the algorithms. The other half is thus used by the *LRU* policy as kind of an automatic prefetching buffer.

1.4.3 Performance results

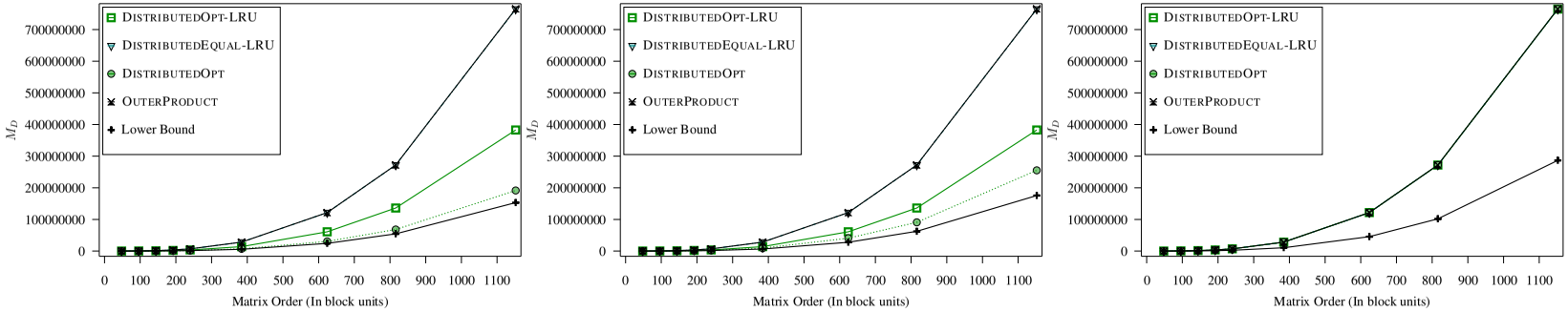
SHAREDMMRA Figure 1.7 depicts the number of shared cache misses achieved by SHAREDOPT versions LRU and IDEAL, in comparison with OUTERPRODUCT, SHAREDEQUAL and the lower bound $m^3\sqrt{\frac{27}{8C_S}}$, according to the matrix dimension m. For every block sizes q, we see that SHAREDOPT performs significantly better than OUTERPRODUCT and SHAREDEQUAL for the LRU setting. Under the IDEAL setting, it is closer to the lower bound, but this latter setting is not realistic.

DISTRIBUTEDMMRA Figure 1.8 is the counterpart of Figure 1.7 for distributed caches. On Figure 1.8a and Figure 1.8b, we similarly see that DISTRIBUTEDOPT performs significantly better than OUTERPRODUCT and DISTRIBUTEDEQUAL for the LRU setting. Under the IDEAL setting, it is very close to the lower bound $\frac{m^3}{p^2}\sqrt{\frac{27}{8C_D}}$.



(a) $C_S = 977$, $q = 32$ (b) $C_S = 245$, $q = 64$ (c) $C_S = 157$, $q = 80$

Figure 1.7: Shared cache misses M_S in function of matrix order.



(a) $C_D = 21$: $q = 32$, data occupy one half of distributed cache (b) $C_D = 16$: $q = 32$, data occupy two thirds of distributed cache (c) $C_D = 6$: $q = 64$, data occupy one half of distributed cache

Figure 1.8: Distributed cache misses M_D in function of matrix order.

However, if we increase the size of the unit block, say $q = 64$ instead of 32, DISTRIBUTEDOPT has not enough space in memory to perform better than OUTERPRODUCT and DISTRIBUTEDEQUAL: as a matter of a fact, in that case $\mu = 1$. This is shown on Figure 1.8c.

Altogether, unit block of size $q = 64$ or larger is not a relevant choice for Distributed Opt., but with $q = 32$, it will always outperform other algorithms in terms of distributed cache misses.

TRADEOFFMMRA The overall time spent in data movement T_{data} of all six LRU algorithms is shown for each cache configuration on Figure 1.9, Figure 1.10 and Figure 1.11. Using $q = 32$ and LRU setting, as shown on Figures 1.9a and 1.9c , TRADEOFF offers the best overall performance, although SHAREDOPT is very close. Moreover, looking at Figures 1.9b and 1.9d, we see that TRADEOFF outperforms even more significantly other algorithms with the IDEAL setting.

However, using $q = 64$ and LRU setting, as shown on Figures 1.10a and 1.10c , TRADEOFF only offers the best overall performance under the pessimistic distributed-cache usage assumption (data can only occupy one half of distributed caches), although SHAREDOPT is once again very close. If we use the loosened cache usage constraint, SHAREDOPT takes the lead over TRADEOFF. This is due to the fact that the *LRU* policy with a larger distributed-cache benefits more to SHAREDOPT (which is unaware of distributed caches) than to TRADEOFF. This is confirmed by Figures 1.10b and 1.10d, on which we see that TRADEOFF and SHAREDOPT are tie with the IDEAL setting.

Finally, using $q = 80$ with LRU setting, as depicted on Figures 1.11a and 1.11c , TRADEOFF never ranks better than SHAREDOPT under both cache usage assumptions. This trend is probably due to the artificial constraints set on cache-related parameters: for instance we require that α divides m and is a multiple of both p and μ. In the implementations, parameters λ and α can be significantly lower than their optimal numerical value. This is confirmed by Figures 1.11b and 1.11d, on which we see that SHAREDOPT significantly outperforms TRADEOFF even using the IDEAL setting.

We also run another experiment to assess the impact of cache bandwidths on T_{data}. We introduce the parameter r defined as: $r = \frac{\sigma_S}{\sigma_D + \sigma_S}$. Figure 1.12 reports results for square matrices of size $m = 384$ and the several caches configurations used before.

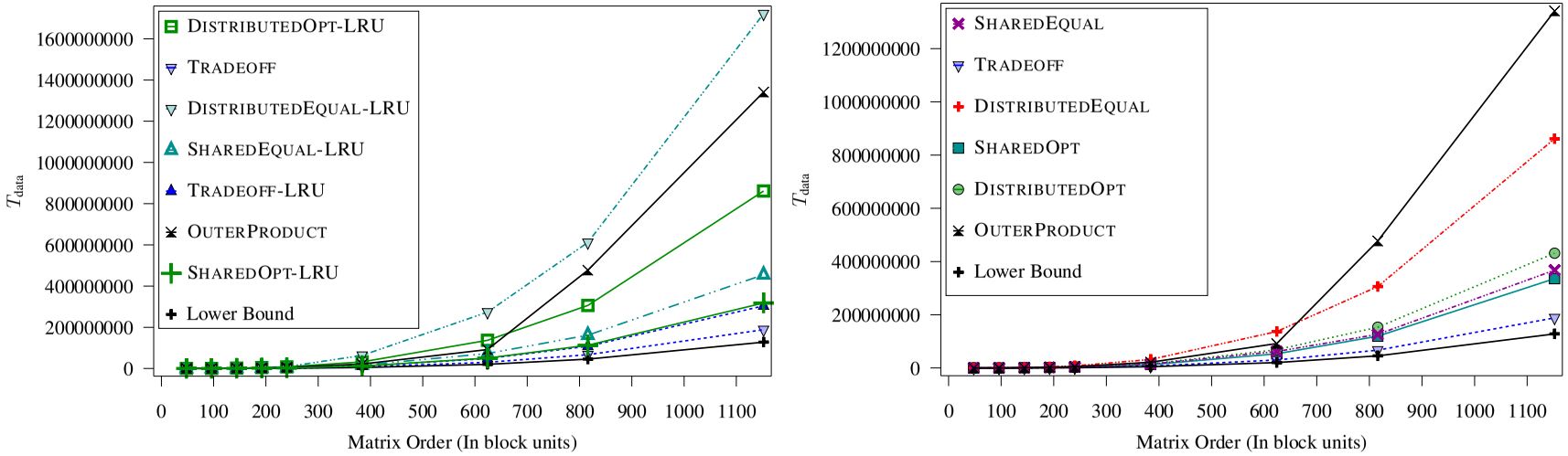
With $q = 32$ we see that TRADEOFF performs best, and still offers the best performance even after distributed misses have become predominant under both optimistic and pessimistic distributed-caches usage assumptions. When the latter event occurs, plots cross over: SHAREDOPT and DISTRIBUTEDOPT achieve the same T_{data}. We also point out that when $r = 0$, TRADEOFF achieves almost the same T_{data} than SHAREDOPT, while when $r = 1$, it ties DISTRIBUTEDOPT.

However, with $q = 64$ and $q = 80$ TRADEOFF does not offer the lowest T_{data}, SHAREDOPT ties or outperforms while shared cache misses are predominant under both distributed-caches assumptions. This is probably because our implementation suffers from the rounding of cache parameters: parameters α and β cannot be adjusted adequately.

1.5 Performance evaluation on CPU

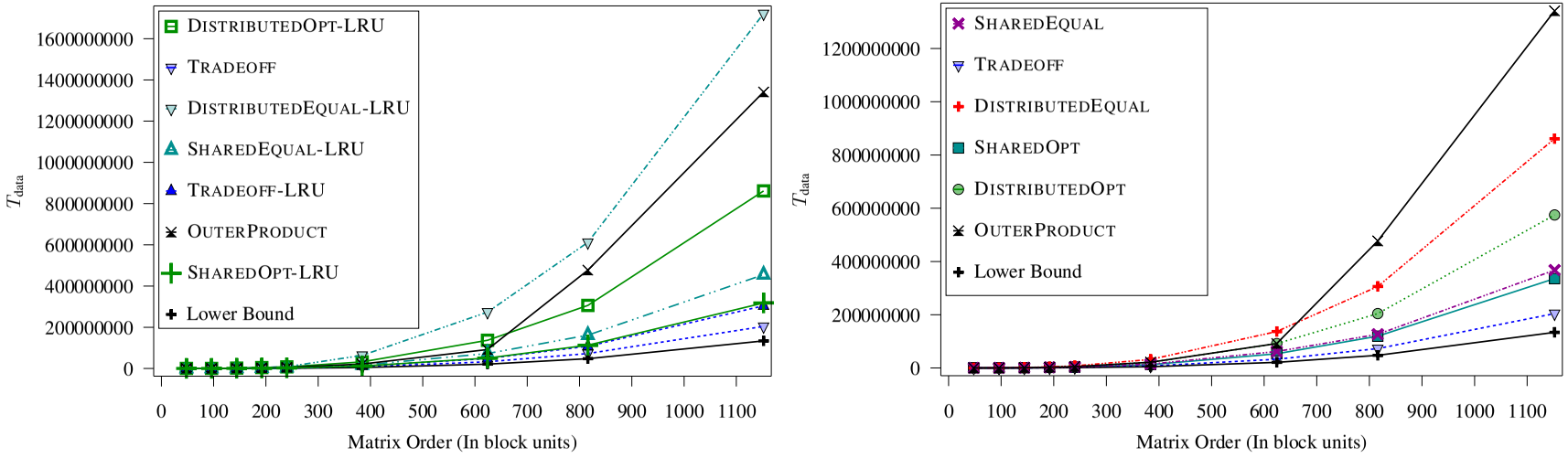
We have designed and analytically evaluated three algorithms minimizing shared cache misses, distributed cache misses, and the overall time spent in data movement. The behavior of all three algorithms has been validated through simulation in Section 1.4. In this section, we provide an implementation of these algorithms on a real quad-core CPU platform. Section 1.6 is the counterpart for a GPU platform.

The goal of this experimental section is to evaluate the impact of real hardware caches, which are not fully associative in practice. Also, we made the hypothesis that the cost of cache misses would dominate execution time, and this hypothesis need be confronted to reality.



(a) LRU setting and $C_D = 21$

(b) IDEAL setting and $C_D = 21$



(c) LRU setting and $C_D = 16$

(d) IDEAL setting and $C_D = 16$

Figure 1.9: Overall data time T_{data}, $C_S = 977$.

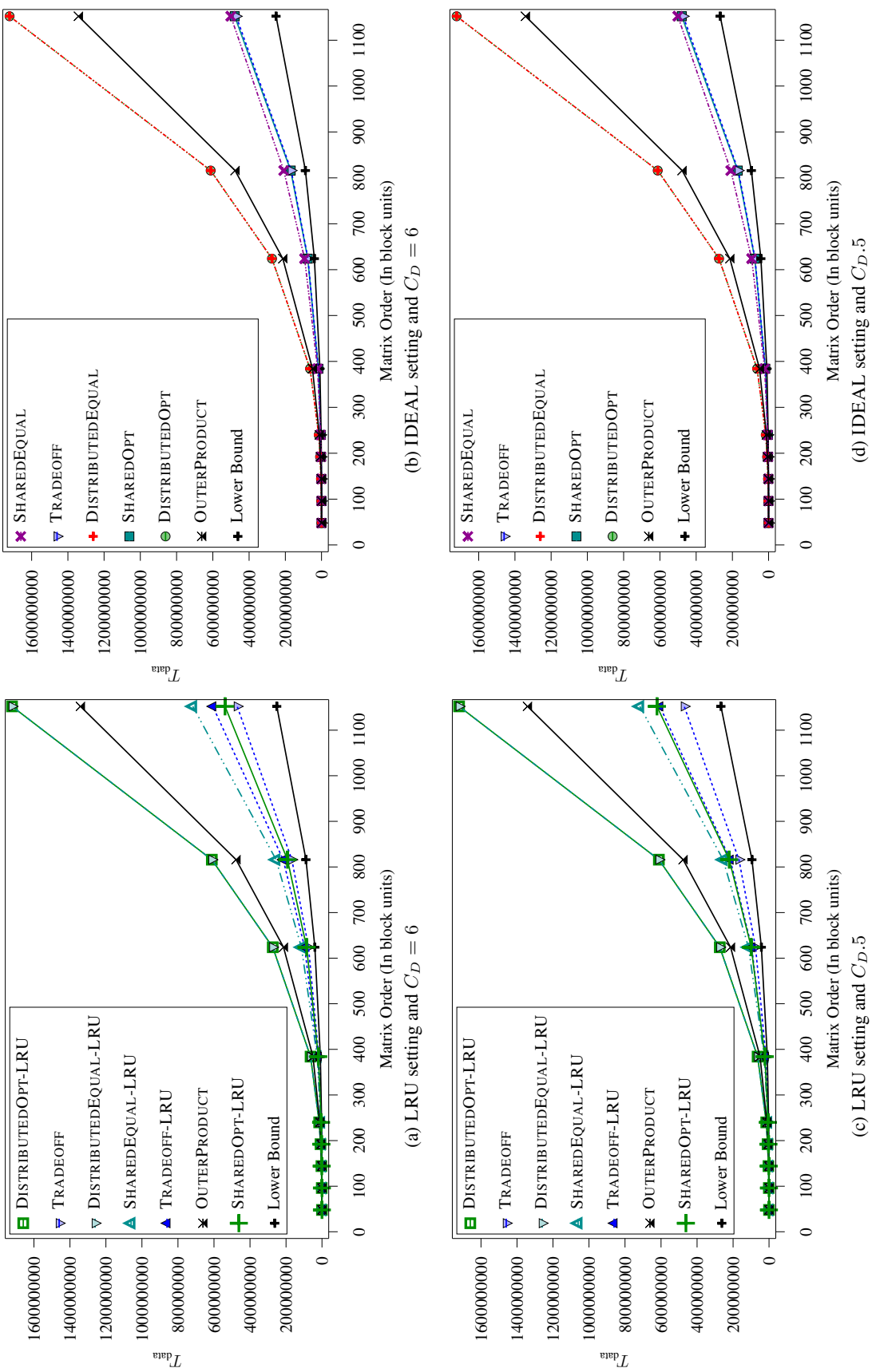
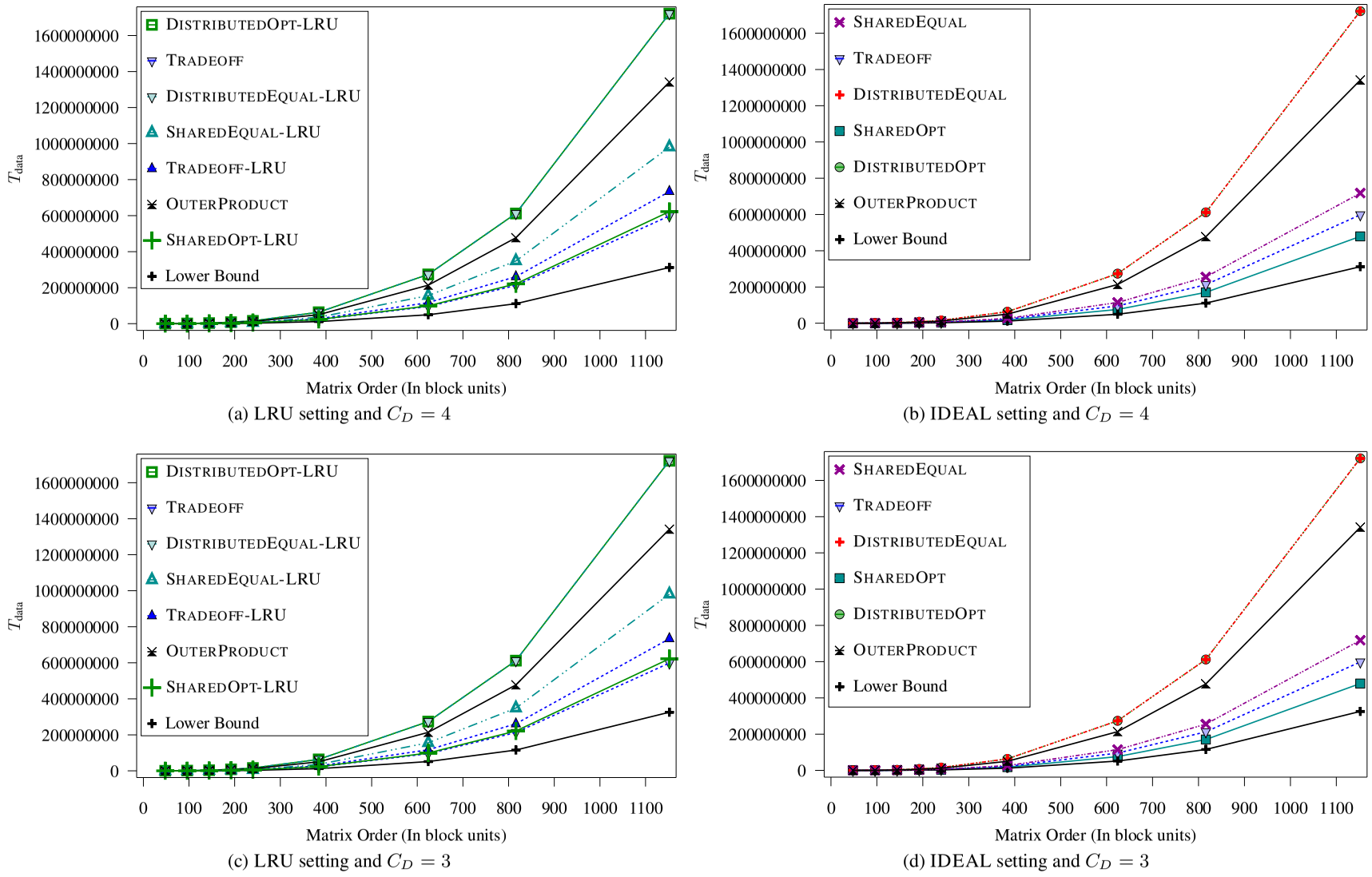


Figure 1.10: Overall data time T_{data}, $C_S = 245$.



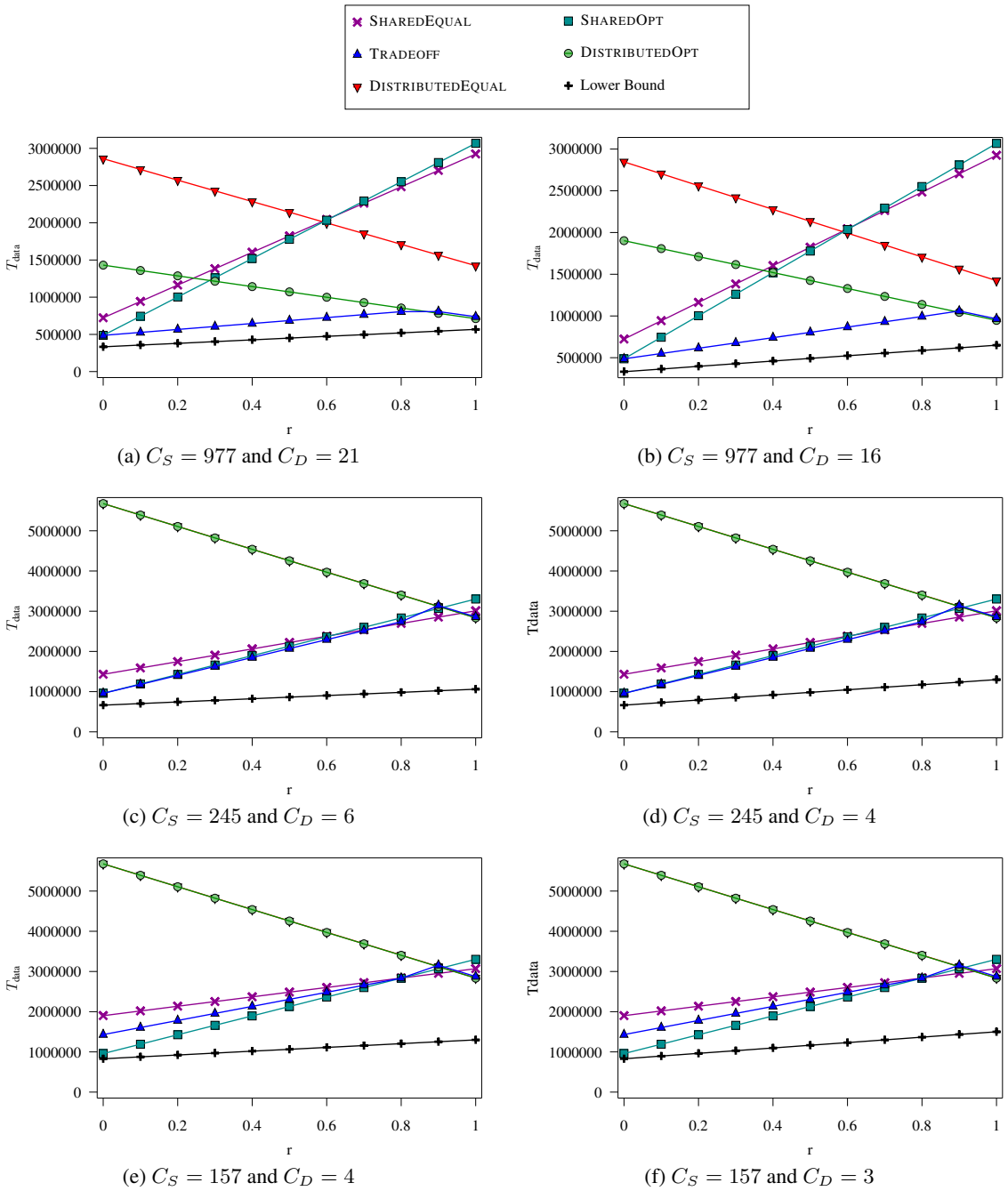
(a) LRU setting and $C_D = 4$

(b) IDEAL setting and $C_D = 4$

(c) LRU setting and $C_D = 3$

(d) IDEAL setting and $C_D = 3$

Figure 1.11: Overall data time T_{data}, $C_S = 157$.



(a) $C_S = 977$ and $C_D = 21$

(b) $C_S = 977$ and $C_D = 16$

(c) $C_S = 245$ and $C_D = 6$

(d) $C_S = 245$ and $C_D = 4$

(e) $C_S = 157$ and $C_D = 4$

(f) $C_S = 157$ and $C_D = 3$

Figure 1.12: Cache bandwidth impact on T_{data} in function of r, the ratio between σ_S and σ_D. Results are given for a square matrix of dimension $m = 384$.

1.5.1 Experimental setting

Hardware platform The hardware platform used in this study is based on two Intel Xeon E5520 processors running at 2.26 GHz. The Intel Xeon E5520 processor is a quad core processor with a 3 level cache hierarchy, L1 and L2 caches being private while L3 cache is shared across all cores. Quite naturally, we ignore the L1 cache and focus on the L2 cache that plays the role of the distributed cache C_D, while the L3 cache will be denoted C_S.

However, despite this simplification, there remains differences between this architecture and the theoretical model. First, these caches are not fully associative : C_D is 8-way associative while C_S is 16-way associative. Moreover, they do not implement an optimal omniscient data replacement policy, but instead they operate with the classical *LRU* policy. Furthermore, C_D is used to cache both data and instructions, while we considered only data in our study.

Processors	2x Intel Xeon E5520
Core arrangement	Processor 0: core 0-3 Processor 1: core 4-7
Core frequency	2.26GHz
QPI bandwidth	23.44 GB/s per link
L1 cache size	32 KB (data) / 32 KB (instruction)
L2 cache size	256 KB
L3 cache size	8 MB
Cache line size	64 B
Page size	4 KB / 2 MB (small/huge pages)
L1 data TLB	48/32 entries for small/huge pages
L2 TLB	512 entries for small pages
Memory type	8x 2 GB DDR3-1066, registered, ECC 3 channels (12 GB)on processor 0 2 channels (4 GB) on processor 1 8.53 GB/s per channel
Operating System	Ubuntu 8.04, Kernel 2.6.28.10 NUMA

Table 1.1: System configuration

The system also embeds 16GB of memory, 12GB being connected to the first processor, and 4GB to the second. This somewhat peculiar memory repartition is explained by the fact that our experiments will be conducted on the first processor only, thus giving it more local memory seems natural. Detailed characteristics of the system are displayed in Table 1.1.

Software framework For the experiments, the following software stack has been used:

- GCC 4.4.2 for compilation
- *Numactl* and *Libnuma* [65] for affinity
- Hwloc [20] for hardware topology detection (number of cores, cache sizes, ...)
- GotoBLAS2 [24] for BLAS Level 3 routines
- Intel MKL [57] for BLAS Level 3 routines
- Intel VTune [58] for cache misses sampling

Algorithms In the following experiments, we compare a large number of algorithmic versions. In addition to the vendor library, we have implemented five algorithms: two reference algorithms, the ScaLaPack outer-product [17] and the equal-layout algorithm [95], and the three optimized algorithms SHAREDMMRA, DISTRIBUTEDMMRA and TRADEOFFMMRA. Each of the latter three algorithms comes in two versions that differ in the size of the cache allocation: either we use the whole cache, as in the *ideal* versions described in Section 1.3, or we use half of it, the other half being used as a buffer for *LRU* policy. The equal-layout algorithm is derived along four versions: shared/distributed version, and *ideal*/*LRU* cache. Altogether, this leads to 12 versions, which are labeled as follows:

- PARALLEL is the vendor library (MKL or GotoBLAS2) parallel matrix product implementation
- OUTERPRODUCT is the outer product algorithm
- SHAREDOPT is SHAREDMMRA using the entire C_S
- SHAREDOPT-LRU is SHAREDMMRA using half of C_S, the other half being used as a buffer for LRU policy
- DISTRIBUTEDOPT is DISTRIBUTEDMMRA using the entire C_D
- DISTRIBUTEDOPT-LRU is DISTRIBUTEDMMRA using half of C_D (other half for LRU)
- TRADEOFF is TRADEOFFMMRA using the entire C_S and C_D
- TRADEOFF-LRU is the TRADEOFFMMRA using half of C_S and C_D (other half for LRU)
- DISTRIBUTEDEQUAL is the equal-layout algorithm [95] using the entire C_D
- DISTRIBUTEDEQUAL-LRU is the equal-layout algorithm [95] using half of C_D (other half for LRU)
- SHAREDEQUAL is the equal-layout algorithm [95] using the entire C_S
- SHAREDEQUAL-LRU is the equal-layout algorithm [95] using half of C_S (other half for LRU)

1.5.2 Performance results: execution time

The first performance metric that we naturally evaluate is the running time. Since our analysis is based on the assumption that execution time is limited by the cost of cache misses, we aim here at validating this hypothesis.

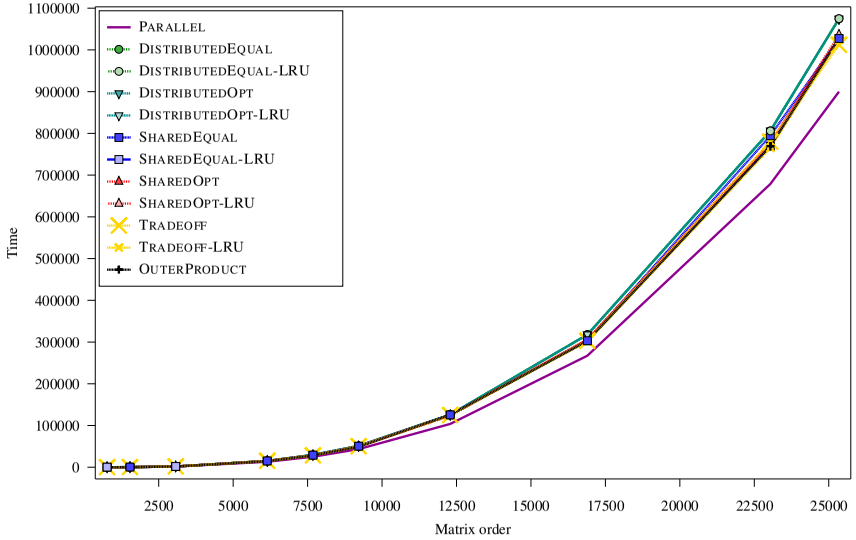
In Figures 1.13a and 1.13b, we plot the runtime of every algorithms using square blocks of 96 by 96 elements for BLAS calls. Indeed, it is widely assumed that memory latencies are hidden when using that block size. We also plot the running times of libraries when called on the whole matrices.

With both libraries, every algorithm offers almost the same performance: the gap between the slowest algorithm and the fastest one is only 5%. Moreover, using those parameters, all algorithms are able to reach up to 89% of the performance of libraries, TRADEOFF-LRU being the best performing algorithm in that experiment.

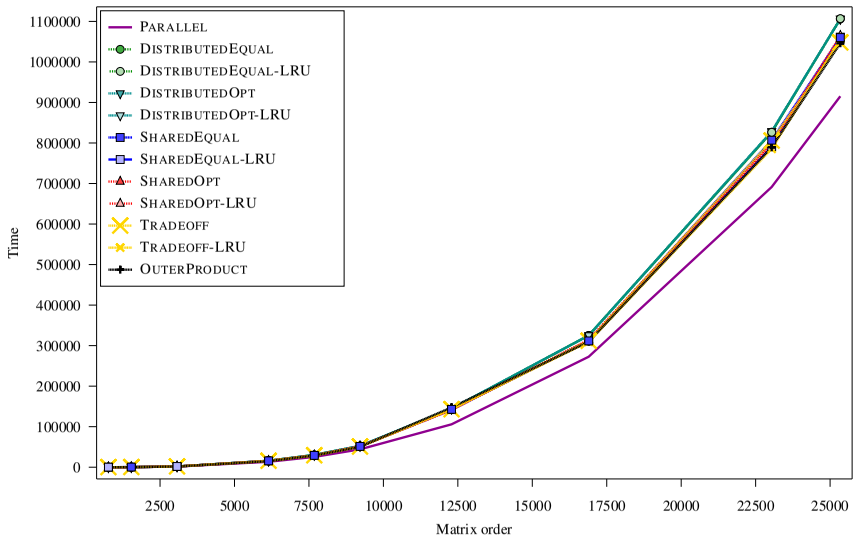
However, this low difference between each algorithms is increased when calling BLAS on smaller blocks. On Figures 1.13c and 1.13d, we did use 48 by 48 blocks of matrix coefficients as the block unit of our algorithms. We can see that the algorithms behave differently to that change, and that algorithms taking C_S into consideration offer the best performance. TRADEOFF-LRU being the best of them.

Altogether, the performance offered by our algorithms do not hold the comparison with both vendor libraries. Considering the fact that libraries are really low-level implementations whereas we aimed at design higher-level strategies, it seems natural. However, developing such low-level libraries requires a huge effort for the programmers, and everything needs to be done from scratch for each architecture. In addition, the scalability of both our algorithms and of these libraries should also be checked, but it is still hard to find general purpose x86 multicore processors with more than 8 cores, thus postponing the assessment for further work.

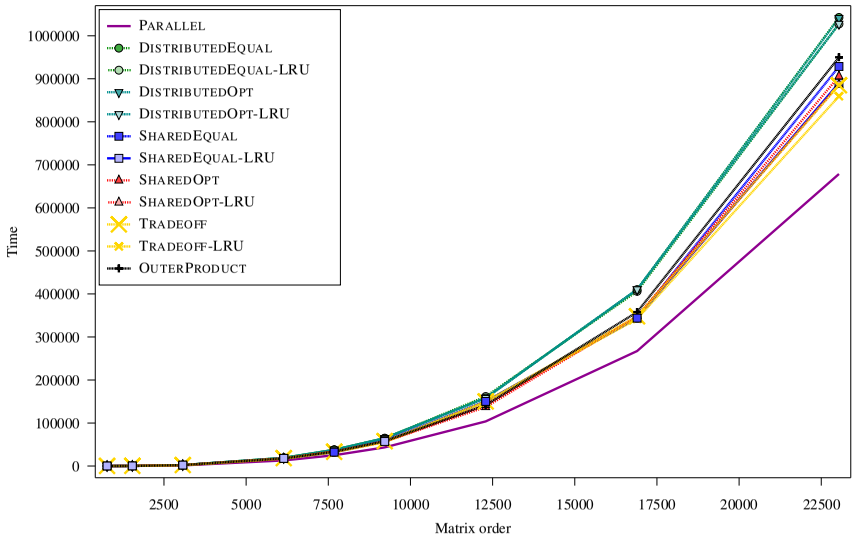
Furthermore, if we restrict the comparison to the algorithms that we have implemented, there is a



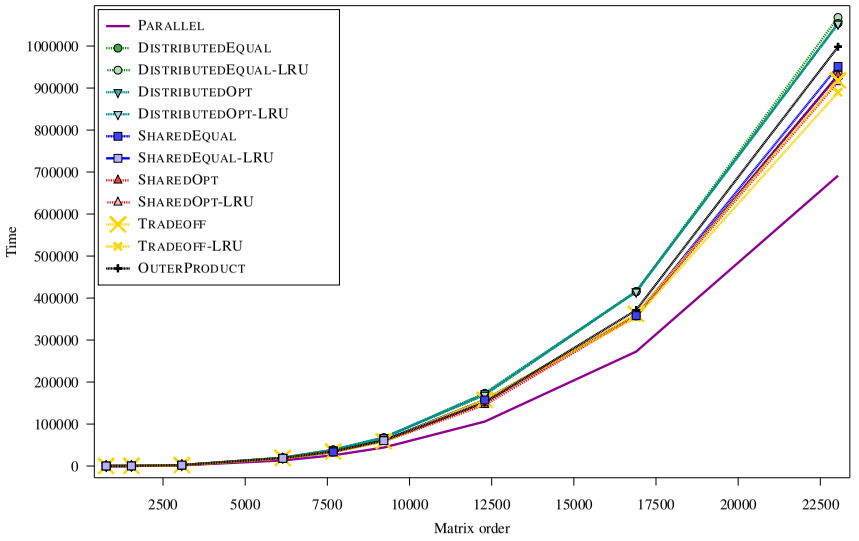
(a) $q = 96$, running times using GotoBLAS2 library.



(b) $q = 96$, running times using MKL library.



(c) $q = 48$, running times using GotoBLAS2 library.



(d) $q = 48$, running times using MKL library.

Figure 1.13: Running times according to matrix order.

significant difference between them only for the smallest block sizes, which are not used in practice because with such blocks, data reads and writes are not overlapped with computations.

In conclusion, cache misses do not seem to have an important impact on run-times (but M_S appears more important than M_D, as expected). Several other mechanisms interact in modern multicore processors, and a cache miss does not necessarily lead to a longer run-time than a cache hit because it might improve prefetching. Unfortunately, refining the model and taking other metrics like TLB misses of prefetching into consideration would render the analysis intractable.

Finally, we observe that three blocks of 96 by 96 matrix elements entirely fill C_D, thus diminishing the ability of MMRA to handle distributed caches properly, because the value of μ is obviously forced to 1. For the same reason, the value of λ is also limited. From this observation we think that MMRA would benefit a lot from larger caches, increasing the decision space.

1.5.3 Performance results: cache misses

We also study the behavior of our algorithms in terms of cache misses. The objective is twofold: we aim at validating the behavior of our algorithms, while giving a more precise analysis of the impact of cache misses on run-time.

Figures 1.14a, 1.14b, 1.14c and 1.14d show that for M_S, we observe what we expected: algorithms focusing on C_S are the best alternatives and algorithms focusing on distributed caches are the worst alternatives. For big matrices, MMRA sometimes performs better than libraries on this precise metric. Moreover, diminishing block size increases the gap between algorithms, similarly to what we observe for run-time.

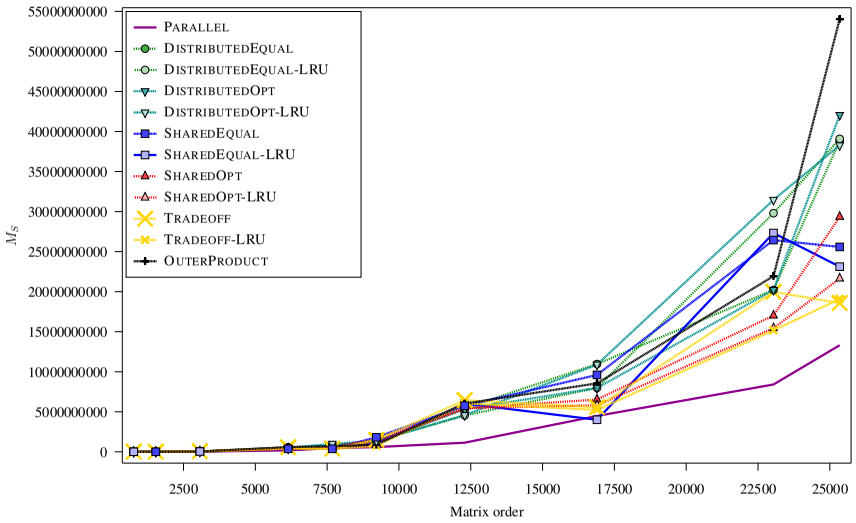
Interestingly, in some cases, increasing the size of the matrices decreases the number of shared cache misses. This behavior is due to the fact that C_S is 16-way associative whereas we assume full-associativity in our study. Fully-associative caches are extremely costly, and makes the mapping between cache and memory a combinatorial problem. That is why real hardware rather implement limited associativity caches. In those cases, the associativity has a non negligible impact on M_S.

However, even though we are able to reduce M_S, the run-time of our algorithms does not follow the same trend. On Figures 1.15a, 1.15b, 1.15c and 1.15d, we measure the number of shared caches misses that are not prefetched by the processor. Theses misses are the ones that actually slowdown the processor, thus having a direct impact on run-time. Our algorithms suffer from their more complicated access pattern, which drastically reduce prefetching. Conversely, libraries and outer product keep their access patterns simple enough, to fully benefit from hardware prefetchers, thus making most of their cache misses for free.

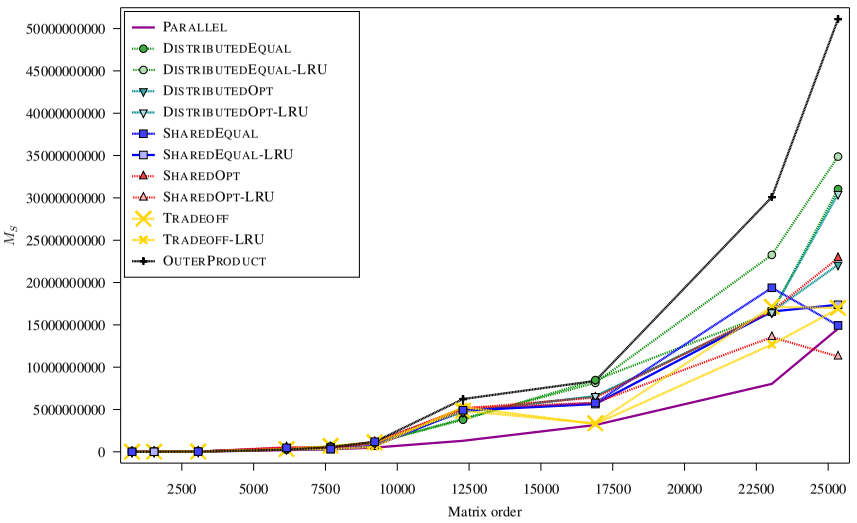
When considering the total number of distributed caches misses, the experiments (see Figures 1.16a, 1.16b, 1.16c and 1.16d) show that with blocks of 96 by 96 elements, the difference in performance between each algorithm is low, and algorithms focusing on C_D do not significantly reduce the number of distributed cache misses. Surprisingly in that case, both libraries make significantly higher M_D.

When we use smaller blocks, the difference in performance increases but we cannot observe the dominance of DISTRIBUTEDMMRA, though it is theoretically optimal. This small impact of algorithms on M_D is probably due to the fact that C_D are not dedicated to data only but also cache instructions. Therefore we do not benefit of the entire space for our tiling algorithms. It also seems that the behavior of data replacement policy is biased because data gets evicted in order to load instructions.

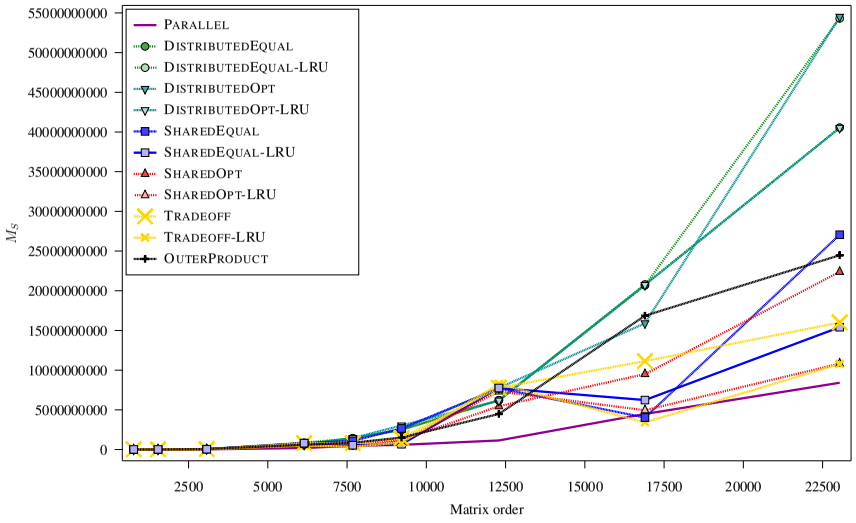
We now focus on the number of unprefetched distributed caches misses depicted in Figures 1.17a, 1.17b, 1.17c and 1.17d. Libraries clearly prefetch most part of their distributed cache misses and therefore pay significantly less misses than the other algorithms. Moreover, the algorithms focusing on M_D are the best alternatives with both block sizes, though results using smaller blocks are more regular.



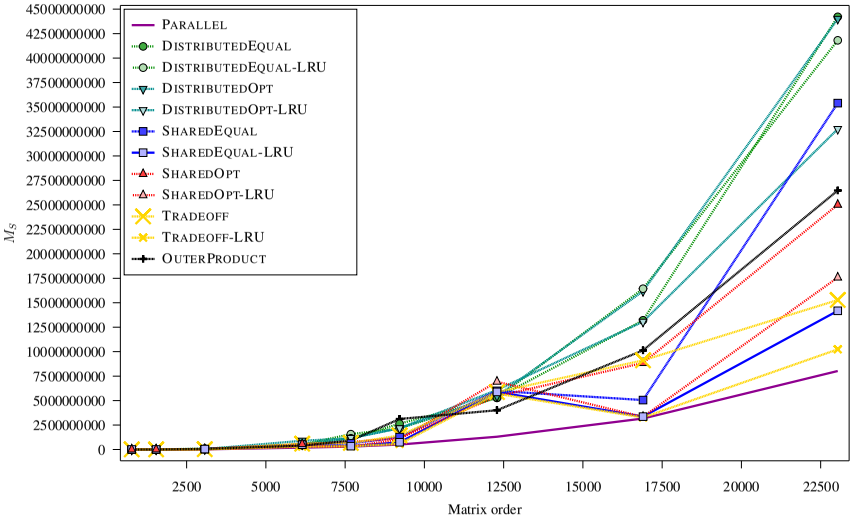
(a) $q = 96$, total M_S using GotoBLAS2 library.



(b) $q = 96$, total M_S using MKL library.

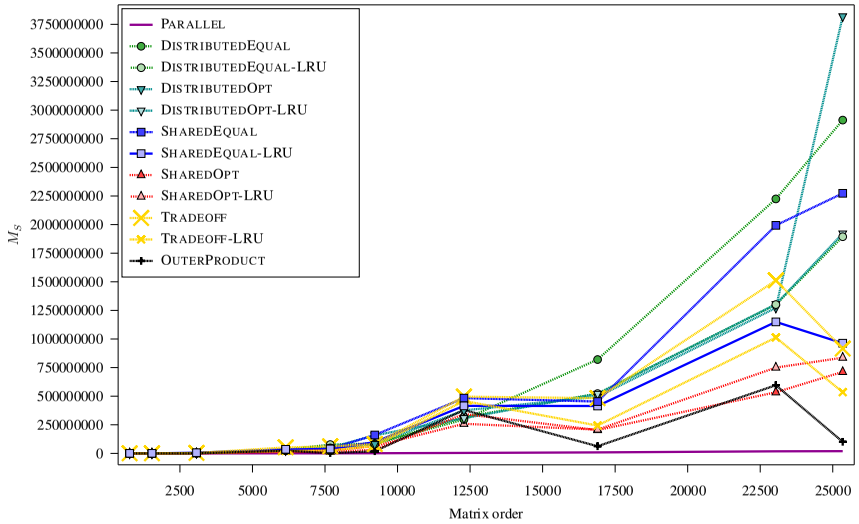


(c) $q = 48$, total M_S using GotoBLAS2 library.

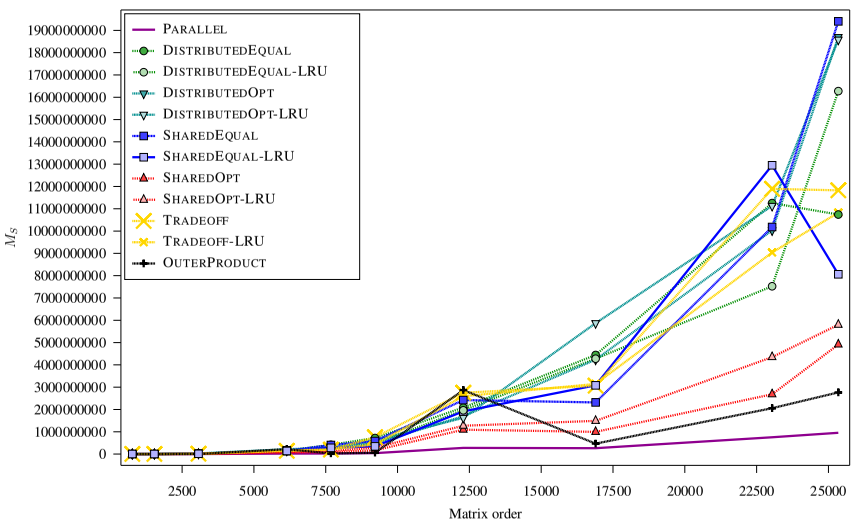


(d) $q = 48$, total M_S using MKL library.

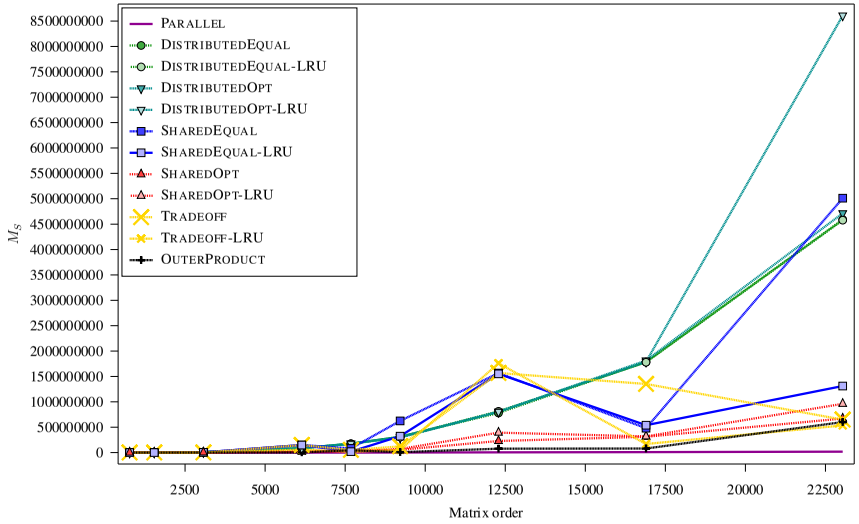
Figure 1.14: Total shared cache misses M_S according to matrix order.



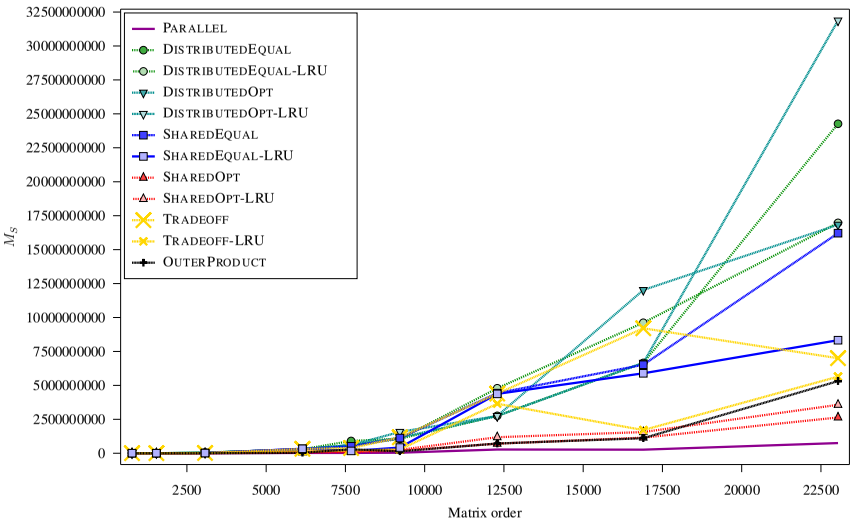
(a) $q = 96$, non prefetched M_S using GotoBLAS2 library.



(b) $q = 96$, non prefetched M_S using MKL library.

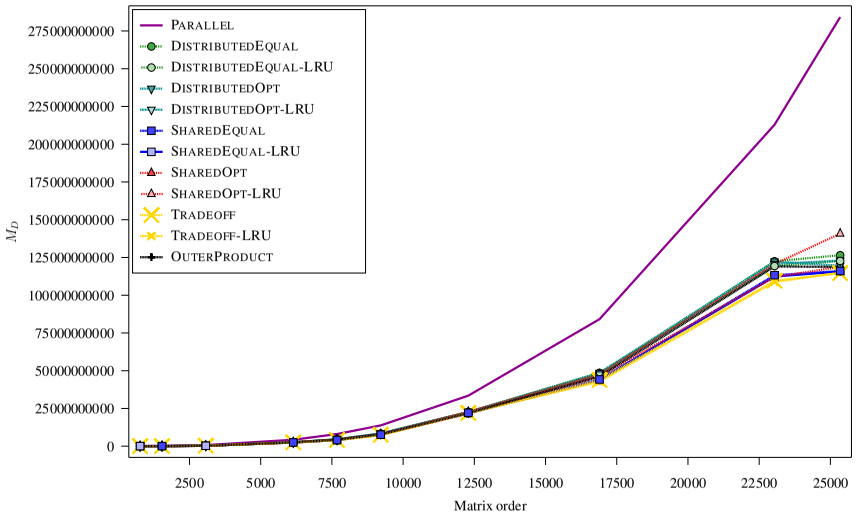


(c) $q = 48$, non prefetched M_S using GotoBLAS2 library.

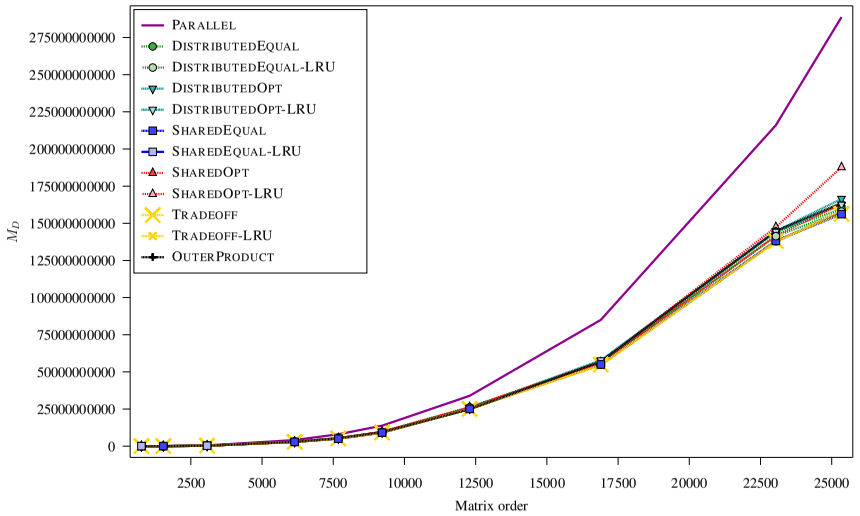


(d) $q = 48$, non prefetched M_S using MKL library.

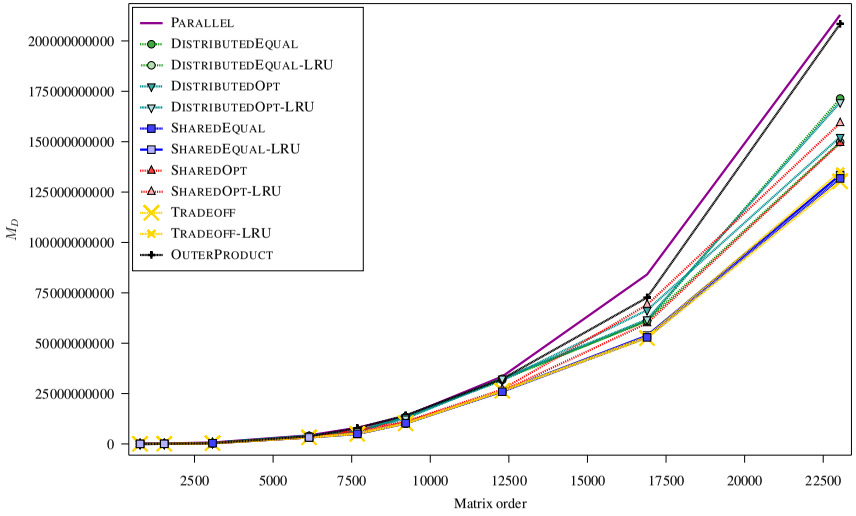
Figure 1.15: Non prefetched shared cache misses M_S according to matrix order.



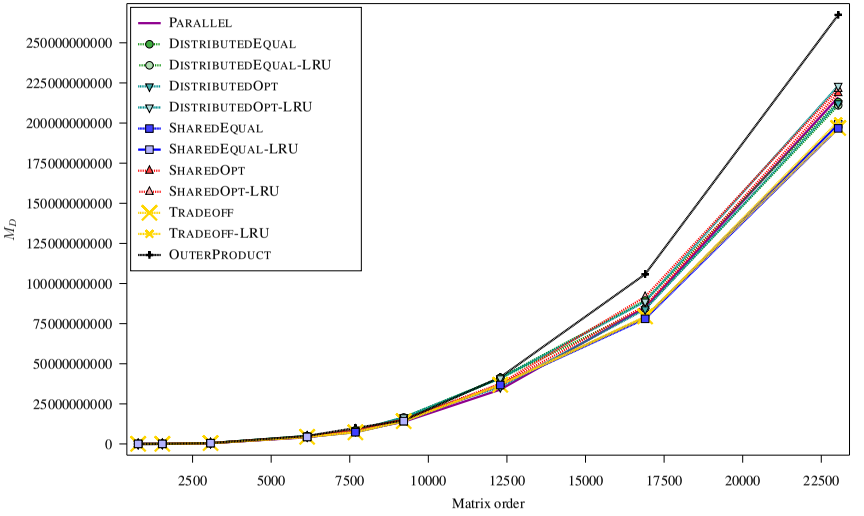
(a) $q = 96$, total M_D using GotoBLAS2 library.



(b) $q = 96$, total M_D using MKL library.

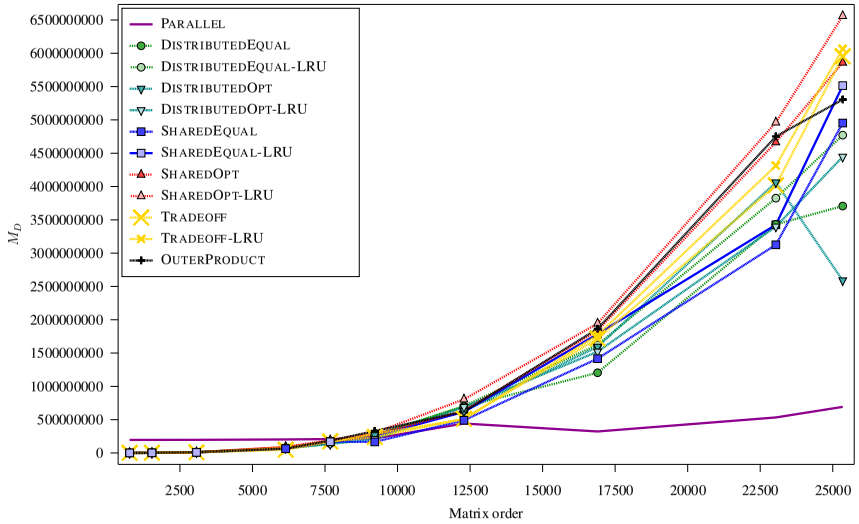


(c) $q = 48$, total M_D using GotoBLAS2 library.

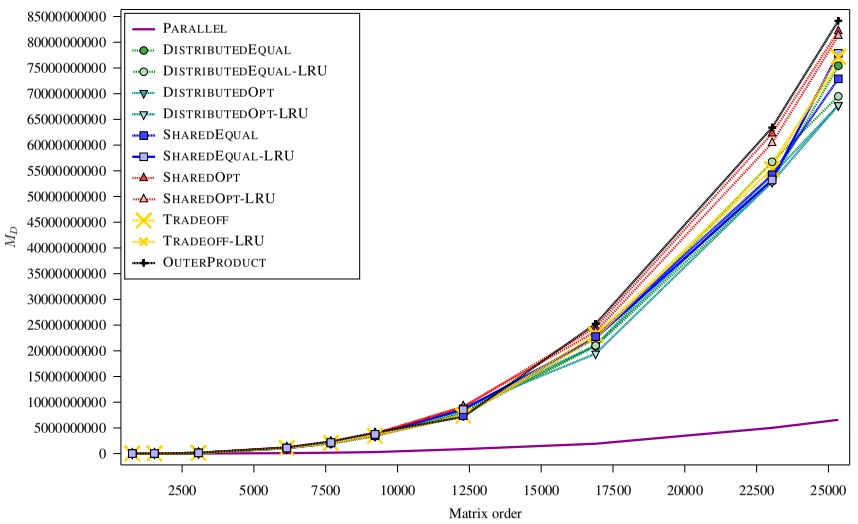


(d) $q = 48$, total M_D using MKL library.

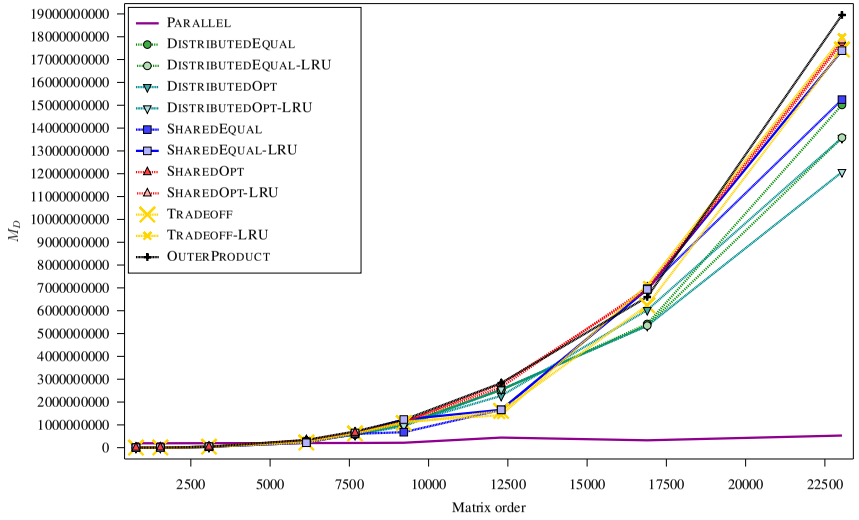
Figure 1.16: Total distributed cache misses M_D according to matrix order.



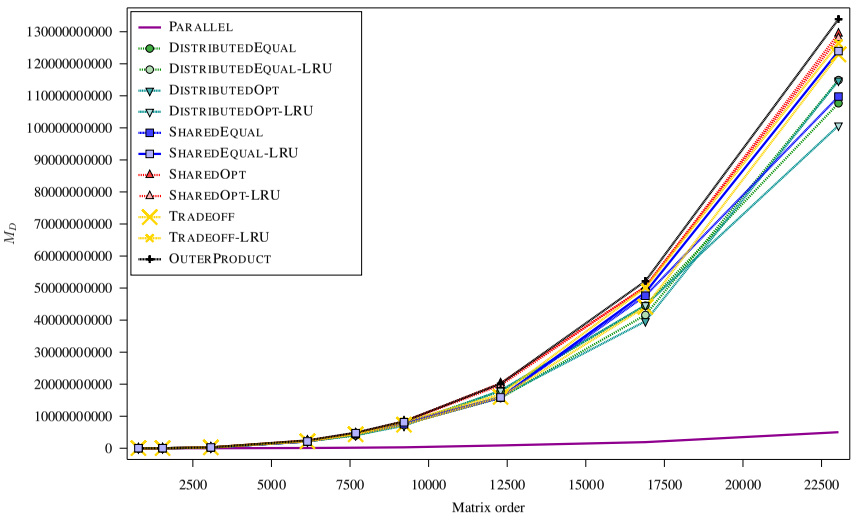
(a) $q = 96$, non prefetched M_D using GotoBLAS2 library.



(b) $q = 96$, non prefetched M_D using MKL library.



(c) $q = 48$, non prefetched M_D using GotoBLAS2 library.



(d) $q = 48$, non prefetched M_D using MKL library.

Figure 1.17: Non prefetched distributed cache misses M_D according to matrix order.

Furthermore, for distributed cache oriented algorithms, we may face false sharing issues. This problem occurs when two matrix elements of two different blocks of C are within the same cache line. The line gets modified by a core, thus triggering coherency protocol between the cores sharing that same line of cache. On Intel Nehalem processors, the cost is twice that of an unshared access [82], thus implying significant impact on run-time. Note that here again, the lower associativity of C_D has a higher impact on cache misses and significantly increases M_D.

Finally, it is difficult to ensure total ownership on the processor to a given application. Therefore, a few other processes sometimes "trash" the cache with their data, thus evicting matrix data from cache. On top of this, BLAS libraries often reorganize data so as to fulfill their own requirements, by sometimes making copies. All in all, these two trends disturb the behavior of our algorithms and increase the number of cache misses at each level.

As a conclusion, these experiments show that on current CPU architectures, it is hard to precisely predict and thus control the cache behavior of a given algorithm. Well known algorithms like matrix product, where memory behavior could be precisely handled, could benefit from having total control over caches. Having such a control over caches is equivalent to consider them as fast local memories. Such memories are often encountered on today's accelerator architectures like the Cell processor [61] or GPUs [6] [75]. We study the latter platform in Section 1.6.

1.6 Performance evaluation on GPU

As outlined above, one of the main drawbacks of current CPU architectures is the fact that caches are always managed by hardware, and hence cannot be used as regular low-latency memories. However, in the context of simple linear algebra kernels like matrix product, data could be handled manually at reasonable cost. Such low-latency memories can be found and programmed on GPUs. In this part, we hence focus on the adaptation of MMRA to GPUs and assess its performance on such hardware platforms.

1.6.1 Experimental setting

For the experiments, we use the same hardware platform as in Section 1.5.1, with an additional GPU card based on GT200 architecture from NVidia. The system runs on Yellow Dog Enterprise Linux for NVIDIA CUDA based on the 2.6.18 Linux kernel. The GT200 architecture is widespread in the HPC community through Tesla GPU computing devices, and is therefore a representative candidate for this study.

The card is a GeForce GTX 285, which embeds 240 cores running at 1.48 GHz, and 2GB of GDDR3 memory; CUDA 2.3 has been chosen as the programming environment. The 240 cores or Stream Processors (SPs) are grouped in 30 clusters called multi-processors (MPs). These multi-processors have a "shared" memory of 16KB shared across every SPs within the multi-processor. They also have 16384 32-bit private registers distributed among the SPs. Finally, the card has a memory space called "global" memory (of 2GB with in this setup) which is shared among every SPs, but which is significantly slower than "shared" memory and registers. The interested reader can find more details on GT200 and CUDA thread management model in [74].

1.6.2 Adaptation of MMRA to the GT200 Architecture

The hardware architecture of GPUs leads to revisit our tiling algorithm in order to cope with its specifics. GPUs have several levels of memory, including on-board RAM memory as well as on-chip

memory, which is order of magnitudes faster than on-board memory. This speed heterogeneity should therefore be taken into account, by fine tuning memory accesses at every level, thus leading us to choose TRADEOFFMMRA. The main idea of this adaptation is to consider the on-board RAM memory (or "global memory") as a shared cache of size C_S whereas each multi-processor will have a distributed cache of size C_D located in on-chip memory (which are called "shared" memories and registers in CUDA vocabulary).

Moreover our algorithm was originally designed for p^2 processors, which is not flexible and leads to underuse of the hardware. Therefore, we decided to modify TRADEOFFMMRA so as to handle $p \times q$ processors. Instead of tiling the $\alpha \times \alpha$ block of C in p^2 subblocks of size $\alpha/p \times \alpha/p$, we rather cut the block of C in $p \times q$ subblocks of size $\alpha/p \times \alpha/q$ (represented as light gray blocks in Figure 1.18).

Furthermore, we need to reserve additional memory in C_S for blocks of A and B in order to overlap PCI-E transfers between host and GPU with computation. We thus need to consider this in the computation of α. The new equation becomes:

$$\alpha^2 + 4\alpha\beta \leq C_S$$

For the computation of μ, it is necessary to consider both the fact that GPUs are efficient only with a high number of concurrent threads, and the fact that the block of C is not partitioned in square tiles anymore.

On GT200, the best hardware occupation is obtained when running more than 512 threads per multiprocessor. Note that on GPUs, threads truly are lightweight, which means that in the context of this study, a thread will process only a few matrix elements. Let γ be the number of threads around the first dimension (i.e. number of lines), and τ be the number of threads around the second dimension.

In order to handle rectangular tiles, let η be the number of columns of the block in C_D, and χ be its counterpart for the number of rows. Altogether, η must divide α/q and be a multiple of τ; while χ must divide α/p and be a multiple of γ. Such blocks are depicted in dark grey in Figure 1.18. All in all, each thread will process $(\eta \times \chi)/(\tau \times \gamma)$ matrix elements of C.

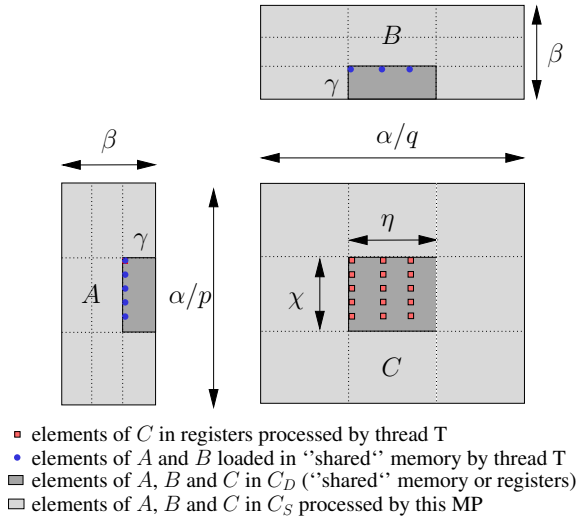


Figure 1.18: Tiling of matrices within a multi-processor

However, contrarily to the original TRADEOFFMMRA, $\chi \times \eta$ subblocks of C will not reside in the same memory than subblocks of A and B: in fact, C_D is no longer a homogeneous memory space. This feature comes from the fact that C is not shared among cores (or SPs), and can therefore be stored in

the private registers available on GT200. The red squares in Figure 1.18 are the elements of C loaded in registers and processed by a given thread T.

Conversely, A and B coefficients are shared, hence stored in "shared" memory. Moreover, for efficiency reasons, instead of loading only one column of A and one row of B, block-columns and block-rows of respectively $\gamma \times \chi$ and $\eta \times \gamma$ elements are loaded. Each thread loads therefore χ/γ elements of A and η/τ elements of B in "shared" memory at each step. For a given thread T, it corresponds to the blue circles depicted in Figure 1.18. Altogether, the parameter used by MMRA on GTX285 are given in Table 1.2.

Multi-processors on lines	$p = 6$
Multi-processors on columns	$q = 5$
Thread count on lines	$\gamma = 16$
Thread count on columns	$\tau = 32$
Total number of threads	$\tau \times \gamma = 512$
Number of lines per MP	$\chi = 80$
Number of columns per MP	$\eta = 96$
Registers used by elements of C per MP	50%
"Shared" memory used by elements of A and B per MP	70%

Table 1.2: Parameters for GeForce GTX 285 (GT200)

1.6.3 Performance results: execution time

In the following experiments, the results are obtained using three different algorithms:
- TRADEOFF is the adaptation of MMRA to GT200 architecture.
- SHAREDEQUAL allocates one third of C_S to each matrix. This setup is built on top of the MMRA kernel with different tiling parameters.
- CUBLAS calls the so-called vendor library made by NVidia. The calls are made in the same order than the classical outer product algorithm.

Every result given in the following corresponds to the average of 10 experimental results.

The first experiment focuses on the efficiency as well as the scalability of TRADEOFF compared to NVidia's library CUBLAS. As depicted on Figure 1.19, the run times of each algorithm is depicted according to the matrix dimension. TRADEOFF is the fastest algorithm in two cases, reducing the time required by CUBLAS by up to 13%. In the other cases, TRADEOFF is slower than CUBLAS by up to 40%. SHAREDEQUAL always is slower to TRADEOFF, but remains close.

Note that the runtime of CUBLAS does not scale linearly when the size of the matrices increases. In order to understand this behavior, the execution of each case was profiled using CUDA Visual Profiler, allowing the identification of the actual computation kernel call underlying below CUBLAS calls. As expected, different kernels are called, as shown on Table 1.3, and every case where TRADEOFF outperforms CUBLAS, CUBLAS uses the same CUBLAS computation kernel, which does not make any usage of GPU's specific hardware features like texture units. Moreover, in the adaptation of TRADEOFFMMRA to GPU, this kind of hardware units were ignored on purpose, in order to stay close to the theoretical model, and to keep the independence of the approach with respect to to the underlying hardware architecture. However, in order to be able to compete with vendor's libraries like CUBLAS, it would be necessary to use these specific dedicated hardware units (but it is not in the scope of this study).

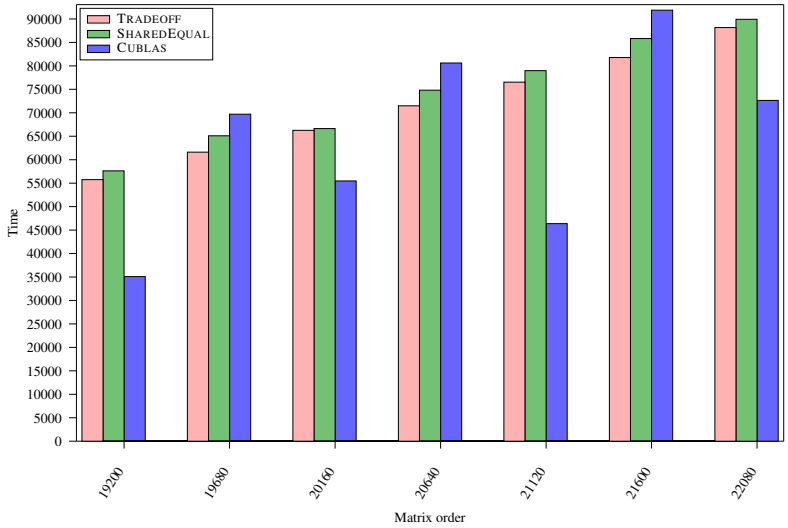


Figure 1.19: Running times on GTX285

19200	*sgemmNN*
19680	*sgemm_main_gld_hw_na_nb*
20160	*sgemm_main_gld_hw_na_nb_fulltile*
20640	*sgemm_main_gld_hw_na_nb*
21120	*sgemmNN*
21600	*sgemm_main_gld_hw_na_nb*
22080	*sgemm_main_gld_hw_na_nb_fulltile*

Table 1.3: CUBLAS kernel used in function of matrix size

1.6.4 Performance results: cache misses

In the next experiment, the number of shared cache misses M_S committed by each algorithm is evaluated. Results are depicted on Figure 1.20. As expected, TRADEOFF provide the best result since it makes up to 70% less shared cache misses than CUBLAS, thus proving the efficiency of our approach on GPUs. SHAREDEQUAL provides the same results than CUBLAS since it uses the same tiling at this level of cache.

On Figure 1.21, the actual time spent by the GPU moving data in order to serve the shared cache (i.e. the time penalty associated to M_S) is depicted. Unsurprisingly, TRADEOFF spend the least time, reducing the time spent by CUBLAS and SHAREDEQUAL by up to 32%.

Finally, in the last experiment, whose results are given in Figure 1.22, the number of distributed caches misses M_D committed by a multi-processor is depicted. In that case, the lowest number of M_D is given by SHAREDEQUAL. This is due to the fact that, even though they share the same kernel, SHAREDEQUAL uses a higher value of β than TRADEOFF. However in most cases, TRADEOFF ties SHAREDEQUAL, depending on its value of β. CUBLAS experiences between 1.9 and 3.4 times more distributed caches misses. However, if we consider for instance the performance on a matrix of size 21120, CUBLAS offers the best execution time while encountering the highest M_D. Therefore, the impact of M_D on performance seems to be low.

Altogether, the approach used by the MMRA proves its efficiency on alternative architectures like GPUs leading to cache efficient computation kernels. As the memory wall is closing in, the need of such

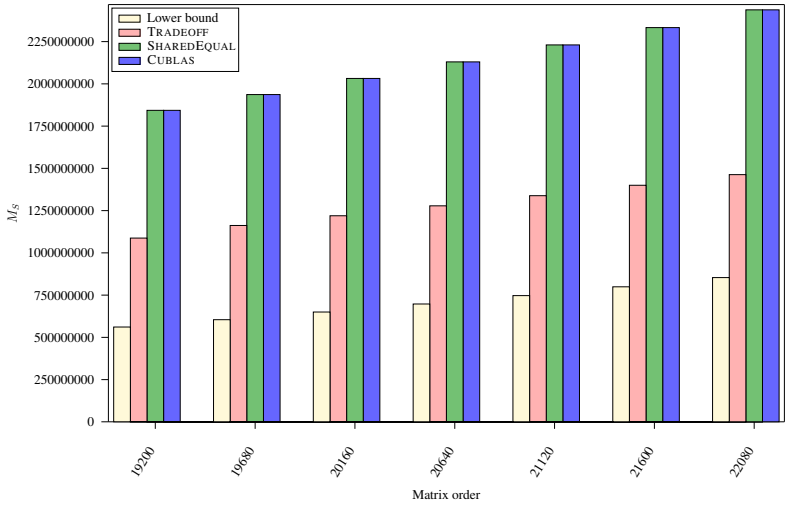


Figure 1.20: M_S on GTX285

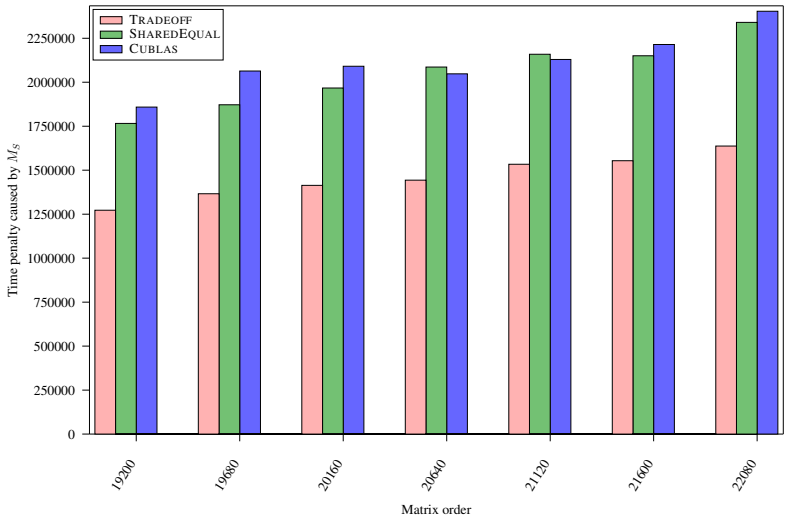


Figure 1.21: Time penalty caused by M_S on GTX285

algorithms is getting more important. This study proves that good results can be obtained using a high level, analytical and architecture-independent approach.

1.7 Conclusion

In this chapter, we have proposed cache-aware matrix product algorithms for multicore processors. Using a simple yet realistic model for multicore memory layout, we have extended a lower bound on cache misses, and proposed cache-aware algorithms. For both types of caches, shared an distributed, our algorithms reach a CCR which is close to the corresponding lower bound for large matrices. We also propose an algorithm for minimizing the overall data access time, which realizes a tradeoff between shared and distributed cache misses.

Using a dedicated cache simulator, we have assessed the impact of cache management policy on the number of cache misses. We have also verified that reserving half of the cache for prefetching

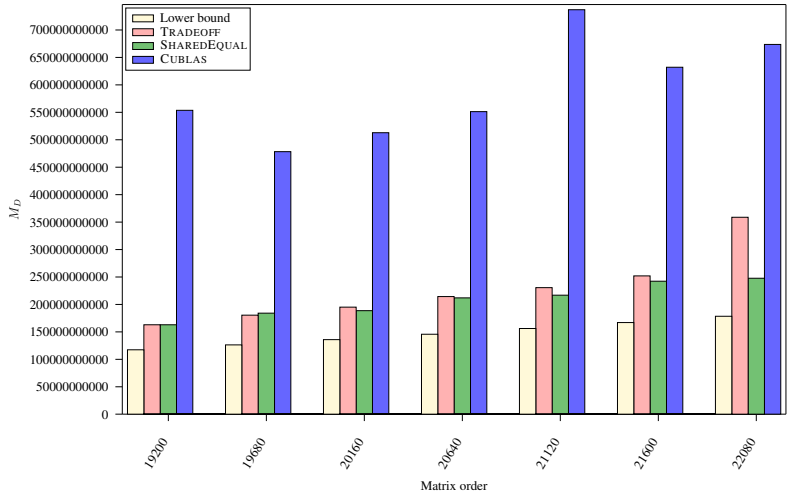


Figure 1.22: M_D on GTX285

alleviates the impact of *LRU* replacement policy. All algorithms have been validated on various cache sizes. Altogether, simulations have confirmed the dominance of our cache-aware algorithms in terms of cache misses and overall data access time.

We have also provided an extensive set of experimental results to compare the effective performance of the previous algorithms on actual multicore platforms. The objective was twofold: (i) assess the impact of cache misses onto runtime performance; and (ii) validate our high-level approach by comparing the algorithms to the vendor library routines. We have obtained mixed results with CPU platforms: due to intensive prefetching, cache misses have a lesser impact than the theoretical analysis has predicted. On the contrary, results with GPU platforms were quite encouraging and have nicely demonstrated the accuracy of our model, and the efficiency of our architecture-independent partitioning approach. Low-latency memories such as those provided by GPUs are much more promising devices than general purpose *LRU* caches for the tiling and partitioning strategies that lie at the heart of state-of-the-art linear algebra kernels.

Chapter 2

Tiled QR factorization algorithms

2.1 Introduction

In the previous chapter, we studied the matrix-matrix product on a multicore architecture, focusing on the impact of cache misses on performance. This kernel plays an important part in many scientific applications. In particular, it is generally used as a building block of more complex and widely used linear algebra operations. In this chapter, we aim at characterizing precisely the performance of such an operation: the QR factorization. However, contrarily to the previous chapter the focus of this study will not be set on the memory architecture. As a matter of a fact, QR algorithms rely on several highly optimized kernels which are themselves optimized for cache hierarchies. We will rather focus on exploiting the potential parallelism of QR factorization so as to match that offered by multicore processors.

In this context, we will study the QR factorization especially in the case of tall rectangular matrices. More precisely, given an m-by-n matrix A with $n \leq m$, we consider the computation of its QR factorization, which is the factorization $A = QR$, where Q is an m-by-n unitary matrix ($Q^H Q = I_n$), and R is upper triangular.

The QR factorization is the time consuming stage of some important numerical computations. The QR factorization of an m-by-n matrix with $n \leq m$ is needed for solving a linear least squares with m equations (observations) and n unknowns. The QR factorization of an m-by-n matrix with $n \leq m$ is used to compute an orthogonal basis (the Q-factor) of the column span of the initial matrix A. For example, all block iterative methods (used to solve large sparse linear systems of equations or computing some relevant eigenvalues of such systems) require orthogonalizing a set of vectors at each step of the process. For these two usage examples, while $n \leq m$, n can range from $n \ll m$ to $n \lessapprox m$. We note that the extreme case $n = m$ is also relevant: the QR factorization of a matrix can be used to solve (square) linear systems of equations (in that case $n = m$). While this requires twice as many flops as an LU factorization, using a QR factorization (a) is unconditionally stable (Gaussian elimination with partial pivoting or pairwise pivoting is not) and (b) avoids pivoting so it may well be faster in some cases (despite requiring twice as many flops).

To obtain a QR factorization, we consider algorithms which apply a sequence of m-by-m unitary transformations, U_i, ($U_i^H U_i = I$), $i = 1, \ldots, \ell$, on the left of the matrix A, such that after ℓ transformations the resulting matrix $R = U_\ell \ldots U_1 A$ is upper triangular, in which case, R is indeed the R-factor of the QR factorization. The Q-factor (if needed) can then be obtained by computing $Q = U_1^H \ldots U_\ell^H$. These types of algorithms are in regular use, e.g. in the LAPACK and ScaLAPACK libraries, and are favored over others algorithms (Cholesky QR or Gram-Schmidt) for their stability.

The unitary transformation U_i is chosen so as to introduce some zeros in the current update matrix $U_{i-1} \ldots U_1 A$. The two basic transformations are Givens rotations and Householder reflections. One

Givens rotation introduces one additional zero; the whole triangularization requires $mn - n(n+1)/2$ Givens rotations for $n < m$. One elementary Householder reflection simultaneously introduces $m - i$ zeros in position $i + 1$ to m in column i; the whole triangularization requires n Householder reflections for $n < m$. (See LAPACK subroutine $GEQR2$.) The LAPACK $GEQRT$ subroutine constructs a compact WY representation to apply a sequence of i_b Householder reflections, this enables one to introduce the appropriate zeros in i_b consecutive columns and thus leverage optimized Level 3 BLAS subroutines during the update. The blocking of Givens rotations is also possible but is more costly in terms of flops.

The main interest of Givens rotations over Householder transformations is that one can concurrently introduce zeros using disjoint pairs of rows, in other words, two transformations U_i and U_{i+1} may be applicable concurrently. This is not possible using the original Householder reflection algorithm since the transformations work on whole columns and thus does not exhibit this type of intrinsic parallelism forcing this kind of Householder reflections to be applied sequentially. The advantage of Householder reflections over Givens rotations is that, first, Householder reflections perform less flops, and second, the compact WY transformation enables high sequential performance of the algorithm. In a multicore setting, where data locality and parallelism are crucial algorithmic characteristics for enabling performance, the tiled QR factorization algorithm combines both ideas: use of Householder reflections for high sequential performance and use of a scheme ala Givens rotations to enable parallelism within cores. In essence, one can think (i) either of the tiled QR factorization as a Givens rotation scheme but on tiles (m_b-by-n_b submatrices) instead of on scalars (1-by-1 submatrices) as in the original scheme, (ii) or of it as a blocked Householder reflection scheme where each reflection is confined to an extent much less than the full column span, which enables concurrency with other reflections.

Tiled QR factorization in the context of multicore architectures has been introduced in [21, 22, 81]. Initially the focus was on square matrices and the sequence of unitary transformations presented was analogous to SAMEH-KUCK [83], which corresponds to reducing the panels with flat trees. The possibility of using any tree in order to either maximize parallelism or minimize communication is explained in [31].

The focus of this work is in maximizing parallelism. Stemming from the observation that a binary tree is best for tall and skinny matrices and a flat tree is best for square matrices, Hadri et al. [46], propose to use trees which combine flat trees at the bottom level with a binary tree at the top level in order to exhibit more parallelism. Our theoretical and experimental work explains that we can adapt FIBONACCI [73] and GREEDY [27, 28] to tiles, resulting in yet better algorithms in terms of parallelism. Moreover our new algorithms do not have any tuning parameter such as the domain size in the case of [46].

The focus of this study is not in trying to reduce communication (data movement between memory hierarchy) to a minimum. Relatively low level of communication is naturally achieved by the algorithm by tiling the operations. How to optimize the trade-off communication and parallelism is out of the scope of this chapter. For this reason, we consider square tiling with constant tile size. In order to increase parallelism, we use so called TT kernels which are more parallel but performs potentially more communication and are less efficient in sequential than the TS kernels. (A longer discussion on the issue can be found in Section 2.2.1.) This is another trade-off that we made and we opted for as much parallelism as possible.

We can quote three studies which use some kind of rectangular tiling. Demmel et al. [31] sequentially process rectangular tiles with a recursive QR factorization algorithm (which is communication optimal in sequential) and then uses reduction trees to perform the QR factorization in parallel. Experimental results are given using a binary tree on tall and skinny matrices. The same algorithms is used on the grid (grid of clusters) in [1]. The ScaLAPACK algorithm is used independently on each cluster on a large parallel distributed rectangular tile; then, a binary tree is used at the grid level among the clusters.

Demmel et al. [32] use a binary tree on top of a flat tree for tall and skinny matrices. The binary tree is therefore used on rectangular tiles. The flat tree is used locally on the nodes to reduce sequential communication, while the binary tree is used within the nodes to increase parallelism. Finally, the approach of Hadri et al. [46] is not only interesting in term of parallelism to tackle various matrix shapes, it is also interesting in reducing communication (same approach in this case as in [32]) and enables the use of TS kernels.

The sequential kernels of the Tiled QR factorization (executed on a core) are made of standard blocked algorithms ala LAPACK encoded in kernels; the development of these kernels is well understood. The focus of this chapter is on improving the overall degree of parallelism of the algorithm. Given a p-by-q tile matrix, we seek to find an appropriate sequence of unitary transformations on the tiled matrix so as to maximize parallelism (minimize critical path length). We will get our inspiration in previous work from the 70s/80s on Givens rotations where the question was somewhat related: given an m-by-n matrix, find an appropriate sequence of Givens rotations as to maximize parallelism. This question is essentially answered in [27, 28, 73, 83]; we call this class of algorithms "coarse-grain algorithms."

Working with tiles instead of scalars, we introduce four essential differences between the analysis and the reality of the tiled algorithms and the coarse-grain algorithms. First, while there are only two states for a scalar (nonzero or zero), a tile can be in three states (zero, triangle or full). Second, there are more operations available on tiles to introduce zeros; we have a total of three different tasks which can introduce zeros in a matrix. Third, the factorization and the update are dissociated to enable factorization stages to overlap with update stages. In the coarse-grain algorithm, the factorization and the associated update are considered as a single stage. Fourth and last, while coarse-grain algorithms have only one task, we end up with six different tasks, which have different computational weights; this dramatically complicates the critical path analysis of the tiled algorithms.

While the GREEDY algorithm is optimal for "coarse-grain algorithms", we show that it is not in the case of tiled algorithms. We are unable to devise an optimal algorithm at this point, but we can prove that both GREEDY and FIBONACCI are asymptotically optimal for all matrices of size $p = q^2 f(q)$, where f is any function such that $\lim_{+\infty} f = 0$. This result applies to all matrices where p and q are proportional, $p = \lambda q$, with $\lambda \geq 1$, thereby encompassing many important situations in practice (least squares).

This chapter is organized as follows. Section 2.2 reviews the numerical kernels needed to perform a tiled QR factorization, and introduces elimination lists, which enable us to formally define tiled algorithms. Section 2.3 presents the core algorithmic contributions of this chapter. One major result is the asymptotic optimality of two new tiled algorithms, FIBONACCI and GREEDY. Section 2.4 is devoted to numerical experiments on multicore platforms. For tall matrices ($p \geq 2q$), these experiments confirm the superiority of the new algorithms over state-of-the-art solutions of the PLASMA library [21, 22, 31, 46]. Finally, we provide some concluding remarks in Section 2.5.

2.2 The QR factorization algorithm

Tiled algorithms are expressed in terms of tile operations rather than elementary operations. Each tile is of size $n_b \times n_b$, where n_b is a parameter tuned to squeeze the most out of arithmetic units and memory hierarchy. Typically, n_b ranges from 80 to 200 on state-of-the-art machines [3]. Algorithm 4 outlines a naive tiled QR algorithm, where loop indices represent tiles:

In Algorithm 4, k is the panel index, and $elim(i, piv(i, k), k)$ is an orthogonal transformation that combines rows i and $piv(i, k)$ to zero out the tile in position (i, k). However, this formulation is somewhat misleading, as there is much more freedom for QR factorization algorithms than, say, for Cholesky algorithms (and contrarily to LU elimination algorithms, there are no numerical stability issues). For

Algorithm 4: Naive QR algorithm for a tiled $p \times q$ matrix.

```
for k = 1 to min(p, q) do
    for i = k + 1 to p do
        elim(i, piv(i, k), k)
```

Operation	Panel		Update	
	Name	Cost	Name	Cost
Factor square into triangle	$GEQRT$	4	$UNMQR$	6
Zero square with triangle on top	$TSQRT$	6	$TSMQR$	12
Zero triangle with triangle on top	$TTQRT$	2	$TTMQR$	6

Table 2.1: Kernels for tiled QR. The unit of time is $\frac{n_b^3}{3}$ floating-point operations, where n_b is the block-size.

instance in column 1, the algorithm must eliminate all tiles $(i, 1)$ where $i > 1$, but it can do so in several ways. Take $p = 6$. Algorithm 4 uses the transformations

$$elim(2, 1, 1), elim(3, 1, 1), elim(4, 1, 1), elim(5, 1, 1), elim(6, 1, 1)$$

But the following scheme is also valid:

$$elim(3, 1, 1), elim(6, 4, 1), elim(2, 1, 1), elim(5, 4, 1), elim(4, 1, 1)$$

In this latter scheme, the first two transformations $elim(3, 1, 1)$ and $elim(6, 4, 1)$ use distinct pairs of rows, and they can execute in parallel. On the contrary, $elim(3, 1, 1)$ and $elim(2, 1, 1)$ use the same pivot row and must be sequentialized. To complicate matters, it is possible to have two orthogonal transformations that execute in parallel but involve zeroing a tile in two different columns. For instance we can add $elim(6, 5, 2)$ to the previous transformations and run it concurrently with, say, $elim(2, 1, 1)$. Any tiled QR algorithm will be characterized by an *elimination list*, which provides the ordered list of the transformations used to zero out all the tiles below the diagonal. This elimination list must obey certain conditions so that the factorization is valid. For instance, $elim(6, 5, 2)$ must follow $elim(6, 4, 1)$ and $elim(5, 4, 1)$ in the previous list, because there is a flow dependence between these transformations. Note that, although the elimination list is given as a totally ordered sequence, some transformations can execute in parallel, provided that they are not linked by a dependence: in the example, $elim(6, 4, 1)$ and $elim(2, 1, 1)$ could have been swapped, and the elimination list would still be valid.

Before formally stating the conditions that guarantee the validity of (the elimination list of) an algorithm, we explain how orthogonal transformations can be implemented.

2.2.1 Kernels

To implement a given orthogonal transformation $elim(i, piv(i, k), k)$, one can use six different kernels, whose costs are given in Table 2.1. In this table, the unit of time is the time to perform $\frac{n_b^3}{3}$ floating-point operations.

There are two main possibilities to implement an orthogonal transformation $elim(i, piv(i, k), k)$: The first version eliminates tile (i, k) with the *TS (Triangle on top of square)* kernels, as shown in Algorithm 5:

Algorithm 5: Elimination $elim(i, piv(i,k), k)$ via *TS (Triangle on top of square)* kernels.

$GEQRT(piv(i,k), k)$
$TSQRT(i, piv(i,k), k)$
for j $= k+1$ *to* q **do**
  $UNMQR(piv(i,k), k, j)$
  $TSMQR(i, piv(i,k), k, j)$

Here the tile panel $(piv(i,k), k)$ is factored into a triangle (with $GEQRT$). The transformation is applied to subsequent tiles $(piv(i,k), j)$, $j > k$, in row $piv(i,k)$ (with $UNMQR$). Tile (i,k) is zeroed out (with $TSQRT$), and subsequent tiles (i,j), $j > k$, in row i are updated (with $TSMQR$). The flop count is $4 + 6 + (6+12)(q-k) = 10 + 18(q-k)$ (expressed in same time unit as in Table 2.1). Dependencies are the following:

$$
\begin{array}{ll}
GEQRT(piv(i,k),k) \prec TSQRT(i,piv(i,k),k) & \\
GEQRT(piv(i,k),k) \prec UNMQR(piv(i,k),k,j) & \text{for } j > k \\
UNMQR(piv(i,k),k,j) \prec TSMQR(i,piv(i,k),k,j) & \text{for } j > k \\
TSQRT(i,piv(i,k),k) \prec TSMQR(i,piv(i,k),k,j) & \text{for } j > k
\end{array}
$$

Note that $TSQRT(i, piv(i,k), k)$ and $UNMQR(piv(i,k), k, j)$ can be executed in parallel, as well as $UNMQR$ operations on different columns $j, j' > k$. With an unbounded number of processors, the parallel time is thus $4 + 6 + 12 = 22$ time-units.

The second approach to implement the orthogonal transformation $elim(i, piv(i,k), k)$ is with the *TT (Triangle on top of triangle)* kernels, as shown in Algorithm 6:

Algorithm 6: Elimination $elim(i, piv(i,k), k)$ via *TT (Triangle on top of triangle)* kernels.

$GEQRT(piv(i,k), k)$
$GEQRT(i, k)$
for j $= k+1$ *to* q **do**
  $UNMQR(piv(i,k), k, j)$
  $UNMQR(i, k, j)$
$TTQRT(i, piv(i,k), k)$
for j $= k+1$ *to* q **do**
  $TTMQR(i, piv(i,k), k, j)$

Here both tiles $(piv(i,k), k)$ and (i,k) are factored into a triangle (with $GEQRT$). The corresponding transformations are applied to subsequent tiles $(piv(i,k), j)$ and (i,j), $j > k$, in both rows $piv(i,k)$ and i (with $UNMQR$). Tile (i,k) is zeroed out (with $TTQRT$), and subsequent tiles (i,j), $j > k$, in row i are updated (with $TTMQR$). The flop count is $2(4+6(q-k))+2+6(q-k) = 10+18(q-k)$,

just as before. Dependencies are the following:

$$
\begin{array}{ll}
GEQRT(piv(i,k),k) \prec UNMQR(piv(i,k),k,j) & \text{for } j > k \\
GEQRT(i,k) \prec UNMQR(i,k,j) & \text{for } j > k \\
GEQRT(piv(i,k),k) \prec TTQRT(i,piv(i,k),k) & \\
GEQRT(i,k) \prec TTQRT(i,piv(i,k),k) & \\
TTQRT(i,piv(i,k),k) \prec TTMQR(i,piv(i,k),k,j) & \text{for } j > k \\
UNMQR(piv(i,k),k,j) \prec TTMQR(i,piv(i,k),k,j) & \text{for } j > k \\
UNMQR(i,k,j) \prec TTMQR(i,piv(i,k),k,j) & \text{for } j > k
\end{array}
$$

Now the factor operations in row $piv(i,k)$ and i can be executed in parallel. Moreover, the $UNMQR$ updates can be run in parallel with the $TTQRT$ factorization. Thus, with an unbounded number of processors, the parallel time is $4 + 6 + 6 = 16$ time-units.

In Algorithm 5 and 6, it is understood that if a tile is already in triangle form, then the associated $GEQRT$ and update kernels are not applied.

All the new algorithms introduced in this chapter are based on *TT* kernels. From an algorithmic perspective, *TT* kernels are more appealing than *TS* kernels, as they offer more parallelism. More precisely, we can always break a *TS* kernel into two *TT* kernels: We can replace a $TSQRT(i,piv(i,k),k)$ (following a $GEQRT(piv(i,k),k)$) by a $GEQRT(i,k)$ and a $TTQRT(i,piv(i,k),k)$. A similar transformation can be made for the updates. Hence a *TS*-based tiled algorithm can always be executed with *TT* kernels, while the converse is not true. However, the *TS* kernels provide more data locality, they benefit form a very efficient implementation (see Section 2.4), and several existing algorithms use these kernels. For all these reasons, and for comprehensiveness, our experiments will compare approaches based on both kernel types.

Currently (April 2011), the PLASMA library only contains *TS* kernels. We have mapped the PLASMA algorithm to *TT* kernel algorithm using this conversion. Going from a *TS* kernel algorithm to a *TT* kernel algorithm is implicitly done by Hadri et al. [45] when going from their "Semi-Parallel" to their "Fully-Parallel" algorithms.

2.2.2 Elimination lists

As stated above, any algorithm factorizing a tiled matrix of size $p \times q$ is characterized by its elimination list. Obviously, the algorithm must zero out all tiles below the diagonal: for each tile (i,k), $i > k$, $1 \leq k \leq \min(p,q)$, the list must contain exactly one entry $elim(i,\star,k)$, where $\star$ denotes some row index $piv(i,k)$. There are two conditions for a transformation $elim(i,piv(i,k),k)$ to be valid:

- both rows i and $piv(i,k)$ must be ready, meaning that all their tiles left of the panel (of indices (i,k') and $(piv(i,k),k')$ for $1 \leq k' < k$) must have already been zeroed out: all transformations $elim(i,piv(i,k'),k')$ and $elim(piv(i,k),piv(piv(i,k),k'),k')$ must precede $elim(i,piv(i,k),k)$ in the elimination list
- row $piv(i,k)$ must be a potential annihilator, meaning that tile $(piv(i,k),k)$ has not been zeroed out yet: the transformation $elim(piv(i,k),piv(piv(i,k),k),k)$ must follow $elim(i,piv(i,k),k)$ in the elimination list

Any algorithm that factorizes the tiled matrix obeying these conditions is called a *generic tiled algorithm* in the following.

A critical result is that no matter what elimination list is used the total weight of the tasks for performing a tiled QR factorization algorithm is constant and equal to $6pq^2 - 2q^3$. Using our unit task weight of $n_b^3/3$, with $m = pn_b$, and $n = qn_b$, we obtain $2mn^2 - 2/3n^3$ flops which is the exact same number as for a standard Householder reflection algorithm as found in LAPACK (e.g., [18]). We

note that this results is true if (a) we use TS kernels as well and if (b) we use any tiling, (e.g. rectangular tiles).

2.2.3 Execution schemes

In essence, the execution of a generic tiled algorithm is fully determined by its elimination list. This list is statically given as input to the scheduler, and the execution progresses dynamically, with the scheduler executing all required transformations as soon as possible. More precisely, each transformation involves several kernels, whose execution starts as soon as they are ready, i.e., as soon as all dependencies have been enforced. Recall that a tile (i, k) can be zeroed out only after all tiles (i, k'), with $k' < k$, have been zeroed out. Execution progresses as follows:

- Before being ready for elimination, tile (i, k), $i > k$, must be updated $k - 1$ times, in order to zero out the $k - 1$ tiles to its left (of index (i, k'), $k' < k$). The last update is a transformation $TTMQR(i, piv(i, k-1), k-1, k)$ for some row index $piv(i, k-1)$ such that $elim(i, piv(i, k-1), k-1)$ belongs to the elimination list. When completed, this transformation triggers the transformation $GEQRT(i, k)$, which can be executed immediately after the completion of the $TTMQR$. In turn, $GEQRT(i, k)$ triggers all updates $UNMQR(i, k, j)$ for all $j > k$. These updates are executed as soon as they are ready for execution.
- The elimination $elim(i, piv(i, k), k)$ is performed as soon as possible when both rows i and $piv(i, k)$ are ready. Just after the completion of $GEQRT(i, k)$ and $GEQRT(piv(i, k), k)$, kernel $TTQRT(i, piv(i, k), k)$ is launched. When finished, it triggers the updates $TTMQR(i, piv(i, k), k, j)$ for all $j > k$.

Obviously, the degree of parallelism that can be achieved depends upon the eliminations that are chosen. For instance, if all eliminations in a given column use the same factor tile, they will be sequentialized. This corresponds to the flat tree elimination scheme described below: in each column k, it uses $elim(i, k, k)$ for all $i > k$. On the contrary, two eliminations $elim(i, piv(i, k), k)$ and $elim(i', piv(i', k), k)$ in the same column can be fully parallelized provided that they involve four different rows. Finally, note that several eliminations can be initiated in different columns simultaneously, provided that they involve different pairs of rows, and that all these rows are ready (i.e., they have the desired number of leftmost zeros).

The following lemma will prove very useful; it states that we can assume w.l.o.g. that each tile is zeroed out by a tile above it, closer to the diagonal.

Lemma 2.1. *Any generic tiled algorithm can be modified, without changing its execution time, so that all eliminations* $elim(i, piv(i, k), k)$ *satisfy to* $i > piv(i, k)$.

Proof. Define a *reverse* elimination as an elimination $elim(i, piv(i, k), k)$ where $i < piv(i, k)$. Consider a generic tiled algorithm whose elimination list contains some reverse eliminations. Let k_0 be the first column to contain one of them. Let i_0 be the largest row index involved in a reverse elimination in column k_0. The elimination list in column k_0 may contain several reverse eliminations $elim(i_1, i_0, k_0)$, $elim(i_2, i_0, k_0)$, ..., $elim(i_r, i_0, k_0)$, in that order, before row i_0 is eventually zeroed out by the transformation $elim(i_0, piv(i_0, k_0), k_0)$. Note that $piv(i_0, k_0) < i_0$ by definition of i_0. We modify the algorithm by exchanging the roles of rows i_0 and i_1 in column k_0: the elimination list now includes $elim(i_0, i_1, k_0)$, $elim(i_2, i_1, k_0)$, ..., $elim(i_r, i_1, k_0)$, and $elim(i_1, piv(i_0, k_0), k_0)$. All dependencies are preserved, and the execution time is unchanged. Now the largest row index involved in a reverse elimination in column k_0 is strictly smaller than i_0, and we repeat the procedure until there does not remain any reverse elimination in column k_0. We proceed inductively to the following columns, until all reverse eliminations have been suppressed. ■

2.3 Critical paths

In this section we describe several generic tiled algorithms, and we provide their critical paths, as well as optimality results. These algorithms are inspired by algorithms that have been introduced twenty to thirty years ago [83, 73, 28, 27], albeit for a much simpler, *coarse-grain* model. In this "old" model, the time-unit is the time needed to execute an orthogonal transformation across two matrix rows, regardless of the position of the zero to be created, hence regardless of the length of these rows. Although the granularity is much coarser in this model, any existing algorithm for the old model can be transformed into a generic tiled algorithm, just by enforcing the very same elimination list provided by the algorithm. Critical paths are obtained using a discrete event based simulator specially developed to this end, based on the Simgrid framework [90, 67]. It carefully handles dependencies across tiles, and allows for the analysis of both static and dynamic algorithms. [1]

2.3.1 Coarse-grain algorithms

We start with a short description of three algorithms for the coarse-grain model. These algorithms are illustrated in Table 2.2 for a 15×6 matrix.

SAMEH-KUCK algorithm The SAMEH-KUCK algorithm [83] uses the panel row for all eliminations in each column, starting from below the diagonal and proceeding downwards. Time-steps indicate the time-unit at which the elimination can be done, assuming unbounded resources. Formally, the elimination list is

$$\left\{ \Big(elim(i,k,k), i = k+1, k+2, \ldots, p \Big), k = 1, 2, \ldots, \min(p,q) \right\}$$

FIBONACCI algorithm The FIBONACCI algorithm is the Fibonacci scheme of order 1 in [73]. Let $coarse(i,k)$ be the time-step at which tile (i,k), $i > k$, is zeroed out. These values are computed as follows. In the first column, there are one 5, two 4's, three 3's, four 2's and four 1's (we would have had five 1's with $p = 16$). Given x as the least integer such that $x(x+1)/2 \geq p-1$, we have $coarse(i,1) = x - y + 1$ where y is the least integer such that $i \leq y(y+1)/2 + 1$. Let the row indices of the z tiles that are zeroed out at step s, $1 \leq s \leq x$, range from i to $i+z-1$. The elimination list for these tiles is $elim(i+j, piv(i+j,1), 1)$, with $piv(i+j) = i+j-z$ for $0 \leq j \leq z-1$. In other words, to eliminate a bunch of z consecutive tiles at the same time-step, the algorithm uses the z rows above them, pairing them in the natural order. Now the elimination scheme of the next column is the same as that of the previous column, shifted down by one row, and adding two time-units: $coarse(i,k) = coarse(i-1,k-1) + 2$, while the pairing obeys the same rule.

GREEDY algorithm At each step, the GREEDY algorithm [28, 27] eliminates as many tiles as possible in each column, starting with bottom rows. The pairing for the eliminations is done exactly as for FIBONACCI. There is no closed-form formula to compute $coarse(i,k)$, the time-step at which tile (i,k) is eliminated, but it is possible to provide recursive expressions (see [28, 27]).

Consider a rectangular $p \times q$ matrix, with $p > q$. With the coarse-grain model, the critical path of SAMEH-KUCK is $p+q-2$, and that of FIBONACCI is $x+2q-2$, where x is the least integer such that $x(x+1)/2 \geq p-1$. The critical path of GREEDY is unknown, but two important results are known: (i) the critical path of GREEDY is optimal; (ii) its value tends to $2q$ if p is negligible in front of q^2, i.e., if

1. The discrete event based simulator, together with the code for all tiled algorithms, is publicly available at http://graal.ens-lyon.fr/~mjacquel/tiledQR.html

(a) SAMEH-KUCK						(b) FIBONACCI						(c) GREEDY					
⋆						⋆						⋆					
1	⋆					5	⋆					4	⋆				
2	3	⋆				4	7	⋆				3	6	⋆			
3	4	5	⋆			4	6	9	⋆			3	5	8	⋆		
4	5	6	7	⋆		3	6	8	11	⋆		2	5	7	10	⋆	
5	6	7	8	9	⋆	3	5	8	10	13	⋆	2	4	7	9	12	⋆
6	7	8	9	10	11	3	5	7	10	12	15	2	4	6	9	11	14
7	8	9	10	11	12	2	5	7	9	12	14	2	4	6	8	10	13
8	9	10	11	12	13	2	4	7	9	11	14	1	3	5	8	10	12
9	10	11	12	13	14	2	4	6	9	11	13	1	3	5	7	9	11
10	11	12	13	14	15	2	4	6	8	11	13	1	3	5	7	9	11
11	12	13	14	15	16	1	4	6	8	10	13	1	3	4	6	8	10
12	13	14	15	16	17	1	3	6	8	10	12	1	2	4	6	8	10
13	14	15	16	17	18	1	3	5	8	10	12	1	2	4	5	7	9
14	15	16	17	18	19	1	3	5	7	10	12	1	2	3	5	6	8

Table 2.2: Time-steps for coarse-grain algorithms.

we have $p = q^2 f(q)$ where f is any function such that $\lim_{+\infty} f = 0$ (and $f(q) > 1/q$ so that $p > q$). In particular, let p and q be proportional, $p = \lambda q$, with a constant $\lambda > 1$: FIBONACCI is asymptotically optimal, because x is of the order of $\sqrt{q}$, hence its critical path is $2q + o(q)$. On the contrary, SAMEH-KUCK is not asymptotically optimal since its critical path is $(1 + \lambda)q - 2$. For square $q \times q$ matrices, critical paths are slightly different ($2q - 3$ for SAMEH-KUCK, $x + 2q - 4$ for FIBONACCI), but the important result is that all three algorithms are asymptotically optimal in that case.

2.3.2 Tiled algorithms

As stated above, each coarse-grain algorithm can be transformed into a tiled algorithm, simply by keeping the same elimination list, and triggering the execution of each kernel as soon as possible. However, because the weights of the factor and update kernels are not the same, it is much more difficult to compute the critical paths of the transformed (tiled) algorithms. Table 2.3 is the counterpart of Table 2.2, and depicts the time-steps at which tiles are actually zeroed out. Note that the tiled version of SAMEH-KUCK is indeed the FLATTREE algorithm in PLASMA [21, 22], and we have renamed it accordingly. As an example, Algorithm 7 shows the GREEDY algorithm for the tiled model.

A first (and quite unexpected) result is that GREEDY is no longer optimal, as shown in the first two columns of Table 2.4a for a 15×2 matrix. In each column and at each step, "*the* ASAP *algorithm*" starts the elimination of a tile as soon as there are at least two rows ready for the transformation. In the following, s denotes the number of tiles ready to be eliminated. When $s \geq 2$ eliminations can start simultaneously, ASAP pairs the $2s$ rows just as FIBONACCI and GREEDY, the first row (closest to the diagonal) with row $s + 1$, the second row with row $s + 2$, and so on. As a matter of a fact, when processing the second column, both ASAP and GREEDY begin with the elimination of lines 10 to 15 (at time step 20). However, once tiles $(13, 2)$, $(14, 2)$ and $(15, 2)$ are zeroed out (i.e. at time step 22), ASAP eliminates 4 zeros, in rows 9 through 12. On the contrary, GREEDY waits until time step 26 to eliminate 6 zeros in rows 6 through 12. In a sense, ASAP is the counterpart of GREEDY at the tile level. However, ASAP is not optimal either, as shown in Table 2.4a for a 15×3 matrix. On larger examples, the critical path of GREEDY is better than that of ASAP, as shown in Table 2.4b.

We have seen that, for a 15×2 matrix, ASAP is better than GREEDY and that, for a 15×3 matrix, GREEDY is better than ASAP. We can further improve upon GREEDY in the 15×3 case. We consider the GRASAP(k) algorithm defined as: following the GREEDY algorithm up from columns 1 to $q - k$ and then switching in ASAP mode for the last k columns. GRASAP(0) is GREEDY, while GRASAP(q) is

Algorithm 7: GREEDY algorithm via *TT* kernels.

```
for j = 1 to q do
    /* nz(j) is the number of tiles which have been eliminated in column j */
    nZ(j) = 0
    /* nT(j) is the number of tiles which have been triangularized in column j */
    nT(j) = 0
while column q is not finished do
    for j = q down to 1 do
        if j == 1 then
            /* Triangularize the first column if not yet done */
            nT_new = nT(j) + (p − nT(j))
            if p − nT(j) > 0 then
                for k = p down to 1 do
                    GEQRT(k, j)
                    for jj = j + 1 to q do
                        UNMQR(k, j, jj)
        else
            /* Triangularize every tile having a zero in the previous column */
            nT_new = nZ(j − 1)
            for k = nT(j) to nT_new − 1 do
                GEQRT(p − k, j)
                for jj = j + 1 to q do
                    UNMQR(p − k, j, jj)
        /* Eliminate every tile triangularized in the previous step */
        nZ_new = nZ(j) + ⌊(nT(j) − nZ(j))/2⌋
        for kk = nZ(j) to nZ_new − 1 do
            piv(p − kk) = p − kk − nZ_new + nZ(j)
            TTQRT(p − kk, piv(p − kk), j)
            for jj = j + 1 to q do
                TTMQR(p − kk, piv(p − kk), j, jj)
        /* Update the number of triangularized and eliminated tiles at the next step */
        nT(j) = nT_new
        nZ(j) = nZ_new
```

ASAP. In Table 2.4a(c), we give the results for GRASAP(1). In this case (a 15×3 matrix), GRASAP(1) is better than GREEDY. GRASAP(1) finishes at time-step 62, while GREEDY finishes at time-step 64. Of course it would be interesting to determine the best value of k as a function of p and q, for the execution of GRASAP(k) on a $p \times q$ matrix.

We have a closed-form formula for the critical path of tiled FLATTREE, but not for that of tiled FIBONACCI (contrarily to the coarse-grain case). But we provide an asymptotic expression, both for FIBONACCI and for GREEDY. More importantly, we show that both tiled algorithms are asymptotically optimal. We state our main result:

Theorem 2.1. *For a tiled matrix of size $p \times q$, where $p \geq q$:*

1. *The critical path length of* FLATTREE *is*

$$\begin{cases} 2p + 2 & \text{if } p \geq q = 1 \\ 6p + 16q - 22 & \text{if } p > q > 1 \\ 22p - 24 & \text{if } p = q > 1 \end{cases}$$

2. *The critical path length of* FIBONACCI *is at most $22q + 6\lceil\sqrt{2p}\rceil$, and that of* GREEDY *is at most $22q + 6\lceil\log_2 p\rceil$*

(a) SAMEH-KUCK						(b) FIBONACCI						(c) GREEDY						(d) BINARYTREE						(e) PLASMATREE ($BS = 5$)					
⋆						⋆						⋆						⋆						⋆					
6	⋆					14	⋆					12	⋆					6	⋆					6	⋆				
8	28	⋆				12	48	⋆				10	42	⋆				8	28	⋆				8	28	⋆			
10	34	50	⋆			12	46	70	⋆			10	40	64	⋆			6	36	56	⋆			10	34	50	⋆		
12	40	56	72	⋆		10	42	68	92	⋆		8	36	62	86	⋆		10	34	70	90	⋆		12	40	56	72	⋆	
14	46	62	78	94	⋆	10	40	64	90	114	⋆	8	34	56	84	106	⋆	6	44	68	104	124	⋆	14	46	62	78	94	⋆
16	52	68	84	100	116	10	40	62	86	112	136	8	34	56	78	102	128	8	28	78	102	138	158	6	54	74	90	106	122
18	58	74	90	106	122	8	36	62	84	108	134	8	30	52	78	100	122	6	42	62	112	136	172	8	28	82	102	118	134
20	64	80	96	112	128	8	34	58	84	106	130	6	28	50	72	100	118	12	40	76	96	146	170	10	34	50	110	130	146
22	70	86	102	118	134	8	34	56	80	106	128	6	28	50	72	94	116	6	46	74	110	130	180	12	40	56	72	138	158
24	76	92	108	124	140	8	34	56	78	102	128	6	28	50	68	94	116	8	28	80	108	144	164	16	52	68	84	100	166
26	82	98	114	130	146	6	28	56	78	100	122	6	28	44	66	88	110	6	36	56	114	142	178	6	56	80	96	112	128
28	88	104	120	136	152	6	28	50	78	100	122	6	22	44	66	88	110	10	34	64	84	148	176	8	28	84	108	124	140
30	94	110	126	142	158	6	28	44	72	100	122	6	22	44	60	82	104	6	38	62	92	112	182	10	34	50	112	136	152
32	100	116	132	148	164	6	22	44	60	94	116	6	22	38	60	76	98	8	28	66	90	114	134	12	40	56	72	140	164

Table 2.3: Time-steps for tiled algorithms.

(a) GREEDY			(b) ASAP			(c) GRASAP(1)		
⋆			⋆			⋆		
12	⋆		12	⋆		12	⋆	
10	42	⋆	10	40	⋆	10	42	⋆
10	40	64	10	36	86	10	40	62
8	36	62	8	34	80	8	36	58
8	34	56	8	32	74	8	34	56
8	34	56	8	30	68	8	34	56
8	30	52	8	28	62	8	30	50
6	28	50	6	28	56	6	28	50
6	28	50	6	26	50	6	28	48
6	28	50	6	24	46	6	28	46
6	28	44	6	24	44	6	28	44
6	22	44	6	22	44	6	22	44
6	22	44	6	22	40	6	22	40
6	22	38	6	22	38	6	22	38

(a) GREEDY nor ASAP are optimal.

		q			
p	Algorithm	16	32	64	128
16	GREEDY	310			
	ASAP	310			
32	GREEDY	360	650		
	ASAP	402	656		
64	GREEDY	374	726	1342	
	ASAP	588	844	1354	
128	GREEDY	396	748	1452	2732
	ASAP	966	1222	1748	2756

(b) GREEDY generally outperforms ASAP.

Table 2.4: Neither GREEDY nor ASAP are optimal.

3. *The optimal critical path length is at least* $22q - 30$
4. FIBONACCI *is asymptotically optimal if* $p = q^2 f(q)$*, where* $\lim_{+\infty} f = 0$
5. GREEDY *is asymptotically optimal if* $\log_2 p = qf(q)$*, where* $\lim_{+\infty} f = 0$

Proof. **Proof of (1).** Consider first the case $p \geq q = 1$. We shall proceed by induction on p to show that the critical path of FLATTREE is of length $2p + 2$, If $p = 1$, then from Table 2.1 the result is obtained since only $GEQRT(1, 1)$ is required. With the base case established, now assume that this holds for all $p - 1 > q = 1$. Thus at time $t = 2(p - 1) + 2 = 2p$, we have that for all $p - 1 \geq i \geq 1$ tile $(i, 1)$ has been factorized into a triangle and for all $p - 1 \geq i > 1$, tile $(i, 1)$ has been zeroed out. Therefore, tile $(p, 1)$ will be zeroed out with $TTQRT(p, 1)$ at time $t + 2 = 2(p - 1) + 2 + 2 = 2p + 2$.

Consider now the case $p > q > 1$. We show by induction on k that tile (i, k), for $i > k \geq 2$, is zeroed out in FLATTREE at time unit $6i + 16k - 22$. For $k = 2$, tile $(2, 2)$ is updated from step $k = 1$ at time $4+6+6 = 16$, and it is factored into a triangle at time 20. Tile $(3, 2)$ is updated from step $k = 1$ at time 22 factored into a triangle at time 26 and then zeroed out at time $26+2 = 28 = 6\times3+16\times2-22$. A new tile in column 2 is zeroed out every 6 time units, hence the initialization of the induction for $k = 2$. Assume now that the formula holds up to column k, and let $t = 6(k+1)+16k-22$ be the time at which tile $(k + 1, k)$ is zeroed out. Tile $(k + 1, k + 1)$ is updated from step k at time $t - 2 + 6 + 6 = t + 10$

and factored into a triangle at time $t+14$. By induction, tile $(k+2,k)$ is zeroed out at time $t+6$, hence triangularized at time $t+4$. The corresponding $UNMQR$ update of tile $(k+2,k+1)$ ends at time $t+10$, its $TTMQR$ update ends at time $\max(t+14,t+10)+6=t+20$. Hence tile $(k+2,k+1)$ can indeed be zeroed out at time $\max(t+12,t+20)+2=t+22$. A new tile in column $k+1$ can be zeroed out every 6 time units, hence the induction formula for $k+1$.

Finally, for a square matrix of size $q \times q$, consider the above formula for a rectangular matrix with $p=q+1$. Instead of zeroing out the last tile $(q+1,q)$ with tile (q,q), simply need to factor tile (q,q) into a triangle with $GEQRT(q,q)$. This costs 4 time units instead of 6 when adding $TTQRT(q+1,q,q)$, and explains the difference of 2 in the formula for square matrices.

Proof of (2). FIBONACCI and GREEDY are more difficult to analyze than FLATTREE, but we provide an upper bound of their critical path. The approach is the same for both algorithms, and hereafter ALG denotes either FIBONACCI or GREEDY. Let $coarse(i,k)$ be the time-step at which tile (i,k) is zeroed out in ALG with the coarse-grain model (see Table 2.2 for examples). We derive a "slowed down" version of the tiled version of ALG by terminating the zeroing out of tile (i,k) at time-step

$$6\,coarse(2,1)+22(k-1)-6(coarse(k+1,k)-coarse(i,k)).$$

We say that this version is slowed down because we do not start the zeroing out of the tiles as soon as possible. For instance in the first column, tile $(i,1)$ is zeroed out at time $6\,coarse(i,1)$, which is larger than the value given in Table 2.3. However, we keep the same elimination list as in the original version of ALG, and we trigger the update and factor operations as soon as possible when the zeroing out operation is completed. We only delay these latter operations.

The intuitive idea for delaying the eliminations is that the corresponding updates will be fully overlapped, within a given column, or when proceeding from one column to the next: in this case, allowing for a time-shift of 22 smooths the chaining of the updates. The regular and repetitive spacing of the eliminations allows us to check (just as we did to prove (1)) that all dependencies are enforced in the slowed down version of ALG. Because the case-analysis is tedious, we have written a program for a sanity check of the validity of ALG [2].

In the coarse-grain model, ALG terminates the first column in time x, so the critical path of its slowed down version is $6x+22(q-1)$. For FIBONACCI, x is the least integer such that $x(x+1)/2 \geq p-1$, hence $x \leq \lceil\sqrt{2p}\rceil$. For GREEDY, $x=\lceil\log_2(p-1)\rceil \leq \lceil\log_2 p\rceil$, hence the result.

Proof of (3). Consider a square $q \times q$ matrix B, with $q \geq 2$. Assume that there are only three non-zero sub-diagonals, i.e., that tile (i,k) is initially zero in B for $i > k+3$. Because there are only three non-zero tiles below the diagonal, there is a constant number of possible row pairings in each column. An exhaustive search is to try all possible pairings in the first column, followed by all possible pairings in the second column, and so on. After a few columns, a pattern emerges, and we can identify that any optimal algorithm (there are several of them) needs at least 22 time-steps to proceed from one column to the next. It is possible to save a few steps at the beginning and end of the execution, and the optimal critical path is $22q-30$. Here also, because the case-analysis is long and tedious, we have written a program for a sanity check of the latter value.

Now we show that the optimal critical path for a general $p \times q$ matrix A, with $p \geq q \geq 2$, is at least equal to the critical path of the previous $q \times q$ matrix B with three sub-diagonals. Indeed, Lemma 2.1 shows that there exist optimal algorithms for factoring A without any reverse elimination. Consider such an algorithm, and discard all eliminations that involve zeroing out elements below the third sub-diagonal, or outside the $q \times q$ top square: the critical path cannot increase, and we have an elimination scheme for B, which proves the desired result.

2. All program sources are publicly available at http://graal.ens-lyon.fr/~mjacquel/tiledQR.html

Note that using B instead of A is the key to the proof: in each column of B, there is only a constant number of possible row pairings, which makes it possible to try all combinations for several consecutive columns. Reasoning with A instead would need a completely different proof (yet to be invented).

Proof of (4) and (5). These are a direct consequence of (3) and (4). ■

Remarks:

1. We express all critical path lengths in terms of p and q, with an unit of $n_b^3/3$ floating-point operations. It is easy to get critical path lengths in term of m, n, and n_b, and with elementary floating-point operations as unit, assuming that all tiles are full. (In other words, m and n are multiple of n_b.) For example for FLATTREE, we get $(2/3)mn_b^2 + (2/3)n_b^3$ if $m \geq n = n_b$, $2mn_b^2 + 16/3nn_b^2 - (22/3)n_b^3$ if $m > n > n_b$ and $(22/3)nn_b^2 - (24/3)n_b^3$ if $m = n > n_b$.
2. From this formula, it is clearer that, if one wants to minimize the number of floating-point operations on the critical path, one needs to take $n_b = 1$. However, such an action would have disastrous consequences. The communication increase would be way too high, and the increase gain in parallelism would not be worth the overhead. More importantly, the efficiency of the elimination kernels would be much lower. In this chapter, we consider n_b constant, large enough so that elimination kernels operate at full Level 3 BLAS performance, and so that communication costs remain relatively low.
3. In the square case, we see that the critical path length of the tiled algorithms is typically in $\mathcal{O}(nn_b^2)$. This is in sharp contrast with the current LAPACK algorithm $GEQRF$. If we assume that the panel is not parallelizable, and that the block size for the LAPACK algorithm is n_b, then counting the length of the chain of panel factorization steps leads to a critical path length in $\mathcal{O}(n^2 n_b)$. There is therefore much more parallelism to exploit in the tiled algorithms than in the current LAPACK algorithms. Or, stated differently [21, 22], the granularity of the tiled algorithms is finer than that of the LAPACK algorithm.

In Table 2.3 we also report time-steps for the BINARYTREE algorithm. As its name indicates, this algorithm performs a binary tree reduction to zero out tiles in each column. Here is an asymptotic expression of its critical path:

Proposition 2.1. *Consider a tiled matrix of size $p \times q$, where $p \geq q$. The critical path length of* BINARYTREE *is* $6q \log_2 p + o(q \log_2 p)$.

Proof. It is possible to derive an exact expression for the critical path length of BINARYTREE in the special case where p and q are both exact powers of two, with $q < p$. We obtain the value $(10 + 6 \log_2 p)q - 4 \log_2 p - 6$. As before, the proof goes by (tedious) induction. Here again, we have written a program for a sanity check of the latter value. The asymptotic value follows easily for an arbitrary matrix, by enlarging each dimension to the nearest power of two. ■

Proposition 2.1 shows that BINARYTREE is not asymptotically optimal. The PLASMA library provides more algorithms, that can be informally described as trade-offs between FLATTREE and BINARYTREE. (We remind the reader that FLATTREE is the same as algorithm as SAMEH-KUCK.) These algorithms are referred to as PLASMATREE in all the following, and differ by the value of an input parameter called the *domain size* BS. This domain size can be any value between 1 and p, inclusive. Within a domain, that includes BS consecutive rows, the algorithm works just as FLATTREE: the first row of each domain acts as a local panel and is used to zero out the tiles in all the other rows of the domain. Then the domains are merged: the panel rows are zeroed out by a binary tree reduction, just as in BINARYTREE. As the algorithm progresses through the columns, the domain on the very bottom

is reduced accordingly, until such time that there is one less domain. For the case that $BS = 1$, PLASMATREE follows a binary tree on the entire column, and for $BS = p$, the algorithm executes a flat tree on the entire column. It seems very difficult for a user to select the domain size BS leading to best performance, but it is known that BS should increase as q increases. Table 2.3 shows the time-steps of PLASMATREE with a domain size of $BS = 5$. In the experiments of Section 2.4, we use all possible values of BS and retain the one leading to the best value.

So far our study has only been concerned with algorithms based on TT kernels. Indeed, in this chapter, FLATTREE stands for TT-FT. We now give the critical path of the algorithm TS-FT. This corresponds to the FLATTREE algorithm (i.e., SAMEH-KUCK) with TS kernels. This algorithm was introduced in [21, 22, 81] and is available in PLASMA for performing the QR factorization of a matrix on multicore architecture.

Proposition 2.2. *The critical path length for* TS-FT *is*

$$\begin{aligned} 6p - 2 \quad & \textit{for } p \geq q = 1 \\ 12p + 18q - 32 \quad & \textit{for } p > q > 1 \\ 30p - 34 \quad & \textit{for } p = q > 1 \end{aligned}$$

Proof. Consider the case of $p \geq q = 1$. In order to show that for any p, with $q = 1$, the critical path is of length $6p - 2$, we shall proceed by induction on p. If $p = q = 1$, then from Table 2.1 the result is obtained since only $GEQRT(1, 1)$ is required. With the base case established, now assume that this holds for all $p - 1 > q = 1$. Thus at time $t = 6(p - 1) - 2 = 6p - 8$, we have that tile $(1, 1)$ has been factorized into a triangle and for all $p - 1 \geq i > 1$, tile $(i, 1)$ has been zeroed out. Therefore, tile $(p, 1)$ will be zeroed out with $TSQRT(p, 1)$ at time $t + 6 = 6(p - 1) - 2 + 6 = 6p - 2$.

Assume that $p > q > 1$. We show by induction on k that tile (i, k), for $i > k \geq 2$, is zeroed out at time unit $12i + 18k - 32$. Tile $(2, 2)$ is updated from step $k = 1$ at time $6(2) - 2 + 12 = 22$, it is factored into a triangle at time 28. Tile $(3, 2)$ is zeroed out at time $28 + 12 = 40 = 12 \times 3 + 18 \times 2 - 32$, and a new tile in column 2 is zeroed out every 12 time units, hence the initialization of the induction for $k = 2$.

Assume now that the formula holds up to column k, and let $t = 12(k + 1) + 18k - 32$ be the time at which tile $(k + 1, k)$ is zeroed out. Tile $(k + 1, k + 1)$ is updated from step k at time $t + 12$ and factored into a triangle at time $t + 18$. By induction, tile $(k + 2, k)$ is zeroed out at time $t + 12$. Hence tile $(k + 2, k + 1)$ can indeed be zeroed out at time $\max(t + 12, t + 18) + 12 = t + 30$. A new tile in column $k + 1$ can be zeroed out every 12 time units, hence the induction formula for $k + 1$.

For a square matrix of size $q \times q$, consider the above formula for a rectangular matrix with $p = q + 1$. Instead of zeroing out the last tile $(q + 1, q)$ with tile (q, q) in 6 time units with $TSQRT(q + 1, q)$, we simply need to factor tile (q, q) into a triangle with $GEQRT(q, q)$. This costs 4 time units instead of 6, and explains the difference of 2 in the formula for square matrices. ■

As we can see, the critical path of TS-FT (Proposition 2.2) is longer than the one of FLATTREE (Theorem 2.1(1)). This stems from the facts that (1) a TS algorithm can be converted into a TT algorithm, and (2) this conversion increases the parallelism, and, consequently, reduces the critical path length.

2.4 Experimental results

All experiments were performed on a 48-core machine composed of eight hexa-core AMD Opteron 8439 SE (codename Istanbul) processors running at 2.8 GHz. Each core has a theoretical peak of 11.2

Gflop/s with a peak of 537.6 Gflop/s for the whole machine. The Istanbul micro-architecture is a NUMA architecture where each socket has 6 MB of level-3 cache and each processor has a 512 KB level-2 cache and a 128 KB level-1 cache. After having benchmarked the AMD ACML and Intel MKL BLAS libraries, we selected MKL (10.2) since it appeared to be slightly faster in our experimental context. Linux 2.6.32 and Intel Compilers 11.1 were also used in conjunction with PLASMA 2.3.1.

For all results, we show both double and double complex precision, using all 48 cores of the machine. The matrices are of size $m = 8000$ and $200 \leq n \leq 8000$. The tile size is kept constant at $n_b = 200$, so that the matrices can also be viewed as $p \times q$ tiled matrices where $p = 40$ and $1 \leq q \leq 40$. All kernels use an inner blocking parameter of $i_b = 32$.

Asymptotically all operations in a QR factorization are FMAs ("*fused multiply-add*", $y \leftarrow \alpha x + y$). In real arithmetic, an FMA involves three double precision numbers for two flops, but these two flops can be combined into one FMA instruction and thus completed in one cycle. In complex arithmetic, the operation $y \leftarrow \alpha x + y$ involves six double precision numbers for eight flops; we also note that there is no such thing as a complex-arithmetic FMA. The ratio of computation/communication is therefore, potentially, four times higher in complex arithmetic than in real arithmetic. Communication aware algorithms are much more critical in real arithmetic than in complex arithmetic. This is the reason why we present results in complex arithmetic and in real arithmetic. Our new algorithms will be at their best in the complex arithmetic case where parallelism is most important while communication less. In the real arithmetic case, we will see that TS kernels which perform potentially less communication than TT kernels have the advantage as soon as there is enough parallelism from the algorithm (q large enough).

The PLASMA interface allows one to specify the dependencies between tasks by designating the data as either INPUT, OUTPUT, INOUT, or NODEP. Currently, the update kernels ($UNMQR$, $TTMQR$, and $TSMQR$) introduced false dependencies between the tasks which sequentializes the execution of update with the factorization kernels $TTQRT$ or $TSQRT$. In order to alleviate these, we altered the dependency designation within each of the update kernels for the matrix of Householder reflectors, V, from INPUT to NODEP as is further explained in [66]. The dependencies between the tasks are still consistent since the T matrix within each update kernel continues to be designated as INPUT so that any subsequent task which overwrites this T matrix cannot be executed.

For each experiment, we provide a comparison of the theoretical performance to the actual performance. The theoretical performance is obtained by modeling the limiting factor of the execution time as either the critical path, or the sequential time divided by the number of processors. This is similar in approach to the Roofline model [97]. Taking γ_{seq} as the sequential performance, T as the total number of flops, cp as the length of the critical path, and P as the number of processors, the predicted performance, γ_{pred}, is

$$\gamma_{pred} = \frac{\gamma_{seq} \cdot T}{\max\left(\frac{T}{P}, cp\right)}$$

Figures 2.1a and 2.1c depict the predicted performance of all algorithms which use the *Triangle on top of triangle* kernels. For double complex precision, sequential kernels reach 3.1860 GFlop/s while in double precision, the peak performance is 3.8440 GFlop/s. Since PLASMATREE provides an additional tuning parameter of the domain size, we show the results for each value of this parameter as well as the composition of the best of these domain sizes. Again, it is not evident what the domain size should be for the best performance, hence our exhaustive search.

Part of our comprehensive study also involved comparisons made to the Semi-Parallel Tile and Fully-Parallel Tile CAQR algorithms found in [45] which are much the same as those found in PLASMA. As with PLASMA, the tuning parameter BS controls the domain size upon which a flat tree is used to zero out tiles below the root tile within the domain and a binary tree is used to merge these domains. Unlike PLASMA, it is not the bottom domain whose size decreases as the algorithm progresses through the

columns, but instead is the top domain. In this study, we found that the PLASMA algorithms performed identically or better than these algorithms and therefore we do not report these comparisons.

p	q	GREEDY	PLASMATREE (TT)	BS	Overhead	Gain
40	1	36.9360	37.5020	1	1.0153	-0.0153
40	2	58.5090	52.7180	3	0.9010	0.0990
40	4	103.2670	90.7940	10	0.8792	0.1208
40	5	115.3060	100.5540	5	0.8721	0.1279
40	10	153.5180	145.8200	17	0.9499	0.0501
40	20	170.8730	171.8270	27	1.0056	-0.0056
40	40	184.5220	182.8160	19	0.9908	0.0092

Table 2.5: Experimental performance of GREEDY versus PLASMATREE (TT) (double)

p	q	GREEDY	PLASMATREE (TT)	BS	Overhead	Gain
40	1	42.0710	42.7120	1	1.0152	-0.0152
40	2	60.4420	52.1970	5	0.8636	0.1364
40	4	95.1820	84.1120	5	0.8837	0.1163
40	5	107.6370	96.7530	5	0.8989	0.1011
40	10	135.0270	128.4320	17	0.9512	0.0488
40	20	144.4010	146.4220	28	1.0140	-0.0140
40	40	152.9280	151.9090	8	0.9933	0.0067

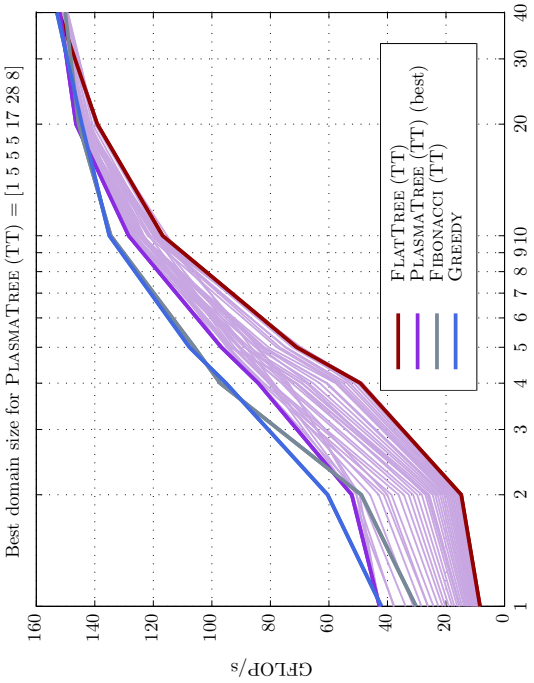
Table 2.6: Experimental performance of GREEDY versus PLASMATREE (TT) (double complex)

p	q	GREEDY	FIBONACCI	Overhead	Gain
40	1	36.9360	26.5610	0.7191	0.2809
40	2	58.5090	49.4870	0.8458	0.1542
40	4	103.2670	100.1440	0.9698	0.0302
40	5	115.3060	115.0020	0.9974	0.0026
40	10	153.5180	152.0090	0.9902	0.0098
40	20	170.8730	170.4780	0.9977	0.0023
40	40	184.5220	180.2990	0.9771	0.0229

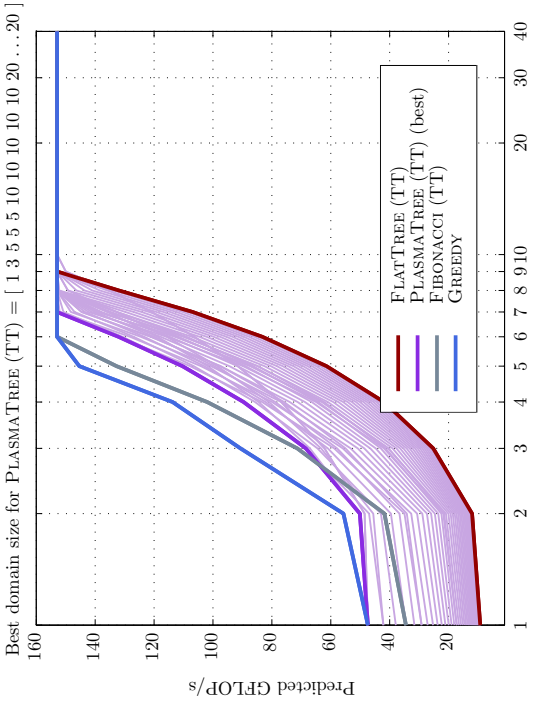
Table 2.7: Experimental performance of Greedy versus FIBONACCI (double)

Figure 2.1b and 2.1d illustrate the experimental performance reached by GREEDY, FIBONACCI and PLASMATREE algorithms using the *TT (Triangle on top of triangle)* kernels. In both cases, double or double complex precision, the performance of GREEDY is better than PLASMATREE even for the best choice of domain size. Moreover, as expected from the analysis in Section 2.3.2, GREEDY outperforms FIBONACCI the majority of the time. Furthermore, we see that, for rectangular matrices, the experimental performance in double complex precision matches the prediction. This is not the case for double precision because communications have higher impact on performance. Detailed results are given in Tables 2.5, 2.6, 2.7 and 2.8.

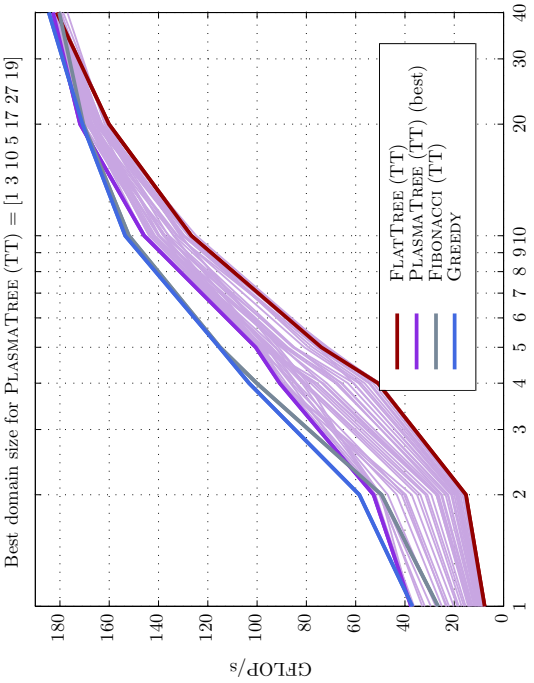
While it is apparent that GREEDY does achieve higher levels of performance, the percentage may not be as obvious. To that end, taking GREEDY as the baseline, we present in Figure 2.2 the theoretical, double, and double complex precision overhead for each algorithm that uses the *Triangle on top of*



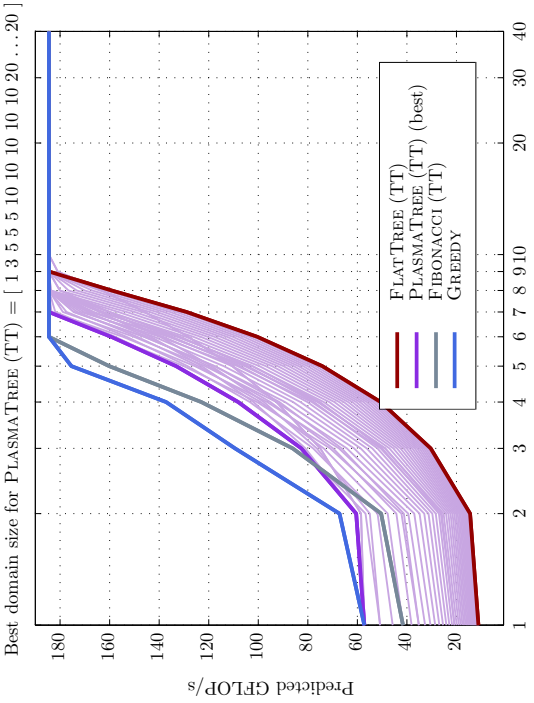
(b) Experimental (double complex)



(a) Predicted (double complex)

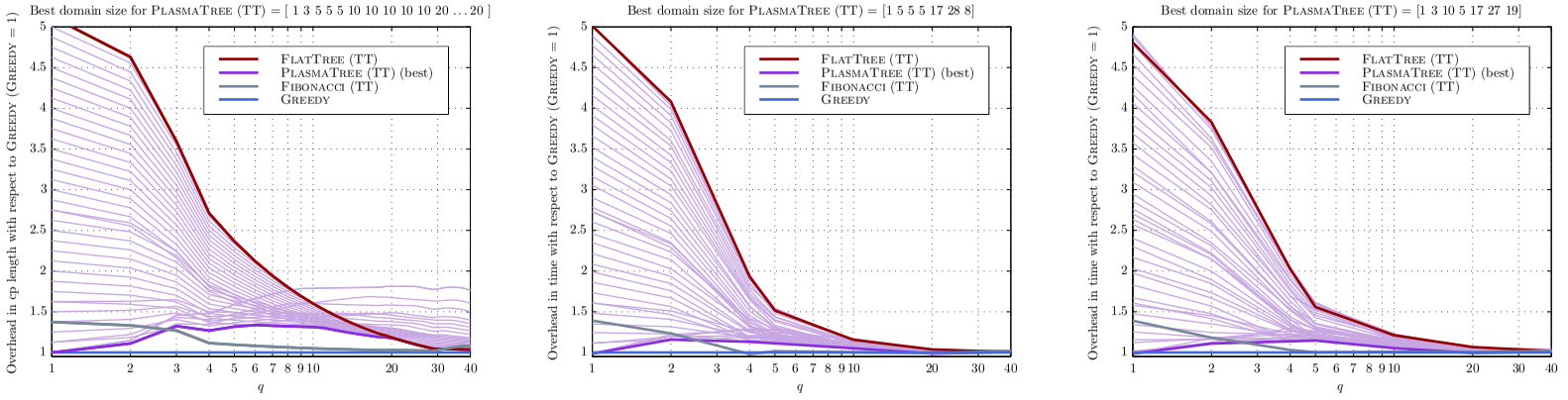


(d) Experimental (double)



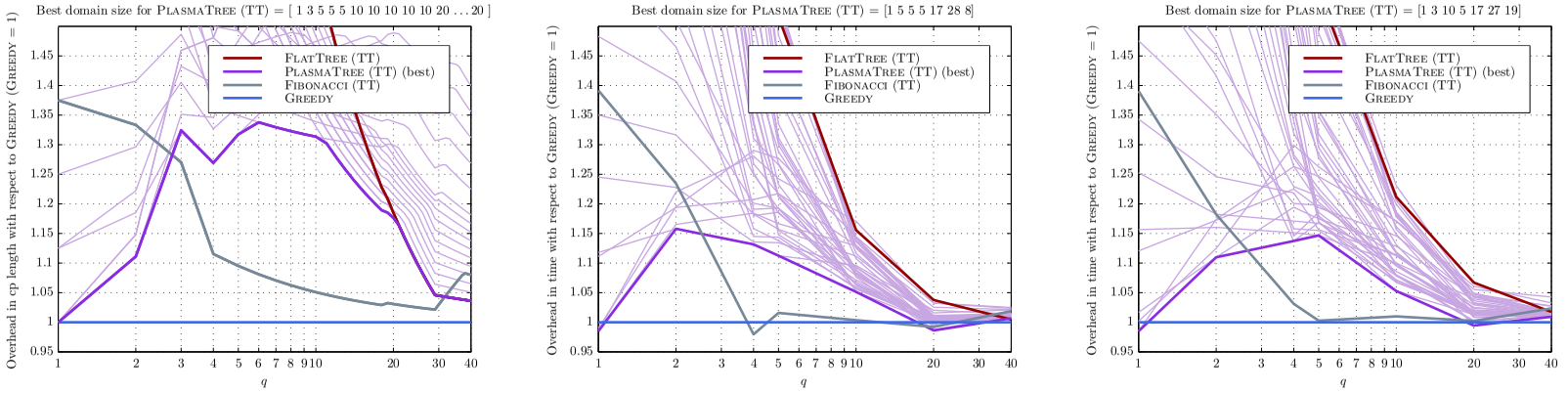
(c) Predicted (double)

Figure 2.1: Predicted and experimental performance of QR factorization - *Triangle on top of triangle* kernels



(a) Theoretical CP length (b) Experimental (double complex) (c) Experimental (double)

Figure 2.2: Overhead in terms of critical path length and time with respect to GREEDY (GREEDY = 1)



(a) Theoretical CP length (b) Experimental (double complex) (c) Experimental (double)

Figure 2.3: Detailed view of the overhead in terms of critical path length and time with respect to GREEDY (GREEDY = 1)

p	q	GREEDY	FIBONACCI	Overhead	Gain
40	1	42.0710	30.2280	0.7185	0.2815
40	2	60.4420	48.9570	0.8100	0.1900
40	4	95.1820	97.1650	1.0208	-0.0208
40	5	107.6370	105.9610	0.9844	0.0156
40	10	135.0270	134.5500	0.9965	0.0035
40	20	144.4010	145.5530	1.0080	-0.0080
40	40	152.9280	150.0980	0.9815	0.0185

Table 2.8: Experimental performance of GREEDY versus FIBONACCI (double complex)

triangle kernel as compared to GREEDY. These overheads are respectively computed in terms of critical path length and time. At a smaller scale (Figure 2.3), it can be seen that GREEDY can perform up to 13.6% better than PLASMATREE.

For all matrix sizes considered, $p = 40$ and $1 \leq q \leq 40$, in the theoretical model, the critical path length for GREEDY is either the same as that of PLASMATREE ($q = 1$) or is up to 25% shorter than PLASMATREE ($q = 6$). Analogously, the critical path length for GREEDY is at least 2% to 27% shorter than that of FIBONACCI. In the experiments, the matrix sizes considered were $p = 40$ and $q \in \{1, 2, 4, 5, 10, 20, 40\}$. In double precision, GREEDY has a decrease of at most 1.5% than the best PLASMATREE ($q = 1$) and a gain of at most 12.8% than the best PLASMATREE ($q = 5$). In double complex precision, GREEDY has a decrease of at most 1.5% than the best PLASMATREE ($q = 1$) and a gain of at most 13.6% than the best PLASMATREE ($q = 2$). Similarly, in double precision, GREEDY provides a gain of 2.6% to 28.1% over FIBONACCI and in double complex precision, GREEDY has a decrease of at most 2.1% and a gain of at most 28.2% over FIBONACCI. Detailed results are given in Table 2.9.

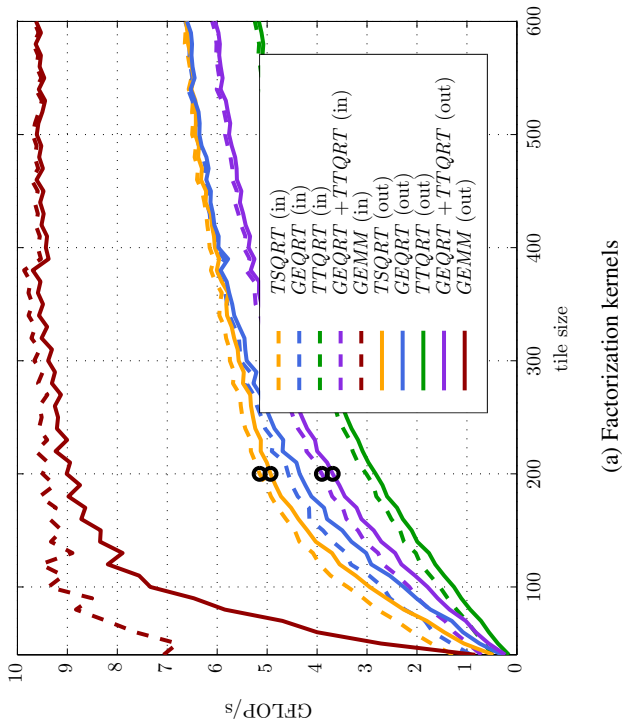
Although it is evidenced that PLASMATREE does not vary too far from GREEDY or FIBONACCI, one must keep in mind that there is a tuning parameter involved and we choose the best of these domain sizes for PLASMATREE to create the composite result, whereas with GREEDY, there is no such parameter to consider. Of particular interest is the fact that GREEDY always performs better than any other algorithm [3] for $p \gg q$. In the scope of PLASMATREE, a domain size $BS = 1$ will force the use of a binary tree so that both GREEDY and PLASMATREE behave the same. However, as the matrix tends more to a square, i.e., q tends toward p, we observe that the performance of all of the algorithms, including FLATTREE, are on par with GREEDY. As more columns are added, the parallelism of the algorithm is increased and the critical path becomes less of a limiting factor, so that the performance of the kernels is brought to the forefront. Therefore, all of the algorithms are performing similarly since they all share the same kernels.

In order to accurately assess the impact of the kernel selection towards the performance of the algorithms, Figures 2.4 and 2.5 show both the in cache and out of cache performance using the *No Flush* and *MultCallFlushLRU* strategies as presented in [2, 96]. Since an algorithm using *TT* kernels will need to call $GEQRT$ as well as $TTQRT$ to achieve the same as the *TS* kernel $TSQRT$, the comparison is made between $GEQRT$ + $TTQRT$ and $TSQRT$ (and similarly for the updates). For $n_b = 200$, the observed ratio for in cache kernel speed for $TSQRT$ to $GEQRT$ + $TTQRT$ is 1.3374, and for $TSMQR$ to $UNMQR$ + $TTMQR$ is 1.3207. For out of cache, the ratio for $TSQRT$ to $GEQRT$ + $TTQRT$ is 1.3193 and for $TSMQR$ to $UNMQR$ + $TTMQR$ it is 1.3032. Thus, we can expect about a 30% difference between the selection of the kernels, since we will have instances of using in cache and out of

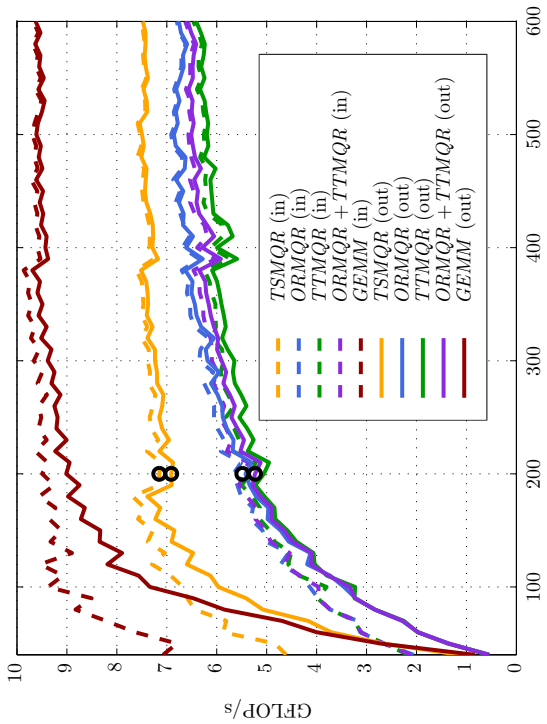
3. When $q = 1$, GREEDY and FLATTREE exhibit close performance. They both perform a binary tree reduction, albeit with different row pairings.

p	q	GREEDY	PLASMATREE (TT)	BS	Overhead	Gain	FIBONACCI	Overhead	Gain
40	1	16	16	1	1.0000	0.0000	22	1.3750	0.2727
40	2	54	60	3	1.1111	0.1000	72	1.3333	0.2500
40	3	74	98	5	1.3243	0.2449	94	1.2703	0.2128
40	4	104	132	5	1.2692	0.2121	116	1.1154	0.1034
40	5	126	166	5	1.3175	0.2410	138	1.0952	0.0870
40	6	148	198	10	1.3378	0.2525	160	1.0811	0.0750
40	7	170	226	10	1.3294	0.2478	182	1.0706	0.0659
40	8	192	254	10	1.3229	0.2441	204	1.0625	0.0588
40	9	214	282	10	1.3178	0.2411	226	1.0561	0.0531
40	10	236	310	10	1.3136	0.2387	248	1.0508	0.0484
40	11	258	336	20	1.3023	0.2321	270	1.0465	0.0444
40	12	280	358	20	1.2786	0.2179	292	1.0429	0.0411
40	13	302	380	20	1.2583	0.2053	314	1.0397	0.0382
40	14	324	402	20	1.2407	0.1940	336	1.0370	0.0357
40	15	346	424	20	1.2254	0.1840	358	1.0347	0.0335
40	16	368	446	20	1.2120	0.1749	380	1.0326	0.0316
40	17	390	468	20	1.2000	0.1667	402	1.0308	0.0299
40	18	412	490	20	1.1893	0.1592	424	1.0291	0.0283
40	19	432	512	20	1.1852	0.1562	446	1.0324	0.0314
40	20	454	534	20	1.1762	0.1498	468	1.0308	0.0299
40	21	476	554	20	1.1639	0.1408	490	1.0294	0.0286
40	22	498	570	20	1.1446	0.1263	512	1.0281	0.0273
40	23	520	586	20	1.1269	0.1126	534	1.0269	0.0262
40	24	542	602	20	1.1107	0.0997	556	1.0258	0.0252
40	25	564	618	20	1.0957	0.0874	578	1.0248	0.0242
40	26	586	634	20	1.0819	0.0757	600	1.0239	0.0233
40	27	608	650	20	1.0691	0.0646	622	1.0230	0.0225
40	28	630	666	20	1.0571	0.0541	644	1.0222	0.0217
40	29	652	682	20	1.0460	0.0440	666	1.0215	0.0210
40	30	668	698	20	1.0449	0.0430	688	1.0299	0.0291
40	31	684	714	20	1.0439	0.0420	710	1.0380	0.0366
40	32	700	730	20	1.0429	0.0411	732	1.0457	0.0437
40	33	716	746	20	1.0419	0.0402	754	1.0531	0.0504
40	34	732	762	20	1.0410	0.0394	776	1.0601	0.0567
40	35	748	778	20	1.0401	0.0386	798	1.0668	0.0627
40	36	764	794	20	1.0393	0.0378	820	1.0733	0.0683
40	37	780	810	20	1.0385	0.0370	842	1.0795	0.0736
40	38	796	826	20	1.0377	0.0363	862	1.0829	0.0766
40	39	812	842	20	1.0369	0.0356	878	1.0813	0.0752
40	40	826	856	20	1.0363	0.0350	892	1.0799	0.0740

Table 2.9: Critical path lengths of GREEDY versus PLASMATREE (TT) and FIBONACCI

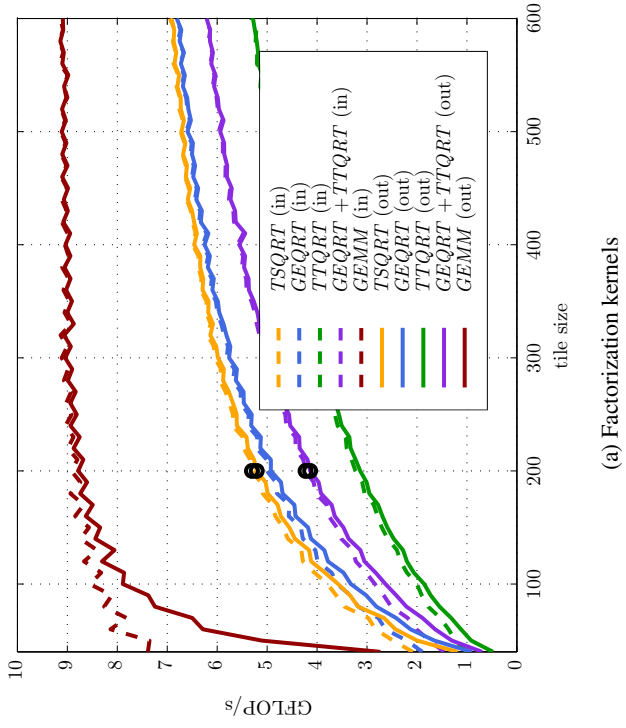


(a) Factorization kernels

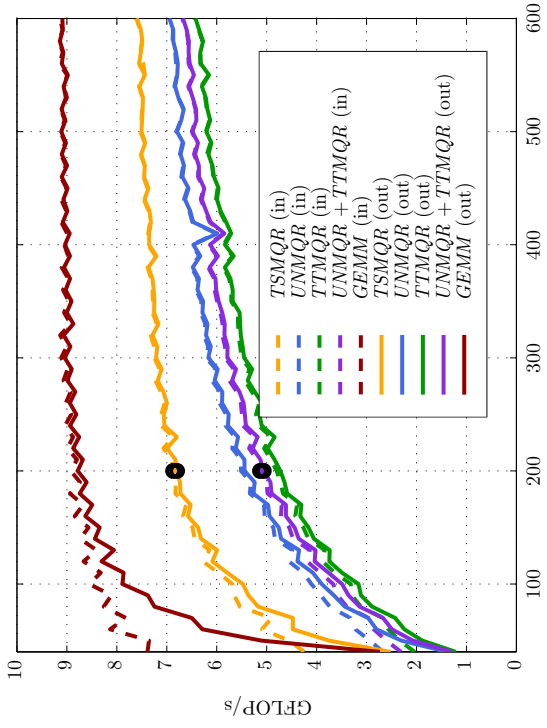


(b) Update kernels

Figure 2.5: Kernel performance for double precision



(a) Factorization kernels



(b) Update kernels

Figure 2.4: Kernel performance for double complex precision

cache throughout the run. Most of this difference is due to the higher efficiency and data locality within the *TT* kernels as compared to the *TS* kernels.

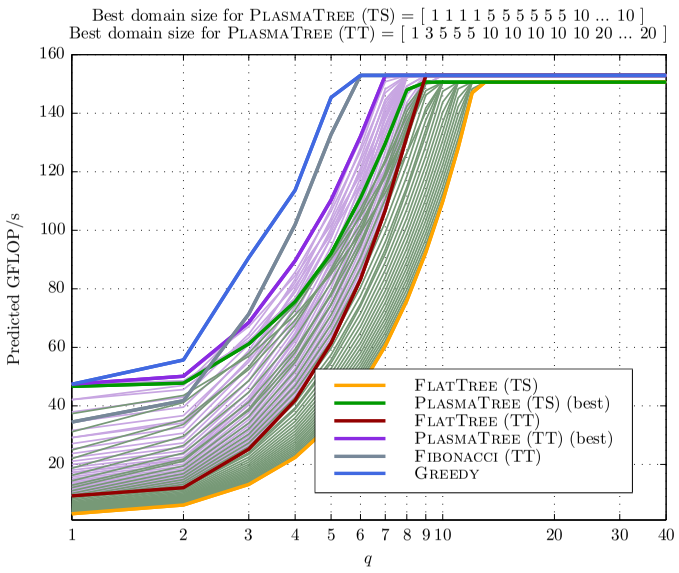
Having seen that kernel performance can have a significant impact, we also compare the *TT* based algorithms to those using the *TS* kernels. The goal is to provide a complete assessment of all currently available algorithms, as shown in Figure 2.6. For double precision, the observed difference in kernel speed is 4.976 GFLOP/sec for the *TS* kernels versus 3.844 GFLOP/sec for the *TT* kernels which provides a ratio of 1.2945 and is in accordance with our previous analysis. It can be seen that as the number of columns increases, whereby the amount of parallelism increases, the effect of the kernel performance outweighs the benefit provided by the extra parallelism afforded through the *TT* algorithms. Comparatively, in double complex precision, GREEDY does perform better, even against the algorithms using the *TS* kernels. As before, one must keep in mind that GREEDY does not require the tuning parameter of the domain size to achieve this better performance.

From these experiments, we showed that in double complex precision, GREEDY demonstrated better performance than any of the other algorithms and moreover, it does so without the need to specify a domain size as opposed to the algorithms in PLASMA. In addition, in double precision, for matrices where $p \gg q$, GREEDY continues to excel over any other algorithm using the *TT* kernels, and continues to do so as the matrices become more square.

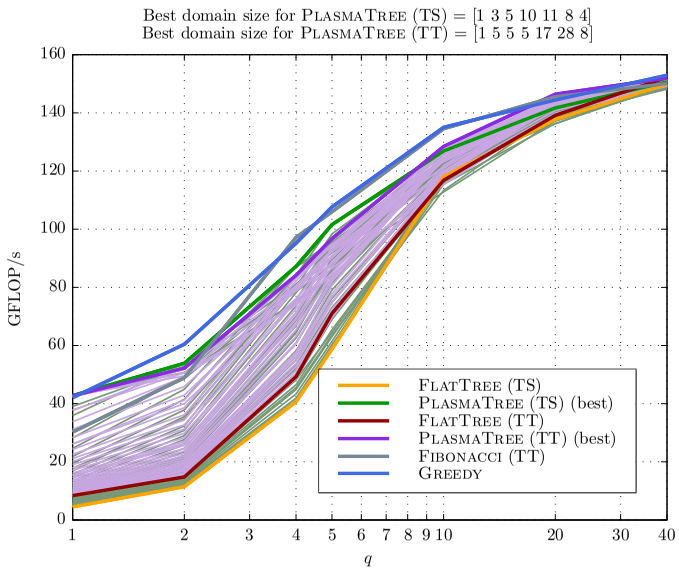
2.5 Conclusion

In this chapter, we have presented FIBONACCI, and GREEDY, two new algorithms for tiled QR factorization. These algorithms exhibit more parallelism than state-of-the-art implementations based on reduction trees. We have provided accurate estimations for the length of their critical path, and we have proven that they were asymptotically optimal for a wide class of matrix shapes, including all cases where the number of tile rows p and tile columns q are proportional, $p = \lambda q$, $\lambda \geq 1$. To the best of our knowledge, this proof is the first complexity result in the field of tiled algorithms, and it lays the theoretical foundations for a comparative study of tiled algorithms.

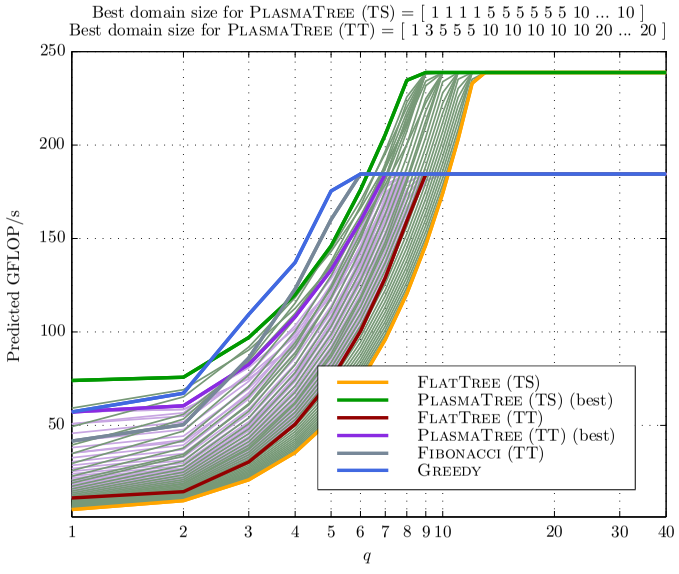
Comprehensive experiments on multicore platforms confirm the superiority of the new algorithms for $p \times q$ matrices, as soon as, say, $p \geq 2q$. This holds true when comparing not only with previous algorithms using TT (*Triangle on top of triangle*) kernels, but also with all known algorithms based on TS (*Triangle on top of square*) kernels. Given that TS kernels offer more locality, and benefit from better elementary arithmetic performance, than TT kernels, the better performance of the new algorithms is even more striking, and further demonstrates that a large degree of a parallelism was not exploited in previously published solutions.



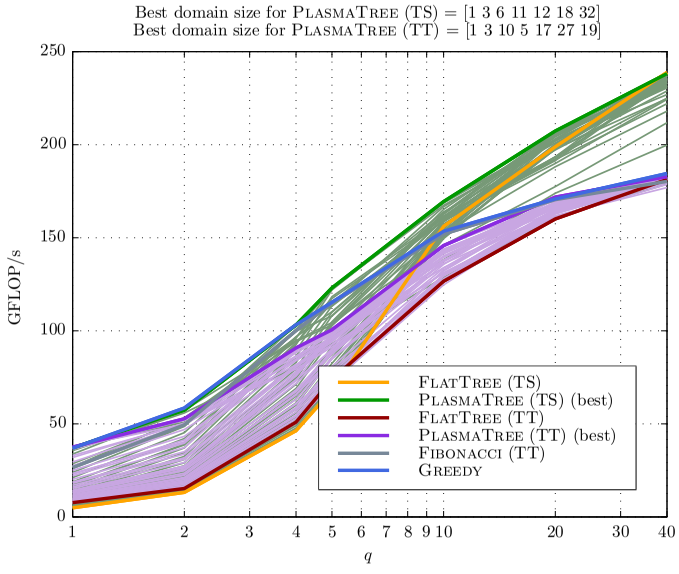
(a) Predicted (double complex)



(b) Experimental (double complex)

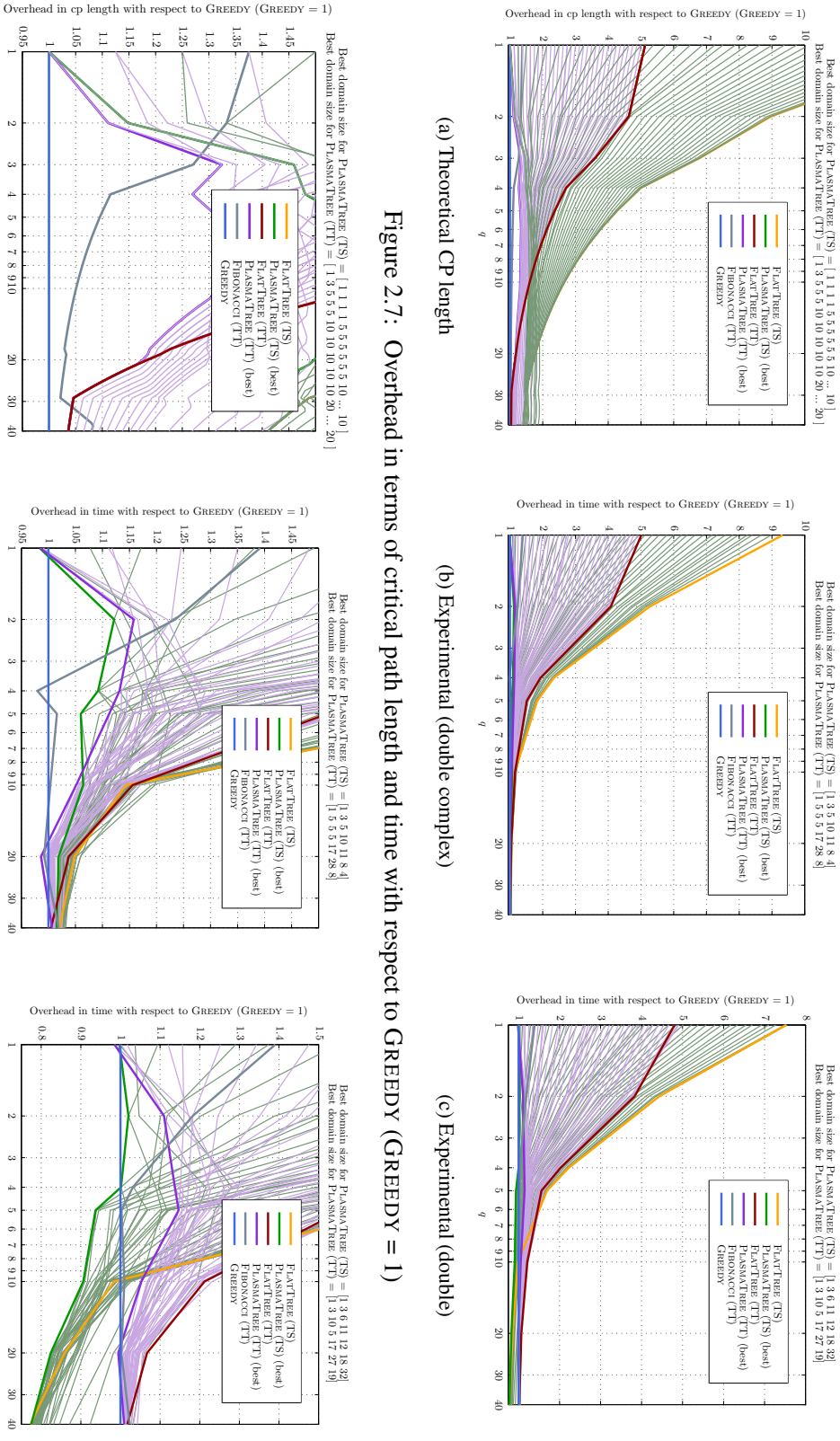


(c) Predicted (double)



(d) Experimental (double)

Figure 2.6: Predicted and experimental performance of QR factorization - All kernels



(a) Theoretical CP length (b) Experimental (double complex) (c) Experimental (double)

Figure 2.7: Overhead in terms of critical path length and time with respect to GREEDY (GREEDY = 1)

(a) Theoretical CP length (b) Experimental (double complex) (c) Experimental (double)

Figure 2.8: Detailed view of the overhead in terms of critical path length and time with respect to GREEDY (GREEDY = 1)

Chapter 3

Scheduling streaming applications on a complex multicore platform

3.1 Introduction

In the previous two chapters, we have focused on linear algebra kernels for traditional homogeneous multicore platforms. Streaming applications represent another example of widely used applications that could benefit from multicore processors. In this chapter, we consider the problem of scheduling such applications on a heterogeneous multicore platform, the IBM QS 22 platform, embedding two STI Cell BE processors, so as to optimize the throughput of the application.

As a matter of a fact, heterogeneous multicore processor has become widespread. Future processors are likely to embed several special-purpose cores –like networking or graphic cores– together with general cores, in order to tackle problems like heat dissipation, computing capacity or power consumption. Deploying an application on this kind of platform becomes a challenging task due to the increasing heterogeneity.

Heterogeneous computing platforms such as Grids have been available for a decade or two. However, the heterogeneity is now likely to exist at a much smaller scale, that is within a single machine, or even a single processor. Major actors of the CPU industry are already planning to include a GPU core to their multicore processors [7]. Classical processors are also often provided with an accelerator (like GPUs, graphics processing units), or with processors dedicated to special computations (like ClearSpeed [26] or Mercury cards [72]), thus resulting in a heterogeneous platform. The best example is probably the IBM RoadRunner, the first supercomputer to break the petaflop barrier, which is composed of an heterogeneous collection of classical AMD Opteron processors and Cell processors.

The STI Cell BE processor is an example of such an heterogeneous architecture, since it embeds both a PowerPC processing unit, and up to eight simpler cores dedicated to vectorial computing. This processor has been used by IBM to design machines dedicated to high performance computing, like the Bladecenter QS 22. We have chosen to focus our study on this platform because the Cell processor is nowadays widely available and affordable, and it is to our mind a good example of future heterogeneous processors.

Deploying an application on such a heterogeneous platform is not an easy task, especially when the application is not purely data-parallel. In this work, we focus on applications that exhibit some regularity, so that we can design efficient static scheduling solutions. We thus concentrate our work on streaming applications. These applications usually concern multimedia stream processing, like video edition software, web radios or Video On Demand applications [98, 49]. However, streaming applications also exist in other domains, like real time data encryption applications, or routing software, which are for example

required to manage mobile communication networks [94]. A stream is a sequence of data that have to go through several processing tasks. The application is generally structured as a directed acyclic task graph, ranging from a simple chain of tasks to a more complex structure, as illustrated in the following.

To process a streaming application on a heterogeneous platform, we have to decide which tasks will be processed onto which processing elements, that is, to find a mapping of the tasks onto the platform. This is a complex problem since we have to take platform heterogeneity, task computing requirements, and communication volume into account. The objective is to optimize the *throughput* of the application: for example in the case of a video stream, we are looking for a solution that maximizes the number of images processed per time-unit.

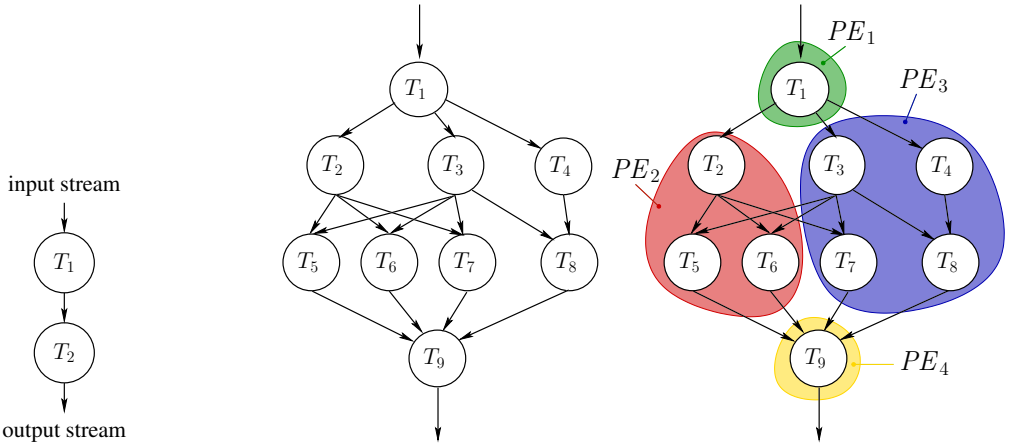
Several streaming solutions have already been developed or adapted for the Cell processor. DataCutter-Lite [48] is an adaptation of the DataCutter framework for the Cell processor, but it is limited to simple streaming applications described as linear chains, so it cannot deal with complex task graphs. StreamIt [47, 91] is a language developed to model streaming applications; a version of the Streamit compiler has been developed for the Cell processor, however it does not allow the user to specify the mapping of the application, and thus to precisely control the application. Some other frameworks allow to handle communications and are rather dedicated to matrix operations, like ALF (part of the IBM Software Kit for Multicore Acceleration [56]), Sequoia [37], CellSs [15] or BlockLib [5].

3.2 General framework and context

Streaming applications may be complex: for example, when organized as a task graph, a simple Vocoder audio filter can be decomposed in 140 tasks. All the data of the input stream must be processed by this task graph in a pipeline fashion. On a parallel platform, we have to decide which processing element will process each task. To optimize the performance of the application, which is usually evaluated using its throughput, we have take into account the computation capabilities of each resource, as well as the incurred communication overhead between processing elements. The platform we target in this study is a heterogeneous multicore processor, which adds to the previous constraints a number of specific limitations (limited memory size on some nodes, specific communication constraints, etc.). Thus, the optimization problem corresponding to the throughput maximization is a complex duty. However, the typical run time of a streaming application is long: a video encoder is likely to run for at least several minutes, and probably several hours. Besides, we target a dedicated environment, with stable run time conditions. Thus, it is worth taking additional time to optimize the application throughput.

In [14], the authors have developed a framework to schedule large instance of similar jobs, known as steady-state scheduling. It aims at maximizing the throughput, that is, the number of jobs processed per time-unit. In the case of a streaming application like a video filter, a job may well represent the complete processing of an image of the stream. Steady-state scheduling is able to deal with complex jobs, described as directed acyclic graphs (DAG) of tasks. The nodes of this graph are the tasks of the applications, denoted by $T_1, \ldots, T_n$, whereas edges represent dependencies between tasks as well as the data associated to the dependencies: $D_{k,l}$ is a data produced by task T_k and needed to process T_l. Figure 3.1a presents the task graph of a simplistic stream application: the stream goes through two successive filters. Applications may be much more complex, as depicted in Figure 3.1b. Computing resources consist in several processing elements $PE_1, \ldots, PE_p$, connected by communication links. We usually adopt the general *unrelated* computation model: the processing time of a task T_k on a processing element PE_i, denoted by $w_{PE_i}(T_k)$ is not necessarily function of the processing element speed, since some processing elements may be faster for some tasks and slower for some others. Communication limitations are taken into account following one of the existing communication models, usually the

one-port or bounded multiport models.



(a) Simple streaming application. (b) Other streaming application. (c) A possible mapping.

Figure 3.1: Applications and mapping.

The application consists of a single task graph, but all data sets of the stream must be processed following this task graph. This results in a large number of copies, or instances, of each task. Consider the processing of the first data set (e.g., the first image of a video stream) for the application described in Figure 3.1b. A possible *mapping* of all these tasks to processing elements is described on Figure 3.1c, which details on which processing element each task will be computed. For the following data sets, there is two possibilities. A possibility is that all data sets use the same mapping, that is, in this example, all tasks T_1 will be processed by processing element PE_1, etc. In this case, a single mapping is used for all instances. Another possibility is that some instances may be processed using different mappings. This multiple-mapping solution allows to split the processing of T_1 among different processors, and thus can usually achieve a larger throughput. However, the control overhead for a multiple-mapping solution is much larger than for a single-mapping solution.

The problem of optimizing the steady-state throughput of an application on heterogeneous platform has been studied in the context of Grid computing. It has been proven that the general problem, with multiple mappings, is NP-complete on general graphs, but can be solved in polynomial time provided that the application graph has a limited depth [13]. Nevertheless, the control policy that would be needed by a multiple-mapping solution would be far to complex for a heterogeneous multicore processor: each processor would need to process a large number of different tasks, store a large number of temporary files, and route messages to multiple destinations depending on their type and instance index. In particular, the limited memory of the Synergistic Processing Element makes it impossible to implement such a complex solution.

Thus, single-mapping solutions are more suited for the targeted platform. These solution have also been studied in the context of Grid computing. In [39], the authors have proven that the general solution is NP-complete, but can be computed using a mixed integer linear program; several heuristics have also been proposed to compute an efficient mapping of the application onto the platform. The present chapter aims at studying the ability of adapting the former solution for heterogeneous multicore processors, and illustrates this on the Cell processor and the QS 22. The task is challenging as it requires to solve the following problems:

– Derive a model of the platform that both allows to accurately predict the behavior of the application, and to compute an efficient mapping using steady-state scheduling techniques.

– As far as possible, adapt the solutions presented in [39] for the obtained model: optimal solution via mixed linear programming and heuristics.
– Develop a light but efficient scheduling software that allows to implement the proposed scheduling policies and test them in real life conditions.

3.3 Adaptation of the scheduling framework to the QS 22 architecture

In this section, we present the adaptation of the steady-state scheduling framework in order to cope with streaming applications on the Cell processor and the QS 22 platform. As outlined, the main changes concern the computing platform. We first briefly present the architecture, and we propose a model for this platform based on communication benchmarks. Then, we detail the application model and some features related to stream applications. Based on these models, we adapt the optimal scheduling policy from [39] using mixed linear programing.

3.3.1 Platform description and model

We detail here the multicore processor used in this study, and the platform which embeds this processor. In order to adapt our scheduling framework to this platform, we need a communication model which is able to predict the time taken to perform a set of transfers between processing elements. As outlined below, no such model is available to the best of our knowledge. Thus, we perform some communication benchmarks on the QS 22, and we derive a model which is suitable for our study.

Description of the QS 22 architecture

The IBM Bladecenter QS 22 is a bi-processor platform embedding two Cell processors and up to 32 GB of DDR2 memory [80]. This platform offers a high computing power for a limited power consumption. The QS 22 has already been used for high performance computing: it is the core of IBM RoadRunner platform, leader of the Top500 from November 2008 to June 2009, the first computer to reach one petaflop [12].

As mentioned in the introduction, the Cell processor is a heterogeneous multicore processor. It has jointly been developed by Sony Computer Entertainment, Toshiba, and IBM [61], and embeds the following components:

– **Power Processing Element (PPE) core.** This two-way multi-threaded core follows the Power ISA 2.03 standard. Its main role is to control the other cores, and to be used by the operating system due to its similarity with existing Power processors.
– **Synergistic Processing Elements (SPE) cores.** These cores constitute the main innovation of the Cell processor and are small 128-bit RISC processors specialized in floating point, SIMD operations. These differences induce that some tasks are by far faster when processed on a SPE, while some other tasks can be slower. Each SPE has its own local memory (called *local store*) of size $LS = 256$ kB, and can access other local stores and main memory only through explicit asynchronous DMA calls.
– **Main memory.** Only PPEs have a transparent access to main memory. The dedicated memory controller is integrated in the Cell processor and allows a fast access to the requested data. Since this memory is by far larger than the SPE's local stores, we do not consider its limited size as a constraint for the mapping of the application. The memory interface supports a total bandwidth of $bw = 25$ GB/s for read and writes combined.

– **Element Interconnect Bus (EIB).** This bus links all parts of the Cell processor to each other. It is composed of 4 unidirectional rings, 2 of them in each direction. The EIB has an aggregated bandwidth $BW = 204.8$ GB/s, and each component is connected to the EIB through a bidirectional interface, with a bandwidth $bw = 25$ GB/s in each direction. Several restrictions apply on the EIB and its underlying rings:
  – A transfer cannot be scheduled on a given ring if data has to travel more than halfway around that ring
  – Only one transfer can take place on a given portion of a ring at a time
  – At most 3 non-overlapping transfers can take place simultaneously on a ring.
– **FLEXIO Interfaces.** The Cell processor has two FLEXIO interfaces ($IOIF_0$ and $IOIF_1$). These interfaces are used to communicate with other devices. Each of these interfaces offers an input bandwidth of $bwio_{\text{in}} = 26$ GB/s and an output bandwidth of $bwio_{\text{out}} = 36.4$ GB/s.

Both Cell processors of the QS 22 are directly interconnected through their respective $IOIF_0$ interface.We will denote the first processor by $Cell_0$, and the second processor $Cell_1$. Each of these processors is connected to a bank of DDR memory; these banks are denoted by $Memory_0$ and $Memory_1$. A processor can access the other bank of memory, but then experiences a non-uniform memory access time.

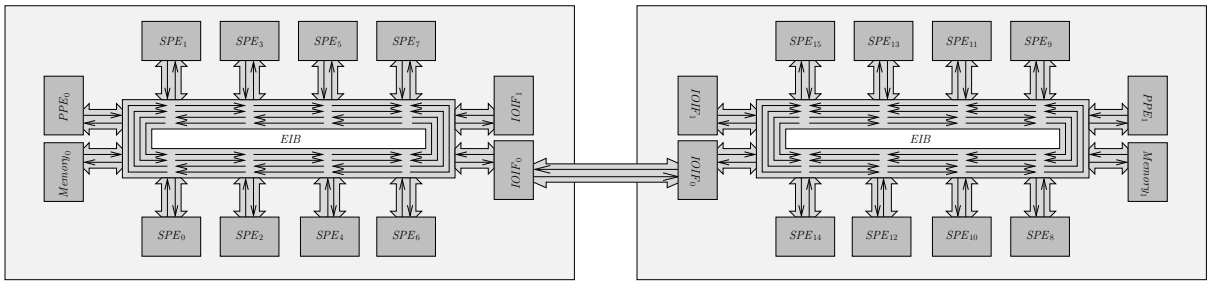


Figure 3.2: Schematic view of the QS 22.

Since we are considering a bi-processor Cell platform, there are $n_P = 2$ PPE cores, denoted by PPE_0 and PPE_1. Furthermore, each Cell processor embedded in the QS 22 has eight SPEs, we thus consider $n_S = 16$ SPEs in our model, denoted by $SPE_0, \ldots, SPE_{15}$ ($SPE_0, \ldots, SPE_7$ belong to the first processor). To simplify the following computations, we gather all processing elements under the same notation PE_i, so that the set of PPEs is $\{PE_0, PE_1\}$, while $\{PE_2, \ldots, PE_{17}\}$ is the set of SPEs (again, PE_2 to PE_9 are the SPEs of the first processor). Let n be the total number of processing elements, i.e., $n = n_P + n_S = 18$. All processing elements and their interconnection are depicted on Figure 3.2. We have two classes of processing elements, which fall under the *unrelated* computation model: a PPE can be fast for a given task T_k and slow for another one T_l, while a SPE can be slower for T_k but faster for T_l. Each core owns a dedicated communication interface (a DMA engine for the SPEs and a memory controller for the PPEs), and communications can thus be overlapped with computations.

Benchmarks and model of the QS 22

We need a precise performance model for the communication within the Bladecenter QS 22. More specifically, we want to predict the time needed to perform any pattern of communication among the computing elements (PPEs and SPEs). However, we do not need this model to be precise enough to predict the behavior of each packet (DMA request, etc.) in the communications elements. We want a simple (thus tractable) and yet accurate enough model. To the best of our knowledge, there does not exist such a

model for irregular communication patterns on the QS 22. Existing studies generally focus on the aggregated bandwidth obtained with regular patterns, and are limited to a single Cell processor [64]. This is why, in addition to the documentation provided by the manufacturer, we perform communication benchmarks. We start by simple scenarios with communication patterns involving only two communication elements, and then move to more complex scenarios to evaluate communication contention.

To do so, we have developed a communication benchmark tool, which is able to perform and time any pattern of communication: it launches (and pins) threads on all the computing elements involved in the communication pattern, and make them read/write the right amount of data from/to a distant memory or local store. For all these benchmarks, each test is repeated a number of times (usually 10 times), and only the average performance is reported. We report the execution times of the experiments, either in nanoseconds (ns) or in cycles: since the time-base frequency of the QS 22 Bladecenter is 26.664 MHz, a cycle is about 37.5 ns long.

In this section, we will first present the timing result for single communications. Then, since we need to understand communication contention, we present the results when performing several simultaneous communications. Thanks to these benchmarks, we are finally able to propose a complete and tractable model for communications on the QS 22.

Note that for any communication between two computing elements PE_i and PE_j, we have to choose between two alternatives: (i) PE_j reads data from PE_i local memory, or (ii) PE_i writes data into PE_j local memory. In order to keep our scheduling framework simple, we have chosen to implement only one of these alternatives. When performing the tests presented in this section, we have noticed a slightly better performance for read operations. Thus, we focus only on read operations: the following benchmarks are presented only for read operations, and we will use only these operations when implementing the scheduling framework. However, write operations would most of the time behave similarly.

Single transfer performance. First, we deal with the simplest kind of transfers: SPE to SPE data transfers. In this test, we choose a pair of SPE in the same Cell, and we make one read in the local store of the other, by issuing the corresponding DMA call. We present in Figure 3.3 the duration of such a transfer for various data sizes, ranging from 8 bytes, to 16 kB (the maximum size for a single transfer). When the size of the data is small, the duration is 112 ns (3 cycles), which we consider as the latency for this type of transfers. For larger data sizes, the execution time increases quasi-linearly. For 16 kB, the bandwidth obtained (around 25 GB/s) is close the theoretical one.

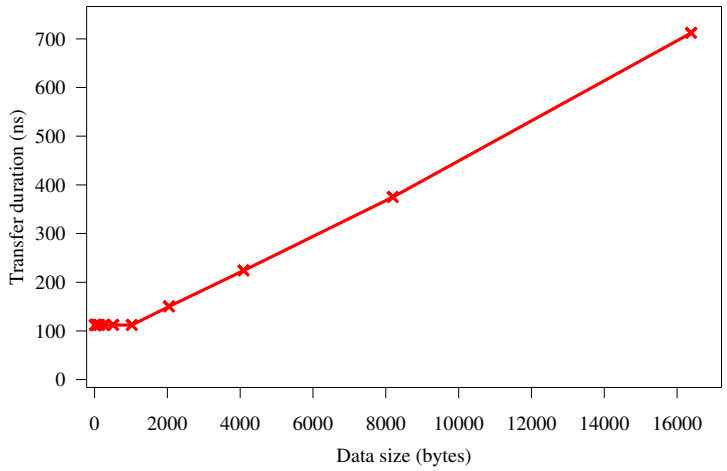


Figure 3.3: Data transfer time between two SPE from the same Cell

When performing the same tests among SPEs from both Cells of the QS 22, the results are not the same.

We measured both bandwidth and latency when performing a "read" operation from a SPE to a distant local store. Table 3.1 presents the average latency for such a transfer, while Table 3.2 shows the obtained bandwidth for a large transfer.

Reader SPE	Distant local store															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	120	128	120	128	113	128	113	113	428	443	443	450	450	443	443	450
1	113	128	113	120	113	143	113	128	458	420	450	443	450	413	465	465
2	113	113	128	120	128	120	113	113	450	443	428	420	443	435	443	450
3	120	113	128	120	120	128	113	128	450	458	450	443	435	435	428	435
4	120	135	128	113	113	113	113	128	465	458	458	435	443	443	443	443
5	113	113	113	113	113	113	113	120	450	450	443	435	450	450	458	450
6	113	113	113	113	113	120	113	113	458	435	443	435	405	443	450	450
7	113	113	128	113	113	113	113	120	435	450	450	450	435	458	450	450
8	480	458	488	473	473	473	458	443	113	113	113	113	113	113	113	113
9	450	480	473	495	428	458	480	458	113	113	113	113	113	113	113	113
10	443	443	428	465	443	450	443	443	113	113	113	113	113	113	113	113
11	480	473	458	488	465	473	443	465	113	120	113	113	113	113	113	113
12	458	450	450	458	458	465	443	435	113	113	113	113	113	113	113	128
13	458	420	435	450	480	443	443	450	113	113	120	113	113	113	113	113
14	443	443	428	450	458	443	435	450	128	113	120	113	113	113	113	113
15	450	488	435	435	443	450	458	435	113	113	113	113	113	113	113	113

Table 3.1: Latency when a SPE reads from a distant local store, in nanoseconds.

The results can be summarized as follows: when a SPE from $Cell_0$ reads from a local store located in $Cell_1$, the average latency of the transfer is 444 ns, and the average bandwidth is 4.91 GB/s. When on the contrary, a SPE from $Cell_1$ reads some data in local store of $Cell_0$, then the average latency is 454 ns, and the average bandwidth is 3.38 GB/s. This little asymmetry can be explained by the fact that both Cells do not play the same role: the data arbiter and address concentrators of $Cell_0$ act as masters for communications involving both Cells, which explains why communications originating from different Cells are handled differently.

Finally, we have to study the special case of a PPE reading from a SPE's local store. This transfer is particular, since it involves multiple transfers: in order to perform the communication, the PPE adds an DMA instruction in the transfer queue of the corresponding SPE. Then, the DMA engine of the SPE reads this instruction and perform the copy into the main memory. Then, in order for the data to be loaded available for the PPE, it is loaded in its private cache. Due to this multiple transfers, and to the fact that DMA instructions issued by the PPE have a smaller priority than the local instructions, the latency of these communications is particularly large, about 300 cycles (11250 ns). With such a high latency, the effect of the size of the data on the transfer time is almost negligible, which makes it difficult to compute a maximal bandwidth.

Concurrent transfers. The previous benchmarks gives us an insight of the performance for a single transfer. However, when performing several transfers at the same time, other effect may appear: concurrent transfers may be able to reach a larger aggregated bandwidth, but each of them may also suffer from the contention and have its bandwidth reduced. In the following, we try to both exhibit the maximum bandwidth of each connecting element, and to understand how the available bandwidth is shared

Reader SPE	Distant local store															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	23.1	23.4	24.2	24.8	24.8	23.9	23.5	23.9	4.9	4.8	4.9	4.9	4.9	5.1	4.9	5.1
1	23.9	23.7	23.4	24.5	23.2	24.5	23.6	24.2	4.9	4.9	4.8	4.9	5.0	4.8	4.9	4.8
2	23.9	24.2	23.1	24.2	23.0	23.4	23.7	23.9	5.0	4.9	4.9	4.8	5.0	4.9	5.0	4.9
3	24.5	23.9	24.2	23.5	22.3	24.0	25.1	24.8	5.1	4.7	4.9	5.0	4.9	4.9	4.9	5.0
4	25.4	25.4	23.9	25.1	25.4	25.4	25.4	25.4	4.9	5.0	5.0	5.0	5.0	5.0	4.8	4.8
5	25.4	25.4	24.0	25.4	25.4	25.4	24.8	25.4	4.9	4.8	4.8	4.8	5.0	4.9	4.9	4.9
6	25.1	25.1	25.1	25.4	24.8	25.4	25.4	24.8	4.9	4.9	4.9	4.9	4.9	5.0	5.0	5.0
7	24.3	25.1	24.5	25.1	23.1	24.6	25.1	24.5	4.9	5.0	4.8	4.8	4.9	4.8	4.9	5.1
8	3.3	3.3	3.5	3.4	3.4	3.4	3.5	3.5	25.4	25.4	24.8	25.1	25.4	25.4	25.4	25.4
9	3.5	3.3	3.5	3.3	3.3	3.4	3.5	3.5	25.4	25.4	24.8	25.1	24.5	25.1	25.4	25.4
10	3.3	3.3	3.4	3.4	3.4	3.4	3.5	3.3	25.4	25.4	24.6	25.1	24.8	25.4	25.4	25.1
11	3.5	3.3	3.4	3.3	3.4	3.4	3.3	3.4	25.4	25.4	25.1	25.4	25.4	25.4	25.4	25.4
12	3.3	3.3	3.4	3.3	3.5	3.4	3.4	3.4	25.4	25.4	25.4	25.1	25.4	25.4	24.6	25.4
13	3.3	3.2	3.4	3.3	3.3	3.3	3.3	3.4	25.4	25.4	25.1	24.8	25.4	25.4	25.4	25.4
14	3.4	3.3	3.4	3.4	3.4	3.3	3.5	3.5	25.4	25.1	24.8	25.4	25.4	25.4	25.4	25.4
15	3.4	3.4	3.5	3.4	3.4	3.3	3.4	3.4	25.1	25.4	24.3	25.4	25.4	25.1	25.1	25.4

Table 3.2: Bandwidth when a SPE reads from a distant local store, in GB/s.

between concurrent flows.

In a first step, we try to see if the bandwidths measured for single transfers and presented above can be increased when using several transfers instead of one. For SPE-SPE transfers, this is not the case: when performing several read operations on different SPEs from the same local store (on the same Cell), the 25 GB/s limitation makes it impossible to go beyond the single transfer performance.

The Element Interconnect Bus (EIB) has a theoretical capacity of 204.8 GB/s. However, its structure consisting of two bi-directional rings makes certain communication pattern more efficient than others. For patterns which are made of short-distance transfers, and for which can be performed without overlap within one ring, we can almost get 200GB/s out of the EIB. On the other hand, if we concurrently schedule several transfers between elements that are far away from each other, the whole pattern cannot be scheduled on the rings without overlapping some transfers, and the bandwidth is reduced substantially.

We illustrate this phenomenon by two examples. Consider the layout of the SPEs as depicted by Figure 3.2. In the first scenario, we perform the following transfers ($SPE_i \leftarrow SPE_j$ means "SPE_i reads from SPE_j"): $SPE_1 \leftarrow SPE_3$, $SPE_3 \leftarrow SPE_1$, $SPE_5 \leftarrow SPE_7$, $SPE_7 \leftarrow SPE_5$, $SPE_0 \leftarrow SPE_2$, $SPE_2 \leftarrow SPE_0$, $SPE_4 \leftarrow SPE_6$, and $SPE_6 \leftarrow SPE_4$. Then, the aggregated bandwidth is 200 GB/s, that is all transfers get their maximal bandwidth (25 GB/s). Then, we perform another transfer pattern: $SPE_0 \leftarrow SPE_7$, $SPE_7 \leftarrow SPE_0$, $SPE_1 \leftarrow SPE_6$, $SPE_6 \leftarrow SPE_1$, $SPE_2 \leftarrow SPE_5$, $SPE_5 \leftarrow SPE_2$, $SPE_3 \leftarrow SPE_4$, and $SPE_4 \leftarrow SPE_3$. In this case, the aggregated bandwidth is only 80 GB/s, because of the large overlap between all transfers. However, these cases are extreme ones; Figure 3.4 shows the distribution of the aggregated bandwidth when we run one hundred random transfer patterns. The average bandwidth is 149 GB/s.

We have measured earlier that the bandwidth of a single transfer between the two Cells (through the FlexIO) was limited to either 4.91 GB/s or 3.38 GB/s depending on the direction of the transfer. When we aggregate several transfer through the FlexIO, a larger bandwidth can be obtained: when several SPEs from $Cell_0$ reads from local stores on $Cell_1$, the maximal cumulated bandwidth is 13 GB/s, where as it is 11.5 GB/s for the converse scenario. Finally, we have measured the overall bandwidth of the FlexIO, when performing transfers in both direction. The cumulated bandwidth is only 19 GB/s, and it is interesting to note that all transfers initiated by $Cell_0$ gets an aggregated bandwidth of 10 GB/s, while

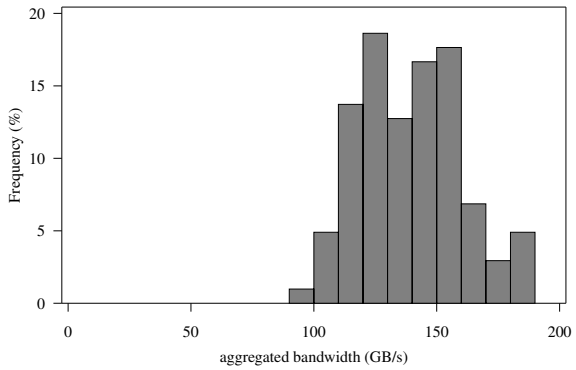


Figure 3.4: Bandwidth distribution

all transfers initiated by $Cell_1$ gets 9 GB/s. Except for the last scenario, bandwidth is always shared quite fairly among the different flows.

Toward a communication model for the QS 22. We are now able to propose a communication model of the Cell processor and its integration in the Bladecenter QS 22. This model has to be a trade-off between accuracy and tractability. Our ultimate goal is to estimate the time needed to perform a whole pattern of transfers. The pattern is described by the amount of data exchanged between any pair of processing elements. Given two processing elements PE_i and PE_j, we denote by $\delta_{i,j}$ the size (in GB) of the data which is read by PE_j from PE_i's local memory. If PE_i is a SPE, its local memory is naturally its local store, and for the sake of simplicity, we consider that the local memory of a PPE is the main memory of the same Cell. Note that the quantity $\delta_{i,j}$ may well be zero if no communication happens between PE_i and PE_j. We denote by $\mathcal{T}$ the time needed to complete the whole transfer pattern.

We will use this model to optimize the schedule of a stream of task graphs, and this purpose has an impact on the model design. In our scheduling framework, we try to overlap communications with computations: a computing resource processes a task while it receives the data for the next task, as outlined later when we discuss about steady-state scheduling. Thus, we are more concerned by the capacity of the interconnection network, and the bandwidth of different resources underlined above, than by the latencies. In the following, we approximate the total completion time $\mathcal{T}$ by the maximum occupation time of all communication resources. For the sake of simplicity, all communication times will be expressed linearly with the data size. This restriction may lower the accuracy of our model, as the behavior of communication in the Cell may not always follow linear laws: this is especially true in the case of multiple transfers. However, a linear model renders the optimization of the schedule tractable. A more complex model would result in more complex, thus harder, scheduling problem. Instead, we consider that a linear model is a very good trade-off between the simplicity of the solution and its accuracy.

In the following, we denote by $chip(i)$ the index of the Cell where processing element PE_i lies ($chip(i)$ = 0 or 1):

$$chip(i) = \begin{cases} 0 & \text{if } i = 0 \text{ or } 2 \leq i \leq 9 \\ 1 & \text{if } i = 1 \text{ or } 10 \leq i \leq 17 \end{cases}$$

We first consider the capacity of the input port of every processing element (either PPE or SPE), which is limited to 25 GB/s:

$$\forall PE_i, \quad \sum_{j=0}^{17} \delta_{i,j} \times \frac{1}{25} \leq \mathcal{T} \tag{3.1}$$

The output capacity of the processing elements and the main memory is also limited to 25 GB/s.

$$\forall PE_i, \quad \sum_{j=0}^{17} \delta_{j,i} \times \frac{1}{25} \leq \mathcal{T} \tag{3.2}$$

When a PPE is performing a read operation, the maximum bandwidth of this transfer cannot exceed 2 GB/s.

$$\forall PE_i \text{ such that } 0 \leq i \leq 1), \quad \forall PE_j, \delta_{i,j} \times \frac{1}{2} \leq \mathcal{T} \tag{3.3}$$

The average aggregate capacity of the EIB of one Cell is limited to 149 GB/s.

$$\forall Cell_k, \quad \sum_{\substack{0 \leq i,j \leq 17 \text{ with} \\ chip(i)=k \text{ or } chip(j)=k}} \delta_{i,j} \times \frac{1}{149} \leq \mathcal{T} \tag{3.4}$$

When a SPE of $Cell_0$ reads from a local memory in $Cell_1$, the bandwidth of the transfer is to 4.91 GB/s.

$$\forall PE_i \text{ such that } chip(i) = 0, \quad \sum_{j,\ chip(j)=1} \delta_{j,i} \times \frac{1}{4.91} \leq \mathcal{T} \tag{3.5}$$

Similarly, when a SPE of $Cell_1$ reads from a local memory in $Cell_0$, the bandwidth is limited to 3.38 GB/s.

$$\forall PE_i \text{ such that } chip(i) = 1, \quad \sum_{j,\ chip(j)=0} \delta_{j,i} \times \frac{1}{3.38} \leq \mathcal{T} \tag{3.6}$$

On the whole, all processing elements of $Cell_0$ cannot read data from $Cell_1$ at a rate larger than 13 GB/s.

$$\sum_{\substack{PE_i, PE_j, \\ chip(i)=0 \text{and} chip(j)=1}} \delta_{j,i} \times \frac{1}{13} \leq \mathcal{T} \tag{3.7}$$

Similarly, all processing elements of $Cell_1$ cannot read data from $Cell_0$ at rate larger than 11.5 GB/s/

$$\sum_{\substack{PE_i, PE_j, \\ chip(i)=1 \text{and} chip(j)=0}} \delta_{j,i} \times \frac{1}{11.5} \leq \mathcal{T} \tag{3.8}$$

The overall capacity of the FlexIO between both Cells is limited to 19 GB/s.

$$\sum_{\substack{PE_i, PE_j \\ chip(i) \neq chip(j)}} \delta_{i,j} \times \frac{1}{19} \leq \mathcal{T} \tag{3.9}$$

We have tested this model on about one hundred random transfer patterns, comprising between 2 and 49 concurrent transfers. We developed a simple testing tool which enables us to transfer a given amount of (random) data between any pair of nodes (PPE or SPE), and more generally to perform any pattern of communication. For each randomly generated transfer pattern, using this tool, we measured the time t_{exp} needed to perform all communications. This time is compared with the theoretical time t_{th} predicted by the previous constraints, and the ratio $t_{\text{th}}/t_{\text{exp}}$ is computed. We consider the theoretical time t_{th} required to perform one communication pattern to be the minimum $\mathcal{T}$ satisfying Equations 3.1 to 3.9

Figure 3.5 presents the distribution of this ratio for all transfer patterns. This figure shows that on average, the predicted communication time is close to the experimental one (the average absolute between the theoretical and the experimental time is 13%). We can also notice that our model is slightly pessimistic (the average ratio is 0.89). This is because the congestion constraints presented above correspond to scenarios where all transfers must go through the same interface, which is unlikely. In practice, communications are scattered among all communication units, so that the total time for communications is slightly less than what is predicted by the model. However, we choose to keep our conservative model, to prevent an excessive usage of communications.

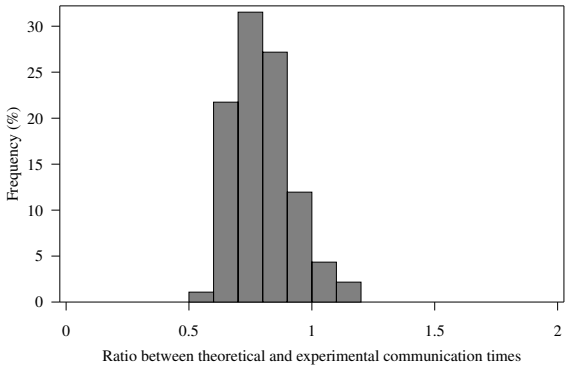


Figure 3.5: Model accuracy

Communications and DMA calls

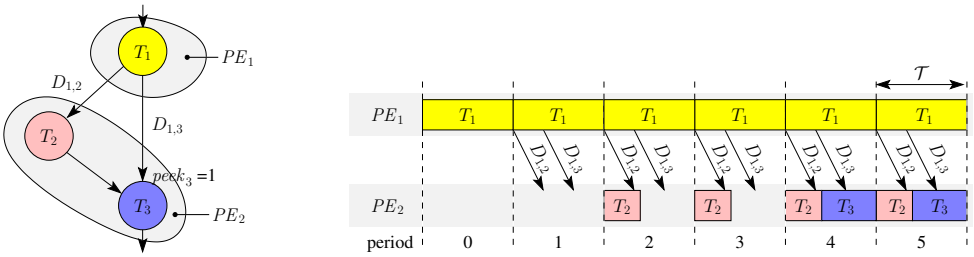
The Cell processor has very specific constraints, especially on communications between cores. Even if SPEs are able to receive and send data while they are doing some computation, they are not multi-threaded. The computation must be interrupted to initiate a communication (but the computation is resumed immediately after the initialization of the communication). Due to the absence of auto-interruption mechanism, the thread running on each SPE has regularly to suspend its computation and check the status of current DMA calls. Moreover, the DMA stack on each SPE has a limited size. A SPE can issue at most 16 simultaneous DMA calls, and can handle at most 8 simultaneous DMA calls issued by the PPEs. Furthermore, when building a steady-state schedule, we do not want to precisely order communications among processing elements. Indeed, such a task would require a lot of synchronizations. On the contrary, we assume that all the communications of a given period may happen simultaneously. These communications correspond to edges $D_{k,l}$ of the task graph when tasks T_k and T_l are not mapped on the same processing element. With the previous limitation on concurrent DMA calls, this induces a strong limitation on the mapping: each SPE is able to receive at most 16 different data, and to send at most 8 data to PPEs per period.

3.3.2 Mapping a streaming application on the Cell

Thanks to the model obtained in the previous section, we are now able to design an efficient strategy to map a streaming application on the target platform. We first recall how we model the application. We then detail some specifics of the implementation of a streaming application on the Cell processor.

Complete application model

As presented above, we target complex streaming applications, as the one depicted on Figure 3.1b. These applications are commonly modeled with a Directed Acyclic Graph (DAG) $G_A = (V_A, E_A)$. The set V_A of nodes corresponds to tasks $T_1, \ldots, T_K$. The set E_A of edges models the dependencies between tasks, and the associated data: the edge from T_k to T_l is denoted by $D_{k,l}$. A data $D_{k,l}$, of size $data_{k,l}$ (in bytes), models a dependency between two task T_k and T_l, so that the processing of the ith instance of task T_l requires the data corresponding to the ith instance of data $D_{l,k}$ produced by T_k. Moreover, it may well be the case that T_l also requires the results of a few instances following the ith instance. In other words, T_l may need information on the near future (i.e., the next instances) before actually processing an instance. For example, this happens in video encoding software, when the program only encodes the difference between two images. We denote by $peek_k$ the number of such instances. More formally, instances $i, i+1, \ldots, i+peek_k$ of $D_{k,l}$ are needed to process the ith instance of T_l. This number of following instances is important not only when constructing the actual schedule and synchronizing the processing elements, but also when computing the mapping, because of the limited size of local memories holding temporary data.



(a) Application and mapping. $peek_1 = peek_2 = 0$ and $peek_3 = 1$

(b) Periodic schedule.

Figure 3.6: Mapping and schedule

Given an application, our goal is to determine the best mapping of the tasks onto the processing element. Once the mapping is chosen, a periodic schedule is automatically constructed as illustrated on Figure 3.6. After a few periods for initialization, each processing element enters a *steady state* phase. During this phase, a processing element in charge of a task T_k has to simultaneously perform three operations. First it has to process one instance of T_k. It has to send the result $D_{k,l}$ of the previous instance to the processing element in charge of each successor task T_l. Finally, it has to receive the data $D_{j,k}$ of the next instance from the processing element in charge of each predecessor task T_j. The exact construction of this periodic schedule is detailed in [13] for general mappings. In our case, the construction of the schedule is quite straightforward: a processing element PE_i in charge of a task T_k simply processes it as soon as its input data is available. In other words, as soon as PE_i has received the data for the current instance and potentially the $peek_k$ following ones. For the sake of simplicity, we do not consider the precise ordering of communications within a period. On the contrary, we assume that all communications can happen simultaneously in one period as soon as the communication constraints expressed in the previous section are satisfied.

Complexity of the problem

In the previous section, we have seen that the mapping of an application onto the computing platform totally defines the schedule and its throughput. In this section, we discuss the complexity of the problem

of finding a mapping with optimal throughput. In this section, we forget about memory constraints, and constraints on the number of DMA transfers: those constraints are not needed to prove that the problem is difficult. The associated decision problem is formally defined as follows.

Definition 3.1 (Cell-Mapping)**.** *Given a directed acyclic application graph G_A, a Cell processor with n_P PPE cores and n_S SPE cores and a bound B, is there a mapping with throughput $\rho \geq B$?*

Theorem 3.1. *Cell-Mapping is NP-complete in the strong sense.*

Proof. First note that we can check in polynomial time if a mapping has a given throughput: we simply have to make sure that the occupation time of each resource (processing element or communication interface) for processing one instance is not larger than $1/B$. Thus, this problem belongs to the NP class.

We prove the NP-completeness using a straightforward reduction from the Minimum Multiprocessor Scheduling problem restricted to two machines, which is known to be NP-complete even with a fixed number of machines [40]. An instance $\mathcal{I}_1$ of Minimum Multiprocessor Scheduling with two machines consists in n tasks $T_1, \ldots, T_n$ provided with a length $l(k,i)$ for each task T_k and each processor $i = 1, 2$, and a bound B'. The goal is to find an allocation f of the tasks on the processors such that $\sum_{k,f(k)=i} l(k,i) \leq B$ for each processor $i = 1, 2$.

We construct an instance $\mathcal{I}_2$ of our problem with one PPE (corresponding to processor 1) and one SPE (corresponding to processor 2). The streaming application consists in a simple chain of n tasks: $V_A = \{T_1, \ldots, T_n\}$ and there is a dependency $D_{k,k+1}$ for all $k = 1, \ldots n-1$. The computation times are $w_{PPE}(T_t) = l(t,1)$ and $w_{SPE}(T_t) = l(t,2)$, and communication costs are neglected: $data_{k,k+1} = 0$ for all $k = 1, \ldots n-1$. Finally, we set $B = 1/B'$. The construction of I_2 is obviously polynomial.

Assume that $\mathcal{I}_1$ has a solution f. Then, f defines a mapping of the tasks onto the processing elements. Using this mapping, the maximum occupation time of each processing element is at most B. Therefore, the throughput of this mapping is at least $1/B = B'$. f is thus a solution for instance $\mathcal{I}_1$. Similarly, a solution for $\mathcal{I}_2$ is to a solution for $\mathcal{I}_1$. ■

The proof does not make use of communications:the problem is NP-complete even without any communication costs. Note that the restriction of Minimum Multiprocessor Scheduling with a fixed number of machines admits a fully polynomial approximation scheme (FPTAS) [54]. However, when considering a general application, communications have to be mapped together with computations, and the previous FPTAS cannot be applied.

Determining buffer sizes

Since SPEs have only 256 kB of local store, memory constraints on the mapping are tight. We need to precisely model them by computing the exact buffer sizes required by the application.

Mainly for technical reasons, the code of the whole application is replicated in the local stores of SPEs (of limited size LS) and in the memory shared by PPEs. We denote by $code$ the size of the code deployed on each SPE, so that the available memory for buffers is $LS - code$. A SPE processing a task T_k has to devote a part of its memory to the buffers dedicated to hold incoming data $D_{j,k}$, as well as for outgoing data $D_{k,l}$. Note that both buffers have to be allocated into the SPE's memory even if one of the neighbor tasks T_j or T_l is mapped on the same SPE. In a future optimization, we could save memory by avoiding the duplication of buffers for neighbor tasks mapped on the same SPE.

As presented above, before computing an instance of a task T_k, a processing element has to receive all the corresponding data, that is the data $D_{j,k}$ produced by each predecessor task T_j, both for the current instance and for the $peek_k$ following instances. Moreover processing elements are not synchronized on

the same instance. Thus, the results of several instances need to be stored during the execution. In order to compute the number of stored data, we first compute the index of the period in the schedule when the first instance of T_k is processed. The index of this period is denoted by $\mathit{firstPeriod}(T_k)$, and is expressed by:

$$\mathit{firstPeriod}(T_k) = \begin{cases} 0 & \text{if } T_k \text{ has no predecessor,} \\ \max\limits_{D_{j,k}} \left(\mathit{firstPeriod}(T_j)\right) + \mathit{peek}_k + 2 & \text{otherwise.} \end{cases}$$

All predecessors of an instance of task T_k are processed after $\max_{D_{j,k}} \left(\mathit{firstPeriod}(T_j)\right)+1$ periods. We have also to wait for peek_k additional periods if some following instances are needed. An additional period is added for the communication from the processing element handling the data, hence the result. By induction on the structure of the task graph, this allows to compute $\mathit{firstPeriod}$ for all tasks. For example, with the task graph and mapping described on Figure 3.6, we have $\mathit{firstPeriod}(1) = 0$, $\mathit{firstPeriod}(2) = 2$, and $\mathit{firstPeriod}(3) = 4$. Again, we could have avoided the additional period dedicated for communication when tasks are mapped on the same processor (e.g., we could have $\mathit{firstPeriod}(3) = 3$). However, we left this optimization as a future work to keep our scheduling framework simple.

Once the $\mathit{firstPeriod}(T_k)$ value of a task T_k is known, buffer sizes can be computed. For a given data $D_{k,l}$, the number of temporary instances of this data that have to be stored in the system is $\mathit{firstPeriod}(T_l) - \mathit{firstPeriod}(T_k)$. Thus, the size of the buffer needed to store this data is $\mathit{temp}_{k,l} = \mathit{data}_{k,l} \times (\mathit{firstPeriod}(T_l) - \mathit{firstPeriod}(T_k))$.

3.3.3 Optimal mapping through mixed linear programming

In this section, we present a mixed linear programming approach that allows to compute a theoretically optimal mapping. We recall that our objective is to map the task of the application on the available processing elements to get the largest throughput for the application, that is to maximize the number of instances that can be processed by time-unit. Typically, for a visual application, our goal is to deliver the maximum number of images per second.

The solution proposed in this section allows to compute a mapping of the tasks onto the processing elements which achieve the optimal throughput according to the model presented in the previous section. It is derived from [39], but takes into account the specific constraints of the QS 22 platform. The problem is expressed as a linear program where integer and rational variables coexist. Although the problem remains NP-complete, in practice, some software are able to solve such linear programs [29]. Indeed, thanks to the limited number of processing elements in the QS 22, we are able to compute the optimal solution for task graphs of reasonable size (up to a few hundreds of tasks).

Our linear programming formulation makes use of both integer and rational variables. The integer variables are described below. They can only take values 0 or 1.

- α's variables which characterize where each task is processed: $\alpha_i^k = 1$ if and only if task T_k is mapped on processing element PE_i.
- β's variables which characterize the mapping of data transfers: $\beta_{i,j}^{k,l} = 1$ if and only if data $D_{k,l}$ is transferred from PE_i to PE_j (note that the same processing element may well handle both task if $i = j$).

Obviously, these variables are related. In particular, $\beta_{i,j}^{k,l} = \alpha_i^k \times \alpha_j^l$, but this redundancy allows us to express the problem as a set of linear constraints. The objective of the linear program is to minimize the duration $\mathcal{T}$ of a period, which corresponds to maximizing the throughput $\rho = 1/\mathcal{T}$. The constraints

of the linear program are detailed below. Remember that processing elements $PE_0, \ldots, PE_{n_P-1}$ are PPEs whereas $PE_{n_P}, \ldots, PE_n$ are SPEs.

– α and β are integers.

$$\forall D_{k,l}, \forall PE_i \text{ and } PE_j, \qquad \alpha_i^k \in \{0,1\}, \beta_{i,j}^{k,l} \in \{0,1\} \tag{3.10}$$

– Each task is mapped on one and only one processing element.

$$\forall T_k, \qquad \sum_{i=0}^{n-1} \alpha_i^k = 1 \tag{3.11}$$

– The processing element computing a task holds all necessary input data.

$$\forall D_{k,l}, \forall j, 0 \leq j \leq n-1, \qquad \sum_{i=0}^{n-1} (\beta_{i,j}^{k,l}) \geq \alpha_j^l \tag{3.12}$$

– A processing element can send the output data of a task only if it processes the corresponding task.

$$\forall D_{k,l}, \forall i, 0 \leq i \leq n-1, \qquad \sum_{j=0}^{n-1} (\beta_{i,j}^{k,l}) \leq \alpha_i^k \tag{3.13}$$

– The computing time of each processing element (PPE or SPE) is no larger that $\mathcal{T}$.

$$\forall i, 0 \leq i < n_P, \qquad \sum_{T_k} (\alpha_i^k w_{PPE}(T_k)) \leq \mathcal{T} \tag{3.14}$$

$$\forall i, n_P \leq i < n, \qquad \sum_{T_k} (\alpha_i^k w_{SPE}(T_k)) \leq \mathcal{T} \tag{3.15}$$

– All temporary buffers allocated on the SPEs fit into their local stores.

$$\forall i, n_P \leq i < n, \qquad \sum_{T_k} \left(\alpha_i^k \left(\sum_{D_{k,l}} temp_{k,l} + \sum_{D_{l,k}} temp_{l,k} \right) \right) \leq LS - code \tag{3.16}$$

– A SPE can perform at most 16 simultaneous incoming DMA calls, and at most eight simultaneous DMA calls are issued by PPEs on each SPE.

$$\forall j, n_P \leq j < n, \qquad \sum_{0 \leq i < n, i \neq j} \sum_{D_{k,l}} \beta_{i,j}^{k,l} \leq 16 \tag{3.17}$$

$$\forall i, n_P \leq i < n, \qquad \sum_{0 \leq j < n_P} \sum_{D_{k,l}} \beta_{i,j}^{k,l} \leq 8 \tag{3.18}$$

– The amount of data communicated among processing elements during one period can be deduced from β.

$$\forall PE_i \text{ and } PE_j, \qquad \delta_{i,j} = \sum_{D_{k,l}} \beta_{i,j}^{k,l} data_{k,l} \tag{3.19}$$

Using this definition, Equations (3.1), (3.2), (3.3), (3.4), (3.5), (3.6), (3.7), (3.8), and (3.9) ensure that all communications are performed within period $\mathcal{T}$.

The size of the linear program depends on the size of the task graphs: it counts $n|V_A| + n^2|E_A| + 1$ variables.

We denote by $\rho_{opt} = 1/\mathcal{T}_{opt}$ the optimal throughput, where $\mathcal{T}_{opt}$ is the value of $\mathcal{T}$ in any optimal solution of the linear program.This linear program computes a mapping of the application that reaches the maximum achievable throughput. By construction, α is a valid mapping, and all possible mappings can be described as α and β variables, which obey the constraints of the linear program.

3.4 Low-complexity heuristics

For large task graphs, solving the linear program may not be possible, or may take a very long time. This is why we propose in this section heuristics with lower complexity to find a mapping of the task graph on the QS 22. As explained in Section 3.3.2, the schedule which takes into account precedence constraints, naturally derives from the mapping. We start with straightforward greedy mapping algorithms, and then move to more involved strategies.

We recall a few useful notations for this section: $w_{PPE}(T_k)$ denotes the processing time of task T_k on any PPE, while $w_{SPE}(T_k)$ is its processing time on a SPE. For data dependency $D_{k,l}$ between tasks T_k and T_l, we have computed $temp_{k,l}$, the number of temporary data that must be stored in steady state. Thus, we can compute the overall buffer capacity needed for a given task T_k, which corresponds to the buffers for all incoming and outgoing data: $buffers[k] = \sum_{j \neq k}(temp_{j,k} + temp_{k,j})$.

3.4.1 Communication-unaware load-balancing heuristic

The first heuristic is a greedy load-balancing strategy, which is only concerned with computation, and does not take communication into account. Usually, such algorithms offer reasonable solutions while having a low complexity.

This strategy first consider SPEs, since they hold the major part of the processing power of the platform. The tasks are sorted according to their affinity with SPEs, and the tasks with larger affinity are load-balanced among SPEs. Tasks that do not fit in the local store of SPEs are mapped on PPEs. After this first step, some PPE might be underutilized. In this case, we then move some tasks which have affinity with PPEs from SPEs to PPEs, until the load is globally balanced. This heuristic will be referred to as GREEDY in the following, and is detailed in Algorithm 8.

3.4.2 Prerequisites for communication-aware heuristics

When considering complex task graphs, handling communications while mapping tasks onto processors is a hard task. This is especially true on the QS 22, which has several heterogeneous communication links, and even within each of its Cell processors. The previous heuristic ignore communications for the sake of simplicity. However, being aware of communications while mapping tasks onto processing element is crucial for performance. We present here a common framework to handle communications in our heuristics.

Partitioning the Cell in clusters of processing elements

As presented above, the QS 22 is made of several heterogeneous processing elements. In order to handle communications, we first simplify its architecture and aggregate these processing elements into coarse-grain groups sharing common characteristics.

If we take a closer look on the QS 22, we can see that some of the communication links are likely to become bottlenecks. This is the case for the link between the PPEs and their respective set of SPEs, and for the link between both Cell chips. Based on this observation, the QS 22 can therefore be partitioned into four sets of processing elements, as shown in Figure 3.7. The communications within each set are supposed to be fast enough. Therefore, their optimization is not crucial for performance, and only the communications among the sets will be taken into account.

In order to estimate the performance of any candidate mapping of a given task graph on this platform, it is necessary to evaluate both communication and computation times. We adopt a similar view as developed above for the design of the linear program. Given a mapping, we estimate the time taken by

Algorithm 8: GREEDY(G_A)

foreach T_k **do** $affinity(T_k) \leftarrow \frac{w_{SPE}(T_k)}{w_{PPE}(T_k)}$
foreach SPE_i **do**
  $workload[SPE_i] \leftarrow 0$
  $memload[SPE_i] \leftarrow 0$
foreach PPE_j **do**
  $workload[PPE_j] \leftarrow 0$
foreach T_k *in non decreasing order of affinity* **do**
  Find SPE_i such that $workload[SPE_i]$ is minimal and $memload[SPE_i] + buffers[k] \leq LS$
  if *there is such a* SPE_i **then**
    $mapping[T_k] \leftarrow SPE_i$ `/* Map` T_k `onto` SPE_i `*/`
    $workload[SPE_i] \leftarrow workload[SPE_i] + w_{SPE}(T_k)$
    $memload[SPE_i] \leftarrow memload[SPE_i] + buffers[k]$
  else
    Find PPE_j such that $workload[PPE_j]$ is minimal
    $mapping[T_k] \leftarrow PPE_j$ `/* Map` T_k `onto` PPE_j `*/`
    $workload[PPE_j] \leftarrow workload[PPE_j] + w_{PPE}(T_k)$
while *the maximum workload of PPEs is smaller than the maximum workload of SPEs* **do**
  Let SPE_i be the SPE with maximum workload
  Let PPE_j be the PPE with minimum workload
  Consider the list of tasks mapped on SPE_i, ordered by non-increasing order of affinity
  Find the first task T_k in this list such that $workload[PPE_j] + w_{PPE}(T_k) \leq workload[SPE_i]$ and $workload[SPE_i] - w_{SPE}(T_k)) \leq workload[SPE_i]$
  if *there is such a task* T_k **then**
    Move T_k from SPE_i to PPE_j:
    $mapping[T_k] \leftarrow PPE_j$
    $workload[PPE_j] \leftarrow workload[PPE_j] + w_{PPE}(T_k)$
    $workload[SPE_i] \leftarrow workload[SPE_i] - w_{SPE}(T_k)$
    $memload[SPE_i] \leftarrow memload[SPE_i] - buffers[k]$
  else
    get out of the *while* loop
return $mapping$

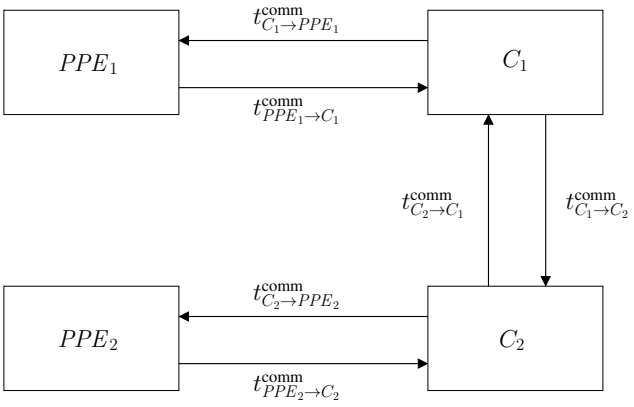


Figure 3.7: Partition of the processing elements within the Cell

all computations and communications, and we compute the period as the maximum of all processing times.

As far as computations are concerned, we estimate the computation time t^{comp}_P of each part P of processing element as the maximal workload of any processing element within the set. The workload of a processing element PE is given by $\sum_{T_k \in S} w_{SPE}(T_k)$ in case of SPE, and by $\sum_{T_k \in S} w_{PPE}(T_k)$ in case of a PPE, where S is the set of tasks mapped on the processing element.

For communications, we need to estimate the data transfer times on every link interconnecting the sets of processing elements, as represented on Figure 3.7. For each unidirectional communication link between two parts P_1 and P_2, we define $t^{\text{comm}}_{P_1 \to P_2}$ as the time required to transfer every data going through that link. We adopt a linear cost model, as in the previous section. Hence, the communication time is equal to the amount of data divided by the bandwidth of the link. We rely on the previous benchmarks for the bandwidths:

- The links links between the two Cell chips have a bandwidth of 13 GB/s ($C_1 \to C_2$) and 11.5 GB/s ($C_2 \to C_1$), as highlighted in Equations (3.7) and (3.8);
- The links from a PPE to the accompanying set of SPEs ($PPE_1 \to C_1$ and $PPE_2 \to C_2$), which correspond to a read operation from the main memory, have a bandwidth of 25 GB/s (see Equation (3.2)).

The routing in this simplified platform is straightforward: for instance, the data transiting between PPE_1 and PPE_2 must go through links $PPE_1 \to C_1$, $C_1 \to C_2$ and finally $C_2 \to PPE_2$. Formally, when $D_{P_1 \to P_2}$ denotes the amount of data transferred from part P_1 to part P_2, the communication time of link $C_1 \to C_2$ is given by:

$$t^{\text{comm}}_{C_1 \to C_2} = \frac{D_{C_1 \to C_2} + D_{PPE_1 \to C_2} + D_{C_1 \to PPE_2} + D_{PPE_1 \to PPE_2}}{13}$$

Load-balancing procedure among the SPEs of a given Cell

All heuristics that will be introduced in the following need to know whether the load-balancing of a given task list across a set of SPEs is feasible or not, and what is the expected computation time. We thus propose a handy greedy mapping policy which balances the load on each SPE, and will be used in more complex heuristics. The provided task list L is first sorted using task weights $w_{SPE}(T_k)$. Then we map the heaviest task T_k on the least loaded SPE SPE_i, provided that it has enough memory left to host T_k. If there is no such SPE, we return an error. Otherwise, we repeat this step until every tasks are mapped onto a given SPE. This procedure is described in Algorithm 9.

Algorithm 9: CELLGREEDYSET(L, C_j)

Input: a list L of tasks, a set C_j of SPEs
Sort L in non increasing order of $w_{SPE}(T_k)$;
foreach $SPE_i \in C_j$ **do**
  $workload[SPE_i] \leftarrow 0$
  $memload[SPE_i] \leftarrow 0$
foreach T_k *in* L **do**
  Search SPE_i in C_j such that $workload[SPE_i]$ is minimal and
  $memload[SPE_i] + buffers[k] \leq LS$
  if *there is such a* SPE_i **then**
    $mapping[T_k] \leftarrow SPE_i$ /* Map T_k on SPE_i */
    $memload[SPE_i] \leftarrow memload[SPE_i] + buffers[k]$
    $workload[SPE_i] \leftarrow workload[SPE_i] + w_{SPE}(T_k)$
  else
    return $FAILURE$
return $mapping$

3.4.3 Clustering and mapping

A classical approach when scheduling a complex task graph on a parallel platform with both communication and computation costs is two use a two-step strategy. In a first set, the tasks are grouped into clusters without considering the platform. In a second step, these clusters are mapped onto the actual computation resources. In our case, the computing resources are the groups of processing elements determined in the previous sections: PPE_1, PPE_2, C_1, and C_2.

The classical greedy heuristic due to Sarkar [84] is used to build up task clusters. The rationale of this heuristic is to consider each expensive communication, and when it is interesting, to group the tasks involved in this communication. This is achieved by merging the clusters containing the source and destination tasks. The merging decision criterion is based on the makespan obtained for a single instance of the task graph. To estimate this makespan, it is assumed that each cluster is processed by a dedicated resource. The processing time of task T_k on a resource is set to $(w_{SPE}(T_k) + w_{PPE}(T_k))/2$, and that the communication bandwidth between these resources is set to $bw = 10\ GB/s$ (the available bandwidth between both Cell chips). In addition, when building clusters, Algorithm 11 bounds the memory footprint of each cluster so that it can be mapped onto any resource (PPE or set of SPEs). In a second step, this heuristic maps the clusters onto the real resources by load-balancing cluster loads across PPEs and sets of SPEs, as described in Algorithm 12.

3.4.4 Iterative refinement using DELEGATE

In this Section, we present an iterative strategy to build an efficient allocation. This method, called DELEGATE, is adapted from the heuristic introduced in [39]. It consists in iteratively refining an allocation by moving some work from a highly loaded resource to a less loaded one, until the load is equally balanced on all resources. Here, resources can be either PPEs (PPE_1 or PPE_2) or set of SPEs (C_1 or C_2). In the beginning, all tasks are mapped on one of the PPE (PPE_1). Then, a (connected) subset of tasks is selected and its processing is delegated to another resource. The new mapping is selected if it respects memory constraints and improves the performance. This refinement procedure is then repeated

Algorithm 10: $EPT(C)$: Estimated parallel time of the clustering C

foreach *task* T_k *in reverse topological order* **do**

$$max^{out}_{local} = \max\Big(bl(T_m) \text{ such that } (T_k, T_m) \in E \text{ and } C(T_k) = C(T_m)\Big)$$

$$max^{out}_{remote} = \max\Big(bl(T_m) + \frac{data_{k,m}}{bw} \text{ such that } (T_k, T_m) \in E \text{ and } C(T_k) \neq C(T_m)\Big)$$

$$bl(T_k) = \frac{w_{PPE}(T_k) + w_{SPE}(T_k)}{2} + \max(max^{out}_{local}, max^{out}_{remote})$$

foreach *task* T_k *in topological order* **do**

$$max^{in}_{local} = \max\Big(tl(T_m) + \frac{w_{PPE}(T_m) + w_{SPE}(T_m)}{2} \text{ such that } (T_m, T_k) \in E \text{ and } C(T_m) = C(T_k)\Big)$$

$$max^{in}_{remote} = \max\Big(tl(T_m) + \frac{w_{PPE}(T_m) + w_{SPE}(T_m)}{2} + \frac{data_{m,k}}{bw} \text{ such that } (T_m, T_k) \in E \text{ and } C(T_m) \neq C(T_k)\Big)$$

$$tl(T_k) = \max(max^{in}_{local}, max^{in}_{remote})$$

return $\max(bl(T_k) + tl(T_k))$

Algorithm 11: $CellClustering(G_A)$

In the initial clustering, each tasks of G_A has its own cluster:
foreach *task* T_k **do** $Clustering(T_k) = k$
compute $EPT(Clustering)$
$L \leftarrow$ list of edges of the task graph
Sort L by non-increasing communication weight
foreach *edge* (T_k, T_l) *in* L **do**
  $newClustering \leftarrow Clustering$
  Merge the clusters containing T_k and T_l:
  foreach *task* T_i **do**
    if $Clustering(T_i) = Clustering(T_l)$ **then**
      $newClustering(T_i) \leftarrow Clustering(T_k)$
  compute $EPT(newClustering)$
  if $EPT(newClustering) \leq EPT(Clustering)$ *and the* CELLGREEDYSET *greedy heuristic successfully maps cluster* $newClustering(T_k)$ *on a set of SPEs* **then**
    $Clustering \leftarrow newClustering$

Algorithm 12: $CellMapClusters(Clustering)$: Map clusters into computing resources

Consider a set of clusters $Clustering$ built with Algorithm 11
foreach *cluster* C *in* $Clustering$ **do**
  Compute the weight of cluster C: $w(c) = \sum_{T_k \in C} w_{PPE}(T_k) + w_{SPE}(T_k)$
Sort L by non-increasing weights
$load(PPE_1) \leftarrow 0$, $load(PPE_2) \leftarrow 0$
foreach *cluster* C *in* $Clustering$ **do** (we try to map C on each resource)
  $newload(PPE_1) \leftarrow load(PPE_1) + \sum_{T_k \in C} w_{PPE}(T_k)$
  $newload(PPE_2) \leftarrow load(PPE_2) + \sum_{T_k \in C} w_{PPE}(T_k)$
  Run CELLGREEDYSET on C_1 using the clusters already mapped to C_1 plus C, compute $newload(C_1)$ as the maximum load of all SPEs in C_1 (let $newload(C_1) \leftarrow \infty$ if CELLGREEDYSET fails)
  Do the same on C_2 to compute $newload(C_2)$
  Select the resource R such that $newload(R)$ is minimum
  Map C on resource R: $mapping[C] \leftarrow R$
  if R *is a PPE* **then** $load(R) \leftarrow newload(R)$
return $mapping$

until no more transfer is possible. If the selected resource for a move is a PPE, it is straightforward to compute the new processing time. If it is a set of SPEs, then the procedure CELLGREEDYSET is used to check memory constraints and compute the new processing time. The algorithm is given in Algorithm 14.

As in [39], at each step, a large number of moves are considered: for each task, all d-neighborhoods of this task are generated (with $d = 1, 2 \ldots d_{\max}$), and mapped onto each available resource. Among this large set of moves, we select the one with best performance. This heuristic needs a more involved way to compute the performance than simply using the period. Consider for example that both PPEs are equally loaded, but all SPEs are free. Then no move can directly decrease the period, but two moves are needed to decrease the load of both PPEs. Thus, for a given mapping, we compute all contributions to the period: the load of each resource, and the time needed for communications between any pair of resources (the period is the maximum of all contributions). The list of these contributions is sorted by non-increasing value. To compare two mappings, the one with the smallest contribution list in lexicographical order is selected, as described in Algorithm 13.

Algorithm 13: $Eval(mapping)$

Compute the list $Score$ of every computation time and communication times of this initial solution :

$$Score = \{t^{\text{comp}}_{C_1}, t^{\text{comp}}_{C_2}, t^{\text{comp}}_{PPE_1}, t^{\text{comp}}_{PPE_2}, t^{\text{comm}}_{C_1 \to C_2}, t^{\text{comm}}_{C_2 \to C_1}, t^{\text{comm}}_{PPE_1 \to C_1}, t^{\text{comm}}_{C_1 \to PPE_1}, t^{\text{comm}}_{PPE_2 \to C_2}, t^{\text{comm}}_{C_2 \to PPE_2}\}$$

Sort $Score$ in non increasing order.
return $Score$

Algorithm 14: DELEGATE(G_A)

```
Map all tasks on PPE_1: foreach T_k do mapping[T_k] ← PPE_1
move_possible ← 1
while move_possible do
    best_move ← ∅
    foreach T_k do
        foreach neighborhood N of T_k do
            foreach S ∈ {PPE_1, PPE_2, C_1, C_2} do
                move ← mapping
                move[N] ← S
                if S is a set of SPEs and CELLGREEDYSET fails to map N on S then
                    Try another S
                if Eval(move) <_lex Eval(best_move) then
                    best_move ← move
    if Eval(best_move) <_lex Eval(mapping) then
        mapping ← best_move
        move_possible ← 1
    else
        move_possible ← 0
```

3.5 Experimental validation

This section presents the experiments conducted to validate the scheduling framework introduced above. We first present the scheduling software developed to run steady-state schedules on the QS 22, then the application graphs in use, and finally report and comment the results.

3.5.1 Scheduling software

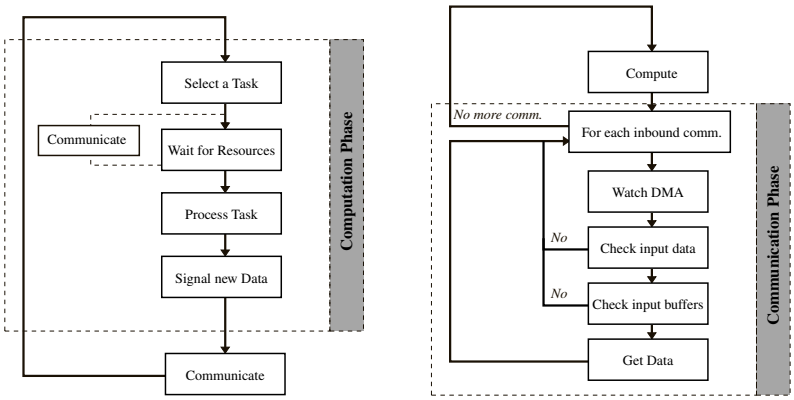
In order to run steady-state schedules on the QS 22, a complex software framework is needed: it has to map tasks on different types of processing elements and to handle all communications. Although there already exists some frameworks dedicated to streaming applications [47, 48], none of them is able to deal with complex task graphs while allowing to statically select the mapping. Thus, we have decided to develop our own framework [1]. Our scheduler requires the description of the task graph, its mapping on the platform, and the code of each task. Even if it was designed to use the mapping returned by the linear program, it can also use any other mapping, such as the ones dictated by the previously described heuristic strategies.

For now, our scheduling framework is able to handle only one PPE. For the following, we thus consider $n_P = 1$ PPE and $n_P = 16$ SPE.

Once all tasks are mapped onto the right processing elements and the process is properly initialized, the scheduling procedure is divided into two main phases: the *computation phase*, when the scheduler selects a task and processes it, and the *communication phase*, when the scheduler performs asynchronous

1. An experimental version of our scheduling framework is available online, at http://graal.ens-lyon.fr/~mjacquel/cell_ss.html

communications. These steps, depicted on Figure 3.8, are executed by every processing element. Moreover, since communications have to be overlapped with computations, our scheduler cyclically alternates between those two phases.



(a) Computation Phase. (b) Communication Phase.

Figure 3.8: Scheduler state machine.

The computation phase, which is shown on Figure 3.8a, begins with the selection of a runnable task according to the provided schedule, and waits for the required resources (input data and output buffers) to be available. If all required resources are available, the selected task is processed, otherwise, it moves to the communication phase. Whenever a new data is produced, the scheduler signals it to every dependent processing elements. Note that in our framework, computation tasks are not implemented as OS tasks but rather as function calls. One computing thread is launched on each processing element (PPE or SPEs). Each thread then internally chooses the next function (or node of the task graph) to run according to the provided schedule. This allows us to avoid fine task scheduling at the OS level.

The communication phase, depicted in Figure 3.8b, aims at performing every incoming communication, most often by issuing DMA calls. Therefore, the scheduler starts watching every previously issued DMA calls in order to unlock the output buffer of the sender as soon as data had been received. Then, the scheduler checks whether there is new incoming data. In that case, and if enough input buffers are available, it issues the proper "Get" command.

Libnuma [65] is used for both thread and memory affinity. Since a PPE can simultaneously handle two threads, the affinity of every management threads is set to the first multi-threading unit, while PPE computing thread's affinity is set to the second multi-threading unit. Therefore, management threads and computing thread runs on different multi-threading units, and do not interfere. Moreover, the data used by a given thread running on a PPE is always allocated on the memory bank associated to that PPE.

To obtain a valid and efficient implementation of this scheduler, we had to overcome several issues due to the very particular nature of the Cell processor. First, the main issue is heterogeneity: the Cell processor is made of two different types of cores, which induces additional challenges for the programmer:

- SPE are 32-bit processors whereas the PPE is a 64-bit architecture;
- Different communication mechanisms have to be used depending on which types of processing elements are implied in the communication. To properly issue our "Get" operations, we made use of three different intrinsics: `mfc_get` for SPE to SPE communications, `spe_mfcio_put` for SPE to PPE communication, and `memcpy` for communication between PPE and main memory.

Another difficulty lies in the large number of variables that we need to statically initialize in each lo-

cal store before starting the processing of the stream: the information on the mapping, the buffer for data transfer, and some control variables such as addresses of all memory blocks used for communications. This initialization phase is again complicated by the different data sizes between 32-bit and 64-bit architectures, and the run-time memory allocation.

All these issues show that the Cell processor is not designed for such a complex and decentralized usage. However, our success in designing a complex scheduling framework proves that it is possible to use such a heterogeneous processor for something else than pure data-parallelism.

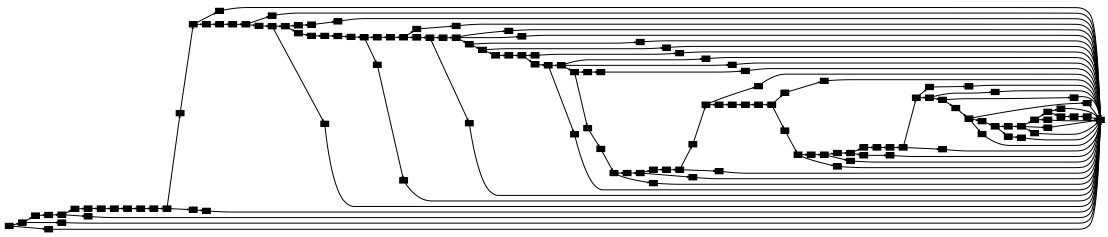
3.5.2 Application scenarios

We test our scheduling framework on 25 random task graphs, obtained with the DagGen generator [92]. This allows us to test our strategy against task graphs with different depths, widths, and branching factors.

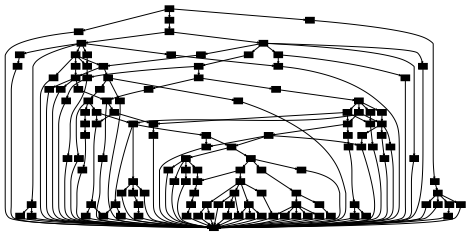
The smallest graph has 20 tasks while the largest has 135 tasks. The smallest graph is a simple chain of tasks, and the largest graph is depicted on Figure 3.9a. On the communication side, the number of edges goes from 19 to 204, the graph with 204 edges is depicted on Figure 3.9b.

We classified the generated graphs into two sets, one for the smaller graphs, having up to 59 tasks, and one for the larger graphs (87 - 135 tasks).

For all graphs, we generated 10 variants with different Communication-to-Computation Ratio (CCR), resulting in 250 different random applications and 230 hours of computation. We define the CCR of a scenario as the total number of transferred elements divided by the total number of operations on these elements. In the experiments, the CCR ranges from 0.001 (computation-intensive scenario) to 0.1 (communication-intensive scenario).



(a) Graph with largest number of tasks



(b) Graph with largest number of edges

Figure 3.9: Largest task graphs used in the experiments

In the experiments, we denote by MIP the scheduling strategy using Mixed Integer Programming to compute an optimal mapping, and described in Section 3.3.3. ILOG CPLEX [29] is used to solve the linear program with rational and integer variables. To reduce the computation time for solving the linear program, we used the ability of CPLEX to stop its computation as soon as its solution is within 5% of the optimal solution. While this significantly reduces the average resolution time, it still offers a very

good solution.

3.5.3 Experimental results

In this section, we present the performance obtained by the scheduling framework presented above. We first focus on the initialization phase of streaming applications using our framework, showing that steady-state operation is reached for a reasonably small length of stream. Then, we compare the throughput obtained by the heuristics introduced in Section 3.4 and the algorithm MIP, both using the model of the QS 22 and using experiments on a real platform. Finally, we discuss the impact of the communication-to-computation ration, and estimate the time required to compute mappings using each strategy. Note that except when we study the influence of the number of SPEs, all the results presented below are computed using all 16 SPEs in the QS 22.

Entering steady-state

First, we show that our scheduling framework succeeds in reaching steady-state, and that the throughput is then similar to the one predicted by the linear program. Figure 3.10 shows the throughput obtained with streams of different sizes: for a short stream containing few data sets, or instances, the start-up cost is important, and the overall throughput is small. However, as soon as the number of instances reaches a few thousands, the obtained throughput is stable. Note that the input data of one instance only consists of a few bytes, so the duration of the transient phase before reaching the steady-state throughput is small compared to the total duration of the stream. In steady state, the experimental throughput achieves 95% of the theoretical throughput predicted by the linear program. The small gap is explained by the overhead of our framework, and the synchronizations induced when communications are performed. The schedules given by other mappings, computed using heuristics, also reaches steady state after a comparable transient phase. We discuss their steady-state throughput below.

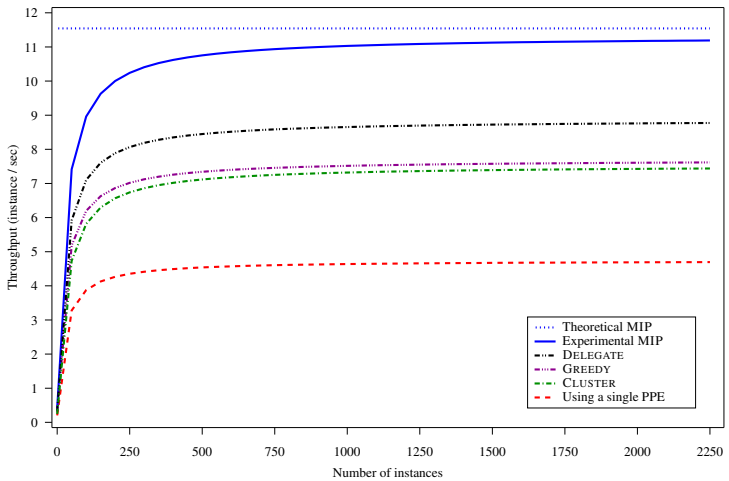


Figure 3.10: Throughput achieved depending on the length of the stream, using the task graph described in Figure 3.9b, with a CCR of 0.004, on the QS 22 using all 16 SPEs.

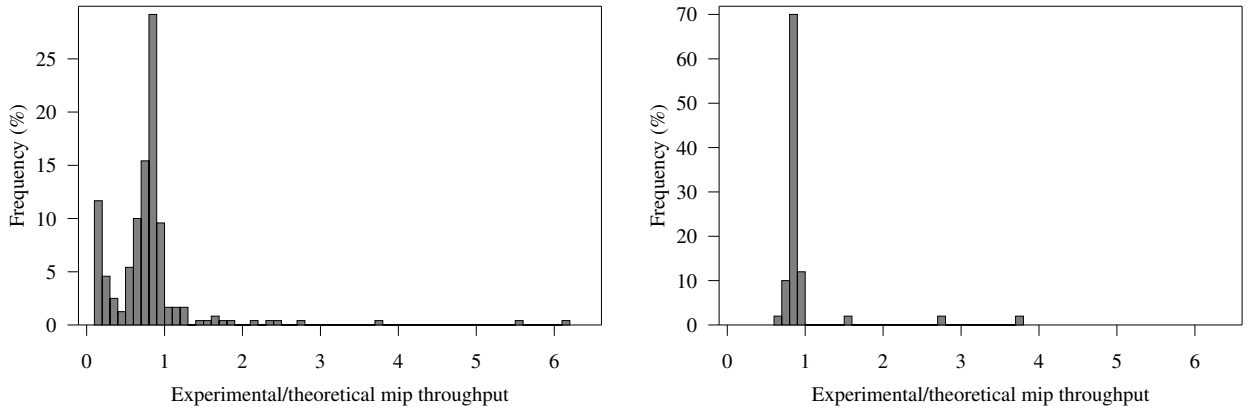
In Table 3.3, we present the number of instances that are required to reach steady state, that is, to reach 99% of the maximum (experimental) throughput, for all possible scenarios. The average value is around 2000 instances, but the mapping computed using MIP reaches steady state faster than the other solutions. We have observed the same behavior on small and large task graphs. This shows that

our scheduling framework is able to reach steady-state operation for any mapping within a reasonable amount of instances compared to the usual length of streaming applications.

Algorithm	Min.	Max.	Average	Std. dev.
GREEDY	300	18,500	2,234	2,589
DELEGATE	250	16,500	2,249	2,381
CLUSTER	300	14,600	2,214	2,586
MIP	300	14,500	2,050	2,182

Table 3.3: Number of instances required to reach steady state for all scenarios.

Figure 3.11a presents the distribution of the ratio $\rho_{\text{exp}}^{\text{MIP}}/\rho_{\text{th}}^{\text{MIP}}$, where $\rho_{\text{exp}}^{\text{MIP}}$ is the experimental throughput of MIP, and $\rho_{\text{th}}^{\text{MIP}}$ is the theoretical throughput (the one predicted by the linear program). The average ratio is 0.91, but we can observe that the experimental throughput is sometimes very different from the theoretical one (as less as 20% or as much as 597%). Cases when the experimental throughput is 5 times smaller than the predicted one mainly happen for small task graphs, when the very few tasks are scheduled on each processing element, which leads to a high synchronization cost. Figure 3.11b shows the results for large task graphs, when the accuracy of the predicted throughput is larger (the average ratio is 1.10). There still remain some cases with a high ratio (the experimental throughput is larger than the predicted throughput). This corresponds to communication-intensive scenarios: in a few cases, our communication model is very pessimistic for communications and overestimates bandwidth contention.



(a) All task graphs (b) Large task graphs

Figure 3.11: Distribution of the ratio between experimental and theoretical throughputs for the MIP strategy.

Theoretical comparison of heuristics with MIP

We first start by comparing the expected throughput of the heuristics described in Section 3.4 with the theoretical throughput of MIP, detailed in Section 3.3.3. This allows to measure the quality of the mapping produced by heuristics without the particularities of the scheduling framework, and the inaccuracies of the QS 22 model.

For this purpose, we first compute the theoretical throughput of all mappings produced by the heuristics. A mapping is first described using the α and β variables from the linear program presented in

Section 3.3.3. The expected period of the mapping can then be computed using Constraints (3.14) and (3.15) (for computations) and Constraints (3.1), (3.2), (3.3), (3.4), (3.5), (3.6), (3.7), (3.8), and (3.9) (for communications).

This theoretical comparison also helps to assess the limitation of the constraints on DMA calls. In the design of the MIP strategy, the limited number of simultaneous DMA calls for PPEs and SPEs is taken into account with two Constraints (Equations (3.17) and (3.18)). This limitation is not taken into account while designing heuristics, because we want to keep their design simple. Our scheduling framework assigns DMA calls dynamically to communication requests, and is able to deal with any number of communications. Of course, when the number of simultaneous communications exceeds the bounds, communications are delayed, thus impacting the throughput. The comparison of theoretical throughput of mappings allows us to measure the limitation on the expected throughput induced by these constraints, which only exist for MIP.

We hence compute the ratio $\rho_{\text{th}}^{\text{H}}/\rho_{\text{th}}^{\text{MIP}}$, where ρ_{th}^{H} is the predicted throughput of any heuristic H, and $\rho_{\text{th}}^{\text{MIP}}$ is the predicted throughput of MIP. Detailed results are given in Table 3.4.

All task graphs					
Algorithm	*Min.*	*Max.*	*Average*	*Std. dev.*	*Best cases*
GREEDY	0.55	1.49	0.82	0.12	6%
DELEGATE	0.75	1.05	0.97	0.07	60%
CLUSTER	0.00	2.46	0.32	0.28	1%
MIP	1.00	1.00	1.00	0.00	80%

Small task graphs					
Algorithm	*Min.*	*Max.*	*Average*	*Std. dev.*	*Best cases*
GREEDY	0.55	1.49	0.82	0.13	8%
DELEGATE	0.87	1.05	0.99	0.04	71%
CLUSTER	0.05	2.46	0.32	0.26	1%
MIP	1.00	1.00	1.00	0.00	78%

Large task graphs					
Algorithm	*Min.*	*Max.*	*Average*	*Std. dev.*	*Best cases*
GREEDY	0.63	0.98	0.79	0.10	0%
DELEGATE	0.75	1.04	0.91	0.09	27%
CLUSTER	0.00	0.81	0.33	0.33	0%
MIP	1.00	1.00	1.00	0.00	88%

Table 3.4: Predicted throughput of heuristics normalized to the predicted throughput of MIP

These results show that in some cases, the limitation on the number of DMA calls has an impact on the throughput: there are cases when the CLUSTER strategy finds a mapping with an expected throughput twice the one of MIP. The MIP strategy is able to reach the best throughput among all strategies in only 80% of the scenarios, due to this limitation. However, MIP still gives a better average throughput than all other strategies. Thus, this limitation has a little impact on performance.

Among heuristics, the DELEGATE strategy ranks first: it is almost within 25% of the performance of MIP, and on average gives the same results, and gives the best performance on 60% of the scenarios. (Note that several heuristics may give the best throughput for a scenario, hence leading to a sum larger than 100%). Quite surprisingly, the CLUSTER strategy gives very poor results: on average its throughput

is only one third of the MIP throughput, whereas the simple GREEDY strategy performs better, with an average throughput around 80% of the MIP throughput. All heuristics perform better on small task graphs, and have more difficulties to tackle the higher complexity of large task graphs.

The bad performance of CLUSTER is surprising because this heuristics takes communication into account, as opposed to GREEDY. However, the communications are considered only when building clusters in the first phase. In the second phase, when clusters are mapped on the resources, communications are totally neglected. However, all communications do not have the same impact, because the communication graph has heterogeneous links. Moreover, since clusters have been made to cancel out the impact of large communications, the load balancing procedure in the second phase has less freedom, and is unable to reach a well balanced mapping. This explains why the mapping produced by CLUSTER are usually worse than the one given by DELEGATE or GREEDY.

Experimental comparison of heuristics with MIP

We move to the experimental comparison of heuristics and MIP, when all mappings are scheduled with our scheduling software on the real QS 22 platform. For the sake of comparison, all throughputs are normalized by the throughput of MIP, and we present in Table 3.5 the ratio $\rho^{H}_{exp}/\rho^{MIP}_{exp}$, where ρ^{H}_{exp} is the experimental throughput of any heuristic H, and ρ^{MIP}_{exp} is the experimental throughput of MIP.

All task graphs					
Algorithm	*Min.*	*Max.*	*Average*	*Std. dev.*	*Best cases*
GREEDY	0.32	3.28	0.91	0.39	8%
DELEGATE	0.75	2.63	1.06	0.27	48%
CLUSTER	0.00	1.80	0.41	0.28	0%
MIP	1.00	1.00	1.00	0.00	54%

Small task graphs					
Algorithm	*Min.*	*Max.*	*Average*	*Std. dev.*	*Best cases*
GREEDY	0.32	3.28	0.93	0.43	11%
DELEGATE	0.78	2.63	1.10	0.28	56%
CLUSTER	0.08	1.80	0.44	0.26	0%
MIP	1.00	1.00	1.00	0.00	46%

Large task graphs					
Algorithm	*Min.*	*Max.*	*Average*	*Std. dev.*	*Best cases*
GREEDY	0.64	1.73	0.85	0.21	0%
DELEGATE	0.75	1.89	0.95	0.20	23%
CLUSTER	0.00	0.82	0.33	0.34	0%
MIP	1.00	1.00	1.00	0.00	80%

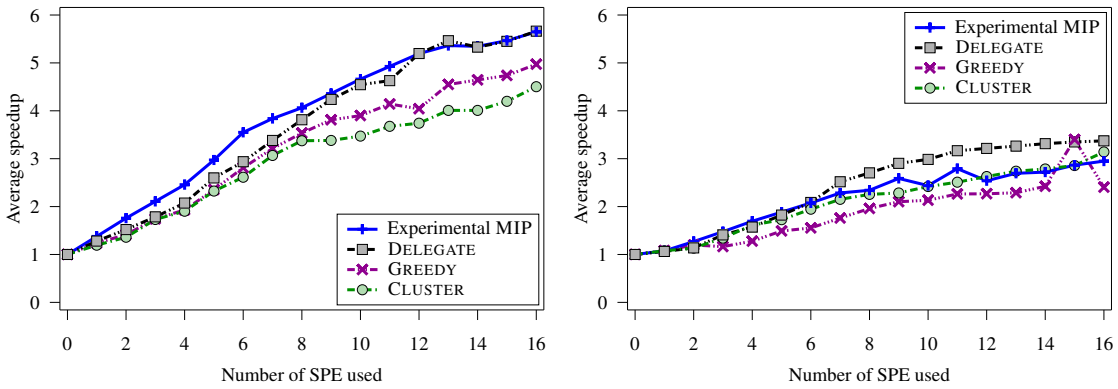
Table 3.5: Experimental throughput of heuristics normalized to the experimental throughput of MIP

The comparison between heuristics and MIP is close to the one with expected throughputs. However, the gap between MIP and each heuristic is smaller: heuristics perform somewhat better on the real platform that on the model. The average throughput of DELEGATE is very close to MIP's throughput (larger for small task graphs, but smaller for large task graphs), and GREEDY gets around 90% of the MIP's throughput. Once again, CLUSTER gives a very limited average throughput, with only 41% of the

MIP's throughput. On large task graphs, MIP is above all heuristics, and reaches the best throughput in 80% of the cases. However, on average, DELEGATE is able to achieve 95% of the MIP's throughput.

Scaling and influence of the communication-to-computation ratio

In this section, we present the speed-up obtained by MIP and heuristics. We also study the influence of the Communication-to-Computation Ratio (CCR), both on the duration of the transient phase before reaching steady state, and on the accuracy of the model.



(a) With CCR=0.001 (b) With CCR=0.1

Figure 3.12: Average speed-up vs. number of SPEs in use.

We first present the performance obtained by all strategies, with different number of SPEs. In the MIP solution, the number of SPEs is a parameter, which may vary between 0 and 16. For all heuristics developed above, it is quite straightforward to adapt them to deal with a variable number of SPEs. Figure 3.12 present the average speed-up of each heuristic obtained for different number of SPEs. We observe that all heuristics are able to take advantage of an increasing number of SPEs. When the CCR is high, the resulting speed-up is smaller than when the CCR is low, because large communications make it more difficult to distribute tasks among processing elements. This is also outlined by Figure 3.13b, which presents the variation of the average speed-up of heuristics (for 16 SPEs) with the CCR. For a CCR smaller than 0.02, the average speed-up is almost constant, while it decreases for a larger CCR. We can observe that DELEGATE is a little less sensitive to the increase of the CCR than MIP. For some strategies, the speed-up slightly increases when the CCR increases from 0.001 to 0.02. This is because to increase the CCR, we first decrease the computation amount of tasks, which makes the overhead of the scheduling software (launching tasks, checking buffers, etc.) more visible when all tasks are scheduled on the PPE.

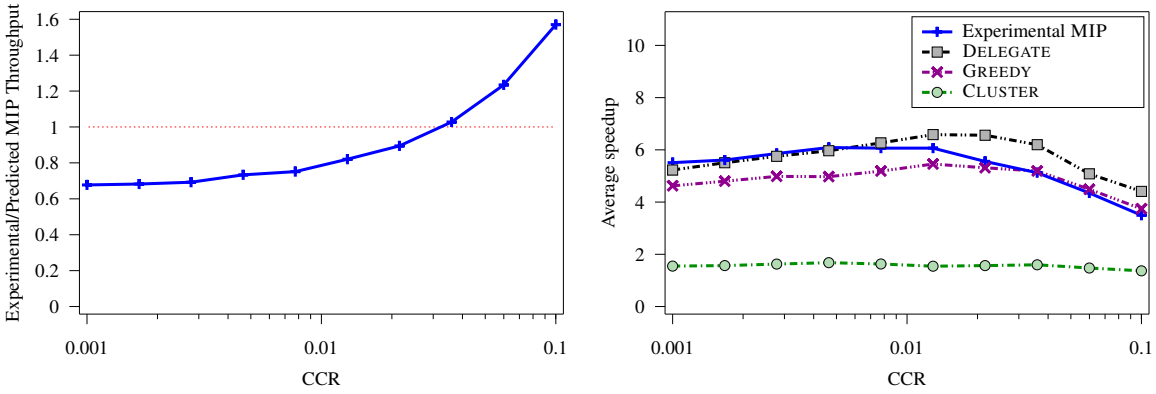
All task graphs, CCR of 0.001				
Algorithm	*Min.*	*Max.*	*Average*	*Std. dev.*
GREEDY	300	400	367	38
DELEGATE	300	400	367	42
CLUSTER	300	400	361	40
MIP	300	400	375	35

All task graphs, CCR of 0.1				
Algorithm	*Min.*	*Max.*	*Average*	*Std. dev.*
GREEDY	200	4750	788	974
DELEGATE	200	4200	750	850
CLUSTER	150	2950	702	681
MIP	150	3200	646	670

Table 3.6: Number of instances required to reach steady state

All mapping heuristics produce a periodic schedule, which needs to be initialized: the first data

of the stream have to be processed specifically before the periodic schedule can be started. Table 3.6 presents the number of instances that are processed in this transient phase, for the two extreme values of CCR: 0.001 (computation-intensive scenario) and 0.1 (communication-intensive scenario). We observe that the larger the CCR, the longer the duration of the transient phase: when communications are predominant, it is more difficult to reach a steady state phase even if communications are well overlapped with computations.



(a) Model accuracy vs. CCR

(b) Average speed-up vs. CCR

Figure 3.13: Evolution of the model accuracy and the speedup with the CCR.

Figure 3.13a presents the evolution of the ratio between the experimental throughput of MIP over its predicted throughput in function of the CCR. We notice that for all CCR smaller than 0.05, our model is slightly optimistic, and predicts a throughput at most 30% larger than the one obtained in the experiments. For larger CCRs, corresponding to communication-intensive scenarios, the model is pessimistic and larger throughputs are obtained in the experiments. Our model is thus best suited for average values of the CCR. In case of extreme values, it is necessary to modify the model, either by taking into account synchronization costs, when CCR is very small, or by better modeling contention between communications, in case of a very large CCR.

Time required to compute schedules

When considering streaming application, it is usually assumed that streams have very long durations. This is why it is worth spending some time to optimize the schedule before the real data processing. Yet, since we are using mixed linear programming, we report the time needed to compute the schedules, to prove that this is done in reasonable time.

Table 3.7 presents the time required to compute mappings using each algorithm. On average, the MIP strategy requires twice as much time than all heuristics, and runs for less than two minutes. In some cases, corresponding to large graphs, the run time of MIP can reach 12 minutes, whereas DELEGATE need at most 3 minutes. These running, although large, are very reasonable in the context of streaming applications that are supposed to run for several minutes or hours.

3.6 Related Work

Streaming applications. Scheduling streaming applications is the subject of a vast literature. Several models and frameworks had been introduced like StreamIt [91], Brook [30] or Data Cutter Lite [16].

Algorithm	*Min.*	*Max.*	*Average*
GREEDY	2.7	189	39
CLUSTER	2.6	190	39
DELEGATE	2.9	205	48
MIP	3.6	706	80

Table 3.7: Scheduling time in seconds

However, our motivation for this work was rather to extract a simple and yet relevant model for the Cell processor, so as to be able to provide a way to compute interesting mapping for pipeline applications. To the best of our knowledge, this has never been considered for the Cell processor, nor for another multicore architecture that exhibits a similar heterogeneity. Note that the small scale of the target architecture makes the problem very particular, and that it was not clear that such complex mapping techniques could result in a feasible and competitive solution for streaming applications. To the best of our knowledge, none of the existing frameworks which target the Cell processor make it possible to implement our own static scheduling strategies. The main two limitations of existing frameworks are the restriction to simple pipeline applications (chain task graphs), whereas we target any DAG, and the fact that only SPEs are used for computation, the PPE being restricted to control, whereas we take advantage of the heterogeneity.

It is worth mentioning that the problem studied in this chapter is close to some research in the Digital Signal Processing (DSP) community. For example, Hoang et al. consider the problem of scheduling a flow graph on such a parallel DSP platform [52]. The advance of multicore DSP [62] makes the problems even closer. However, in DSP literature, it is usually assumed that the program to be implemented on the platform is at hand, and can be modified when needed: code can be rewritten to split tasks, etc., whereas we consider a fixed task graph. Moreover, DSP literature is often concerned with other constraints like energy consumption and heat dissipation for mobile devices or real-time constraints.

Task graph scheduling. When scheduling tasks which have dependencies between them, it is natural to model the resulting problem as a Directed Acyclic Graph (DAG). There are a lot of DAG scheduling studies that are related to the present chapter, especially the ones targeting a heterogeneous environment. The most used techniques are list scheduling [25], clustering [76], and task duplication [4]. However, these studies usually consider a single DAG, and try to minimize the completion time of this DAG, whereas we consider a pipelined version of the problem, when several instances of a common DAG have to be processed.

Memory access and computation overlap. Hiding memory accesses latencies is a crucial need when considering the Cell processor. In [44], the authors introduce an improvement to the Decoupled Threaded Architecture [43] consisting in a prefetch mechanism. They propose an implementation of this mechanism on the Cell processor through the use of DMA calls and extensions to compiler instructions set, which interest is demonstrated through simulation (no real implementation seems to be available). In our scheduler, we propose a real implementation of a similar prefetch mechanism.

3.7 Conclusion

In this chapter, we have studied the scheduling of streaming applications on a heterogeneous platform: the IBM Bladecenter QS 22, made of two heterogeneous multicore Cell processors. The first

challenge was to come up with a realistic and yet tractable model of the Cell processor. We have designed such a model, and we have used it to express the optimization problem of finding a mapping with maximal throughput. This problem has been proven NP-complete, and we have designed a formulation of the problem as a mixed linear program. By solving this linear program with appropriate tools, we can compute a mapping with optimal throughput. We have also proposed a set of scheduling heuristics to avoid the complexity of mixed linear programing.

In a second step, we have implemented a complete scheduling framework to deploy streaming applications on the QS 22. This framework, available for public use, allows the user to deploy any streaming application, described by a potentially complex task graph, on a QS 22 platform or single Cell processor, given any mapping of the application to the platform. Thanks to this scheduling framework, we have been able to perform a comprehensive experimental study of all our scheduling strategies. We have shown that our MIP strategy usually reaches 90% of the throughput predicted by the linear program, that it has a good and scalable speed-up when using up to 16 SPEs. Some of the proposed heuristics, in particular DELEGATE, are also able to reach very good performance. When the task graph to schedule is large, MIP is the one which offers the best performance, but DELEGATE achieves 95% of its throughput on average. We have shown that considering load-balancing among processing elements, and carefully estimating communications between the different components of this bi-Cell platform, is the key to performance.

Overall, this demonstrates that scheduling a complex application on a heterogeneous multicore processor is a challenging task, but that scheduling tools can help to achieve good performance.

Chapter 4

On optimal tree traversals for sparse matrix factorization

4.1 Introduction

As already said in chapters 1 and 3, communication and memory architectures highly impact performance, and must be considered in order to offer high levels of performance. This is especially true for scientific applications. Moreover, on most hardware platforms, the available memory or storage space determines the size of the problems that could be solved by such applications. Therefore, for a given application, it is crucial to minimize the amount of required space.

Generally, applications are modeled by complex data structures, like Directed Acyclic Graphs (or DAGs). However, the problem of scheduling a general DAG so as to minimize the amount of required memory has been shown to be NP-hard in [88]. We will rather focus on applications modeled by rooted trees, which are often encountered in practice, for instance in the multifrontal method of sparse matrix factorization.

In this chapter, we consider the following memory-aware traversal problem for rooted trees. The nodes of the tree correspond to tasks, and the edges correspond to the dependencies among the tasks. The dependencies are in the form of input and output files: each node accepts a large file as input, and produces a set of large files, each of them to be accepted by a different child node. We are to execute such a set of tasks on a two-level memory system. The execution scheme corresponds to a traversal of the tree where visiting a node translates into reading the associated input file and producing output files. How one can traverse the tree so as to optimize memory usage? For convenience we refer to the two-levels of storage as the *main memory* and the *secondary memory*, and also as *in-core* and *out-of-core*. Many combinations such as cache and RAM, or RAM and disk, or even disk and tape, lead to the same association of a faster but smaller storage device together with a larger but slower device. The difficulty remains the same for all combinations: find an execution scheme that makes the best use of the *main memory*, and minimizes accesses to the *secondary memory*.

Throughout the chapter, we consider *out-trees* where a task can be executed only if its parent has already been executed. However, we show in Section 4.3 that all results equivalently apply to *in-trees*, where tasks are processed from the leaves up to the root. Each task (or node) i in the tree is characterized by the size f_i of its input file (data needed before the execution and received from its parent), and by the size n_i of its execution file.

During execution, non-leaf nodes generate several output files, one for each child, which can have different sizes. A task can be processed only in-core; its execution is feasible only if all its files (input, output, and execution) fit in currently available memory. More formally, let M be the size of the main

memory, and S the set of files stored in this memory when the scheduler decides to execute task i. Note that S must contain the input file of task i. The processing of task i is possible if we have:

$$MemReq(i) = f_i + n_i + \sum_{j \in Children(i)} f_j \leq M - \sum_{j \in S, j \neq i} f_j \tag{4.1}$$

where $MemReq(i)$ denotes the memory requirement of task i (and $Children(i)$ its child nodes in the tree). Once i has been executed, its input file and execution file can be discarded, and replaced by other files in main memory; the output files can either be kept in main memory, in order to execute some child of the task, or they can temporarily be stored into secondary memory (and retrieved later when the scheduler decides to execute the corresponding child of i). The volume of accesses (reads or writes) to secondary memory is referred to as the *I/O volume*.

Clearly, the traversal, i.e., the order chosen to execute the tasks, plays a key role in determining which amount of main memory and I/O volume are needed for a successful execution of the whole tree. More precisely, there are two main problems which the scheduler must address:

MINMEMORY Determine the minimum amount of main memory that is required to execute the tree without any access to secondary memory.

MINIO Given the size M of the main memory, determine the minimum I/O volume that is required to execute the tree.

Obviously, a necessary condition for the execution to be successful is that the size M of main memory exceeds the largest memory requirement over all tasks:

$$\max_i MemReq(i) \leq M$$

However, this condition is not sufficient, and a much larger main memory size may be needed for the MINMEMORY problem.

The main motivation for this work comes from numerical linear algebra. Tree workflows (assembly or elimination trees) arise during the factorization of sparse matrices, and the huge size of the files involved makes it absolutely necessary to reduce the memory requirement of the factorization. The trees arising in this context are in-trees (as said before, and as we will discuss later, there is no difference between in-trees and out-trees). We build upon two key results from the literature [68, 69]. Liu [68] discusses how to find the best traversal for the MINMEMORY problem when the traversal is required to correspond to a postorder traversal of the tree. In the follow-up study [69], an exact algorithm is proposed to solve the MINMEMORY problem, without the postorder constraint on the traversal.

In this chapter, we propose a new exact algorithm called $MinMem$ for the MINMEMORY problem. The $MinMem$ algorithm is based upon a novel approach that systematically explores the tree with a given amount of memory. This approach is quite different from the techniques used in [69]. Although the worst-case complexity of the proposed $MinMem$ algorithm is the same as that of Liu's, i.e., quadratic in the number of nodes in the tree, it turns out that it is much more efficient in practice, as demonstrated by our experiments with elimination trees arising in sparse matrix factorization (see Section 4.6 for details). We also compare $MinMem$ and Liu's algorithms with the best postorder traversal (common in sparse matrix factorization packages), in terms of both quality (memory needed) and execution time. We report that the best postorder traversals result in only a little additional memory requirement than the optimal one in practice, which is good news for the current sparse matrix factorization libraries. However, we show that there exist trees where postorder based traversals require arbitrarily larger amounts of main memory than the optimal one.

As for the MINIO problem, we show that it is NP-hard, both for postorder based and for arbitrary traversals, even for simple harpoon graphs, while it is polynomial for arbitrary trees with unit-size files

(in which case MINIO reduces to the *I/O pebble game* introduced by Hong and Kung [53]). This shows that introducing files of different sizes does add a level of difficulty in memory-aware scheduling of tree workflows. We provide a set of heuristics to solve the MINIO problem. Our heuristics use various greedy criteria to select the next node to be scheduled, and those files to be temporarily written to secondary memory. All these heuristics are evaluated using assembly trees arising in sparse matrix factorization methods.

The chapter is organized as follows. We start with an overview of related work in Section 4.2. Then we describe the framework in Section 4.3. The next three sections constitute the heart of this chapter. We deal with the MINMEMORY problem in Section 4.4, presenting complexity results for postorder traversals and proposing the exact $MinMem$ algorithm. Then we consider the MINIO problem in Section 4.5, assessing the NP-hardness of this problem, and designing heuristics. The experimental evaluation of all MINMEMORY algorithms and MINIO heuristics is conducted in Section 4.6. Finally we provide some concluding remarks and hints for future work in Section 4.7.

4.2 Background and Related Work

4.2.1 Elimination tree and the multifrontal method

As mentioned above, determining a memory-efficient tree traversal is very important in sparse numerical linear algebra. The elimination tree is a graph theoretical model that represents the storage requirements, and computational dependencies and requirements, in the Cholesky and LU factorization of sparse matrices. Here we give a brief description of such trees; we refer to [87] for the first formalization of elimination trees, and to [70] for an excellent survey on the subject.

There are at least two interpretations of elimination trees [70]. Among those, the one describing the dependencies of numerical values among the columns of the Cholesky factor serves well for our purposes in this chapter. Assume that A is an $n \times n$ sparse, symmetric, positive definite matrix with a lower triangular Cholesky factor L such that $A = LL^T$. It is known that for $i > j$, the numerical values of column i of L depend on column j of L if and only if $\ell_{ij} \neq 0$. Consider building a directed graph on n vertices with edges representing the column dependencies, i.e., we add an edge from the vertex v_j to the vertex v_i whenever the column i of L depends on the column j. The transitive reduction (if there is a directed path of length at least two from v_j to v_i, then the edge (v_j, v_i) is discarded) of this graph yields the elimination tree. Given such a model, it is clear that the column i of L can only be computed after all the columns corresponding to the children of v_i in the elimination tree.

The multifrontal method of sparse matrix factorization [36, 71] organizes the computations of sparse factorizations as a sequence of dense matrix operations using the elimination tree. The method associates a block 2×2 matrix with each node of the elimination tree—the block matrix contains a diagonal element and the nonzeros in the corresponding row and column of the matrix currently being eliminated. The $(1, 1)$-block of a node can be eliminated (it is called *fully summed*) only if all the updates to the corresponding diagonal entry have been computed. The Schur complement formed by the elimination of the fully summed variable on the $(2, 2)$-block of a node cannot be eliminated until later in the factorization. This Schur complement is called the contribution block, and it is passed to the father node for the *assembly* operation. Therefore the operations that are at the heart of the multifrontal method are as follows. The first one is to assembly the contribution blocks from the children nodes, and the original entries from the matrix (if we are at a leaf, there is no contribution block); the second one is to eliminate the fully summed variable; and the third one is to compute and send the contribution block to the father. This leads to an in-tree where the computations proceed from the leaves to the root.

Since the elimination tree is defined with one variable (row/column) per node, it only allows one

elimination per node and the (1,1) block would be of order one. Therefore, there would be insufficient computation at a node for efficient implementation. It is thus advantageous to combine or amalgamate nodes of the elimination tree. The amalgamation can be restricted so that two nodes of the elimination tree are amalgamated only if the corresponding columns of the L factor have the same structure below the diagonal [36]. As even this technique may not give a large enough (1,1) block, a threshold based amalgamation strategy can be used in which the columns to be amalgamated are allowed to have discrepancies in their patterns [11]. The resulting tree is often called the assembly tree.

4.2.2 Pebble game and its variants

On the more theoretical side, this work builds upon the many papers that have addressed the pebble game and its variants. The MINMEMORY problem amounts to revisiting the I/O pebble game with pebbles of arbitrary sizes that must be loaded into main memory before *firing* (executing) the task. The pioneering work of Sethi and Ullman [89] deals with a variant of the pebble game that translates into the simplest instance of MINMEMORY, with $f_i = 1$ and $n_i = 0$ for any task i. The concern in [89] was to minimize the number of registers that must be used while computing an arithmetic expression. The problem of determining whether a general DAG can be executed with a given number of pebbles has been shown NP-hard by Sethi [88] if no vertex is pebbled more than once (the general problem allowing recomputation, that is, re-pebbling a vertex which have been pebbled before, has been proven PSPACE complete by Gilbert, Lengauer and Tarjan [41]). However, this problem has a polynomial complexity for tree-shaped graphs [89].

A variant of the game with two levels of storage has been introduced by Hong and Kung [53] under the name of *I/O pebble game*, which was used to derive lower bounds on I/O operations and study the trade-off between I/O operation and main memory size for particular graphs. A comprehensive summary of results for pebble games can be found in the book by Savage [85].

In [89], the algorithm proposed by Sethi and Ullman for processing tree-shaped graphs and minimizing the number of allocated registers also has a minimum number of *store* instructions, which makes it optimal both for memory and for I/O minimization. It is quite interesting to see that the classical pebble game problem with trees remains polynomial with files of arbitrary sizes instead of pebbles (this is the MINMEMORY problem,) while the I/O pebble game becomes NP-hard (see Theorem 4.2).

On the application side, there are many variants of MINMEMORY, some of which being discussed in Section 4.3.3. The execution model summarized by Equation (4.1) applies to a large variety of scenarios, including divide-and-conquer algorithms. For high-degree trees, simultaneously loading all children files into main memory may be a bottleneck requirement. While some applications could allow for processing the children one after the other, like in map-reduce problems, other scenarios call for generating all children data concurrently. Along the same line, a relaxation of the MINIO problem would allow to write *fractions* of files into secondary memory, leading to a *divisible* version of the problem. Again, while this may make sense in some cases (e.g., when the main memory is naturally divided into small pages, and if it is possible to unload some pages containing fractions of files), it is not always possible (e.g., when the main memory is a complex file system).

4.3 Models and Problems

4.3.1 Application model

The tree workflow $\mathcal{T}$ is composed of p nodes, or tasks, numbered from 1 to p. Nodes in the tree have an input file, an execution file (or program), and several output files (one per child). More precisely:

- Each node i has an input file of size f_i. If i is not the root, its input file is produced by its parent $parent(i)$; if i is the root, its input file can be of size zero, or contain input from the outside world.
- Each node i in the tree has an execution file of size n_i.
- Each non-leaf node i in the tree, when executed, produces a file of size f_j for each $j \in Children(i)$. Here $Children(i)$ denotes the set of the children of i. If i is a leaf-node, then $Children(i) = \emptyset$ and i produces a file of null size: we then consider that the terminal data produced by leaves are directly written to the secondary memory or sent to the outside world, independently from the I/O mechanism.

The memory requirement $MemReq(i)$ of node i is the total amount of main memory that is needed to execute node i, as underlined in Equation (4.1). After i has been processed, its input file and program can be discarded, while its output files can either be kept in main memory (to process the children of i) or be stored in secondary memory temporarily.

Algorithm 15: Checking an in-core traversal.

Input: tree $\mathcal{T}$ with p nodes, available memory M, ordering σ of the nodes
Output: whether the traversal is feasible
$Ready \leftarrow \{root\}$
$M_{avail} \leftarrow M - f_{root}$
for $step = 1$ *to* p **do**
  Let i be the task such that $\sigma(i) = step$
  if $i \notin Ready$, *or* $MemReq(i) > M_{avail} + f_i$ **then**
    return FAILURE
  $M_{avail} \leftarrow M_{avail} + f_i - \sum_{j \in Children(i)} f_j$
  $Ready \leftarrow Ready \setminus \{i\} \cup \bigcup_{j \in Children(i)} \{j\}$
return SUCCESS

4.3.2 In-core traversals and the MINMEMORY problem

For the MINMEMORY problem, we are given a tree $\mathcal{T}$ with p nodes and an initial amount of memory M. A traversal is an ordering of the p nodes that specifies at which step they are executed. A traversal must obey precedence constraints (a node is always scheduled after its parent) and must never exceed the available memory. Algorithm 15 checks if a given traversal is feasible: it computes the memory M_{avail} that is available at each step, which corresponds to the original memory M minus the size of the files of ready nodes (nodes which are not executed yet, but whose parents have been processed). A formal definition of a traversal is given below.

Definition 4.1 (INCORETRAVERSAL)**.** *Given a tree $\mathcal{T}$ and an amount M of available memory, the problem* INCORETRAVERSAL$(\mathcal{T}, M)$ *consists in finding a feasible in-core traversal σ described by a permutation of the nodes of a tree $\mathcal{T}$ such that:*

$$\forall i \neq root, \quad \sigma(parent(i)) < \sigma(i) \tag{4.2}$$

$$\forall i, \quad \sum_{\sigma(j)<\sigma(i)} \left(\sum_{k \in Children(j)} (f_k) - f_j \right) + n_i + \sum_{k \in Children(i)} f_k \leq M \tag{4.3}$$

In this definition, Equation (4.2) accounts for precedence constraints and Equation (4.3) deals with memory constraints. A postorder traversal is a traversal where nodes are visited according to some top-down postorder ordering of the tree nodes. Hence, in a postorder traversal, after processing a vertex i, the whole subtree rooted in i is completely processed.

Definition 4.2 (MINMEMORY)**.** *Given a tree $\mathcal{T}$, determine the minimum amount of memory M such that* INCORETRAVERSAL$(\mathcal{T}, M)$ *has a solution.* MINMEMORY-POSTORDER *is the same problem restricted to postorder traversals.*

4.3.3 Model variants

In this section, we discuss three variants of the model.

Bottom-up traversals for in-trees Let $\mathcal{T}$ be an in-tree with p nodes and M the amount of main memory. As the tasks have to be executed from the leaves to the root, a task now has many input files and a single output file. We do not change the notations and assume that the output file has size f_i (*to* the parent, instead of *from* the parent in an out-tree), and the input files have the size f_j for each child j of i. A valid traversal of such an in-tree should respect the order of the tasks (from the leaves to the root) and should satisfy Equation (4.1) for each task i. Suppose $\sigma(\mathcal{T}, p, M)$ is a valid traversal of the in-tree $\mathcal{T}$. Then $\widetilde{\sigma}(\widetilde{\mathcal{T}}, p, M)$ is a valid traversal of the out-tree $\widetilde{\mathcal{T}}$ where $\widetilde{\sigma}$ denotes the reverse permutation of σ, defined as $\widetilde{\sigma}(i) = p - \sigma(i) + 1$ for all i. This is easy to verify as the reverse permutation guarantees the order of the tasks for $\widetilde{\mathcal{T}}$, and the memory constraint is satisfied for any task. The relation between a valid traversal of an in-tree $\mathcal{T}$ and the inverted traversal of the out-tree $\widetilde{\mathcal{T}}$ holds for the other way round too.

Model with replacement In some variants of the pebble game, the player is allowed to move a pebble from one pebbled node to an unpebbled node. Extending the game to pebbles with non-unit costs, this amounts to the variant of the model where the memory occupied by the input file of node i (of size f_i) is *replaced* by the memory occupied by the output files of node i (of size $\sum_{j \in Children(i)} f_j$). The amount of memory needed to process node i is $\max(f_i, \sum_{j \in Children(i)} f_j)$ (note that in the pebble game, there is no cost n_i). This variant can be simulated by our model as follows: given an instance of the problem with the replacement policy, we add a negative weight $n_i = -\min(f_i, \sum_{j \in Children(i)} f_j)$ to each node i (an example is given in Figure 4.1).

Liu's model In [69], the author introduces a bottom-up framework modelling sparse matrix LU factorization. In this framework, the tree T modelling the application is modified as follows: each original node x of T is expanded in two nodes x^+ and x^-. Here x^+ represents x during the processing of a column, x^- being x after its processing. Note that in this model, parameter f_x is not used.

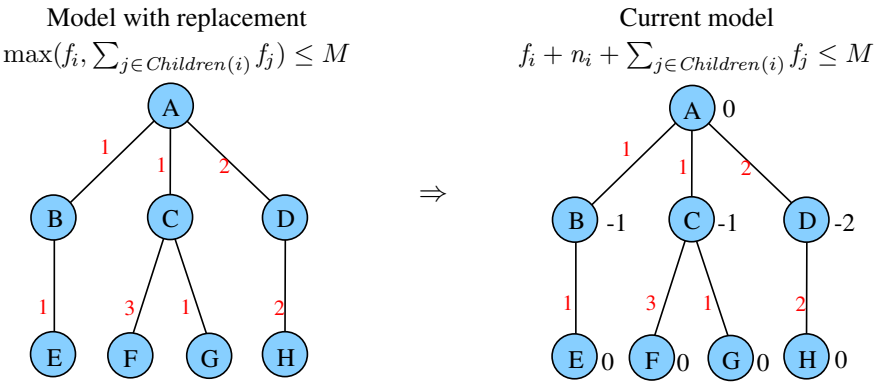


Figure 4.1: Transformation from the model with replacement.

The cost n_{x^+} associated to node x^+ represents the number of nonzeros in columns of the matrix L from the subtree of T rooted in x, that are required during the processing of the column x in the factorization: in other words the memory peak associated to node x. Conversely, the cost n_{x^-} is the number of nonzeros in columns of matrix L associated with the subtree of T rooted in x that are still required after the processing of x, which is the storage requirement of the subtree of T rooted in x.

This variant can be simulated in our framework by merging back each pair of nodes (i^+,i^-) into node i, with an input file of size $f_i = n_{i^-}$ and an extra memory cost during processing $n_i = n_{i^+} - n_{i^-} - \sum_{j^- \in Children(i^+)} n_{j^-}$. An example is given in Figure 4.2.

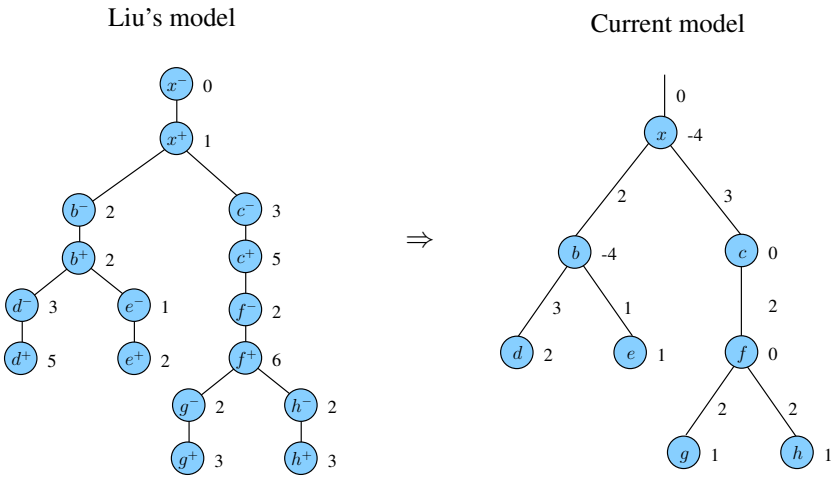


Figure 4.2: Reduction for Liu's model.

4.3.4 Out-of-core traversals and the MINIO problem

Out-of-core processing enables solving large problems, when the size of the data cannot fit into the main memory. In this case, some temporary data are copied into the secondary memory, and unloaded from the main memory, so as to leave room for other computations. Since secondary memory has a smaller access rate, the usual objective is to limit the volume of I/O operations.

Defining traversals that perform I/O operations is more complicated than defining in-core traversals: in addition to determining the ordering of the nodes (the permutation σ), at each step we have to identify

Algorithm 16: Checking an out-of-core traversal.

Input: tree $\mathcal{T}$ with p nodes, available memory M, ordering σ of the nodes, ordering τ of the output transfers to secondary memory
Output: whether the traversal is feasible, and the amount of I/O

$Ready \leftarrow \{root\}$
$M_{avail} \leftarrow M - f_{root}$
$IO \leftarrow 0$
$Written \leftarrow \emptyset$
for $step = 1$ *to* p **do**
  foreach i *such that* $\tau(i) = step$ **do**
    if $\sigma(i) \geq step$ **then**
      return FAILURE
    $Written \leftarrow Written \cup \{i\}$
    $M_{avail} \leftarrow M_{avail} + f_i$
    $IO \leftarrow IO + f_i$
  Let i be the task such that $\sigma(i) = step$
  if $i \in Written$ **then**
    $Written \leftarrow Written \setminus \{i\}$
    $M_{avail} \leftarrow M_{avail} - f_i$
  if $i \notin Ready$, *or* $MemReq(i) > M_{avail} + f_i$ **then**
    return FAILURE
  $M_{avail} \leftarrow M_{avail} + f_i - \sum_{j \in Children(i)} f_j$
  $Ready \leftarrow Ready \setminus \{i\} \cup \bigcup_{j \in Children(i)} \{j\}$
return (SUCCESS, IO)

which files are written into secondary memory (if necessary). When a task i is scheduled for execution but its input file was moved to secondary memory, that file must be read and loaded back into the main memory before processing task i. Thus, a given file is written at most once in the main memory. The ordering of the I/O operations is done via a second function τ, such that $\tau(i)$ is the step when the input file of task i (of size f_i) should be moved to secondary memory ($\tau(i) = \infty$ means that this file is never moved to the secondary memory).

Algorithm 16 is used to check whether an out-of-core traversal is feasible. It makes use of $Written$, the set of files that have been moved to secondary memory. Similarly to the in-core case, M_{avail} denotes the memory which is available at a current step, and $Ready$ the set of ready nodes. The algorithm also computes IO, the total amount of data transferred from/to main memory. Note that each data written (once) to the secondary memory is read only once. At each step, the algorithm checks that the files written to secondary memory have been produced earlier, that precedence constraints are satisfied, and that there is enough memory to process the chosen node. More formally, a valid out-of-core traversal can be defined as follows.

Definition 4.3 (OUTOFCORETRAVERSAL)**.** *Given a tree $\mathcal{T}$ and a fixed amount of main memory M, the problem* OUTOFCORETRAVERSAL$(\mathcal{T}, M)$ *consists in finding an out-of-core traversal, described by a permutation σ of the nodes in $\mathcal{T}$ (corresponding to the schedule of computations), and a set $F \subset$*

$\{1, \ldots, n\}$ *(corresponding to the input files of tasks which are written to secondary memory), such that:*

$$\forall i \neq root, \quad \sigma(parent(i)) < \sigma(i) \tag{4.4}$$

$$\forall i, \quad \sum_{\sigma(j)<\sigma(i)} \left(\left(\sum_{k \in Children(j)} f_k \right) - f_j \right) - \sum_{\substack{j \in F \\ \sigma(j)>\sigma(i) \\ \sigma(parent(j))<\sigma(i)}} f_j + n_i + \sum_{k \in Children(i)} f_k \leq M \tag{4.5}$$

Then the amount of data written in secondary memory is given by

$$IO = \sum_{i \in F} f_i$$

In Equation (4.5), the term $\sum_{j \in F, \sigma(j)>\sigma(i), \sigma(parent(j))<\sigma(i)} f_j$ corresponds to the files that have been written into secondary memory at step $\sigma(i)$. We now define the MINIO problem, which asks for an out-of-core traversal with the minimum amount of I/O volume.

Definition 4.4 (MINIO)**.** *Given a tree $\mathcal{T}$, and a fixed amount of main memory M, determine the minimum I/O volume IO needed by a solution of* OUTOFCORETRAVERSAL$(\mathcal{T}, M)$.

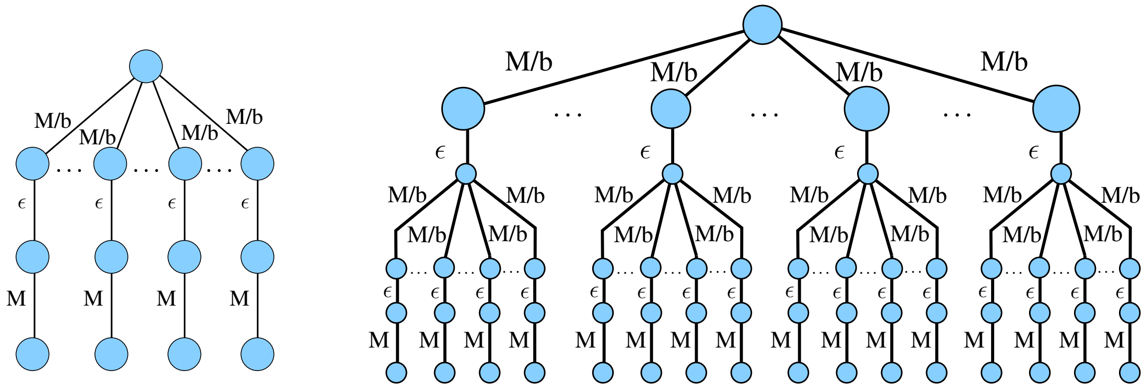
4.4 The MINMEMORY Problem

In this section, we present algorithms for the MINMEMORY problem. We first present the best possible postorder traversal, and show that its performance can be arbitrarily bad. Then we propose an alternative to the optimal algorithm introduced by Liu [69].

4.4.1 Postorder traversals

Postorder traversals are very natural for the MINMEMORY problem, and they are widely used in sparse matrix software like `MUMPS` [9, 10]. Liu [68] has characterized the best postorder traversal, leading to a fast but sub-optimal solution for MINMEMORY. In a nutshell, the best postorder is obtained by guaranteeing that in the resulting order, the children of a node are listed in the increasing order of the memory requirement of their respective subtrees. The algorithm is called $PostOrder$. In another study, Liu [69] has also provided an optimal algorithm for MINMEMORY whose worst case execution time is $O(p^2)$, where p is the number of tree nodes. The algorithm that finds the best postorder runs in $O(p \log(p))$ time, which calls for a tradeoff between speed and performance. But while postorder traversals are widely used in practice, their efficiency has never been thoroughly assessed (to the best of our knowledge). We now show that the best postorder may require arbitrarily more main memory than the optimal traversal.

Theorem 4.1. *Given any arbitrarily large integer K, there exist trees for which the best postorder traversal requires at least K times the amount of main memory needed by the optimal traversal for* MINMEMORY.



(a) One level.

(b) Two levels.

Figure 4.3: First levels of the graph for the proof of Theorem 4.1. Here b is the number of children of the nodes with more than one child.

Proof. Consider the harpoon graph with b branches in Figure 4.3a. All branches are identical and all tasks have a zero length execution file. Any postorder traversal requires an amount of $M+\varepsilon+(b-1)M/b$ main memory, while the optimal traversal (which alternates between branches) only requires $M_{\min} = M + b\varepsilon$. Now replace each leaf by a copy of the harpoon graph, as shown in Figure 4.3b. The value of $M_{\min}$ becomes $M+\varepsilon+2(b-1)\varepsilon$, while a postorder traversal requires $M_{PO} = M+\varepsilon+2(b-1)M/b$. When iterating the process L times, postorder requires $M_{PO} = M+\varepsilon+L(b-1)M/b$ while $M_{\min} = K(b-1)\varepsilon + M$. Thus, for any ratio K, there exists L such that when iterating the process L times, $M_{PO}/M_{\min} > K$. ■

4.4.2 The $Explore$ and $MinMem$ algorithms

Liu [69] proposes an algorithm for MINMEMORY which is optimal among all possible traversals, not only postorder ones. It is a recursive bottom-up traversal of the tree which, at each node of the tree, combines the optimal traversals built for all subtrees. The combination is based on the notion of ***Hill-Valley Segments*** and requires some sophisticated multi-way merging algorithm, in order to reach the $O(p^2)$ complexity. In this section, we introduce $MinMem$, another exact algorithm which proceeds top-down and maintains the best reachable cut of the tree at each step. While the worst-case complexity of $MinMem$ is the same as Liu's exact algorithm, it runs faster in practical cases resulting from multifrontal methods (see Section 4.6). The $MinMem$ algorithm is based on an advanced tree exploration routine: the $Explore$ algorithm.

The $Explore$ algorithm requires a tree T, a node i to start the exploration, and an amount of available memory M_{avail}. The last two parameters (L_{init} and Tr_{init}) are optional, and useful to speed-up the algorithm by avoiding the repeated exploration of some parts of the tree. With these parameters, the algorithm computes the minimal memory consumption that can be reached. If the whole tree can be processed, then the minimal memory is zero. Otherwise, the algorithm stops before reaching the bottom of the tree, because some parts of the tree require more memory than what is available. In this case, the state with minimal memory corresponds to a *cut* in the tree: some subtrees are not yet processed, and the input files of their root nodes are still stored in memory. The $Explore$ algorithm outputs the cut with minimal memory occupation, as well as a possible traversal to reach this state with the provided memory. In the case where the whole subtree cannot be executed, it also gives the minimum amount of memory (called memory *peak*) which is needed to explore an additional node in the subtree.

Algorithm 17: *Explore* $(T, i, M^{\text{avail}}, L_{\text{init}}, Tr_{\text{init}})$

Input: tree T, root i of the subtree to explore, available memory M^{avail}, initial set of nodes L_{init}, initial traversal Tr_{init}
Output: $\langle M_i, L_i, Tr_i, M_i^{\text{peak}} \rangle$, where:
M_i: the minimum memory requirement in the subtree rooted in i, reachable with memory M,
L_i: set of input files related to M_p,
Tr_i: the traversal from node i to L
M_i^{peak}: minimum memory to be able to visit a new node

if *node i is a leaf and* $n_i + f_i \leq M^{\text{avail}}$ **then**
  return $\langle 0, \emptyset, [i], \infty \rangle$
if $n_i + f_i + \sum_{j \in Children(i)} f_j > M^{\text{avail}}$ **then**
  $M_i^{\text{peak}} \leftarrow n_i + f_i + \sum_{j \in Children(i)} f_j$
  return $\langle \infty, \emptyset, [\,], M_i^{\text{peak}} \rangle$
if $L_{\text{init}} \neq \emptyset$ **then**
  $L_i \leftarrow L_{\text{init}}$
  $Tr_i \leftarrow Tr_{\text{init}}$
else
  $L_i \leftarrow Children(i)$
  $Tr_i \leftarrow [i]$
$Candidates \leftarrow L_i$
while $Candidates \neq \emptyset$ **do**
  foreach $j \in Candidates$ **do**
    $\langle M_j, L_j, Tr_j, M_j^{\text{peak}} \rangle \leftarrow Explore(T, j, M^{\text{avail}} - \sum_{k \in L_i \setminus \{j\}} f_k, \emptyset, \emptyset)$ `/* Process j */`
    if $M_j \leq f_j$ **then**
      $L_i \leftarrow L_i \setminus \{j\} \cup L_j$
      $Tr_i \leftarrow Tr_i \oplus Tr_j$ `/* append traversal` Tr_j `to the end of` Tr_i `*/`
  $Candidates \leftarrow \left\{ j \in L_i \text{ such that } M^{\text{avail}} - \sum_{k \in L_i \setminus \{j\}} f_k \geq M_j^{\text{peak}} \right\}$
$M_i \leftarrow \sum_{j \in L_i} f_j$
$M_i^{\text{peak}} \leftarrow min_{j \in L_i} \left(M_j^{\text{peak}} + \sum_{k \in L_i \setminus \{j\}} f_k \right)$
return $\left\langle M_i, L_i, Tr_i, M_i^{\text{peak}} \right\rangle$

When called on a node i, the algorithm first checks if the current node can be executed. If not, the algorithm stops and returns the current requirement as memory peak. Otherwise, it recursively proceeds in its subtree. The optimal cut is initialized with its children, and iteratively improved. All the nodes in the cut are explored: if the cut L_j found in the subtree of a child j has a smaller memory occupation than the child itself, the cut is updated by removing child j, and by adding the corresponding cut L_j. When no more nodes in the cut can be improved (which is easily tested using their respective memory peak), then the algorithm outputs the current cut.

Algorithm 18: $MinMem\ (T)$

Input: tree T
Output: minimum memory M needed to process the whole tree, traversal Tr

```
1 M^peak ← max_{i∈T} MemReq(i)                    /* lower bound */
2 M^avail ← 0
3 L ← ∅
4 Tr ← [ ]
5 while M^peak < ∞ do
6 |   M^avail ← M^peak
7 |   ⟨M, L, Tr, M^peak⟩ ← Explore(T, root, M^avail, L, Tr)
8 return ⟨M^avail, Tr⟩
```

The $Explore$ algorithm can be used to check whether a given tree can be processed using a given memory. If not, it provides a refined lower bound on the necessary memory. The $MinMem$ algorithm makes use of this information to solve the MINMEMORY problem.

To assess the complexity of the MINMEMORY problem, we consider the moment when each node is first visited by $Explore$, that is, for each node i, the first call on $Explore(T, i)$. There are p such events, which we denote as $F_1, \ldots, F_p$. We observe that between two such events, no node is visited more than twice by $Explore$. Firstly, in $Explore$, a subtree is re-visited only if the available memory is larger than its peak, which induces that a new node will be visited. Secondly, $MinMem$ asks $Explore$ to re-visit the whole tree with its peak value, which similarly leads to visit a new node. Thus, there are at most $2p$ calls to $Explore$ between two events F_i and F_{i+1}. Altogether, the overall complexity of the algorithm is $O(p^2)$.

4.5 The MINIO Problem

Contrarily to MINMEMORY, the MINIO problem turns out to be combinatorial. The difficulty goes beyond finding the best traversal. Indeed, even when the traversal is given, it is hard to determine which files should be transferred into secondary memory at each step.

4.5.1 NP-completeness

We prove that the following three variants of the problem are NP-complete.

Theorem 4.2. *Given a tree $\mathcal{T}$ with p nodes, and a fixed amount of main memory M, consider the following problems:*

(i) given a postorder traversal σ of the tree, determine the I/O schedule so that the resulting I/O volume is minimized,

(ii) determine the minimum I/O volume needed by any postorder traversal of the tree,

(iii) determine the minimum I/O volume needed by any traversal of the tree.

The (decision version of) each problem is NP-complete.

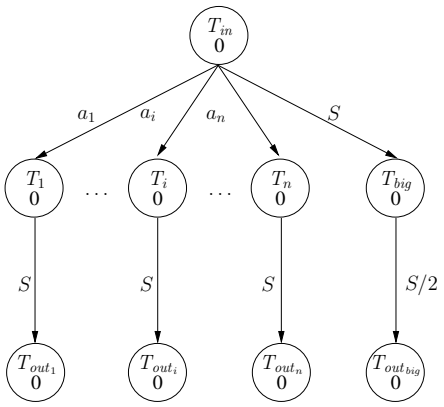


Figure 4.4: Graph corresponding to $Inst_2$ in the proof of Theorem 4.2.

Note that (iii) is the original MINIO problem. Also note that the NP-completeness of (i) does not a priori imply that of (ii), because the optimal postorder traversal could have a particular structure. The same comment applies for (ii) not implying (iii).

Proof. We use the same reduction for the three problems, which clearly all belong to NP. Consider an instance $Inst_1$ of 2-Partition [40], with n integers $\{a_1, a_2, \ldots, a_n | \sum_i a_i = S\}$. The instance $Inst_2$, common for all three problems, consists in the harpoon graph depicted on Figure 4.4, with $2n+3$ nodes. We let $M = 2S$, which is the largest memory requirement of a node (the root node T_{in}). We let the I/O bound be $IO = S/2$. The construction of $Inst_2$ is clearly polynomial in the size of $Inst_1$.

Note that all traversals are postorder traversals. Any traversal must start with the root T_{in}. After it has been processed, $2S$ units of memory are occupied. In order to process the rest of the tree, one has two main choices:

- either execute one of the n tasks T_i first, with $1 \leq i \leq n$. This requires loading the output file of T_i of size S into main memory, hence to transfer some files whose total sizes are at least S into secondary memory. This violates the I/O bound.
- either execute task T_{big} first. This requires to load its output file of size $S/2$ into main memory, hence to transfer some files whose total sizes are at least $S/2$ into secondary memory.

For (ii) and (iii), the reduction from *2-Partition* goes as follows. Suppose first that $Inst_1$ has a solution, with $\sum_{i \in I} a_i = \sum_{i \notin I} a_i = S/2$. Then, after unloading files of size a_i with $i \in I$ (thus increasing the available memory by $S/2$), one is able to process the entire branch of T_{big} up to the root. This means $Inst_2$ has a solution.

Suppose now that $Inst_2$ has a solution. That means that some files were unloaded in order to process T_{big}. The amount of memory to free is at least $S/2$. It is also at most $S/2$ so as to meet the bound IO. Therefore, exactly $S/2$ units of memory were unloaded to be able to host T_{big}. If I is the set of the unloaded files, then $\sum_{i \in I} a_i = S/2$, which means that $Inst_1$ has a solution and concludes our proof.

Now for (i), take any ordering of the nodes σ which executes T_{big} just after the root task T_{in}. The proof is the same, independently of the rest of the ordering. ■

4.5.2 Heuristics

The NP-completeness of problem (i) in Theorem 4.2 shows that it is difficult to select which files to unload to secondary memory, even when the traversal is given. We introduce six heuristics that greedily choose such files. In the following, j denotes the next node to be processed. First, if f_j has been previously unloaded, it must be stored back into main memory. Then an amount of $MemReq(j) - f_j$ of main memory must be available to execute node j. Let M^{avail} be the currently available memory. If $MemReq(j) - f_j \leq M^{\text{avail}}$, then node j can be processed without I/O. Otherwise, we have to unload a volume $IOReq(j) = M^{\text{avail}} - (MemReq(j) - f_j)$. In that case, we order the set $S = \{f_{i_1}, f_{i_2}, \ldots, f_{i_j}\}$ of the files already produced and still residing in main memory, so that $\sigma(i_1) > \sigma(i_2) > \cdots > \sigma(i_j)$. Hence f_{i_1} is the file which will be used at the latest iteration in the traversal, and so on. We greedily select the first files from S according to various criteria which we describe below.

Last Scheduled Node First (LSNF) We select the first files from S until their total size is at least $IOReq(j)$. The rationale is to unload the files that will be used the latest in the traversal, in order to avoid swapping intermediate files. This heuristic can easily be shown to be optimal for the divisible version of MINIO, where fractions of file can be written from and to secondary memory (see Section 4.2.2.)

First Fit This heuristic writes out the first file in S whose size is at least $IOReq(j)$. If no such file exists, the LSNF strategy is used.

Best Fit This heuristic writes out the file in S whose size is the closest of $IOReq(j)$: it chooses i_k such that $|IOReq(j) - f_{i_k}|$ is minimal. This step is repeated until enough space has been freed.

First Fill This heuristic writes out the first file in S whose size is smaller than $IOReq(j)$. This step is repeated until enough space has been freed. If not enough space can be freed, the LSNF strategy is then used. The rationale here is to avoid unduly writing big files out to secondary memory, thus significantly increasing I/O volume. Instead this heuristic tries to ''fill'' out the required I/O volume with the first eligible files.

Best Fill This heuristic writes out the file whose size is the closest to $IOReq(j)$ among those files in S whose size is smaller than $IOReq(j)$. This step is repeated until enough space has been freed. If not enough space can be freed, the LSNF strategy is then used. The rationale here is to ''fill'' out the required I/O volume, but this time with the best eligible files.

Best K Combination This last heuristic considers the first K files in S (we use $K = 5$ in the experiments) and selects the best combination, i.e., the subset whose size is the closest to $IOReq(j)$. This step is then repeated until enough memory has been freed.

4.6 Experiments

In this section, we experimentally compare the three algorithms for MINMEMORY, namely $PostOrder$ (which finds the best postorder traversal of the tree) and the two optimal variants Liu (exact algorithm of [69]) and $MinMem$. We evaluate the deviation of $PostOrder$ form the optimal solution, and we study the execution cost of each algorithm. Next we report on the performance of the heuristics for MINIO.

4.6.1 Setup

Each algorithm has been implemented in highly optimized C++ versions. The $PostOrder$ and Liu algorithms are written as iterative codes while $MinMem$ is a recursive code. Their behavior has been validated on a platform based on an Intel Xeon 5250 processor. Source code for all the algorithms, heuristics and experiments is publicly available [1].

Experiments were conducted within the Matlab environment for commodity reasons, especially ease of access to various data sets. We use a generic tool called *performance profiles* [33] to assess the proposed algorithms and heuristics. The main idea behind performance profiles is to use a cumulative distribution function as the performance metric, instead of taking averages over all test cases. We investigate the performance of the algorithms and heuristics in terms of running times and the quality of the solution (the memory requirement, or the total I/O volume). For a given metric, a profile plot shows the fraction of cases where a specific method gives results which are within some value τ of the best result reached by all algorithms. Therefore the higher the fraction, the more preferable the method. For example, for the runtime metric, a τ value shows the fraction of cases where the running time of the target algorithm is within τ of the fastest algorithm shown in the same plot. Similarly, for the memory requirement metric, a τ value shows the fraction of cases where the memory requirement of the target algorithm is within τ of the best result found by any algorithm shown in the same plot.

4.6.2 The Data Set

The data set contains assembly trees of a set of sparse matrices obtained from the University of Florida Sparse Matrix Collection [2]. The matrices satisfy the following assertions: square, number of rows is between 2×10^4 and 2×10^5, the number of nonzeros per row is at least 2.5, and the number of nonzeros is at most 5×10^6. At the time of testing there were 291 matrices satisfying these properties. We use the symmetrized pattern of the matrices, e.g., the pattern of $|A| + |A|^T + I$. We first order the matrices using MeTiS [63] (through MeshPart toolbox [42]) and `amd` (available in Matlab), and then build the corresponding elimination trees using the `symbfact` routine of Matlab. We also perform relaxed node amalgamation on these elimination trees to create assembly trees. We have created a large set of instances by allowing 1, 2, 4, and 16 (if $n > 1.6 \times 10^5$) relaxed amalgamations per node. We always realize perfect amalgamations, e.g., when a node is the only child of its parent and the parent has only one less entry in the associated column in L, the two nodes are amalgamated. When the current amalgamated node does not contain more than the allowed amalgamation per node, we amalgamate the node with its densest child. At the end we compute the weight of a node as $\eta^2 + 2\eta(\mu - 1)$, where η is the number of nodes amalgamated, and μ is the number of nonzeros in the column of L which is associated with the highest node (in the starting elimination tree). Edge weights are computed as $(\mu - 1)^2$. These numbers correspond respectively to n_i and f_i as described in Section 4.3.

4.6.3 Results for MINMEMORY

The first objective is to evaluate the performance of $PostOrder$ in terms of the memory requirement of the resulting traversal with respect to the optimal value. In 95.8% of the cases, $PostOrder$ is optimal. Only the non-optimal cases are depicted on Figure 4.5, $PostOrder$ requiring up to 18% more memory than the optimal solution. Detailed statistics are given in Table 4.1. As a conclusion, $PostOrder$ statistically gives very good results for assembly trees, except in rare cases where it can require up to 20% more main memory than the optimal traversal.

1. http://graal.ens-lyon.fr/~mjacquel/minmem.html
2. http://www.cise.ufl.edu/research/sparse/matrices/

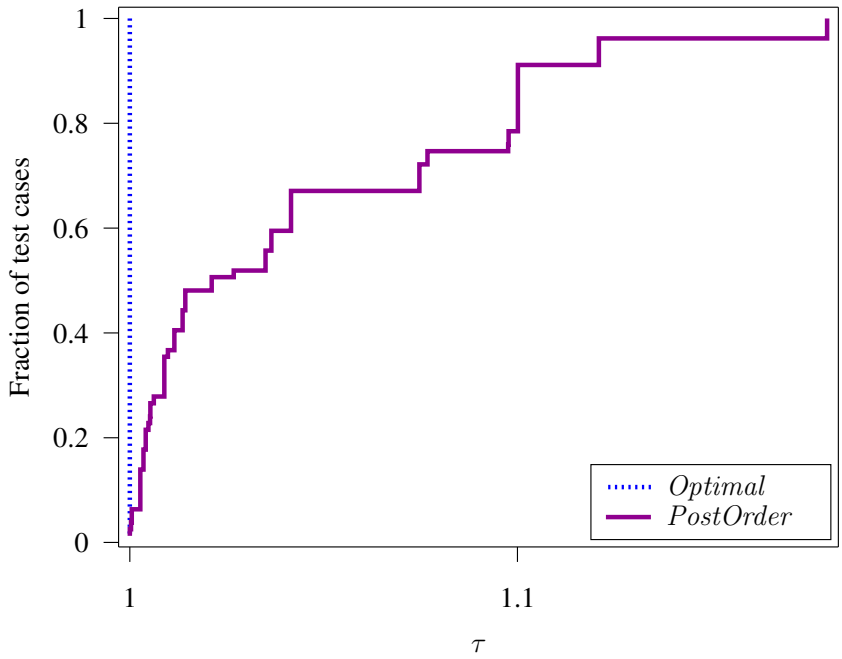


Figure 4.5: Performance profile for comparing the memory requirement obtained by $PostOrder$ with the optimum values for the assembly trees for which $PostOrder$ does not find an optimal solution.

Non optimal $PostOrder$ traversals	4.2%
Max. $PostOrder$ to opt. cost ratio	1.18
Avg. $PostOrder$ to opt. cost ratio	1.01
Std. Dev. of $PostOrder$ to opt. cost ratio	0.01

Table 4.1: Statistics on memory cost of $PostOrder$ for assembly trees.

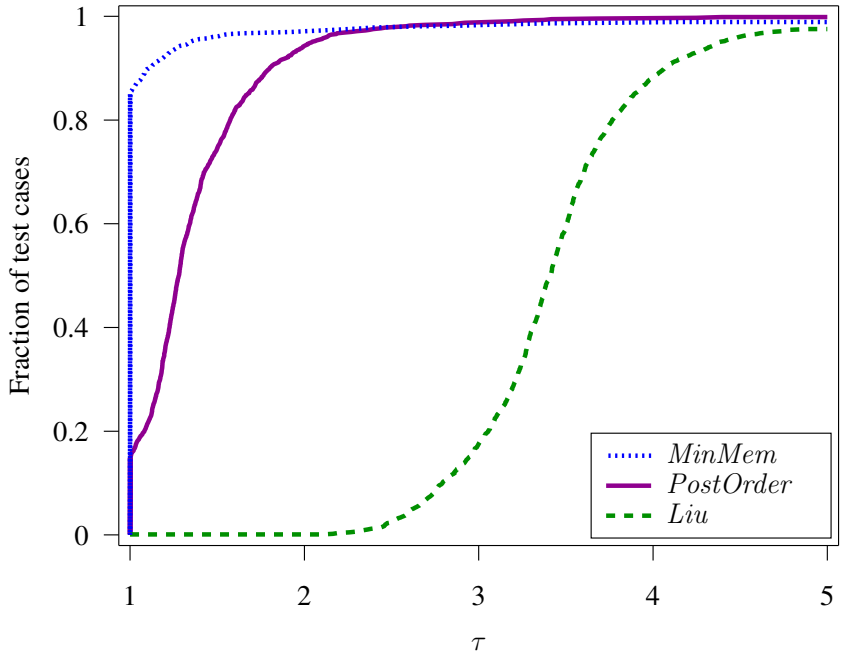


Figure 4.6: Performance profiles for comparing the running time of the three algorithms for the MINMEMORY problem on the assembly trees.

The second objective is to compare the running times of the three algorithms. We observe on Figure 4.6 that $MinMem$ is the fastest algorithm in 80% of the cases, and clearly outperforms Liu.

Altogether, these experiments show that in the context of sparse matrix assembly trees, $PostOrder$ frequently offers optimal or near-optimal results. When $PostOrder$ is not optimal, it is reasonably close to the minimum memory required to process the tree. Nevertheless, whenever memory becomes a key resource, $MinMem$ can compete with $PostOrder$ in terms of running time, and it always produces the optimum memory requirement, therefore constituting an interesting alternative.

4.6.4 Results for MINIO

This experiment aims at evaluating the six heuristics introduced in Section 4.5.2 for the MINIO problem. Tree traversals are obtained using $PostOrder$, Liu and $MinMem$ for the MINMEMORY problem. The available memory ranges from $\max_{i \in T} MemReq(i)$, to the minimal memory required for the traversal.

On Figure 4.7, the performance profile of all heuristics applied on traversals produced by $MinMem$ is depicted. The best heuristic is clearly First Fit, which is almost tied by Best K Combination. Then Best Fill and First Fill provide almost the same I/O volume, and in turn perform better than Last Scheduled Node First and Best Fit, which are very close. As a consequence, because of its good behavior and low complexity, First Fit represents the best alternative among the six policies. This conclusion remains true when applying the heuristics to traversals produced by $PostOrder$ or Liu.

The next experiment aims at characterizing the behavior of the algorithms designed for MINMEMORY in the context of out-of-core traversals. The policy used for I/O is First Fit. The performance profile of every traversal is reported on Figure 4.8. The best results are provided by $PostOrder$. This experiment also shows that $MinMem$ does not produce good out-of-core tree traversals, and is outperformed by Liu which provides better traversals for MINIO. This interesting result is due to the fact that contrar-

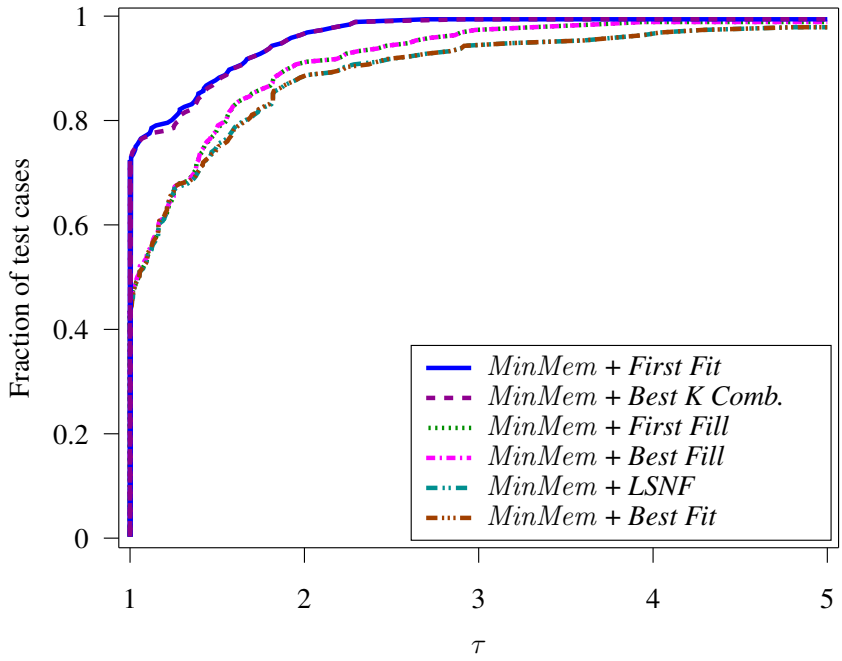


Figure 4.7: Performance profiles for comparing the resulting I/O volume of the heuristics for the *MinMem* algorithm on the assembly trees.

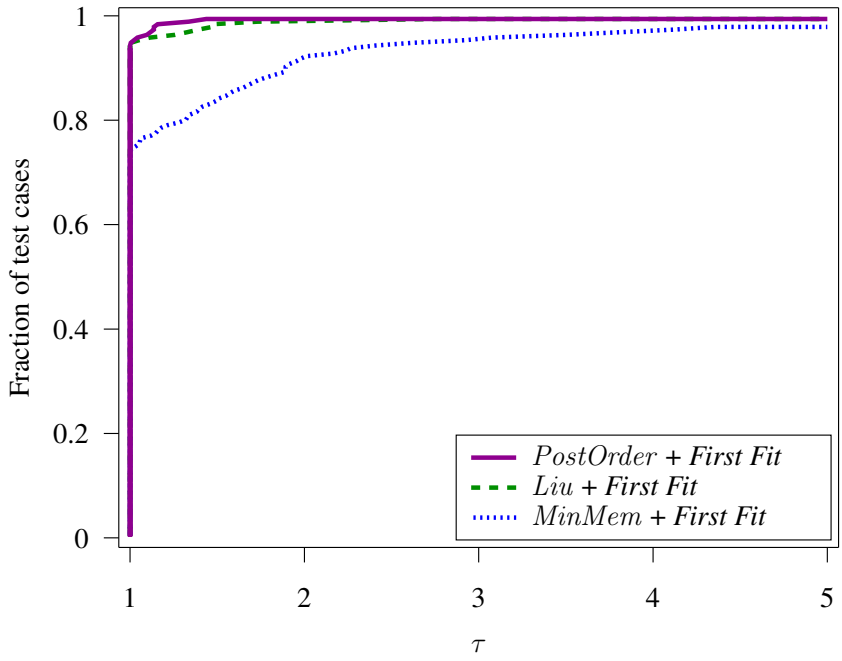


Figure 4.8: Performance profiles for comparing the resulting I/O volume of the three algorithms equipped with the First Fit heuristic on the assembly trees.

ily to $MinMem$, Liu produces long chains of dependent tasks by construction. These chains reduce the pressure on main memory since files produced by a task will be consumed soon, thereby reducing the I/O volume. $PostOrder$ also benefits from this phenomenon.

4.6.5 More on $PostOrder$ Performance

This last experiment comes almost as a digression, because we do not use assembly trees here. The objective is to further assess the performance of $PostOrder$ in terms of the resulting memory requirement. While the theory tells us that $PostOrder$ can be arbitrarily bad (see Theorem 4.1), it turns out that its performance on assembly trees is very good (see Table 4.1). We wanted to assess the performance of $PostOrder$ on randomly generated trees. We keep the structure of every actual assembly tree from the data set discussed above, and assign random integers ranging from 1 to $N/500$ to the node weights and from 1 to N for the edge weights (N denoting the number of tree nodes). This leads to a comprehensive data set containing more than 3200 trees, and allows for a more refined performance evaluation of $PostOrder$.

The experiment (see Figure 4.9 for details on the non-optimal cases) shows that $PostOrder$ requires more than the minimum memory in 61% of the cases. In some cases, $PostOrder$ may require more than twice as much memory as the optimal solution. More details are given in in Table 4.2. All in all, this experiment shows that when dealing with general trees, it is mandatory to use an optimal algorithm if main memory is a scarce resource.

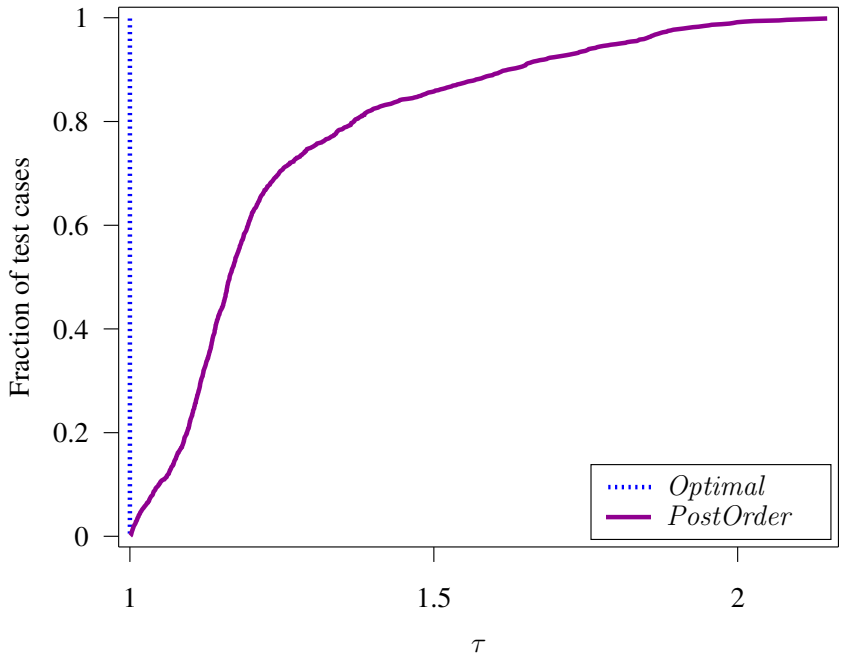


Figure 4.9: Performance profile for comparing the memory requirement obtained by $PostOrder$ with the optimum values on the random trees.

4.7 Conclusion

We have discussed how to traverse the nodes of a tree-shaped workflow so as to optimize the memory used in a two-level memory system. We have investigated two main problems. In the MINMEMORY

Non optimal $PostOrder$ traversals	61%
Max. $PostOrder$ to opt. cost ratio	2.22
Avg. $PostOrder$ to opt. cost ratio	1.12
Std. Dev. of $PostOrder$ to opt. cost ratio	0.13

Table 4.2: Statistics on memory cost of $PostOrder$ for random trees.

problem, the aim is to minimize the memory requirement, while in the MINIO problem, the aim is to minimize the I/O volume, given a limited memory. Our motivating application was the multifrontal method of sparse matrix factorization in which elimination (or assembly) trees are used to reorganize the computations. The MINMEMORY problem corresponds to the problem of minimizing the memory requirement of an in-core execution of the multifrontal method, while the MINIO problem corresponds to the problem of minimizing the I/O requirement in an out-of-core execution.

For the MINMEMORY problem, we have proposed an exact algorithm which runs faster than the reference alternative of Liu [69]. The current state of the art software for sparse matrix factorization finds the best postorder as a solution to the MINMEMORY problem. This is done both for convenience and for the in-core memory requirement. We have investigated how good this choice is, and concluded that in most practical cases, the minimum memory requirement due to a postorder is usually close to the optimal one (in a large set of instances we have seen at most 18% increase with respect to the memory minimizing traversal). However, we also showed that on general trees, the best postorder can result in memory requirements that are arbitrarily large.

We have shown that the MINIO problem is NP-complete, as well as some of its variations (for example, we have shown that finding the postorder traversal that minimizes the I/O volume is NP-complete). We have designed heuristics for the problem and have performed thorough experimental comparisons. Our experiments are based on highly optimized versions of three tree traversal algorithms, and precisely assess the quality of each proposed algorithm and I/O heuristic. We have shown that our $MinMem$ algorithm outperforms the running time of Liu's exact algorithm, but we have also observed that it was less suited for out-of-core execution.

Chapter 5

Comparing archival policies for BLUE WATERS

5.1 Introduction

Hierarchical memory architectures can be found from within multicore processors, as seen in Chapters 1 and 3, up to the storage systems of cutting-edge supercomputers. Indeed, these large scale platforms are likely to also exhibit a two level memory hierarchy made of fast but low capacity storage devices, like hard disk drives, and a higher capacity but slower archival level made of magnetic tape drives for instance.

The new generation of petascale supercomputers will need exabyte-scale archival storage. For example, the BLUE WATERS petascale system that is being installed at the University of Illinois, Urbana-Champaign [19] will have a peak performance of over 10 petaflop/s, over a petabyte of DRAM, and over 18 petabytes of disk storage; yet most of the storage – up to half an exabyte – will be on tapes.

This is consistent with a recent report [50], based on eight year experience at several major High Performance Computing (HPC) centers, that shows a need to archive about 35 TB of new data each year for every TB of memory, not counting archived data that are deleted (20-50%). In a such a system, there is a crucial need for efficient archival policies for writing to and reading from the tape system.

In addition, archive systems need to be reliable. Experience shows that a significant fraction of jobs in HPC centers fail because of some errors in the archive system: inability to load the tape, metadata errors on the tape, tensioning errors, breaking tape, etc. It is estimated in large centers that 1 out of 100 tape handling events leads to a job failure [55]. Moreover, tapes may face unrecoverable data errors leading to permanent data loss. The current solution to prevent data loss and avoid the propagation of a failed tape handling event is RAIT, which stands for *Redundant Array of Independent Tapes* [35] – in analogy to RAID.

On BLUE WATERS, the disk space will be managed by GPFS [86], while the archival tape system will be managed by HPSS [93]. The GPFS-HPSS Interface (GHI) integrates the tape archive into the GPFS namespace, so that the disk storage essentially is a cache for the tape storage, and data migrates transparently between disk and tape.

HPSS supports RAIT Level 0 (mirroring, for reliability) and RAIT Level 1 (stripping, for increased transfer speed). Mirroring doubles the amount of tape storage needed – an expensive proposition. There is ongoing work on HPSS support for schemes similar to RAID 6 [55], Such a RAIT architecture requires $X+Y$ tape drives for archiving a single file. The file is split into blocks; for each X consecutive blocks one computes Y *Erasure Code* (EC) blocks. The $X+Y$ blocks are written on $X+Y$ distinct tapes, with EC blocks rotated across tapes. Such a design can recover from the failure of any Y tapes,

and speed up transfer rate, by a factor of X. (However, start-up time increases, as $X + Y$ tapes need to be loaded.) Values being considered include $X + Y = 4 + 1$, $4 + 2$ or $8 + 2$.

The main drawbacks of such RAIT architecture are the following:

– Because X consecutive blocks of the (currently written) file are stored in parallel, files are scattered across many tapes.
– Each archival request monopolizes $X + Y$ tape drives, which considerably reduces the number of user requests that can be simultaneously processed by the system.

The first problem (file fragmentation) is expected to have a dramatic impact on performance. Contiguous access is faster when reading from tapes, just as it is for disks, but the speedup ratio is much larger. Suppose that, in order to use a tape efficiently, one needs to access a contiguous block of size at least S; then, with RAIT, one needs to access at least $X \times S$ contiguous data to use the tapes efficiently. Many files might be shorter than this threshold. It is possible to solve this issue for writes, by concatenating multiple short files into one larger "container file". However, subsequent reads will have low performance unless all the files concatenated in one container are accessed together – something that is not always true, and cannot be guaranteed, especially in a system such as BLUE WATERS where archival is initiated by the system, not by a user.

The second problem (several tape drives per request) will drastically limit the access concurrency of the system, by increasing the response time when many users aim at archiving their data. If, say, 500 tape drives are available, and if the archival policy requires 10 tape drives per request, then at most 50 requests can be served simultaneously. This may well prove a severe limitation for some usage scenarios of the target supercomputer platform. Furthermore, the average start-up time for file transfers can increase significantly, since the number of robotic arms to move tapes is often lower than the number of tape drives, and the transfer can start only after the last tape was mounted.

Note that, unlike for RAID, it is not always necessary to read the redundancy blocks when a file is accessed: Since tape blocks are long, one can compute and store longitudinal codes to ensure that data read is valid. Also, one can leverage the fact that disk storage (unlike main memory) is persistent to delay the storage of error correcting information to tape, thus enabling more asynchrony. These differences allow for new policies, different from classical RAID.

To overcome the shortcomings of RAIT, we have designed two new archival policies. The first of them, PARALLEL, still uses the same number $X + Y$ of tape drives, and hence suffers from the same problem that it reduces the servicing capacity of the system and increases start-up time. But it does reduce the fragmentation of files, by pre-partitioning such files into X stripes that will be written in contiguous mode on the tapes. Subsequent reads will be able to access larger segments from each tape.

The second policy, VERTICAL, is more drastic and solves both problems, at the price of lower transfer rates. The idea is to write data contiguously and sequentially on X tapes, filling up the tapes one by one, and to delay the archival of the redundancy data on Y tapes after X tapes have been actually written. This requires to update the contents of the Y redundancy tapes on-the-fly. This scheme allows for serving as many requests as the number of available tape drives. However, each request is processed without any parallelism in writing, hence transfer rate decreases. (The problem can be avoided, for very long files, by simultaneously writing or reading tape-sized segments; it is not an issue for very short files, where tape load and seek time dominates access time; it affects files in a range in between these extremes.)

In this chapter, the focus will mainly be set on the evaluation of the three RAIT, PARALLEL and VERTICAL policies within an event-driven simulator, and to compare their performance through extensive simulations. The simulation setting corresponds to realistic execution scenarios (in terms of both hardware platform parameters and I/O request rates) for the future exploitation of BLUE WATERS. After studying their relative performance on different file sizes, we propose a last strategy, which mixes

the best two candidates (PARALLEL and VERTICAL) to outperform them.

This chapter is organized as follows. We first briefly review related work in Section 5.2, and we outline the framework in Section 5.3. Then we detail the three archival policies in Section 5.4. The main scheduler and load balancer are described in Section 5.5. The simulation setting is provided in Section 5.6, as well as the results of the comprehensive simulations. Finally, we state some concluding remarks and hints for future work in Section 5.7.

5.2 Related work

We classify related work into two main categories, those dealing with resilient storage policies, and those discussing tape request scheduling strategies.

Resilient storage policies The first fault-tolerant policy proposed for tapes was inspired from disks. It adapts the classical disk RAID policy for tapes, and thus was called RAIT [35]. An important difference between RAID and RAIT is that the erasure code is computed from disk blocks on RAID 5 and RAID 6, while RAIT compute the erasure code from file stripes. Notes that this approach of encoding the data has also been proposed recently to overcome the issues related to RAID 5 and RAID 6.

Jonhson and Prabhakar proposed to decouple the stripes used to write files to the tapes from the ones used to compute parity, called *regions* [60]. Their basic idea is to group a number of regions from different tapes into a parity group and to compute and store the parity of these regions on another tape. The proposed framework allows for a wide variety of policies, such as the ones introduces in this chapter.

Scheduling tape requests Together with designing tape storage policies, we also need to schedule I/O requests. Some specific problems to I/O on tapes have been considered in the literature. In [51], Hillyer et al. consider the problem of scheduling retrieval requests to data stored on tapes. Using a precise model for the performance of the tape drives, they proved the problem of minimizing the completion time for a set of request NP-hard, and proposed a complex heuristic to solve it.

In [78], Prabhakar et al. consider the problem of scheduling a set of storage requests using a simple model of tape storage, with the objective of minimizing the average waiting time. An optimal scheduling policy is provided for the one tape drive case, and the problem is proven NP-complete for multiple drives.

To the best of our knowledge, this work is the first aimed at designing innovative tape archival policies for petascale computers like BLUE WATERS, and assessing their performance.

5.3 Framework

In this section, we first describe the platform model, and then we state the optimization problem under consideration.

5.3.1 Platform model

We derive a model that is representative of a system such as BLUE WATERS, but does not match exactly its (currently confidential) configuration. Our goal is to simulate the maximum capacity of the BLUE WATERS archival storage (0.5 Exabytes), that is much higher than its initial capacity. Here is a list of key parameters describing the platform:

Tapes The archival system counts 5000,000 serpentine tapes. Each tape stores up to 1 TB of uncompressed data.

Tape Drives There are 500 tape drives to perform read/write operations on these tapes.

Tape Libraries Tapes are gathered into 3 tape libraries, with passthrough to transfer tapes between different libraries

Mover Nodes 50 mover nodes are dedicated to process I/O requests and compute redundancy blocks. Each mover node has 24 cores and 96 GB of RAM; a mover mode is connected to 10 tape drives. We assume that a mover node has access to 10 TB of local disk storage.

Additional computing resources are used by HPSS, e.g., to schedule transfers, and by GPFS to run file system code.

5.3.2 Problem statement

The focus of this study is to handle I/O requests in an efficient way. An I/O request can be sent in the system in response to an explicit user command (to archive or delete data, or move it to disk or off-site); by the automatic disk management system (to migrate from disk data not touched recently); or by the job scheduler (to load to disk files needed by scheduled batch jobs).

An I/O request is characterized by the file that it is accessing, and therefore by the size of this file. The request is also associated with a resiliency scheme $X + Y$, where X denotes the number of data blocks corresponding to Y Erasure Code (EC) blocks. The Y EC blocks are computed by using any EC algorithm over X blocks of data. Finally, an I/O request is defined by the I/O policy which it is using, i.e., the way data and EC blocks are organized onto tapes. As already stated, three I/O policies will be considered in this study: RAIT, VERTICAL and PARALLEL.

The most natural objective function is the *average response time* for a request, which measures the time between the arrival of a request in the system and the completion of its processing. However, this objective is known to unduly favor large request over smaller ones, and the objective of choice is rather the *average weighted response time*, where the response time for a request is divided by its size. The weighted response time is close to the *stretch*, which is a widely used fairness objective [79]. The stretch is the slow-down experienced by the request, i.e., its response time in the actual system divided by its response time if it were alone in the system (this later quantity being roughly proportional to the file size if we neglect all latencies). Another important, platform-oriented objective, is the aggregate throughput, or aggregated bandwidth of I/O operations, achieved by the system.

5.4 Tape archival policies

Writing data to tapes is a challenging task, and particular care is required to design and implement efficient archival policies. In this section, we first review the well-known RAIT policy. Then we introduce two novel I/O policies, VERTICAL and PARALLEL.

5.4.1 RAIT

The first policy is called RAIT, for "Redundant Array of Independent Tapes", and is the counterpart of RAID for tapes [35]. In order to overcome the performance and reliability limitations of each individual tape drives, RAIT writes data in parallel while keeping the usage of all tapes balanced. In addition to the resiliency scheme $X + Y$, which requires $X + Y$ tape drives, RAIT is characterized by a block size B which defines the transfer unit. In order to ensure resiliency, RAIT computes Y EC blocks

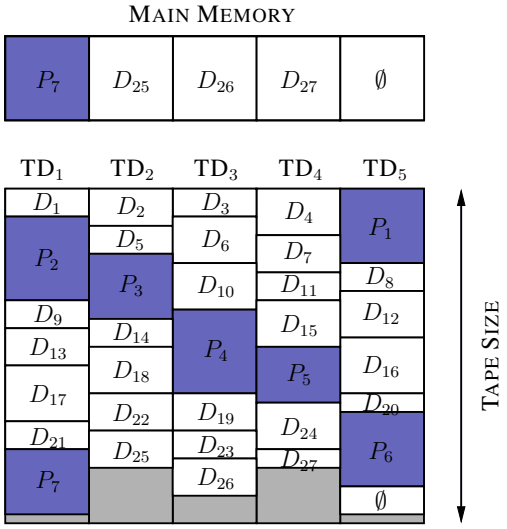


Figure 5.1: Writing data with RAIT policy for $X = 4$ and $Y = 1$.

from X data blocks. These EC blocks are computed in memory before data blocks are actually written to tapes. This implies a memory footprint of $(X + Y) \times B$ for each RAIT request running concurrently.

The behavior of RAIT is depicted on Figure 5.1: if all tapes are empty, the first X blocks are written on tape drives TD_1 to TD_X, while the Y EC blocks are written on TD_{X+1} to TD_{X+Y}. The following sets of X data and Y EC blocks are then periodically shifted (see Figure 5.1). Note that when several consecutive requests of the same policy and resiliency scheme are served using the same tapes, they are processed with consecutive unit shifts as if we had a single large request. Note that in the example depicted on Figure 5.1, the file is made of 37 data blocks.

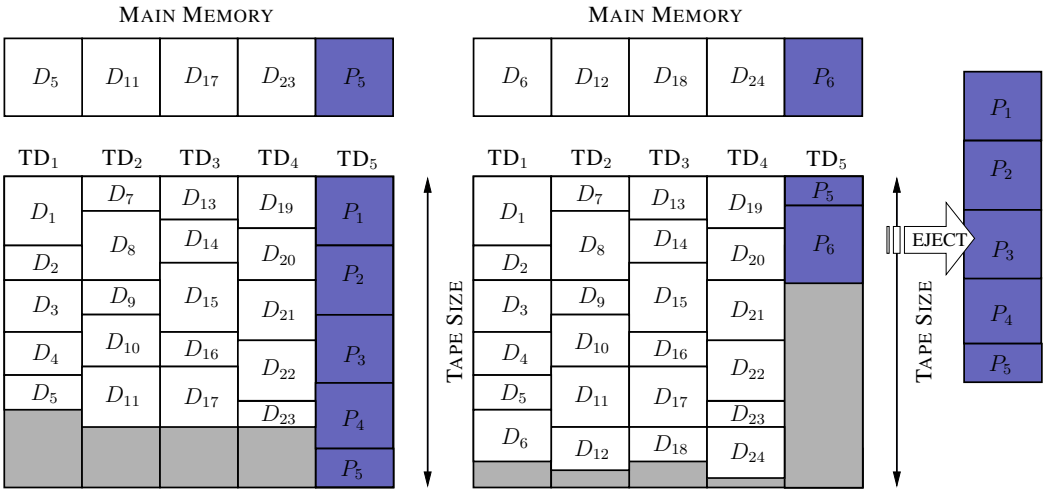
Note that tape drives generally offer a hardware compression mechanism. Hence, although every block has a size of B, the space occupied on tape may differ from block to block. In particular, EC blocks are expected to be much less compressible than data blocks. RAIT balances tape occupation through its periodical shifting mechanism.

The use of $X + Y$ tapes for each file transfer reduces the number of concurrent transfers possible, and increases start-up time. Moreover, as data and EC blocks are periodically shifted across all tape drives, it is not possible to bypass EC blocks on reads.

5.4.2 PARALLEL

One of the main drawbacks of RAIT is that data on tapes is fragmented. In order to solve this issue, we introduce a novel policy called PARALLEL. This policy keeps the parallel I/O operations offered by RAIT but rather writes data as much contiguously as possible.

Just like RAIT, PARALLEL is characterized by a block size B. For a given resiliency scheme $X+Y$, it also requires $X+Y$ tape drives for writing data along with Y tape drives for writing EC blocks. These EC blocks are computed in memory from X data blocks before being actually written to tapes. The memory footprint is therefore $(X + Y) \times B$ for each PARALLEL request running concurrently. However, unlike RAIT, (i) all EC blocks are written on Y separate tapes; and (ii) the system writes on each of the X data tapes the longest possible sequence of consecutive data blocks. Thus, if the file has S blocks, then each of the X data tapes will store either $\lceil S/X \rceil$ or $\lfloor S/X \rfloor$ consecutive data blocks. In



(a) Data is written until EC tape gets filled.

(b) Filled EC tape has been ejected, and replaced by a new one.

Figure 5.2: Writing data with PARALLEL policy with $X = 4$ and $Y = 1$.

the simpler case where $S = W \times X$ then, at step i, the system transfers to X data tapes the X file blocks $D_i, D_{i+W}, \ldots, D_{i+(X-1)W}$, and transfers to the Y EC tapes the Y EC blocks computed from the X data blocks. The scheme is depicted on Figure 5.2: first, blocks D_1, D_7, D_{13} and D_{19} are held in memory and EC block P_1 is computed. Everything is then written on distinct tape drives. Then, next sets of blocks are processed the same way until a tape dedicated to data gets filled. Whenever this happens, every tapes are replaced by new empty tapes. However, it may happen that some (or all) of the tapes dedicated to store EC blocks get filled before data tapes (as depicted on Figure 5.2a), EC blocks being generally less compressible. In such a case, these tapes are ejected and replaced by new ones. Overflowing EC blocks are then written onto those new tapes. This is what happens on Figure 5.2b.

Altogether, PARALLEL maintains the same level of concurrency in the transfer of files to/from tape, but stores contiguously as much data as possible. Because of the hardware compression mechanism embedded in tape drives, and given that EC blocks are generally less compressible than data blocks, tape occupation is slightly unbalanced. This can lead in some cases to the use of extra tapes to hold the Y EC blocks, which could have an effect on performance. Like RAIT, PARALLEL also has a significant impact on the level of parallelism of the system, although lower than RAIT, since $X + Y$ tape drives are required for write operations but only X tape drives are needed for read operations.

5.4.3 VERTICAL

With both previous policies, parallelism and resiliency are tightly coupled, which tends to decrease the overall level of service provided by the system. In order to break this coupling, we propose the VERTICAL policy, which also keeps data entirely contiguous on tapes.

For a given $X + Y$ resiliency scheme, VERTICAL writes X data tapes before writing Y EC tapes (or more, according to the compression rate). All writes are performed serially, on one tape drive. In order to do so, VERTICAL requires enough local storage space to store Y uncompressed tapes. Each time a data block is written, the corresponding Y EC blocks are updated; the computation of these EC blocks is completed after X data blocks have been written. The scheme is depicted on Figure 5.3. Several areas are allocated on disk dedicated to hold the Y ECs. As data is received, ECs areas are updated and data is written onto tape. When the tape (holding data) is filled, it is replaced (as shown on

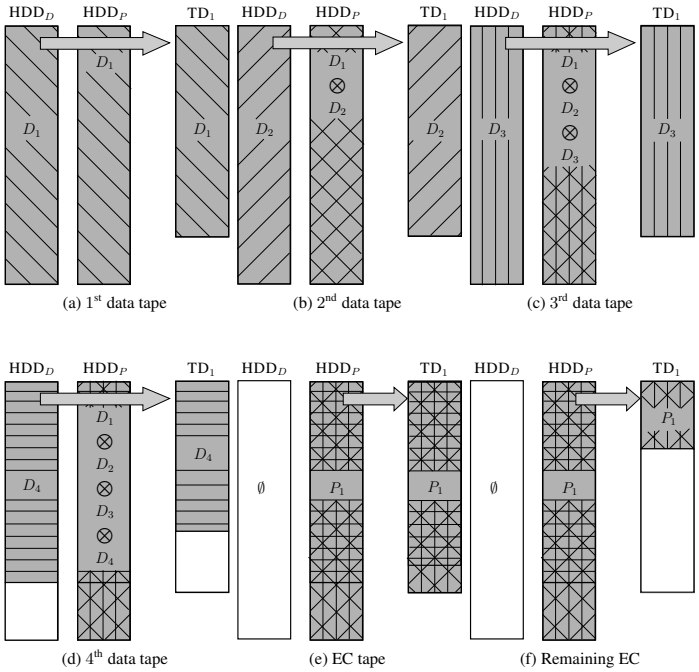


Figure 5.3: Writing data with VERTICAL policy with $X = 4$ and $Y = 1$.

Figures 5.3a, 5.3b and 5.3c).

When the X^{th} data tape has been filled or there is no more data to be written (Figure 5.3d), the tape is replaced by the first tape dedicated to ECs. EC block is then written onto tape (as depicted on Figure 5.3e). However, similarly to the PARALLEL policy, it may happen that this EC tape gets filled before the entire EC block has been written. Whenever this happens, the tape is ejected and replaced by a new tape which will hold the remaining part of the EC block. This case is depicted on Figure 5.3f. This step is then repeated for each remaining EC blocks.

It is clear that VERTICAL has a minimal impact on the level of parallelism of the system since it requires only one tape drive, regardless of the resiliency scheme used. Data is contiguous on tape and EC blocks need not be read when data is read.

However, this approach has several limitations: (i) the erasure code cannot be computed in mover memory because at least $Y + 1$ entire tapes should fit in this memory, which is not possible. Therefore, these tapes must instead be stored on disks. Like the PARALLEL policy, EC blocks are written onto dedicated tapes, meaning that tape occupancy may be less balanced than with RAIT. (ii) although the entire system may be able to handle more requests concurrently, each request will take more time to complete since there is no data parallelism with VERTICAL. (iii) the tape drive is unavailable for application I/O when writing the EC blocks.

5.5 Scheduling archival requests

Scheduling I/O requests on a petascale platform is a hard task. Indeed, the tremendous number of parameters that need to be taken into account makes it challenging.

We introduce an online scheduler which basically maps tape I/O requests submitted to the system onto a mover node. This scheduling process, denoted as MAIN-SCHEDULER in the following, works

hand in hand with a load balancing process, denoted as LOAD-BALANCER, responsible for handling requests which were impossible to schedule by MAIN-SCHEDULER at the time of their arrival.

We choose a "dynamic" approach, where processes corresponding to different I/O policies are created on-the-fly onto the mover nodes, rather than being statically allocated. This creation process is done either by MAIN-SCHEDULER or LOAD-BALANCER whenever a new process is required.

The MAIN-SCHEDULER process works as follows: as soon as a request R is submitted to the system, it is handled by MAIN-SCHEDULER. MAIN-SCHEDULER first checks whether R can be served now, i.e., if there is no previous request(s) regarding the same file or, if R is a read request, if the tapes containing the concerned file are not currently in use. If R is in use, it is delayed and placed in the waiting list.

Requests are identified by their *type*, which is defined as their archival policy together with their resiliency scheme. For instance $(\text{PARALLEL}, 4+1)$ or $(\text{RAIT}, 8+2)$ are possible request types. If the request R can be scheduled, MAIN-SCHEDULER tries the following actions:

- first, MAIN-SCHEDULER tries to find a currently running process which matches the type of R. If such a process P exists, and if no more than MAXLIGHTLOAD requests are already scheduled onto this process, then R is mapped on process P;
- otherwise, MAIN-SCHEDULER tries to find a mover node having enough idle tape drives to host a new process for R, and it creates this process;
- then, if MAIN-SCHEDULER is unable to create a new process for handling R, it tries to schedule it on a currently running process P matching the type of R, but this time regardless of the number of requests already mapped onto P.

The rationale is to allocate requests to already running processes, provided that their load remains reasonable, otherwise it might be better to create new processes. The role of the system parameter MAXLIGHTLOAD is to tune the load threshold of the processes. Finally, if MAIN-SCHEDULER is still not able to schedule R, the request is delayed and placed in the waiting list.

In order to schedule the requests in the waiting list, as well as to keep the load balanced across the system, the LOAD-BALANCER is periodically executed every MINLBINTERVAL units of time. However, in order to keep the number of interventions of LOAD-BALANCER within a reasonable amount, one of the following conditions must be met:

- the oldest request has been delayed for more than MAXWAITINGTIME, and its file is not currently in use;
- the number of non-scheduled pending requests in the waiting list exceeds MAXREQCOUNT;
- the maximum *imbalance* of the system exceeds MAXIMBALANCE. Here, the *imbalance* is defined as the difference between the most and the least loaded types, were the load of a given type is the ratio between the number of requests and the number of processes of that type.

Whenever LOAD-BALANCER is triggered, it resets all pending requests and marks them as unscheduled. Only those requests that are currently executed are not modified (and continue their execution), but their processes are terminated, while all other existing processes are canceled. Then LOAD-BALANCER analyzes which process types are required by the set of unscheduled requests. For each required type, a new process is created on a mover node. Then, if idle tape drives able to host a process still remain, a new process is created, matching the type of the most loaded type. This action is then repeated until no new process can be created.

Once LOAD-BALANCER has created new processes, it tries to map each unscheduled request R. If the file concerned by R is not currently used, or, if R is a read request, if the tapes containing the concerned file are not currently in use, R is mapped on the least loaded process matching its type. Otherwise, R is delayed and placed in the new waiting list.

5.6 Performance evaluation

In order to assess the performance of each I/O policy, and the behavior of our I/O request scheduling algorithm, we have simulated an entire platform resembling that of a current petascale supercomputer. We first describe the environmental framework. We then conduct experiments where all requests obey the same archival policy (RAIT, PARALLEL or VERTICAL), and we discuss the influence of file sizes on the performance. Based upon the results of these experiments, we evaluate a scenario mixing policies, which associates the best-suited policy to each file size category.

5.6.1 Experimental framework

We have developed our own simulator using SimGrid [90, 67], a discrete event simulator framework, in its 3.5 version. The platform is simulated using distributed processes running in parallel on multiple virtual hosts. Each component of the model is represented by such processes. For instance, I/O policies running concurrently on a single mover are simulated by parallel processes on a single host, whereas each tape drive is represented by a host and a dedicated process. The same holds for the main scheduling and the load balancing processes, which are running concurrently on a single host.

The simulated platform is depicted on Figure 5.4. The user process simulates the arrival of the requests in the system. Those requests are handled by the MAIN-SCHEDULER process, which may create new I/O processes on the mover nodes and assign tape drives to them. Unscheduled requests are handled by the LOAD-BALANCER process. Finally, the tape library controls multiple robotic arms dedicated to move the tapes back and forth from the library to the tape drives.

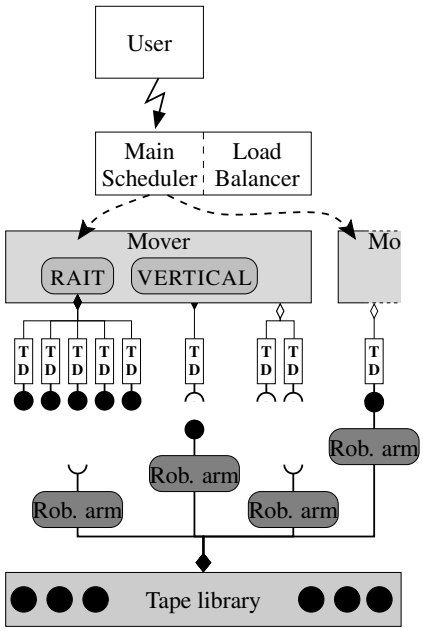


Figure 5.4: Model of the simulated platform

In the experiments, the platform is instantiated using one tape library managing 20 robotic arms and 500 tape drives. These tape drives are connected 10-by-10 to 50 mover nodes, responsible for handling I/O operations. These parameters match the size of today's petascale supercomputers.

Moreover, in order to simulate a typical workload running on such a supercomputer, we generated random workloads following a Poisson process with an arrival rate λ. File sizes are chosen according to

$X+Y$	$1+0$	$2+1$	$3+1$	$3+2$
p_{X+Y}	0.025	0.025	0.05	0.1

$X+Y$	$4+1$	$4+2$	$6+2$	$8+2$
p_{X+Y}	0.1	0.3	0.2	0.2

Table 5.1: Resiliency schemes used in the experiments.

a random log uniform distribution, which is a simple approximation of the lognormal distribution of file sizes observed in file systems [34] for large file sizes (files smaller than a few KB do not participate to the archival process). The type of the I/O operation is chosen uniformly between read and write. A new file is created in 90% of the cases if the request is a write operation, and an existing file is written again otherwise. Each request is provided with a resiliency scheme $X+Y$, which is randomly chosen among a set of representative schemes. These schemes and their respective probability are given in Table 5.1. Finally, for each file, the compression rate of data blocks C_D is chosen within $[1,3]$ while the EC blocks compression rate C_P belongs to $[1, C_D]$.

5.6.2 Results with a single policy

In a first step, only homogeneous scenarios are considered: requests may have different resiliency schemes but use only one I/O policy, either RAIT, PARALLEL or VERTICAL.

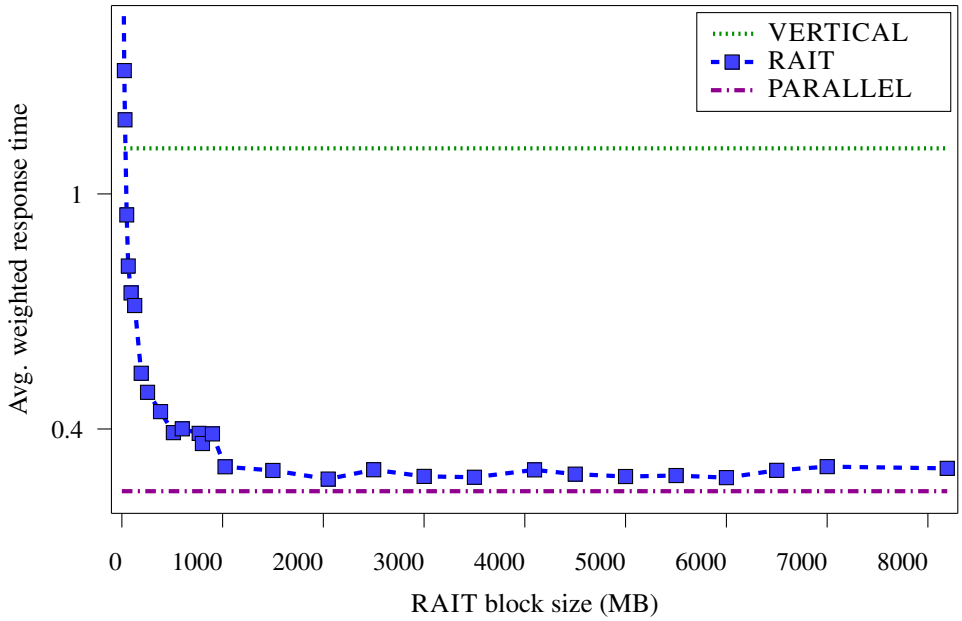


Figure 5.5: Impact of RAIT block size B on average weighted response time.

The first experiment aims at analyzing the impact of the block size B for the RAIT policy (contrarily to PARALLEL and VERTICAL, RAIT requires a block size to be tuned, since it impacts how data is written onto tape). The performance of RAIT is computed in terms of the average weighted response time. Request arrival rate is set to 180 requests per hour, and file sizes range from 1 GB to 1 TB, while B varies between 1 MB and 8 GB.

Results presented on Figure 5.5 show that B has a significant impact on the average weighted re-

sponse time of RAIT. For the smallest values, RAIT performs worse than VERTICAL whereas it almost ties PARALLEL for larger values. With $B = 1$ MB, RAIT is about 80 times slower than with $B = 192$ MB, while the performance is constant between 192 MB and 8 GB. Altogether, this experiment outlines the importance of B value for RAIT policy, which clearly benefits from large enough blocks in order to offer competitive performance. In all the following experiments, B will be chosen according to these results.

The next experiment intends to compare the performance of all I/O policies for various arrival rates. The objective is twofold : assessing the average performance of each policy, and in particular, determining the maximum arrival rate which can be handled by each policy. File sizes are chosen among three subsets: *small* file sizes range from 10 MB to 1 GB, *medium* file sizes from 1 GB to 100 GB, and *large* file sizes from 1 TB to 100 TB. Arrival rates are chosen with respect of these files sizes between a few requests per hour to hundreds of requests per hour.

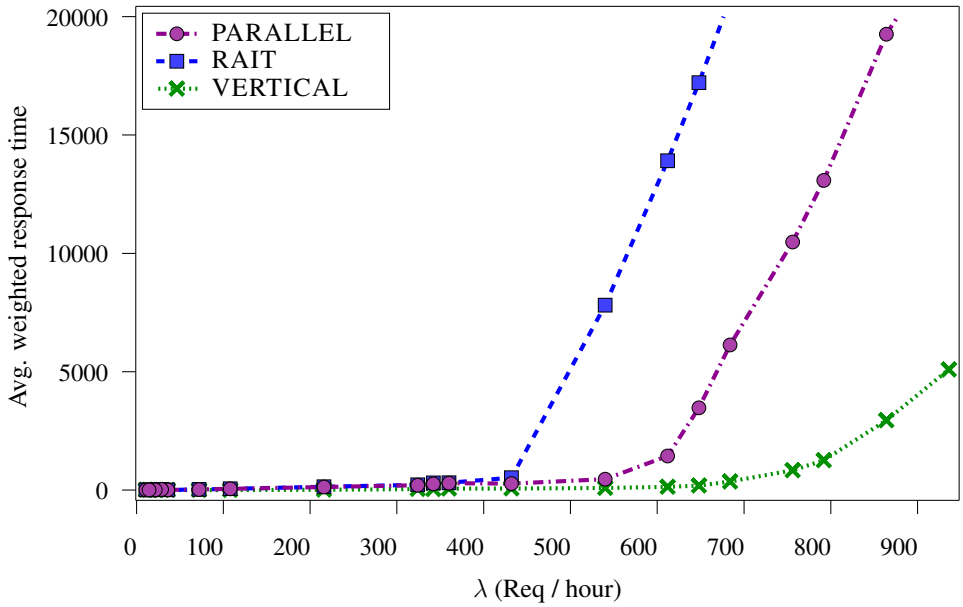


Figure 5.6: Impact of arrival rates on the average weighted response time for small files ($B = 16$ MB for RAIT).

For small files, as depicted in Figure 5.6, the best average weighted response time is offered by VERTICAL, which can sustain higher arrival rates than the other policies. RAIT is able to serve requests with reasonable response time for rates lower than 420 requests per hour. PARALLEL performs better than RAIT since it can keep up with rates lower than 600 requests per hour. The best policy in that case is VERTICAL, which can tolerate rates up to 800 requests per hour. This is due to the fact that with small files, extra latencies paid by data parallel policies (RAIT and PARALLEL) are not negligible. Also, recall that more files can be written concurrently throughout the entire system with VERTICAL. Contrarily to its contenders, VERTICAL can pipeline a large number of files before having to write EC tapes, thereby increasing average performance.

For medium sizes, results presented on Figure 5.7 show that, as expected, the system benefits more from data parallelism, latencies being now negligible. PARALLEL dominates other policies in this case, being able to handle up to 150 requests per hour while RAIT gets overloaded with arrival rates higher than 95 requests per hour. The extra tape drives used by RAIT as well as extra latencies when reading data have a significant impact on the average weighted response time. In that case, VERTICAL

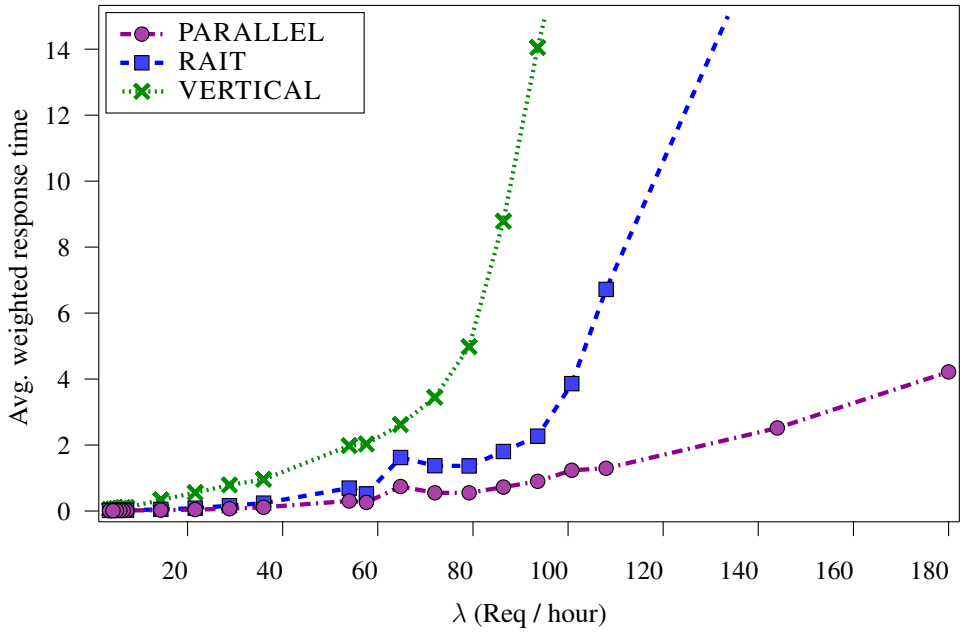


Figure 5.7: Impact of arrival rates on the average weighted response time for medium-size files ($B =$ 256 MB for RAIT).

is the worst policy since it can only sustain up to 80 requests per hour.

Finally for large files, results presented on Figure 5.8 outline a behavior similar to that observed for medium sized files. PARALLEL represents the best policy in terms of average weighted response time. Using this policy, the system is able to cope with 1.3 requests per hour while the second competitor, RAIT, can only serve up to 1 request per hour without being overloaded. Unsurprisingly, VERTICAL suffers from the lack of parallelism with these huge files, and can only bear 0.1 request per hour.

As a conclusion, these experiments show that PARALLEL offers the best results overall. RAIT outperforms VERTICAL whenever files are large enough. The fact that PARALLEL dominates the other solutions means not only that data parallelism is crucial, but also (and less expectedly) that balancing parity across tapes (as in RAIT) has a negligible impact compared to that of enforcing data sequentiality.

The next experiment aims at measuring how the difference of compressibility between data and EC blocks affects the overall performance of each I/O policy. Indeed, neither PARALLEL nor VERTICAL can balance the EC blocks across tapes, possibly leading to more tape loads/unloads for tapes dedicated to EC blocks. The underlying objective of this experiment is therefore to assess the impact of this pitfall on the average weighted response time. In order to do so, the arrival rate is set to 65 requests per hour, RAIT block size is $B = 192\ MB$, and file sizes range between 1 GB and 1 TB. The compression rate of data is set to be 3x. The EC compression rate varies between 1x (no compression) and 3x (same compression rate as data).

As shown on Figure 5.9, I/O policies do not display the same sensitivity to EC compression rate. Both PARALLEL and RAIT are more affected by compression than VERTICAL. As a matter of a fact, while the later almost maintains its performance regardless of EC compression rate, the average weighted response time displayed by RAIT and PARALLEL increases when EC blocks are less compressed. The lower sensitivity of VERTICAL to EC compression comes from the fact that several requests are served before writing EC blocks, because these are aggregated. This is not the case for PARALLEL

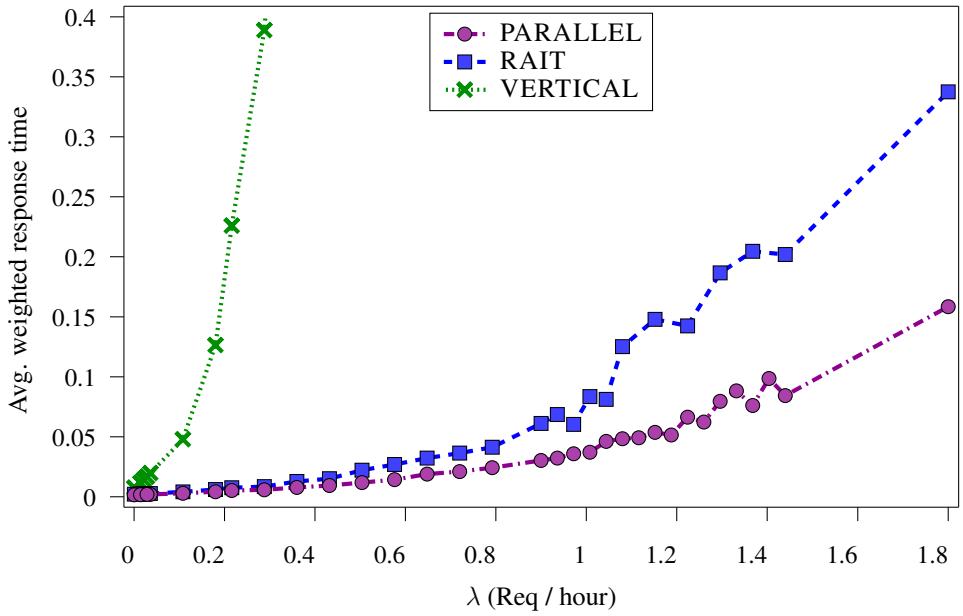


Figure 5.8: Impact of arrival rates on the average weighted response time for large files ($B = 8$ GB for RAIT).

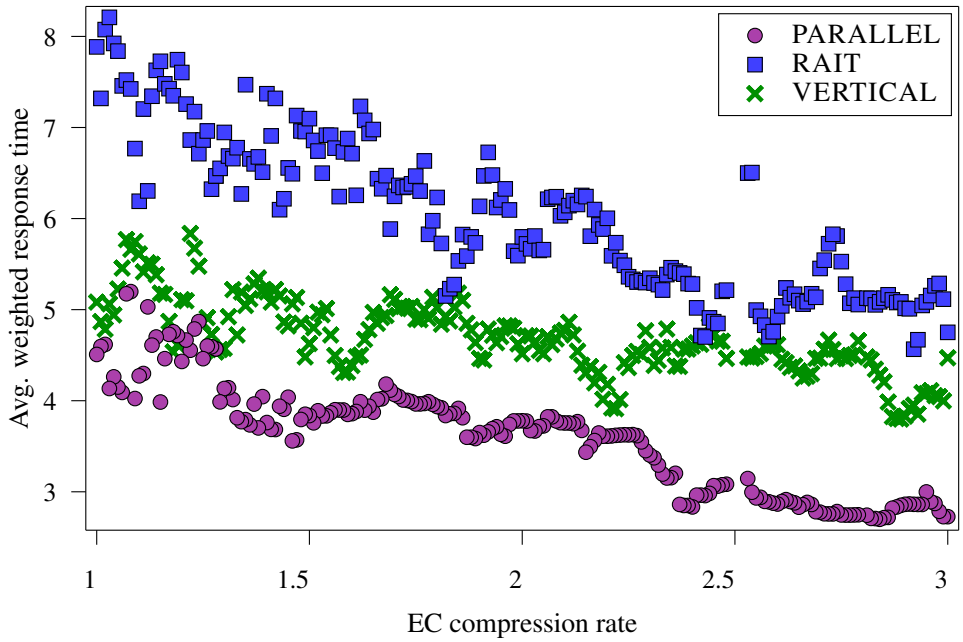


Figure 5.9: Impact of EC blocks compression rates on performance.

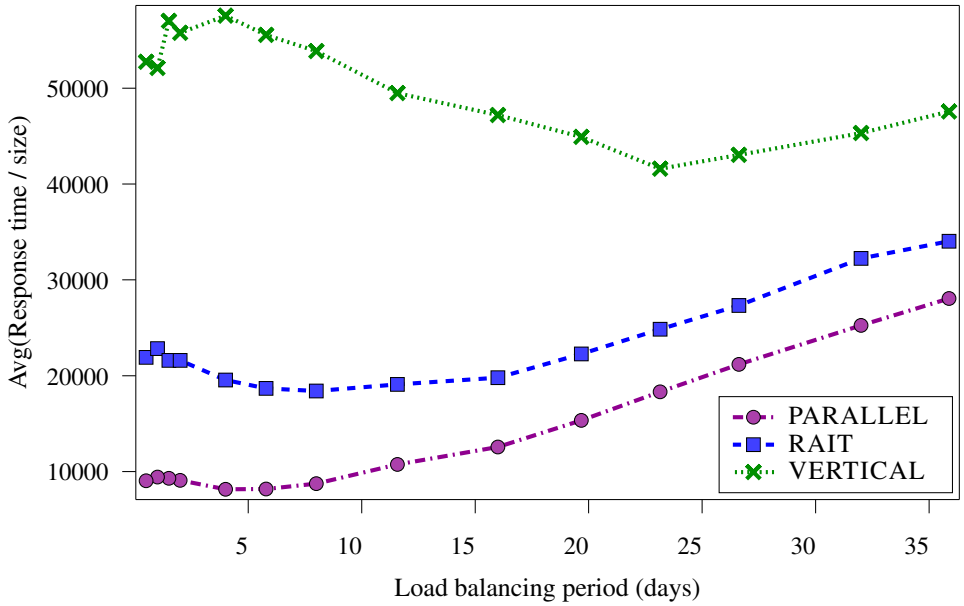


Figure 5.10: Impact of LOAD-BALANCER period on performance.

and RAIT.

Moreover, with both PARALLEL and VERTICAL, data tapes are always entirely filled, and EC blocks are written onto dedicated tapes. When EC blocks are far less compressed than data, tapes dedicated to EC are filled faster and require more frequent EJECT/LOAD operations. This also explain the higher sensitivity of PARALLEL to EC compression. On the contrary, with RAIT, the difference of compression between data and EC blocks degrades the balance of tape occupancy, leading to extra EJECT/LOAD operations on every tape drive (with RAIT, when a tape is filled, all loaded tapes are ejected).

Altogether, this experiment shows that both RAIT and PARALLEL display higher sensitivity to compression than VERTICAL. Whenever data is highly compressible, VERTICAL ties PARALLEL.

The following experiment focuses on the evaluation of the impact of the load balancing period MINLBINTERVAL on the average weighted response time. Indeed, this setting might significantly influence the behavior of the system under intensive workloads. Therefore, MINLBINTERVAL needs to be precisely tuned in order to fully exploit the archival architecture. In this experiment, file sizes are chosen between 1 GB and 1 TB, RAIT block size is $B = 192$ MB, while the mean arrival rate is set to $\lambda = 65$ requests per hour. Results depicted on Figure 5.10 show that the load balancing period indeed has a significant impact on performance: all three policies reach a minimum average weighted response time for a particular value of MINLBINTERVAL. Interestingly, the best value of MINLBINTERVAL is not the same for every policy: although both RAIT and PARALLEL perform better when LOAD-BALANCER is called at most every 6 days, VERTICAL benefit from a significantly higher value: 24 days. This is due to the fact that VERTICAL often has more pending requests than the other policies, and requires more time to serve each request. Remember that whenever an I/O process is destroyed by LOAD-BALANCER, loaded tapes are ejected. Therefore, calling the load balancing process too frequently may cause (in the worst case) tapes to be unloaded after each request has been served, leading to higher response times. All in all, this experiment shows that the load balancing period MINLBINTERVAL need to be precisely tuned in order to fully benefit from the parallel storage system.

5.6.3 Results with multiple policies

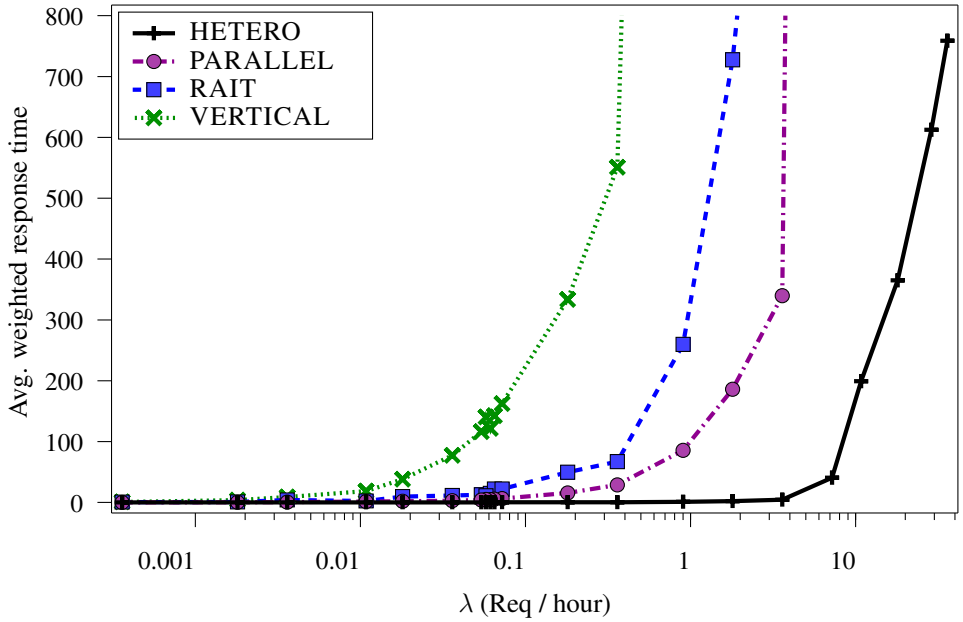


Figure 5.11: Impact of arrival rates on the average weighted response time (with small, medium-size and large file together, and $B = 512$ MB for RAIT).

Based on the previous results, a novel strategy using multiple policies is introduced: HETERO. As seen above, VERTICAL represents the best solution when writing small files, while PARALLEL is the best choice for larger files. In the following experiment, the I/O policy used to write a file is now dynamically chosen by the system, based on its size. The purpose here is to fully benefit from both policies in order to enhance the overall performance of the storage system. In order to assess the corresponding improvement, the impact of the arrival rate on the average weighted response time is again evaluated. File sizes are now chosen among a broader range: from 10 MB to 10 TB. A file smaller than 1 GB will be processed using VERTICAL, while larger files will use PARALLEL.

Results depicted on Figure 5.11 show that the HETERO strategy clearly outperforms all single policy strategies. HETERO can handle up to 7 requests of any size per hour while the best single policy strategy, PARALLEL, is limited to 0.6 requests per hour. The single strategy using VERTICAL does not perform well, since it is able to maintain a reasonable average weighted response time until arrival rate reaches 0.003 requests per hour. RAIT does better with a maximum of 0.3 requests per hour.

Altogether, this experiment shows that a strategy using multiple policies, carefully choosing the I/O policy that will be used to handle a file based on its size, brings a dramatic performance increase. The performance is sustained at significantly higher rates than with any singly policy strategy.

5.7 Conclusion

In this chapter, we have first discussed the well-known RAIT policy for tape archival on a petascale supercomputer, and we have identified its shortcomings. We have introduced two new I/O policies, PARALLEL and VERTICAL, that either reduce file fragmentation, or increase the number of requests that can be served simultaneously, or both. Contrarily to RAIT which requires to carefully choose a blocksize, the new policies do not require any tuning.

We have conducted a comprehensive set of experiments to assess the performance of the three RAIT, PARALLEL and VERTICAL policies. We observed that for small files, VERTICAL provides the best weighted response time, while for medium-size and large files, PARALLEL is the clear winner. This has led us to propose a heterogeneous solution mixing policies (VERTICAL for small files, PARALLEL otherwise). Altogether, this latter approach provides a dramatic ten-fold improvement over each policy taken separately.

We hope that the lessons learnt in this study will help guide the final design decisions of the BLUE WATERS supercomputer, and more generally, of future large-scale platforms that will require even larger storage capacities, and always more efficient archival scheduling policies.

Conclusion

Throughout this thesis, we have focused on designing memory-aware algorithms and schedules tailored for hierarchical memory architectures. Nowadays, these memory layouts can be found from within the heart of a multicore processor, up to the storage architectures of larger-scale platforms and supercomputers. During this thesis, we have studied several platforms of various scales, in order to assess the impact of such memory architectures on performance and peak memory usage. Our main contributions are recalled in the following paragraphs.

Complexity analysis and performance evaluation of matrix product on multicore architectures

The first contribution of this thesis is the complexity analysis and performance evaluation of matrix product on multicore architectures. By introducing a simple and yet realistic model for multicore hierarchical memory layout, we were able to extend lower bounds on cache misses. We have introduced cache-aware matrix product algorithms tailored for multicore processors each focusing on a specific level of cache. These result in a CCR close to their respective lower bound. We have also realized a tradeoff between shared and distributed cache misses through the overall data access time, and proposed an algorithm minimizing this objective.

Another contribution is the experimental validation of this theoretical model, through simulation and using actual hardware platforms. Indeed, thanks to a dedicated cache simulator, we have assessed the impact of cache management policy on the number of cache misses. We have also verified that reserving half of the cache for prefetching alleviates the impact of *LRU* replacement policy and all algorithms have been validated using various cache sizes. Moreover, we have compared the effective performance of all algorithms on actual multicore platforms by conducting an extensive set of experiments. Even though we have obtained mixed results with CPU platforms, we have discussed how the intricate and opaque mechanisms they use affect performance. For instance, due to intensive prefetching, cache misses have a lesser impact than predicted. On the contrary, on GPU platforms, we have nicely demonstrated both the accuracy of our model and the efficiency of our architecture-independent partitioning approach. Low-latency memories such as those provided by GPUs are much more promising devices than general purpose *LRU* caches for the tiling and partitioning strategies that lie at the heart of state-of-the-art linear algebra kernels.

Tiled QR factorization

In this chapter, we have proposed two new algorithms for tiled QR factorization: FIBONACCI and GREEDY. These algorithms exhibit a higher parallelism than that of state-of-the-art implementations. An important contribution is the accurate estimations for the length of their critical path. We have proven that they were asymptotically optimal for a wide class of matrix shapes, including all cases where the number of tile rows p and tile columns q are proportional, $p = \lambda q$, $\lambda \geq 1$. To the best of

our knowledge, this proof is the first complexity result in the field of tiled algorithms, and it lays the theoretical foundations for a comparative study of tiled algorithms.

Comprehensive experiments on multicore platforms confirm the superiority of the new algorithms for $p \times q$ matrices, as soon as, say, $p \geq 2q$. This holds true when comparing not only with previous algorithms using TT (*Triangle on top of triangle*) kernels, but also with all known algorithms based on TS (*Triangle on top of square*) kernels. Given that TS kernels offer more locality, and benefit from better elementary arithmetic performance, than TT kernels, the better performance of the new algorithms is even more striking, and further demonstrates that a large degree of a parallelism was not exploited in previously published solutions.

Scheduling streaming applications on a complex multicore platform

We have studied the scheduling of streaming applications on a heterogeneous platform: the IBM Bladecenter QS 22, made of two heterogeneous multicore Cell processors. The first contribution is a realistic and yet tractable model of the Cell processor. We have used this model to express the optimization problem of finding a mapping with maximal throughput. We have proven this problem to be NP-complete, and have designed a formulation of the problem as a mixed linear program. Solving this linear program with appropriate tools allowed us to compute a mapping with optimal throughput. We have also proposed a set of scheduling heuristics exhibiting a lower complexity.

Another important contribution is the experimental validation of both the model and our strategies. To this end, we have implemented a complete scheduling framework to deploy streaming applications on the QS 22, which is available for public use. It allows the user to deploy any streaming application, described by a potentially complex task graph, on a QS 22 platform or single Cell processor, given any mapping of the application to the platform. Using this scheduling framework, we have been able to perform a comprehensive experimental study of all our scheduling strategies. We have shown that our MIP strategy usually reaches 90% of the throughput predicted by the linear program, and has a good and scalable speed-up when using up to 16 SPEs. Some of the proposed heuristics, in particular DELEGATE, are also able to reach very good performance. Altogether, we have demonstrated that even though scheduling a complex application on a heterogeneous multicore processor is a challenging task, scheduling tools can help to achieve good performance.

On optimal tree traversals for sparse matrix factorization

We have discussed how to traverse the nodes of a tree-shaped workflow so as to optimize the memory used in a two-level memory system. We have investigated two main problems. In the MINMEMORY problem, the aim is to minimize the memory requirement, while in the MINIO problem, the aim is to minimize the I/O volume, given a limited memory. Our motivating application was the multifrontal method of sparse matrix factorization in which elimination (or assembly) trees are used to reorganize the computations.

Our first contribution is an exact algorithm for the MINMEMORY problem, which runs faster than the reference alternative of Liu [69]. The current state of the art software for sparse matrix factorization finds the best postorder as a solution to the MINMEMORY problem. This is done both for convenience and for the in-core memory requirement. We have investigated the relevance of this choice, and concluded that in most practical cases, the minimum memory requirement due to a postorder is usually close to the optimal one (in a large set of instances we have seen at most 18% increase with respect to the memory minimizing traversal). However, we also showed that on general trees, the best postorder can result in memory requirements that are arbitrarily large.

The MINIO problem has been proven to be NP-complete, as well as some of its variations (for example, we have shown that finding the postorder traversal that minimizes the I/O volume is NP-complete). We have proposed heuristics for the problem and have thoroughly evaluated them through experiments. Our experiments are based on highly optimized versions of three tree traversal algorithms, and precisely assess the quality of each proposed algorithm and I/O heuristic.

Altogether, we have shown that our $MinMem$ algorithm outperforms the running time of Liu's exact algorithm, but we have also observed that it was less suited for out-of-core execution.

Comparing archival policies for BLUE WATERS

All previous chapters focus on taking into consideration memory hierarchy in order to increase performance. Chapter 5 rather focuses on the performance of hierarchical storage systems. Our first contribution is the discussion of the state-of-the-art RAIT policy and the identification of its shortcomings. We have also introduced two novel I/O policies, PARALLEL and VERTICAL, either reducing file fragmentation, or increasing the number of requests that can be served simultaneously, or both. These new policies do not require any tuning since that, unlike RAIT, they do not require to carefully choose a block-size.

We have conducted a comprehensive set of experiments to assess the performance of the three archival policies. Either VERTICAL or PARALLEL policy is best suited to respectively small, or from medium up to large files. This observation led us to introduce a heterogeneous solution mixing policies (VERTICAL for small files, PARALLEL otherwise), providing a ten-fold improvement over each policy taken separately.

Altogether, this study should be considered as a tool which may help guide the final design decisions of the BLUE WATERS supercomputer, and more generally, of future large-scale platforms. Indeed, such platforms will require even larger storage capacities, and always more efficient archival scheduling policies.

Perspectives

In this thesis, we have studied several problems in the context of hierarchical memory architectures. There are multiple possible extensions to this work.

Multicore algorithms

On the short term, we would like to run comparisons of matrix product algorithms on larger scale platforms, with 32 or 64 cores. Indeed, on such platforms, memory accesses are likely to have a higher cost and might significantly impact performance. The need of memory aware algorithms therefore becomes more important. Regarding tiled QR factorization, several directions seems promising. First, using rectangular tiles instead of square tiles could lead to efficient algorithms, with more locality and still the same potential level of parallelism. Another extension would be to refine the model in order to account for communications, and extending it to fully distributed architectures. It would lay the ground to the design of MPI implementations of the new algorithms, unleashing their high level of performance on larger platforms.

On the longer term, we envision to design efficient linear algebra algorithms for clusters of multicores. Indeed, we expect yet another level of hierarchy (or tiling) in the algorithmic specification to be required in order to match the additional complexity of such platforms. Such an approach could also benefit to streaming applications. The model of the QS 22 could indeed be refined to more complex

platforms, like a cluster of QS 22, of other multicore platform. Furthermore, the increasing number of cores embedded within these multicore (or many-core) platforms will inevitably increase the failure rate and introduce variations in processor speeds. The design of robust algorithms, for both matrix product and tiled QR factorization, and capable of achieving efficient performance despite such variations, or even resource failures, is a challenging but crucial task to fully benefit from future platforms with a huge number of cores.

Scheduling with two memory levels

There are several direct extensions to the study of optimal tree traversals for sparse matrix factorization. Many problems are still waiting to be tackled with respect to the MINIO problem. Among those, finding a lower bound for the minimum I/O volume when a fixed amount of main memory is permitted seems to be very promising. Indeed, this would allow to assess the absolute performance of the heuristics. Furthermore, though such a result seems out of reach for general traversals, derive an approximation algorithm for postorder traversals would be a very interesting contribution. More generally, multicore platforms with non-uniform memory access are now becoming widespread. They introduce new levels of hierarchy in the whole memory system, from distributed caches to shared caches, to main memory, and to disk. Such platforms call for re-designing the whole computational chain of sparse matrix factorization, by introducing memory-aware computational kernels at every level.

A longer term direction would be to extend this study to more complex applications, modeled by Directed Acyclic Graphs (DAGs). Although both MINIO and MINMEMORY problems have been shown NP-complete in this context, finding efficient heuristics, or even better, approximation algorithms, would significantly impact a wider range of applications.

Moreover, allowing for parallel tree traversals would be a major contribution. Indeed, parallel explorations with a single shared memory are required to exploit the computing power offered by multicore processor or even larger shared memory or Non Uniform Memory Accesses (NUMA) platforms, and would significantly impact sparse matrix factorization. A more ambitious study would be to handle parallel traversals with distributed memories, thus allowing for larger-scale platforms through MPI implementations of the multifrontal method for sparse matrix factorization. Such a study would also greatly benefit to hybrid methods, where multiple lower subtrees are processed in parallel using the multifrontal method, before that the upper part of the main tree is processed using iterative methods.

Archival policies for BLUE WATERS

The comparative study of archival policies for BLUE WATERS has a natural and important extension, namely to collect real application traces. Such real-world traces would give the opportunity to better assess the performance of all three policies.

Moreover, on the long term, hard disk drives are likely to be considered as a level of cache for the archival system, thus calling for the design of an ambitious data management policy. In this context, real-life application traces would be a strong prerequisites for an “automated” policy, moving data between hard disks and magnetic tapes. Analyzing these traces, and designing a hard disk management policy based on this information, would represent a dramatic advance toward seamlessly exposing hierarchical storage architectures to the user.

Bibliography

[1] Emmanuel Agullo, Camille Coti, Jack Dongarra, Thomas Herault, and Julien Langou. QR factorization of tall and skinny matrices in a grid computing environment. In *IPDPS'10: Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium*. IEEE Computer Society Press, 2010.

[2] Emmanuel Agullo, Jack Dongarra, Rajib Nath, and Stanimire Tomov. A fully empirical autotuned dense QR factorization for multicore architectures. Technical Report 242, LAPACK Working Note, March 2011.

[3] Emmanuel Agullo, Bilel Hadri, Hatem Ltaief, and Jack Dongarra. Comparative study of one-sided factorizations with multiple software packages on multi-core hardware. In *SC'09: Proceedings of the 2009 ACM/IEEE conference on Supercomputing*, pages 1–12. IEEE Computer Society Press, 2009.

[4] Ishfaq Ahmad and Yu-Kwong Kwok. On exploiting task duplication in parallel program scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 9(9):872–892, 1998.

[5] M. Ålind, M. Eriksson, and C. Kessler. BlockLib: a skeleton library for Cell broadband engine. In *IWMSE'08: Proceedings of the First International Workshop on Multicore Software Engineering*, pages 7–14. ACM Press, 2008.

[6] AMD. ATI stream technology. Official website, AMD, 2010.

[7] AMD Fusion. http://fusion.amd.com.

[8] G. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS'67: Proceedings of the 1967 AFIPS spring Joint Computer Conference*, pages 483–485, New York, NY, USA, 1967. ACM Press.

[9] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[10] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel Computing*, 32(2):136–156, 2006.

[11] Cleve Ashcraft and Roger Grimes. The influence of relaxed supernode partitions on the multifrontal method. *ACM Transactions on Mathematical Software*, 15(4):291–309, 1989.

[12] Kevin J. Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Mike Lang, Scott Pakin, and Jose C. Sancho. Entering the petaflop era: the architecture and performance of roadrunner. In *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11. IEEE Computer Society Press, 2008.

[13] O. Beaumont, A. Legrand, L. Marchal, and Y. Robert. Steady-state scheduling on heterogeneous clusters. *International Journal of Foundations of Computer Science*, 16(2):163–194, 2005.

[14] Olivier Beaumont and Loris Marchal. Steady-state scheduling. In *Introduction to Scheduling*, pages 159–186. Chapman and Hall/CRC Press, 2010.

[15] Pieter Bellens, Josep M. Pérez, Felipe Cabarcas, Alex Ramírez, Rosa M. Badia, and Jesús Labarta. CellSs: Scheduling techniques to better exploit memory hierarchy. *Scientific Programming*, 17(1-2):77–95, 2009.

[16] Michael D. Beynon, Tahsin M. Kurç, Ümit V. Çatalyürek, Chialin Chang, Alan Sussman, and Joel H. Saltz. Distributed processing of very large datasets with datacutter. *Parallel Computing*, 27(11):1457–1478, 2001.

[17] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM Press, 1997.

[18] Susan Blackford and Jack J. Dongarra. Installation guide for LAPACK. Technical Report 41, LAPACK Working Note, June 1999. originally released March 1992.

[19] Blue Waters. http://www.ncsa.illinois.edu/BlueWaters.

[20] François Broquedis, Jérôme Clet Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In *PDP'10: Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*. IEEE Computer Society Press, 2010.

[21] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. Parallel tiled QR factorization for multicore architectures. *Concurrency and Computation: Practice and Experience*, 20(13):1573–1590, 2008.

[22] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, 2009.

[23] L. E. Cannon. *A cellular computer to implement the Kalman filter algorithm*. PhD thesis, Montana State University, 1969.

[24] Texas Advanced Computing Center. GotoBLAS2. http://www.tacc.utexas.edu/tacc-projects/gotoblas2/.

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

[26] ClearSpeed technology. http://www.clearspeed.com/technology/index.php.

[27] Michel Cosnard, Jean-Michel Muller, and Yves Robert. Parallel QR decomposition of a rectangular matrix. *Numerische Mathematik*, 48:239–249, 1986.

[28] Michel Cosnard and Yves Robert. Complexity of parallel QR factorization. *Journal of the ACM*, 33(4):712–723, 1986.

[29] ILOG CPLEX: High-performance software for mathematical programming and optimization. http://www.ilog.com/products/cplex/.

[30] William J. Dally, Francois Labonte, Abhishek Das, Patrick Hanrahan, Jung-Ho Ahn, Jayanth Gummaraju, Mattan Erez, Nuwan Jayasena, Ian Buck, Timothy J. Knight, and Ujval J. Kapasi. Merrimac: Supercomputing with streams. In *SC'03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 35, Washington, DC, USA, 2003. IEEE Computer Society Press.

[31] J. W. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Communication-avoiding parallel and sequential QR and LU factorizations: theory and practice. Technical Report arXiv:0806.2159, 2008.

[32] James Demmel, Mark Hoemmen, Marghoob Mohiyuddin, and Katherine Yelick. Minimizing communication in sparse matrix solvers. In *SC'09: Proceedings of the 2009 ACM/IEEE conference on Supercomputing*, pages 1–12. IEEE Computer Society Press, 2009.

[33] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *MOS Mathematical Programming*, 91(2):201–213, 2002.

[34] Allen B. Downey. The structural cause of file size distributions. In *MASCOTS'01: Proceedings of the 9th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 361–370, 2001.

[35] A.L. Drapeau and R.H. Katz. Striped tape arrays. In *MASS'93: Proceedings of the 12th IEEE Symposium on Mass Storage Systems*, pages 257 –265. IEEE Computer Society Press, April 1993.

[36] Iain S. Duff and John K. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Transactions on Mathematical Software*, 9:302–325, 1983.

[37] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Memory - sequoia: programming the memory hierarchy. In *SC'06: Proceedings of the ACM/IEEE conference on Supercomputing*, page 83. IEEE Computer Society Press, 2006.

[38] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *FOCS'99: Proceedings of the 40th IEEE Symposium on Foundations of Computer Science*, pages 285–298. IEEE Computer Society Press, 1999.

[39] Matthieu Gallet, Loris Marchal, and Frédéric Vivien. Efficient scheduling of task graph collections on heterogeneous resources. In *IPDPS'09: Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*. IEEE Computer Society Press, 2009.

[40] M. R. Garey and D. S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[41] John R. Gilbert, Thomas Lengauer, and Robert Endre Tarjan. The pebbling problem is complete in polynomial space. *SIAM Journal on Computing*, 9(3):513–524, 1980.

[42] John R. Gilbert, Gary L. Miller, and Shang-Hua Teng. Geometric mesh partitioning: Implementation and experiments. *SIAM Journal on Scientific Computing*, 19(6):2091–2110, 1998.

[43] Roberto Giorgi, Zdravko Popovic, and Nikola Puzovic. Dta-c: A decoupled multi-threaded architecture for CMP systems. In *SBAC-PAD'07: Proceedings of the 19th IEEE International Symposium on Computer Architecture and High Performance Computing*, pages 263–270. IEEE Computer Society Press, 2007.

[44] Roberto Giorgi, Zdravko Popovic, and Nikola Puzovic. Exploiting DMA to enable non-blocking execution in decoupled threaded architecture. In *IPDPS'09: Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*. IEEE Computer Society Press, 2009.

[45] Bilel Hadri, Hatem Ltaief, Emmanuel Agullo, and Jack Dongarra. Enhancing parallelism of tile QR factorization for multicore architectures, 2009.

[46] Bilel Hadri, Hatem Ltaief, Emmanuel Agullo, and Jack Dongarra. Tile QR factorization with parallel panel processing for multicore architectures. In *IPDPS'10: Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium*. IEEE Computer Society Press, 2010.

[47] X. Hang. A streaming computation framework for the Cell processor. Master's thesis, Massachusetts Institute of Technology, 2007.

[48] T. Hartley and U. Catalyurek. A component-based framework for the Cell broadband engine. In *IPDPS'09: Proceedings of Workshops of the 23rd IEEE International Parallel and Distributed Processing Symposium*. IEEE Computer Society Press, 2009.

[49] Stephen L. Hary and Fusun Ozguner. Precedence-constrained task allocation onto point-to-point networks for pipelined execution. *IEEE Transactions on Parallel and Distributed Systems*, 10(8):838–851, 1999.

[50] Jason Hick. HPSS in the Extreme Scale. In *Report to DOE Office of Science on HPSS in 2018-2022, LBNL Paper LBNL-3877E, http://www.escholarship.org/uc/item/4wn1s2d3*. Lawrence Berkeley National Laboratory, 2009.

[51] B.K. Hillyer, R. Rastogi, and A. Silberschatz. Scheduling and data replication to improve tape jukebox performance. In *ICDE'99: Proceedings of the 15th IEEE International Conference on Data Engineering*, pages 532 –541. IEEE Computer Society Press, March 1999.

[52] P.D. Hoang and J.M. Rabaey. Scheduling of dsp programs onto multiprocessors for maximum throughput. *IEEE Transactions on Signal Processing*, 41(6):2225 –2235, 1993.

[53] J.-W. Hong and H.T. Kung. I/O complexity: The red-blue pebble game. In *STOC'81: Proceedings of the 13th ACM Symposium on Theory of Computing*, pages 326–333. ACM Press, 1981.

[54] Ellis Horowitz and Sartaj Sahni. Exact and approximate algorithms for scheduling nonidentical processors. *Journal of the ACM*, 23(2):317–327, 1976.

[55] James Hughes, Dave Fisher, Kent Dehart, Benny Wilbanks, and Jason Alt. HPSS RAIT Architecture. In *White paper of the HPSS collaboration, www.hpss-collaboration.org/documents/HPSS_RAIT_Architecture.pdf*, 2009.

[56] IBM software kit for multicore acceleration. http://www.ibm.com/chips/techlib/techlib.nsf/products/IBM_SDK_for_Multicore_Acceleration.

[57] Intel. Intel Math Kernel Library. http://software.intel.com/en-us/intel-mkl/, 2010.

[58] Intel. Intel VTune Performance Analyzer. http://software.intel.com/en-us/intel-vtune/, 2010.

[59] Dror Ironya, Sivan Toledo, and Alexander Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *Journal of Parallel and Distributed Computing*, 64(9):1017–1026, 2004.

[60] T. Johnson and S. Prabhakar. Tape group parity protection. In *MASS'99: Proceedings of the 16th IEEE Symposium on Mass Storage Systems*, pages 72 –79. IEEE Computer Society Press, 1999.

[61] James A. Kahle, Michael N. Day, H. Peter Hofstee, Charles R. Johns, Theodore R. Maeurer, and David J. Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4-5):589–604, 2005.

[62] L.J. Karam, I. AlKamal, A. Gatherer, G.A. Frantz, D.V. Anderson, and B.L. Evans. Trends in multicore dsp platforms. *IEEE Signal Processing Magazine*, 26(6):38 –49, 2009.

[63] G. Karypis and V. Kumar. *MeTiS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version 4.0*. University of Minnesota, Department of Comp. Sci. and Eng., Army HPC Research Center, Minneapolis, 1998.

[64] Michael Kistler, Michael Perrone, and Fabrizio Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Micro Magazine*, 26(3):10–23, 2006.

[65] Andreas Kleen. A NUMA API for LINUX. http://andikleen.de/, 2005.

[66] Jakub Kurzak, Hatem Ltaief, Jack Dongarra, and Rosa M. Badia. Scheduling dense linear algebra operations on multicore processors. *Concurrency and Computation: Practice and Experience*, 22(1):15–44, 2010.

[67] A. Legrand, L.Marchal, and H. Casanova. Scheduling Distributed Applications: The SIMGRID Simulation Framework. In *CCGrid'03: Proceedings of the Third IEEE International Symposium on Cluster Computing and the Grid*, pages 138–145. IEEE Computer Society Press, May 2003.

[68] Joseph W. H. Liu. On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Transactions on Mathematical Software*, 12(3):249–264, 1986.

[69] Joseph W. H. Liu. An application of generalized tree pebbling to sparse matrix factorization. *SIAM Journal of Algebraic Discrete Methods*, 8(3), 1987.

[70] Joseph W. H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11(1):134–172, 1990.

[71] Joseph W. H. Liu. The multifrontal method for sparse matrix solution: Theory and practice. *SIAM Review*, 34(1):82–109, 1992.

[72] Mercury technology. http://www.mc.com/technologies/technology.aspx.

[73] J.J. Modi and M.R.B. Clarke. An alternative Givens ordering. *Numerische Mathematik*, 43:83–90, 1984.

[74] NVIDIA. CUDA C Programming Guide. Technical documentation, NVIDIA, 2010.

[75] NVIDIA. CUDA Programming Model. Official website, NVIDIA, 2010.

[76] Michael A. Palis, Jing-Chiou Liou, and David S. L. Wei. Task clustering and scheduling for distributed memory parallel architectures. *IEEE Transactions on Parallel and Distributed Systems*, 7(1):46–55, 1996.

[77] Jean-Francois Pineau, Yves Robert, Frédéric Vivien, and Jack Dongarra. Matrix product on heterogeneous master-worker platforms. In *PPoPP'2008: the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 53–62. ACM Press, 2008.

[78] S. Prabhakar, D. Agrawal, A. El Abbadi, and A. Singh. Scheduling tertiary i/o in database applications. In *DEXA'97: Proceedings of the 8th International Workshop on Database and Expert Systems Applications*, pages 722 –727. IEEE Computer Society Press, September 1997.

[79] Irk Pruhs, Jiri Sgall, and Eric Torng. On-line scheduling. In J. Leung, editor, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, pages 15.1–15.43. CRC Press, 2004.

[80] IBM BladeCenter QS 22. http://www-03.ibm.com/systems/bladecenter/hardware/servers/qs22/.

[81] G. Quintana-Ortí, E. S. Quintana-Ortí, R. A. van de Geijn, F. G. Van Zee, and Ernie Chan. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Transactions on Mathematical Software*, 36(3), 2009.

[82] Trent Rolf. Cache organization and memory management of the Intel Nehalem computer architecture. Research report, University of Utah Computer Engineering, 2009.

[83] A.H. Sameh and D.J. Kuck. On stable parallel linear systems solvers. *Journal of the ACM*, 25:81–91, 1978.

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

[85] John E. Savage. *Models of Computation: Exploring the Power of Computing*. Addison-Wesley, 1997.

[86] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *FAST'02: Proceedings of the First USENIX Conference on File and Storage Technologies*, pages 231–244, 2002.

[87] R. Schreiber. A new implementation of sparse Gaussian elimination. *ACM Transactions on Mathematical Software*, 8(3):256–276, 1982.

[88] Ravi Sethi. Complete register allocation problems. In *STOC'73: Proceedings of the Fifth annual ACM Symposium on Theory of Computing*, pages 182–195. ACM Press, 1973.

[89] Ravi Sethi and J.D. Ullman. The generation of optimal code for arithmetic expressions. *Journal of the ACM*, 17(4):715–728, 1970.

[90] SimGrid. URL: `http://simgrid.gforge.inria.fr`.

[91] StreamIt project. `http://groups.csail.mit.edu/cag/streamit/index.shtml`.

[92] F. Suter. DAG generation program. `http://www.loria.fr/~suter/dags.html`.

[93] Danny Teaff, Dick Watson, and Bob Coyne. The architecture of the high performance storage system (hpss). In *MSST'95: Proceedings of the Fourth NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 28–30. IEEE Computer Society Press, 1995.

[94] William Thies. *Language and Compiler Support for Stream Programs*. PhD thesis, Massachusetts Institute of Technology, 2001.

[95] Sivan Toledo. A survey of out-of-core algorithms in numerical linear algebra. In *Proceedings of the DIMACS Workshop on External Memory Algorithms and Visualization*, pages 161–180. American Mathematical Society Press, 1999.

[96] R. Clint Whaley and Anthony M. Castaldo. Achieving accurate and context-sensitive timing for code optimization. *Software: Practice and Experience*, 38:1621–1642, December 2008.

[97] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52:65–76, April 2009.

[98] Q. Wu, J. Gao, M. Zhu, N.S.V. Rao, J. Huang, and S.S. Iyengar. On optimal resource utilization for distributed remote visualization. *IEEE Transactions on Computers*, 57(1):55–68, 2008.

Publications

Articles in international refereed conferences

[A1] Henricus Bouwmeester, Mathias Jacquelin, Julien Langou, and Yves Robert. Tiled QR factorization algorithms. In *Proceedings of the IEEE conference on Supercomputing 2011 (SC'11) (to appear)*. IEEE Computer Society Press, 2011.

[A2] Mathias Jacquelin, Loris Marchal, Yves Robert, and Bora Uçar. On optimal tree traversals for sparse matrix factorization. In *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS'11)*. IEEE Computer Society Press, 2011.

[A3] Matthieu Gallet, Mathias Jacquelin, and Loris Marchal. Scheduling complex streaming applications on the Cell processor. In *Proceedings of the Workshop on Multithreaded Architectures and Applications (MTAAP'10), in conjunction with the 24th International Parallel and Distributed Processing Symposium (IPDPS'10)*, pages 1–8. IEEE Computer Society Press, 2010.

[A4] Mathias Jacquelin, Loris Marchal, and Yves Robert. Complexity analysis and performance evaluation of matrix product on multicore architectures. In *Proceedings of the 38th IEEE International Conference on Parallel Processing (ICPP'09)*, pages 196–203. IEEE Computer Society Press, 2009.

Posters

[B1] Mathias Jacquelin. Memory-aware algorithms and scheduling techniques: from multi-core processors to supercomputers. In *Proceedings of the PhD Forum of the 25th International Parallel and Distributed Processing Symposium (IPDPS'11), Winner of the TCPP Best Poster Award.* IEEE Computer Society Press, 2011.

Research reports

[C1] Henricus Bouwmeester, Mathias Jacquelin, Julien Langou, and Yves Robert. Tiled QR factorization algorithms. Research Report RR-7601, INRIA, 2011.

[C2] Franck Cappello, Mathias Jacquelin, Loris Marchal, Yves Robert, and Marc Snir. Comparing archival policies for Blue Waters. Research Report RR-7583, INRIA, 2011.

[C3] Mathias Jacquelin, Loris Marchal, and Yves Robert. The impact of cache misses on the performance of matrix product algorithms on multicore platforms. Research Report RR-7456, INRIA, 2010.

[C4] Mathias Jacquelin, Loris Marchal, Yves Robert, and Bora Uçar. On optimal tree traversals for sparse matrix factorization. Research Report RRLIP2010-30, LIP, ENS Lyon, 2010.

[C5] Tudor David, Mathias Jacquelin, and Loris Marchal. Scheduling streaming applications on a complex multicore platform. Research Report RRLIP2010-25, LIP, ENS Lyon, 2010.

[C6] Matthieu Gallet, Mathias Jacquelin, and Loris Marchal. Scheduling complex streaming applications on the Cell processor. Research Report RRLIP2009-29, LIP, ENS Lyon, 2009.

[C7] Mathias Jacquelin, Loris Marchal, and Yves Robert. Complexity analysis and performance evaluation of matrix product on multicore architectures. Research Report RRLIP2009-9, LIP, ENS Lyon, 2009.

[C8] Loris Marchal Mathias Jacquelin and Yves Robert. Complexity analysis of matrix product on multicore architectures. Research Report RR2008-41, LIP, ENS Lyon, 2008.

Résumé :
L'ensemble de cette thèse s'articule autour de la conception d'algorithmes et d'ordonnancements de calculs adaptés aux architectures mémoire hiérarchiques, rencontrées notamment dans le contexte des processeurs multi-cœurs. Nous étudions plusieurs plates-formes de tailles différentes afin de mesurer l'impact de ces hiérarchies mémoire et de leur hétérogénéité sur les performances.

Dans un premier temps, nous nous intéressons aux performances du produit de matrices sur les processeurs multi-cœurs. En effet, c'est au cœur de ce type de noyau de calcul que résident les performances de la plupart des applications scientifiques. Nous adaptons donc le produit de matrices aux hiérarchies de caches des architectures multi-cœur. Nous présentons un modèle d'un tel processeur et calculons des bornes inférieures sur le volume de communication du produit de matrices. Nous introduisons trois algorithmes ayant pour but de minimiser ce volume de communication et validons leurs performances à travers une large campagne expérimentale.
Dans un deuxième temps, nous étudions un algorithme plus complexe : la factorisation QR, couramment utilisée dans de nombreuses applications scientifiques. Nous nous plaçons dans le contexte des matrices ayant significativement plus de lignes que de colonnes. Nous revisitons les algorithmes existants dans le but d'exploiter le niveau de parallélisme supplémentaire offert par les processeurs multi-cœurs. Nous évaluons la longueur de leurs chemins critiques et montrons que certains d'entre eux sont asymptotiquement optimaux. Enfin, nous analysons expérimentalement leurs performances.
Nous nous intéressons ensuite à une plate-forme multi-cœur hétérogène, le QS 22, à travers l'ordonnancement d'applications pipelinées. Nous établissons un modèle de la plate-forme fondé sur des mesures de performances expérimentales. Nous appliquons alors les techniques d'ordonnancement en régime permanent afin de maximiser le débit de traitement. Nous introduisons un programme linéaire mixte permettant d'obtenir un ordonnancement optimal, ainsi qu'un ensemble d'heuristiques. Nous validons de manière expérimentale les performances de l'ensemble des stratégies proposées.
Dans l'étude suivante, nous visons d'abord à minimiser la quantité de mémoire nécessaire à une application donnée. Nous nous concentrons sur les applications modélisées par des arbres, tels que ceux rencontrés lors de la factorisation de matrices creuses. La plate-forme étudiée est un système à deux niveaux de mémoire, les entrées/sorties correspondent aux transferts d'un niveau à un autre. Nous présentons un algorithme optimal, offrant de meilleures performances en pratique qu'un algorithme optimal existant. Nous montrons en outre qu'il existe des arbres tels que les parcours postfixes nécessitent arbitrairement plus de mémoire que la quantité optimale. Nous étudions ensuite le problème de la minimisation du volume d'E/S à quantité de mémoire donnée, et montrons que ce problème est NP-complet que le parcours soit restreint aux parcours postfixes ou non. Nous présentons plusieurs heuristiques elles-aussi validées expérimentalement.
Enfin, le dernier chapitre est dédié à la comparaison de politiques d'archivage pour BLUE WATERS. En effet, les hiérarchies mémoire sont également présentes au sein des systèmes de stockage des derniers supercalculateurs, tels que BLUE WATERS. Nous introduisons deux nouvelles politiques d'archivage améliorant les performances de la politique classique RAIT. Nous modélisons totalement la plate-forme et simulons son fonctionnement. Nous montrons alors que RAIT n'offre jamais de meilleures performances que les deux politiques proposées, et qu'une politique hétérogène mêlant ces deux nouvelles politiques offre des performances 10 fois supérieures aux autres politiques testées.

Mots-clés :
Multi-cœur, hiérarchies mémoire, ordonnancement, régime permanent, plates-formes hétérogènes, optimisation, heuristiques, algèbre linéaire, programmes linéaires, maximisation du débit, contraintes mémoire.

Abstract:

Throughout this thesis, we focus on designing memory-aware algorithms and schedules tailored for hierarchical memory architectures. Nowadays, these memory layouts can be found from within the heart of a multicore processor to the storage architectures of larger-scale platforms like supercomputers. Several platforms of various scale are studied in order to assess the impact on performance of such memory architectures.

We first study both the complexity and the performance of matrix product on multicore architectures. Indeed, dense linear algebra kernels are the key to performance for many scientific applications. We introduce a realistic but still tractable model of a multicore processor, and derive lower bounds on the communication volume. We adapt matrix product algorithm to multicore architectures by taking caches into account, leading to three algorithms. Hence, the focus is set on minimizing cache misses. We assess both model relevance and performance of algorithms through an extensive set of experiments, ranging from simulation to real implementation on a GPU.

We then target a more complex operation: the QR factorization of rectangular matrices composed of $p \times q$ tiles, where $p \geq q$. This dense linear algebra kernel lies at the foundation of many scientific applications. We thus revisit existing algorithms so as to better exploit the additional level of parallelism offered by multicore processors. Within this framework, we study the critical paths and performance of several algorithms and prove some of them to be asymptotically optimal. We conclude this study by an extensive set of experiments that show the superiority of the new algorithms for tall matrices.

In the next study, we focus on scheduling streaming applications onto a heterogeneous multicore platform, the QS 22. We experimentally evaluate communication performance within the QS 22 and introduce a model of the platform based upon these results. We then use steady-state scheduling techniques in order to maximize the throughput, that is the number of instances processed per time-unit. We then present a mixed integer programming (MIP) approach that allows to compute a mapping with optimal throughput, propose simpler heuristics, and experimentally assess the performance of all approaches on the QS 22.

We then focus on minimizing the amount of required memory for a given application. We study the traversal of tree-shaped workflows, which typically arise in sparse matrix factorization, and target a classical two-level memory system. In this context, I/O represent transfers from a memory to the other. We propose a new exact algorithm which is more efficient in practice than an existing optimal algorithm. We also show that there exist trees where commonly used postorder based traversals require arbitrarily larger amounts of main memory than the optimal one. We then study the problem of minimizing the I/O volume for a given memory, and show that it is NP-hard, both for postorder based and for arbitrary traversals. We provide a set of heuristics to solve this problem, and experimentally assess their performance on existing trees.

Finally, we compare archival policies for BLUE WATERS. Indeed hierarchical memory architectures are also found into the storage system of cutting-edge supercomputers, like BLUE WATERS. We hence introduce two archival policies tailored for the tape storage system of BLUE WATERS and adapt the well known RAIT strategy. We provide an analytical model of the tape storage platform, and use it to assess and discuss the performance of the three policies through simulation. We show that RAIT is always outperformed by either VERTICAL or PARALLEL, and introduce the HETERO policy which uses the two latter policies, bringing a ten-fold performance improvement.

Keywords:

Multicore, memory hierarchy, scheduling, steady-state, heterogeneous platforms, optimization, heuristics, linear algebra, linear programs, throughput maximization, memory constraints.