Memory and data aware scheduling

Loris Marchal

MÉMOIRE D'HABILITATION À DIRIGER DES RECHERCHES

présenté le 30 mars 2018

à l'École Normale Supérieure de Lyon

par

Loris Marchal

Memory and data aware scheduling

(Ordonnancement tenant compte de la mémoire et des mouvements de données)

Devant le jury composé de :

Ümit Çatalyürek	*Georgia Tech., USA*	Rapporteur
Pierre Manneback	*Polytech-Mons, Belgique*	Examinateur
Alix Munier Kordon	*Univ. Paris 6*	Examinatrice
Cynthia Phillips	*Sandia Nat. Lab., USA*	Rapportrice
Yves Robert	*ENS Lyon*	Examinateur
Denis Trystram	*Grenoble INP*	Rapporteur

Preamble

In this habilitation thesis, I have chosen to present the work that I have done on memory-aware algorithms, and on related studies on data movement for matrix computations. This choice, needed to give coherence to this document, leaves out a number of contributions, mainly in the area of task scheduling, but still represents the main research domain which I contributed to, since my PhD ten years ago.

This manuscript is organized as follows: Chapter 1 presents the motivation and context of this work, and contains a short survey of the existing studies on memory-aware algorithms that are the basis of the work presented later. Then, my contributions are divided into two parts: Part I gathers the studies on memory-aware algorithms for task graph scheduling, while Part II collects other studies focusing on minimizing data movement for matrix computations. When describing my contributions, I have chosen to present the simplest algorithms in full details, and to give only the intuition for the most complex results. Most proofs are thus omitted. They are available in the referred publications or research reports. Chapter 10 concludes the document and open some perspectives. The appendices contain the bibliography (Appendix A) and my personal publications (Appendix B). Note that references to personal publications start with a letter corresponding to their type and are sorted chronologically by type, while other references are numbered alphabetically.

Last but not least, the "we" pronoun used in this document is not only employed out of politeness, but recalls that all these contributions originate from a collaborative work with all my co-authors. In particular, I owe a lot to my talented PhD students: Mathias Jacquelin, Julien Herrmann and Bertrand Simon.

Contents

List of Algorithms

Chapter 1

Motivation and context

Scientific applications are always looking for increased performance, not only to solve the same problems faster, but also to target larger problems. This is for example the case for numerical simulations: to get more accurate results, one needs to use finer mesh sizes and thus to solve larger systems. Most of the machines devoted to scientific computation, ranging from the small cluster to the supercomputer, are complicated to use efficiently. Firstly, they often embed heterogeneous computing cores, such as dedicated accelerators. Secondly, the memory available for computation is limited and hierarchically organized, leading to non-uniform memory access times. Lastly, supercomputers including a very large number of cores are subject to core failures. In this document, we concentrate on the second point, and more precisely on the fact that the amount of fast memory available for the computation is limited. We propose various scheduling and mapping algorithms that take the memory limit into account and aim at optimizing application performance. We mainly consider two types of applications: well-known linear algebra kernels, such as matrix multiplication, and applications that are described by a directed acyclic graph of tasks. In this latter case, the edges of the graph represent the dependencies between these tasks in the form of input/output data. Note that this task-based representation of computations is very common in the theoretical scheduling literature [34] and sees an increasing interest in High Performance Computing: to cope with the complexity and heterogeneity in modern computer design, many HPC applications are now expressed as task graphs and rely on dynamic runtime schedulers for their execution, such as StarPU [8], KAAPI [44], StarSS [72], and PaRSEC [16]. Even the OpenMP standard now includes task graph scheduling constructs [71]. We assume that the graph of tasks is known before the application and thus does not depend on the data itself. An example of application which is well modeled by a task graph is the QR-MUMPS software [3], which computes a QR factorization of a sparse matrix using multifrontal direct methods.

In this chapter, we first detail the motivation to optimize algorithms for a limited amount of memory. We then present two basic pebble games that were introduced in the literature to formalize the problem of computing with bounded memory. The first one deals with reducing the memory footprint of a computation, while the second one aims at reducing the amount of input/output operations in an out-of-core computation, that is, a computation that does not fit into the available memory. We then take the opportunity of this thesis to present a quick survey on the lower bounds and algorithms that were proposed in the literature for these two problems. Some of them were only published in specific application domains, and to the best of our knowledge, have never been gathered and presented under the same formalism, as done here. Our objective is to give the intuition of these results, which is why we provide their proof when they are short enough. The contributions of this habilitation thesis, presented in the following chapters, are based on these results and extend some of them.

1.1 Motivation

1.1.1 On the importance of data movement

Fast memory, that is the immediate storage space available for the computation, has always been a scarce resource in computers. However, the amount of available memory has largely increased in the previous decade, so that one could imagine that this limitation is about to vanish. However, when looking at the evolution of memory bandwidth and of processor speed, the future is not so bright: it has been acknowledged that the processing speed of a micro-processor (measured in floating point operations per seconds) has increased annually by 59% on average from 1988 to 2004, whereas the memory bandwidth (measured in byte per second) has increased annually only by 26%, and the memory latency (measured in seconds) only by 5% on the same period [48]. This means that the *time to process the data* is reduced at a much higher pace to the *time to move the data* from the memory to the processor. This problem is known as the "memory wall" in micro-processor design, and has been alleviated by the introduction of cache memories: the large but slow memory is assisted with a small but fast *cache*.

Thus, to overcome the decrease of the memory bandwidth/computing speed ratio, chips manufacturer have introduced a hierarchy of caches. This makes the performance of an application very sensitive to data locality: only the applications that exhibit a large amount of temporal locality (data reuse within a short time interval) or spatial locality (successive use of data from close storage locations) may take advantage of the speed of the processor. Furthermore, the cost of data movement is expected to become dominant also in terms of energy [80, 68]. Thus, avoiding data movement and favoring data reuse are crucial both to obtain good performance and to limit en-

ergy consumption. Numerous works have considered this problem. In this introductory chapter, we concentrate on some theoretical studies among them, namely how to design algorithms and schedules that have good and guaranteed performance in a memory-limited computing environment. In particular, we will not cover the numerous studies in hardware design or compilation techniques that target the same problem, because they are mainly orthogonal to the present approach.

1.1.2 Algorithms and memory usage

Let us consider here a very simple problem, the matrix product, to show how the design of the algorithm can influence the memory usage and the amount of data movement[1]. We consider two square $n \times n$ matrices A and B, and we compute their product $C = AB$.

Algorithm 1: NAIVE-MATRIX-MULTIPLY(n, C, A, B)

```
for i = 0 → n − 1 do
    for j = 0 → n − 1 do
        C_{i,j} = 0
        for k = 0 → n − 1 do
            C_{i,j} = C_{i,j} + A_{i,k} B_{k,j}
```

We consider that this algorithm is executed on a simple computer, consisting of a processing unit with a fast memory of size M. In addition to this limited memory, a large but slow storage space is available. In the following, we assume that this space is the disk and has unlimited available storage space.[2] Our objective is to minimize the data movement between memory and disk, also known as the volume of I/O (input/output), that is the number of A, B and C elements that are loaded from the disk to the memory, or written back from the memory to the disk. We assume that the memory is limited, and cannot store more than half a matrix, i.e., $M < n^2/2$.

In Algorithm 1, all B elements are accessed during one iteration of the outer loop. Thus, as the memory cannot hold more than one half of B, at least $n^2/2$ elements must be read. For the n iterations, this leads to $n^3/2$ read operations. This is huge as it is the same order of magnitude of the number of computations (n^3).

Fortunately, it is possible to improve the I/O behavior of the matrix product by changing the algorithm. We set $b = \sqrt{M/3}$ and assume that n is a multiple of b. We consider the blocked version of the matrix product, with

[1] A large part of this section is adapted from [83].

[2] Note that this study detailed for the case "main memory vs. disk" may well apply to other pairs of storage such as "fast small cache vs. large slower memory".

block size b, as detailed in Algorithm 2. In this algorithm $C^b_{i,j}$ denotes the block of size b at position (i, j) (all elements $C_{k,l}$ such that $ib \leq k \leq (i+1)b-1$ and $jb \leq l \leq (j+1)b-1$).

Algorithm 2: BLOCKED-MATRIX-MULTIPLY(n, C, A, B)

```
b ← √(M/3)
for i = 0, → n/b − 1 do
    for j = 0, → n/b − 1 do
        for k = 0, → n/b − 1 do
            Naive-Matrix-Multiply(n, C^b_{i,j}, A^b_{i,k}, B^b_{k,j})
```

Each iteration of the inner loop of the blocked algorithm must access 3 blocks of size b^2. Thanks to the choice of b, this fits in the memory, and thus, each of these $3b^2$ elements are read and written at most once. This leads to at most $2M$ data movements. Since there are $(n/b)^3$ iteration of the inner loop, the volume of I/O of the blocked algorithm is $O((n/b)^3 \times 2M) = O(n^3/\sqrt{M})$.

Changing the algorithm has allowed us to reduce the amount of data movements. The question is now: can we do even better? Interestingly, the answer is no, and it is possible to prove that any matrix product algorithms will perform at least $\Omega(n^3/\sqrt{M})$ I/O operations. We present here the proof of this results by Toledo [83].

Proof. We consider here a "regular" matrix product algorithms, which performs n^3 elementary multiplications (we thus exclude Strassen-like algorithms). We decompose the execution of such an algorithm into phases, such that in all phases, the algorithm exactly performs M I/O operations (except in the last phase, which may contains less). We say that a $C_{i,j}$ element is *alive* in a phase if some $A_{i,k} \times B_{k,j}$ product is computed in this phase.

During phase p, at most $2M$ elements of A can be used for the computation: at most M may initially reside in the memory, and at most M are read. We denote these elements by A_p. Similarly, at most $2M$ elements of B can be accessed during this phase, they are noted B_p. We distinguish two cases for the elements of A_p:

- S^1_p is the set of rows of A with at least $\sqrt{M}$ elements in A_p (thus, $|S^p_1| \leq 2\sqrt{M}$). Each of these rows can be multiplied with at most $|B_p|$ elements of B, resulting in at most $4m^{3/2}$ elementary products.

- S^1_p is the set of rows of A with fewer than $\sqrt{M}$ elements. Each of these rows contributes to a different alive $C_{i,j}$. Thus, they contribute to at most $\sqrt{M} \times 2M$ elementary multiplications.

Overall, a phase consists in at most $6M^{3/2}$ products. The total number of full phases is thus lower bounded by $\lfloor n^3/6M^{3/2} \rfloor \geq n^3/6M^{3/2} - 1$, and the volume of I/O is at least $n^3/6\sqrt{M} - 1/M$ which is in $\Omega(n^3/\sqrt{M})$ as soon as $M < n^2/2$ (our initial assumption). □

This example of the matrix product highlights several facts. Firstly, the choice of the algorithms impacts how the memory is accessed, and several algorithms which have similar computational complexity may exhibit very different memory and I/O behavior. Secondly, it is possible to exhibit lower bounds on the volume of I/O, especially thanks to the analysis in phases. Thirdly, thanks to these lower bounds, we may prove that some algorithms have optimal or asymptotically optimal I/O volume.

In the rest of this introductory chapter, we will present the pebble game models that laid the ground for most theoretical studies on memory or I/O minimization (Section 1.2). We will then present some fundamental results for memory and I/O minimization which are derived from these models and their extensions, and which are the basis of the work presented in this manuscript (Sections 1.3 and 1.4).

1.2 Two pebble game models

In this section, we review two pebble game models that lay the foundations of the problems studied in this manuscript. We first present the classical pebble-game that was introduced to model register allocation problems and is related to memory minimization. We then introduce the I/O pebble-game that deals with data movement minimization.

1.2.1 (Black) pebble game for memory minimization

We present here the first theoretical model that was proposed to study the space complexity of programs. This model, based on a pebble game, was originally used to study register allocation. Registers are the first level of storage, the fastest one, but also a scarce resource. When allocating registers to instructions, it is thus crucial to use them with caution and not to waste them. The objective is thus to find the minimum amount of registers that is necessary for the correct execution of the program. Minimizing the number of registers is similar to minimizing the amount of memory. For the sake of consistency, we present the pebble game as a problem of memory size rather than register number minimization. This does not change the proposed model nor the results, but allows us to present all results of this chapter with the same model and formalism.

The pebble-game was introduced by Sethi [78] to study the space complexity of "straight-line" programs, that is, programs whose control flow

does not depend on the input data. A straight-line program is modeled as a directed acyclic graph (DAG): a vertex represents an instruction, and an arc between two vertices $i \to j$ means that the results of the vertex i is used for the computation of j.

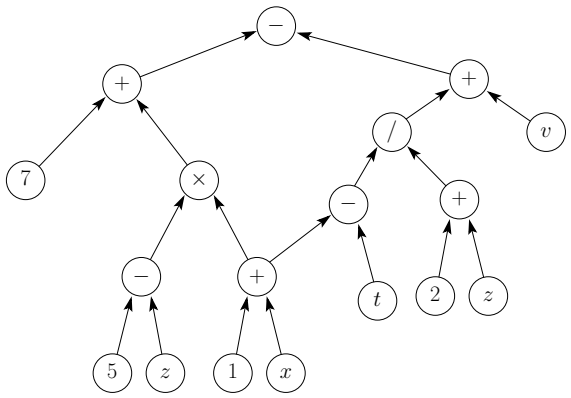


Figure 1.1: Graph corresponding to the computation of the expression $7 + (5 - z) \times (1 + x) - (1 + x - t)/(2 + z) + v$

When processing a vertex, all its inputs (as well as the result) must be loaded in memory, and the goal is to execute the program using the smallest amount of memory. Memory slots are modeled as pebbles, and executing the program is equivalent to playing a game on the graph with the following rules:

- (PG1) A pebble may be removed from a vertex at any time.
- (PG2) A pebble may be placed on a source node at any time.
- (PG3) If all predecessors of an unpebbled vertex v are pebbled, a pebble may be placed on v.

The goal of the game is to put a pebble on each output vertex at some point of the computation, and to minimize the total number of pebbles needed to reach this goal. In this game, pebbling a node corresponds to loading an input in memory (rule PG2) or computing a particular vertex (rule PG3). From a winning strategy of this game, it is thus straightforward to build a solution to the original memory allocation problem.

Note that the game does not ensure that each vertex will be pebbled only once. Actually, in some specific graphs, it may be beneficial to pebble several times a vertex, that is, to compute several times the same values, to save a pebble needed to store its value. A variation of the game, named the *Progressive Pebble Game*, forbids any recomputation, and thus models the objective of minimizing the amount of memory without any increase in the computational complexity. In this latter model, the problem of determining

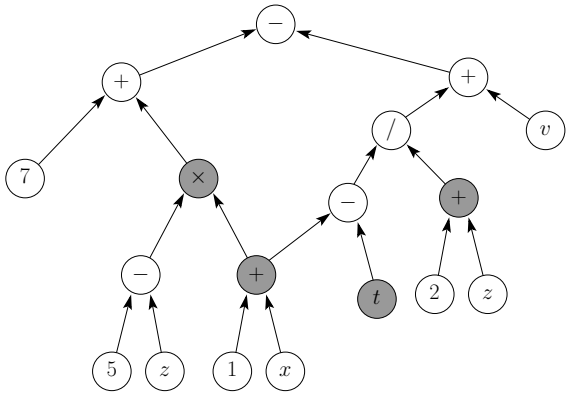


Figure 1.2: Playing the (black) pebble game on the graph of Figure 1.1. Dark nodes are the ones currently pebbled, meaning that four values are now in memory: $(5 - z) \times (1 + x)$, $1 + x$, t and $2 + z$.

whether a directed acyclic graph can be processed with a given number of pebbles has been shown NP-hard by Sethi [78]. The more general problem allowing recomputation is even more difficult, as it has later been proved PSPACE-complete by Gilbert, Lengaeur and Tarjan [45]. Another variant of the game slightly changes rule PG3 and allows to *shift* a pebble to an unpebbled vertex if all its predecessors are pebbled. Van Emde Boas [86] shows that it can at most decrease the number of pebbles required to pebble the graph by one, but in the worst case the saving is obtained at the price of squaring the number of moves needed in the game.

A simpler class of programs consists in trees rather than general graphs: for example, arithmetic expressions are usually described by in-trees (rooted directed trees with all edges oriented towards the root), unlike the one of Figure 1.1 which uses a common sub-expression twice. In this special case, the problem gets much simpler: Sethi and Ullman [79] designed an optimal scheme which relies on the following theorem.[3]

Theorem 1.1. *An optimal solution for the problem of pebbling an in-tree with the minimum number of pebbles using rules PG1 – PG3 is obtained by a depth-first traversal which orders subtrees by non-increasing values of $P(i)$, where the peak $P(v)$ of the subtree rooted at v is recursively defined by:*

$$P(v) = \begin{cases} 1 & \text{if } v \text{ is a leaf} \\ \max_{i=1\ldots k} P(c_i) + i - 1 & \text{where } c_1, \ldots c_k \text{ are the children of } v \\ & \text{such that } P(c_1) \geq P(c_2) \geq \cdots \geq P(c_k) \end{cases}$$

[3] When pebble shifting is allowed, the minimum number of pebbles needed to pebble a tree is the Strahler number (as outlined in [39]), which is a measure of a tree's branching complexity that appears in natural science, such as in the analysis of streams in a hydrographical bassin [82] or in biological trees such as animal respiratory and circulatory systems https://en.wikipedia.org/wiki/Strahler_number.

The first step to prove this result is to show that depth-first traversals are dominant, i.e., that there exists an optimal pebbling scheme which follows a depth-first traversal. Pebbling a tree using a depth-first traversal adds a constraint on the way a tree is pebbled: consider a tree T and any vertex v whose children are $c_1, \dots c_k$, and assume w.l.o.g that the first pebble that is put in the subtree rooted at v is in the subtree rooted at its leftmost child c_1. Then, in a depth-first traversal, the following pebbles must be put in the subtree rooted at c_1, until c_1 itself holds a pebble (and all pebbles underneath may be removed). Then we are allowed to start pebbling other subtrees rooted at $c_2, \dots, c_k$. These traversals are also called *postorder*, as the root of a subtree is pebbled right after its last child.

To prove that depth-first traversals are dominant, we notice that whenever some pebbles have been put on a subtree rooted at some vertex v, it is always beneficial to completely pebble this subtree (until its root v, which uses a single vertex) before starting pebbling other subtrees.

The second step to prove Theorem 1.1 is to justify the order in which the subtrees of a given vertex must be processed to give a minimal number of pebbles. This result follows from the observation that after having pebbles $i-1$ subtrees, $i-1$ pebbles should be kept on their respective roots while the i^{th} subtree is being pebbled.

The pebble-game model has been successfully used to obtain space-time complexity tradeoffs, that is, to derive relations between the number of pebbles and the number of moves, such as in [62]. Another variant, called the black-white pebble game has been proposed to model non deterministic execution [61, 67], where putting a white pebble on a node corresponds to guessing its value; a guessed vertex has to be actually computed later to check the value of the guess. In Section 1.4.1, we will come back to this model to present its extension to heterogeneous pebble sizes.

1.2.2 Red-Blue pebble game for data transfer minimization

In some cases, the amount of fast storage (i.e., memory) is too limited for the complete execution of a program. In that case, communication are needed to move data from/to a second level of storage (i.e., disk), which is usually larger but slower. Because of the limited bandwidth of the secondary storage, the amount of data transfers, sometimes called Input/Output (or simply I/O) volume, is a crucial parameter for performance, as seen in Section 1.1.1. Once again, we present this problem for the pair (main memory,disk), but this may well apply to any pair of storages in the usually deep memory hierarchy going from the registers and fastest caches to the slowest storage systems.

While the first pebble game allows to model algorithms under a limited memory, Hong and Kung have proposed another pebble game to tackle the

I/O volume minimization in their seminal article [50]. This game uses two types of pebbles (of different colors) and thus is also called the red/blue pebble game to distinguish with the original (black) pebble game. The goal is to distinguish between the main memory storage (represented by red pebbles), which is fast but limited, and the disk storage (represented by blue pebbles), which is unlimited but slow. As in the previous model, a computation is represented by a directed acyclic graph. The red and blue pebbles can be placed on vertices according to the following rules:

- (RB1) A red pebble may be placed on any vertex that has a blue pebble.
- (RB2) A blue pebble may be placed on any vertex that has a red pebble.
- (RB3) If all predecessors of a vertex v have a red pebble, a red pebble may be placed on v.
- (RB4) A pebble (red or blue) may be removed at any time.
- (RB5) No more than S red pebbles may be used at any time.

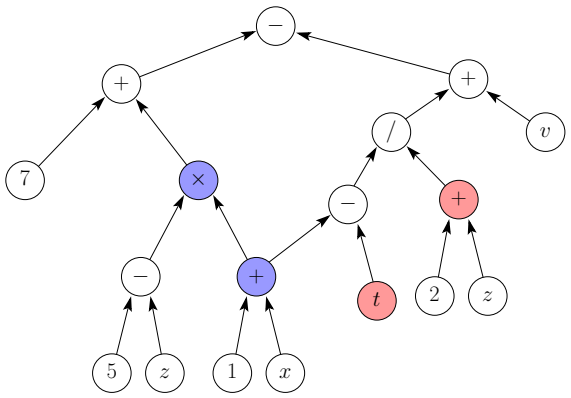


Figure 1.3: Playing the red/blue pebble game on the graph of Figure 1.1. The two red nodes represent values that are currently in memory, whereas the two blue nodes represent values that have been computed but then been evicted to disk. Before pebbling node $(1+x)-t$, a red pebble has to be put again on node $1+x$, corresponding to reading this value from the disk.

The goal of the game is to put a red pebble on each output vertex at some point of the computation, and to use the minimum number of RB1/RB2 rules to reach this goal. Red vertices represents values that currently lies in the main memory, after a computation, while blue pebbles represents values that are on disk. A value on disk may be read from disk (rule RB1) and similarly a value in memory can be stored on disk (rule RB2). Finally, we may compute a value if all its inputs are already in memory (rule RB3).

The volume of I/O is the total number of moves using rules RB1 or RB2, that is the total number of data movements between memory and disk.

The main results of Hong and Kung [50] relies on the decomposition of a computation, that is, a traversal of the graph, into a S-partition. A S-partition is defined as a family of vertex subsets $V_1, \dots V_h$ such that:

- (C1) The V_is define a partition of the graph's vertices;
- (C2) Each V_i admits a *dominator set* of at most S vertices, where a dominator set D of V is a set of vertices such that any path from a source to a vertex of V goes through a vertex of D;
- (C3) The *minimum set* of each V_i has at most S vertices, where the minimum set of V is the set of vertices in V that have no successors in V;
- (C4) There is no cyclic dependence between the V_is (we say there is a dependence between two subsets V_i,V_j if there exists an arc $u \to v$ such that $u \in V_i$ and $v \in V_j$).

Using this S-partition, Hong and Kung proves the following theorem:

Theorem 1.2. *Any complete computation of a directed acyclic graph using at most S red pebbles is associated to a $2S$-partition such that*

$$S(h-1) \leq V \leq Sh$$

where V is the volume of I/O, i.e., the total number of moves using rules RB1 and RB2 in the computation.

One of the corollaries of this result was a first proof of the $O(n^3/\sqrt{M})$ bound on I/O volume for product of two n^2 matrices, which was later simplified by Irony, Toledo and Tiskin [54] to derive the results presented in Section 1.1.2. In the original proof by Hong and Kung, a S-partition is constructed by decomposing any valid computation into phases where at most S moves using rules RB1 and RB2 occur in each phase. Thus, a dominator set of each V_i can be constructed by considering both the vertices holding a red pebble at the beginning of V_i and the vertices red by using rule RB1. As in the previous proof, each case contributes for at most S vertices, thus proving a $2S$-partition.

Thanks to this theorem, Hong and Kung derive the following lemma:

Lemma 1.1. *For any directed acyclic graph G, given a memory of size M, the minimum I/O volume satisfies*

$$Q \geq S(H(2M) - 1)$$

where $H(2M)$ is the minimum size of any $2M$-partition of G.

This lemma is then used to derive I/O lower bounds on some specific computations, such as the Fast Fourier Transform or the matrix multiplication.

In [50], Hong and Kung laid the foundations of I/O analysis, by introducing some key ideas: consider phases where the number of data transfers is equal to the size of the memory, and then bound the number of elementary computation steps that may be perform in each phase. This is precisely what we have done in Section 1.1.2 to exhibit a lower bound on the I/O operations of the matrix product. Since then, it has been largely used for the I/O analysis of numerous operations and extended to more complex memory models. Among many others, the book of Savage [75] presents several extensions, in particular for a hierarchical memory layout. Some recent works still build on the Hong and Kung model to compute I/O lower bounds for complex computational directed acyclic graphs [36]. In the following section, we will present some extensions of these ideas to design algorithms that minimizes I/O volume for linear algebra operations.

1.3 I/O lower bounds for matrix computations

We now present some development of the techniques introduced in the previous section, in order to derive I/O lower bounds for linear algebra operations. We first review some extensions on the matrix product presented in Section 1.1.2, and then move to extensions for other linear algebra operations.

1.3.1 More on matrix product

Before moving to more complex linear algebra operations, we go back to the classical matrix product in order to present some improvements and extensions of the basic result presented in Section 1.1.2.

In [54], Irony, Toledo and Tiskin proposed a new proof of the lower bound on data transfers presented in Section 1.1.2. This proof also relies on a decomposition into phases of M data transfers each, where M is the size of the memory. However, the upper bound on the number of elementary computations that may be performed in one phase comes from the following lemma.

Lemma 1.2. *Consider the conventional matrix multiplication algorithm $C = AB$. With access to N_A elements of A, N_B elements of B and contributions to N_C elements of C, no more than $\sqrt{N_A N_B N_C}$ elementary multiplications can be performed.*

This lemma relies on the discrete Loomis-Whitney inequality [66], which relates the cardinality of a finite subset V of $\mathbb{Z}^D$ to the cardinalities of its

d-dimensional projections, for $1 \leq d \leq D$. In particular, as illustrated on the right part of Figure 1.4, for $D = 3$ and $d = 2$, if V_1, V_2, V_3 denotes the orthogonal projections of V on each coordinate planes, we have

$$|V|^2 \leq |V_1| \cdot |V_2| \cdot |V_3|,$$

where $|V|$ is the volume of V and $|V_i|$ the area of V_i.

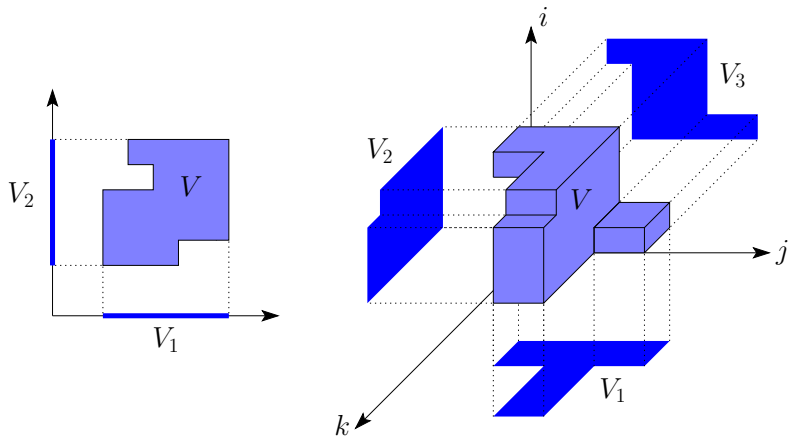


Figure 1.4: Use of Loomis-Whitney inequality, in two and three dimensions (left: $D = 2, d = 1$, which gives $|V| \leq |V_1| \cdot |V_2|$, right: $D = 3, d = 2$, leading to $|V|^2 \leq |V_1| \cdot |V_2| \cdot |V_3|$).

In our context, V is the set of indices (i, j, k) such that the elementary product of $A_{i,k}B_{k,j}$ is performed in a given time interval. The projection V_1 of V along the i axis corresponds to the set of B elements accessed for the computation, the projection V_2 along the j axis corresponds to the set of used A elements, and the projection V_3 along the k axis corresponds to elements of C for which some contribution is computed. The lemma directly follows from the Loomis-Whitney inequality.

This lemma is used in [54] with the same decomposition in phases presented in Section 1.1.2: in a phase, at most M data transfers (read and write) are performed. The number N_A of elements of A which can be used for the computation during one phase is bounded by $2M$, as at most M elements are in the memory, and at most M can be read. Similarly, we have $N_B \leq 2M$ and $N_C \leq 2M$. Thus, the number of elementary products during one phase is at most $\sqrt{N_A N_B N_C} \leq 2\sqrt{2}M^{3/2}$. Therefore, the number of full phases is at least $\lfloor n^3/2\sqrt{2}M^{3/2} \rfloor$ and the number of data transfers is at least

$$\left(\frac{n^3}{2\sqrt{2}M^{3/2}} - 1 \right) M.$$

In [59], Langou carefully optimized this analysis and obtained a lower bound of $2\frac{n^3}{\sqrt{M}} - 2M$ on data transfers.

Similar bounds may also be obtained for parallel algorithms. On P processors, at least one processor is in charge of computing n^3/P elementary products. By applying the previous reasoning to this processor, Irony, Toledo and Tiskin [54] proved that one processor has to perform at least $\frac{n^3}{2\sqrt{2}P\sqrt{M}} - M$ communications. They also studied the case of $2D$-block algorithms algorithm (such as Algorithm 2) where each processors holds $O(n^2/P)$ elements, and the case of $3D$-block algorithms, where the input matrices are replicated and each processor holds $O(n^2/P^{2/3})$ elements. In both cases, they exhibit communication lower bounds and show there exist algorithms in the literature that are asymptotically optimal.

1.3.2 Extension to other matrix computations

In [10], Ballard, Demmel, Holtz and Schwartz extended the previous results to many other matrix computations, by introducing the notion of *generalized matrix computations*. They noticed that many existing matrix algorithms could be formulated as two nested loops on i and j containing an update of the following form:

$$\forall (i,j), \quad C_{i,j} \leftarrow f_{i,j}\big(g_{i,j,k}(A_{i,k}, B_{k,j}) \text{ for } k \in S_{i,j}, K\big) \tag{1.1}$$

where:

- Matrices A,B,C may be reordered, and may even overlap;
- $f_{i,j}$ and $g_{i,j,k}$ represent any functions that depend non-trivially on their arguments, that is, the computation of $f_{i,j}$ requires at least one data to be kept in memory while scanning its arguments, and both $A_{i,k}$ and $B_{k,j}$ must be in memory while computing $g_{i,j,k}$;
- K represents any other arguments for $f_{i,j}$

The formulation of the matrix product as a generalized matrix computation is straightforward: simply take the product as $g_{i,j,k}$, $S_{i,j} = 1, \ldots, n$ and the sum as $f_{i,j}$. Moreover, other matrix computations, such as the LU Gaussian elimination, can also be described in this form. The classical algorithm for LU elimination is made of two nested loops on i and j containing the following updates:

$$\begin{aligned} L_{i,j} &= (A_{i,j} - \sum_{k<j} L_{i,k} \cdot U_{k,j})/U_{j,j} \text{ for } i > j \\ U_{i,j} &= A_{i,j} - \sum_{k<i} L_{i,k} \cdot U_{k,j} \text{ for } i \leq j \end{aligned}$$

It may be transformed into Equation (1.1) by setting $A = B = C$, where the lower triangular part of A classically represents L and its upper triangular part represents U, $g_{i,j,k}$ multiplies $A_{i,k} = L_{i,k}$ with $A_{k,j} = U_{k,j}$ for all indices k in $S_{i,j} = \{k < j\}$ if $i > j$ and $S_{i,j} = \{k < i\}$ otherwise, and finally $f_{i,j}$ performs the sum of its arguments, subtracts the result from $A_{i,j}$ (and divides it by $U_{j,j}$ in the case $i > j$).

Then, a generalized matrix computation may be analysed by decomposing the computations into phases corresponding to a given amount of data transfers, as in the case of the matrix product, in order to derive I/O lower bounds. Again, exactly M data transfers are perform in a phase (except in the last one). In order to bound the number of computations done in a phase, Ballard et al. propose to distinguish both the origin (root) and the destination of each data:

- Root 1: the data is already in the memory in the beginning of the phase, or is read during the phase (at most $2M$ such data);
- Root 2: the data is computed during the phase (no bound);
- Destination 1: the data is still in the memory at the end of the phase, or is written back to disk (at most $2M$ such data);
- Destination 2: the data is discarded (no bound).

Usually, we may discard the case where the algorithm computes some data which is then discarded (Root 2/Destination 2). Then, we consider V the set of indices (i, j, k) of $g_{i,j,k}$ operations computed in a given phase, V_A the set of indices (i, k) of A values used in the computations, V_B the set of indices (k, j) of B values used in the computations, and V_C the set of indices (i, j) of the corresponding $f_{i,j}$ operations. For each of the A, B and C cases, all data either originate from Root 1 or target Destination 1 (since we exclude Root 2/Destination 2), and thus we can bound $V_X \leq 4M$. Using the Loomis-Whitney inequality, we know that $V \leq \sqrt{(4M)^3}$. For a total number G of $g_{i,j,k}$ computations, at least $\lfloor G/V \rfloor$ phases are full, and thus the number of data transfer is lower bounded by

$$M \left\lfloor \frac{G}{\sqrt{(4M)^3}} \right\rfloor \geq \frac{G}{8\sqrt{M}} - M.$$

In [10], this analysis is also successfully applied to Cholesky and QR factorizations, and to algorithms computing eigenvalues or singular values.

The discovery of these lower bounds led to the design of parallel algorithms for many linear algebra operations that aim at minimizing the amount of communications while keeping up with perfect load balancing

in order to optimize performance. These algorithms are usually known as *communication-minimizing*, *communication-avoiding* or *communication-optimal* (see [9, 29, 30] among others).

1.4 Memory-aware task graph scheduling

We now move to the problem of scheduling applications described by directed acyclic task graphs, as presented in the introduction of this chapter. A major difference with what was presented above is the heterogeneity of the data sizes: up to now, we only considered data of unit size. However, task graphs usually model computations of coarser grain, so that tasks may produce data of larger and heterogeneous sizes.

Most existing studies on memory-aware task graph scheduling were done in a very particular application domain, namely the factorization of sparse matrices through direct methods. We gather here their main results as well as the main proof arguments. For the sake of coherence, we translate these results into our own formalism.

1.4.1 Generalized pebble games for task graph scheduling

In [65], Liu introduced a variant of the (black) pebble game, called the *generalized pebble game* to account for data with different sizes. In this model, a vertex of the directed acyclic graph is no longer a single instruction which needs a single register, but a coarser-grain task with a given memory requirement. His objective was to study a particular linear algebra operation (the sparse matrix factorization using direct multifrontal method), which is described as a tree of elementary factorizations. Before processing the actual factorization, the tree of tasks is built, and it is easy to compute what is the amount of memory needed by each task. To model the computation, he proposed to associate a value τ_x with each vertex x, which represents the number of pebbles needed to satisfy x. The rules of the (black) pebble game are then adapted to this new model:

- (GPG1) A pebble may be removed from a vertex at any time.
- (GPG2) A pebble may be placed on a source node at any time.
- (GPG3) If all predecessors of an unpebbled vertex v are satisfied, new pebbles may be placed on v, or moved to v from its predecessors.

The objective is, starting with no pebbles on the graph, to satisfy all outputs of the graph (that is, the root of the tree in this particular case). Note that because rule GPG3 allows to move pebbles, this is an extension of the variant with pebble shifting of the black pebble game presented in Section 1.2.1. In this model, the number of pebbles needed to satisfy a vertex v represents the

size of the output of the computational task of v, as these pebbles must stay on v until its successor is computed (i.e., pebbled). Rather than number of pebbles, we will now refer to memory amounts: τ_k represents the size of the output data of task k, and the goal is to minimize the *peak memory*, that is the maximum amount of memory used by a traversal at any time.

1.4.2 Peak memory minimizing traversals for task trees

Since the generalized pebble game is an extension of the pebble game, it naturally inherits its complexity: computing a schedule of a given graph that minimizes the peak memory is NP-complete. Adapting the algorithm described in Section 1.2.1 to compute optimal tree traversals is an interesting question, that has been studied by Liu [64, 65]. First, we should note that postorder traversals are not dominant anymore in this heterogeneous model: it may well happen that a vertex v has a large output τ_v while its input (the sum of its children outputs) is relatively small. In that case, it is not true anymore that v must be processed right after its last child, as this will increase the memory consumption, and thus decrease the memory available for other parts of the tree. However, postorder traversals are simple to implement and it is thus interesting to compute the one with smallest peak memory.

Best postorder traversal

The following theorem, proposed by Liu [64] is an adaptation of Theorem 1.1 for heterogeneous trees. It allows to compute the best postorder traversal for peak memory minimization, which is given in Algorithm 3.

Theorem 1.3. *Let T be a vertex-weighted tree rooted at r, where τ_v is the size of the data output by vertex v. Let $c_1, \ldots, c_k$ be the children of r and P_i be the smallest peak memory of any postorder traversal of the subtree rooted at c_i. Then, an optimal postorder traversal is obtained by processing the subtrees by non-increasing $P_i - \tau_{c_i}$, followed by r. Assuming that $P_1 - \tau_{c_1} \geq P_2 - \tau_{c_2} \geq \cdots P_k - \tau_{c_k}$, the number of pebbles needed by the best postorder traversal is:*

$$P = \max\left(P_1, \tau_{c_1} + P_2, \tau_{c_1} + \tau_{c_2} + P_3, \ldots \sum_{i=1}^{k-1} \tau_{c_i} + P_k, \tau_r\right) \tag{1.2}$$

Proof. This results can be proven by a simple exchange argument: Consider two subtrees rooted at c_i and c_j such as $P_i - \tau_{c_i} \leq P_j - \tau_{c_j}$. If the subtree rooted at c_i is processed right before c_j, their contribution to the peak memory is:

$$P_{i,j} = \sum_{k \in B} \tau_{c_k} + \max\left(P_i, c_i + P_j\right)$$

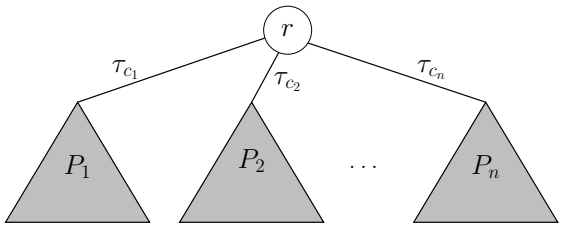


Figure 1.5: Illustration of Theorem 1.3.

where $\{c_k, k \in B\}$ is the set of roots of subtrees processed before i. If we exchange the two subtrees, that is, process i right after j, we get:

$$P_{j,i} = \sum_{k \in B} \tau_{c_k} + \max\left(P_j, c_j + P_i\right)$$

Note that $P_j \leq c_i + P_j$ and $c_j + P_i \leq c_i + P_j$ since $P_i - \tau_{c_i} \leq P_j - \tau_{c_j}$. Thus $P_{j,i} \leq P_{i,j}$. □

Algorithm 3: PostorderMinMem (T)

Data: $T = (V, E, \tau)$: tree with vertex-weights
Result: (π, P): Postorder schedule and its minimal peak memory
if $V = \{u\}$ **then return** $((u), \tau_u)$
Let r be the root of T, $c_1, c_2 \dots c_k$ its children and $T_1, T_2, \dots T_k$ the corresponding subtrees
for $i = 1$ *to* k **do**
  $(\pi_i, P_i) \leftarrow$ PostorderMinMem (T_i)
Sort the k subtrees such that $P_1 - \tau_{c_1} \geq \dots \geq P_k - \tau_{c_k}$
$\pi \leftarrow$ Concatenate $\pi_1, \pi_2, \dots, p_k$ and add r at the end
Compute P as in Equation (1.2)
return (π, P)

Best general traversal

Despite their simplicity, postorder traversals are not optimal. In a second study, Liu proposed a more complex algorithm to compute optimal traversals for peak memory minimization [65]. This strategy, detailled in Algorithm 4 consists in a recursive algorithm. To compute an optimal traversal for a tree rooted at vertex k, it first recursively computes an optimal traversal for the subtrees rooted at each of its children, then merges them and finally adds the root at the end. The difficult part is of course how to merge the children traversals. Contrarily to a postorder traversal, in a general traversal, there is no reason to process the subtrees one after the other. One may want to

interleave the processing on one subtree with that of the others. Liu makes the observation that, in an optimal traversal, the switching points between the subtrees' processing have to be local minima in the memory profile: while processing one subtree T_i, there is no reason to switch to T_j if one can reduce the memory needed for T_i by processing one more task. This remark leads to slicing the traversal into atomic parts, called *segments*. The end-points of segments (which are local minima in the memory profile) are called *valleys*, while the peak memory vertices of each segment are called *hills*. The memory consumption at valleys and hills are denoted by V and H and are defined as follows, for a given traversal:

- H_1 is the peak memory of the whole traversal;
- V_1 is the minimum amount of memory which occurs after the step when H_1 is (last) reached;
- H_2 is the peak memory after the step when V_1 is (last) reached;
- V_2 is the minimum amount of memory occurring after the step when H_2 is (last) reached;
- etc.

The sequence of hill-valley ends when the last vertex is reached. The segments consist of the vertices comprised between two consecutive valleys $]V_i, V_{i+1}]$. Take for example the tree depicted on Figure 1.6, and consider first the subtree rooted in C. It has a single valid traversal, (A, B, C) which is decomposed in two segments $[A, B]$ $(H = 10, V = 1)$ and $[C]$ $(H = 5, V = 5)$. Its sibling subtree, rooted at E, has a traversal consisting in a single segment $[D, E]$ $(H = 5, V = 1)$.

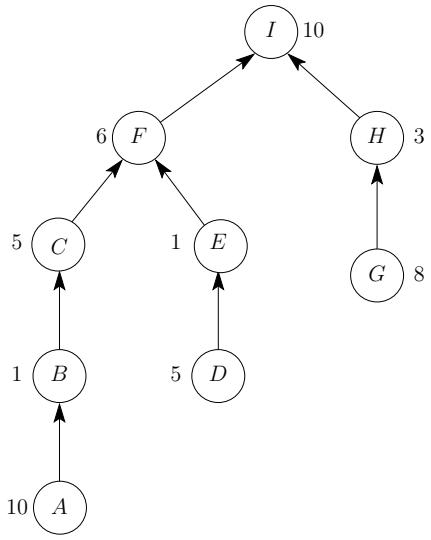


Figure 1.6: Example of tree in Liu's model.

To merge the traversals of the subtrees, the first step is to compute all hill-valley segments. Then the lists of segments are merged using the fol-

lowing criterion: if several segments are available (one for each subtree in the beginning), it is always beneficial to start with the segment with the maximum ($hill - valley$) difference. Intuitively, the residual memory will only increase when processing segments, so it is better (i) to start with the segment with larger peak memory (hill) to avoid large memory consumption later, and (ii) to start with the segment with smaller residual memory (valley) to ensure an increase of memory as small as possible. Note that this is the same argument that made us process subtrees in decreasing $P_i - \tau_i$ to derive the best postorder.

In the example of Figure 1.6, when merging the traversals from subtrees rooted at C and E, we will first take segment $[A, B]$ ($H - V = 9$), then $[D, E]$ ($H - V = 4$) and finally $[C]$ ($H - V = 0$). By adding the root, we get the optimal traversal (A, B, D, E, F) for this subtree, now decomposed in the following segments:

segment	hill	valley
$[A, B]$	10	1
$[D, E]$	6	2
$[C, F]$	6	6

The subtree rooted at H is a single segment $[G, H]$ ($H = 8, V = 3$). We repeat the same merging procedure and get the following optimal traversal: $(A, B, G, H, D, E, C, F, I)$, of peak memory 10. Note that the best postorder traversal reaches a peak memory of 13 $(G, H, A, B, C, D, E, F, I)$.

In [65], Liu proves that the worst-case runtime complexity of the algorithm is in $O(n^2)$, where n is the number of nodes in the tree. To reach this complexity, a multiway-merge algorithm is used to merge t hill-valley segments in time $O(n \log t)$. Surprisingly, this algorithm seems to have been independently rediscovered by Lam et al. [58]. Using optimized data structures for storing the segments, they decreased its complexity to $O(n \log^2 n)$.

Algorithm 4: LiuMinMem (T)

Data: $T = (V, E, \tau)$: tree with vertex-weights
Result: π: Schedule with minimal peak memory
if $V = \{u\}$ **then return** (u)
Let r be the root of T and $T_1, T_2, \ldots T_k$ its subtrees
for $i = 1$ *to* k **do**
    $\pi_i \leftarrow$ LiuMinMem (T_i)
    Decompose π_i in hill-valley segments $(s_i^1, s_i^2, \ldots)$ as described in the text, where H_i^j (resp. V_i^j) is the hill (resp. valley) of segment s_i^j
$\pi \leftarrow$ Merge segments s_i^j sorted by non-increasing $H_i^j - V_i^j$, add the root r at the end
return π

This algorithm derives from Yannakakis work on min-cut linear arrangement [89]. This is the problem of finding a linearization of an undirected (unweighted) graph such that for any point of this linear ordering, the number of edges that cross this point (cutwidth) is minimal. This problem was known to be NP-complete for general graphs [42] and Yannakakis proposed a polynomial algorithm for undirected trees. Note that adding weights renders min-cut linear arrangement NP-complete on undirected trees [69]. The peak memory minimization presented above exactly corresponds to the weighted version of min-cut linear arrangement, but on directed trees when all edges are directed towards the root.

1.4.3 Minimizing I/O volume when scheduling task trees

We now move to the case where the available memory is too limited and cannot accomodate all the data needed for the processing of one tree, as presented in Sections 1.2.2 and 1.3. This corresponds to a tree with peak memory larger than the available memory. Our objective is then to make a moderate use of the secondary storage, that is, to minimize the amount of data movement between fast memory and disk. This is frequently called out-of-core computation. In this context, Agullo et al. show how to compute the postorder traversal that minimizes the I/O volume [4].

We consider a given postorder described by a function PO such that $PO(i) < PO(j)$ means that i is processed before j. We define the storage requirement, which may be larger than the available memory M, of a subtree T_i rooted at node i, which is very similar to the peak memory of a postorder seen in the previous section:

$$S_i = \max\left(\tau_i, \max_{j \in Children(i)}\left(\sum_{k \in PS(j)} \tau_k + S_j\right)\right)$$

where $Children(i)$ denotes the set of children of task i, $PS(j)$ is the set of siblings of j that are scheduled before i in PO (formally, $PS(j) = \{k \in Children(i), PO(k) < PO(j)\}$). The amount A_i of main memory that is used to processed this subtree in PO is then $A_i = \min(S_i, M)$. Finally, the I/O volume of this traversal can be computed recursively:

$$V_i = \max\left(\underbrace{\max\left(\tau_i, \max_{j \in Children(i)}\left(\sum_{k \in PS(j)} \tau_k + A_j\right)\right)}_{\text{peak memory demand during the processing of } T_i} - M, 0\right) + \sum_{j \in Children(i)} V_j$$

The only term in the previous formula that depends on the subtree ordering is

$$\max_{j \in Children(i)} \left(\sum_{k \in PS(j)} \tau_k + A_j \right).$$

We recognize the previous expression of the peak memory of a postorder traversal, where P_j has been replaced by A_j. Similarly, this expression is minimized when subtrees are processed in non-increasing order of $A_i - \tau_i$, which is done by Algorithm 5.

Algorithm 5: PostorderMinIO (T, M)

Data: $T = (V, E, \tau)$: tree with vertex-weights, M: memory bound
Result: (π, S): postorder schedule with minimal I/O and its storage requirement S
if $V = \{u\}$ **then return** $((u), \tau_u)$
Let r be the root of T, $c_1, c_2 \dots c_k$ its children and $T_1, T_2, \dots T_k$ the corresponding subtrees
for $i = 1$ *to* k **do**
  $(\pi_i, S_i) \leftarrow$ PostorderMinIO (T_i)
  $A_i \leftarrow \min(S_i, M)$
Sort the k subtrees such that $A_1 - \tau_{c_1} \geq \dots \geq A_k - \tau_{c_k}$
$\pi \leftarrow$ Concatenate $\pi_1, \pi_2, \dots, p_k$ and add r at the end
$S \leftarrow \max \left(\tau_i, \max_{1 \leq j \leq k} \sum_{l=1}^{j-1} \tau_l + S_j \right)$
return (π, S)

1.5 Conclusion

We presented in this chapter a few theoretical studies that allow us to estimate the amount of memory needed for a computation, or the amount of data movement between the main memory and a secondary storage, whenever the former is too limited in size. The two seminal works [78, 50] both rely on playing games with pebbles, which represent storage slots in the primary or secondary storage. They have later been extended to derive I/O lower bounds for several types of computations, including matrix operations, and to schedule task graphs in a memory-limited computing environnement. Note that the work on task graphs is still subject to many limitations: indeed, it is restricted to trees, it does not take into account parallel processing, nor the fact that memory can be distributed. In the following chapters, we will extend these results to overcome some of these limitations.

The choice of the results presented above is subjective and influenced by the rest of the manuscript: we have mainly concentrated on linear algebra operations and task graph scheduling, as it lays the basis for the extensions presented in the following chapters. There exist numerous studies that take a different approach to study algorithms under memory limited conditions.

Models for parallel processing using shared memory were first proposed based on the PRAM model [56]. On the opposite, distributed memory systems were studied through models such as LogP [26]. The External Memory model [1, 88] was proposed to account for I/O between the memory and the cache, of limited size M and block size B, which corresponds to the size of the consecutive data that is read/written in a single step from/to the memory. It has later been extended to several distributed caches sharing the same memory (Parallel External Memory) [7]. Most of the studies in these models aim at deriving the complexity of fundamental operations (sorting, prefix sum), and sometimes more specific operations (graph algorithms, Fast Fourier Transform, LU factorization).

The idea of cache-oblivious algorithms as proposed in [41] is to design algorithms that do not depend on any machine-related parameters, such as cache size, but that minimize the amount of data moved to the cache, or even through a hierarchy of caches. The main idea to design such algorithms is to use a recursive divide-and-conquer approach: the problem is divided into smaller and smaller subproblems. Eventually, the data of a subproblem will totally fit into the cache (and similarly for smaller subproblems) and is computed without any data transfer. Such algorithms have been proposed for several operations that are naturally expressed in a recursive way, such as Fast-Fourier-Transform, matrix multiplication, sorting or matrix transposition.

Part I

Task graph scheduling with limited memory

Foreword

In the following chapters, we present some contributions on memory-aware task graph scheduling: we target either peak-memory minimization, or in the case of a bounded memory, total I/O volume minimization or makespan minimization. Most of these results builds on the previous algorithms designed for trees and described in the previous chapter (more specifically, Algorithms PostorderMinMem, LiuMinMem and PostorderMinIO).

Before presenting our contributions, we first introduce in Chapter 2 a new and more flexible model of memory-aware task graphs, that is used throughout the next chapters, as well as the straightforward adaptation of these three algorithms in this model. Then, Chapter 3 presents new results on the sequential processing of trees, Chapter 4 shows an extension to Series-Parallel graphs, Chapter 5 focuses on processing task graphs on hybrid platforms, and Chapter 6 concentrates on the parallel processing of task trees.

Chapter 2

Memory-aware dataflow model

We start by introducing a new task graph model for structured applications that takes memory footprint into account; this model, called the dataflow model, is used throughout the next four chapters. We prove that it inherits the complexity[1] of the pebble game, shows its equivalence with Liu's generalized pebble game on trees, and then adapt the three algorithms presented in the previous chapter for trees to this model.

2.1 Proposed dataflow model

We consider structured applications described by a directed acyclic graph (DAG) $G = (V, E)$, where vertices represent tasks and edges represent the dependencies between these tasks, in the form of input/output data. The integer n denotes the number of vertices in V, which are indifferently called tasks or nodes. We adopt a different model for introducing weights representing the memory behavior of the tasks, which we believe is more general and better suited for general task graphs (in opposition to trees). In our model, both vertices and edges are weighted: the weight on vertex i, denoted by m_i, accounts for the memory size of the program of task i as well as its temporary data, that is, all the data that must be kept in memory as long as task i is running, but can safely be deleted afterwards. The weight of edge (i, j), denoted by $d_{i,j}$, represents the size of the data produced by task i and used as an input by task j.

Trees are a notable particular type of graphs that we study in the following chapters. We denote by $\mathit{Children}(i)$ the set of children of a node i and by $\mathit{parent}(i)$ its parent. We generally consider in-trees, with all depen-

[1] In all contributions, "complexity" refers to "runtime complexity", unless otherwise stated.

dencies oriented towards the root. For such trees, we simplify the notation $d_{i,parent(i)}$ to d_i.

We now characterize the memory behaviour of a sequential processing of the tree. We assume that during the processing of a task, both the input data, the output data and the program and temporary data must reside in memory. The memory needed for processing of task i is thus

$$MemReq(i) = \sum_{(j,i)\in E} d_{j,i} + \sum_{(i,k)\in E} d_{i,k} + m_i.$$

Note that other data (not involving task i) may be stored in memory during this processing. After processing a subset V' of nodes, if there is no ongoing computation, the size of the data remaining in memory is given by

$$\sum_{\substack{(i,j)\in E \text{ s.t.} \\ i\in V' \text{ and } j\notin V'}} d_{i,j}.$$

Finally, if task k is being processed and tasks in V' are completed, the size of the data in memory is:

$$MemUsage(V', k) = \sum_{\substack{(i,j)\in E \text{ s.t.} \\ i\in V' \text{ and } j\notin V'}} d_{i,j} + m_k + \sum_{(k,j)\in E} d_{k,j}.$$

Note that the first sum accounts among other for the input data of task k, while the second one contains its output data. In the sequential case, a schedule of the tasks (also called traversal of the graph) is defined by a permutation σ of $[1, n]$ such that $\sigma(t) = i$ indicates that task i is processed at step t. For a given schedule σ, the peak memory can be computed as follows:

$$PeakMemory(\sigma) = \max_{t=1,\dots n} MemUsage(\{\sigma(t') \text{ with } t' < t\}, \sigma(t)).$$

2.2 Emulation of the pebble game and complexity

This dataflow model differs from the original pebble game presented in the previous chapter (Section 1.2.1) as it considers that a task with several successors produces a different data for each of them, whereas in the pebble game, a single pebble is placed on such a task to model its result. However, we show in the following theorem that the dataflow model is able to emulate the pebble game, and thus, inherits its complexity.

Theorem 2.1. *Let $G = (V, E, d, m)$ be a vertex- and edge-weighted graph representing a computation in the dataflow model. Computing the minimum peak memory of any schedule of G is PSPACE-complete.*

Proof. The problem is clearly in PSPACE, as checking all $n!$ possible schedules of a graph may easily be done in polynomial space. We prove the completeness by transforming a DAG G representing a computation in the pebble game to another DAG G' under the dataflow model and showing that any schedule in G corresponds to a schedule in G' with the same peak memory. The final result follows from the pebble game being PSPACE complete on DAGs without recomputation [45].

Let $G = (V, E)$ be a DAG representing a computation under the pebble game model. We build the dataflow graph $G' = (V', E', d, m)$ such that:

- For each node i of V, we define two nodes i_1 and i_2 of V', connected by an edge (i_1, i_2) of unit weight ($d_{i_1,i_2} = 1$). Intuitively, i_1 represents the allocation of the pebble on i and i_2 its removal. The edge weight stands for the cost of the pebble between these two events.

- For each edge (i, j) of E, we build two edges (i_1, j_1) and (j_1, i_2) in E' of null weight ($d_{i_1,j_1} = d_{j_1,i_2} = 0$). The first edge states that a pebble must be placed on i before placing one on j, and the second edge ensures that j is pebbled before the removal of the pebble on i.

- All nodes have temporary data of null size ($m_i = 0$ for all i).

Figure 2.1 illustrates this construction on a toy example. We now prove that G can be traversed with B pebbles without recomputation if and only if G' can be scheduled with a memory of size B.

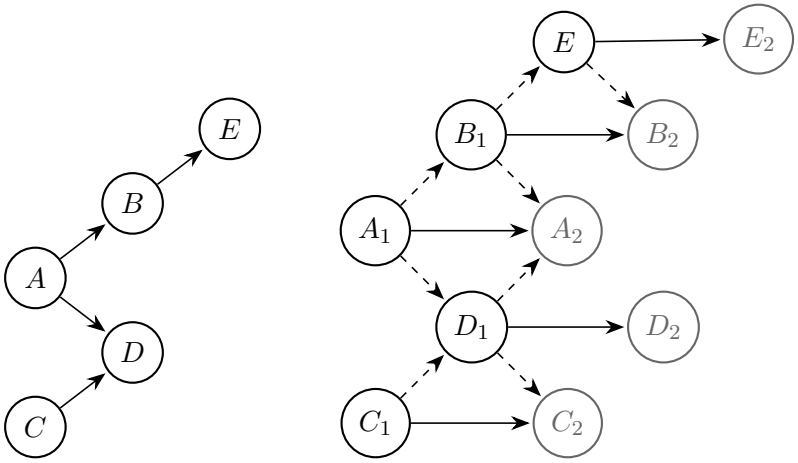


Figure 2.1: Example of the transformation of an instance of the pebble game (left) into an instance of peak memory minimization in the dataflow model (right). Dashed edges have null weight.

First, we suppose that there exists a pebbling scheme π which processes G with B pebbles. We transform π into a schedule σ of G: when π pebbles

a node i of G, σ executes the node i_1 of G'; when π removes a pebble from a node i of G, σ executes the node i_2 of G'. One can easily check that σ is a valid schedule on G' as π is a valid traversal of G without recomputation. Besides, at any time, the memory used by σ is equal to the numbers of nodes i of G such that i_1 is executed but not i_2, which is equal to the number of pebbles required by π, so that σ is a valid schedule of G using a memory of size B.

Now, we suppose there exists a schedule σ of G' with a memory peak equal to B. We transform σ into a pebbling scheme π of G: when σ executes a node i_1, π pebbles the node i, and when σ executes a node i_2, π removes the pebble of node i. Similarly, we easily verify that π is a valid pebbling scheme of G without recomputation since σ is a valid traversal of G', and that π uses no more than B pebbles. □

2.3 Equivalence with Liu's model on trees

The proposed dataflow model differs from the generalized pebble game on trees of Liu presented in the previous chapter (Section 1.4.1) as it distinguishes between the size of the memory needed during the computation and the size of the output of a task. However, on task trees, both models are general enough to emulate each other:

- Given a tree $T = (V, E, \tau)$ in Liu's model, we construct a tree $T' = (V, E, n, f)$ in the dataflow model with $d_i = \tau_i$ and $m_i = -\tau_i$. We easily check that the memory usage after processing a subset of nodes and during the computation of a node is the same in T and in T'.

- Given a tree $T = (V, E, n, f)$ describing a computation in the dataflow model, we build a tree $T' = (V', E', \tau)$ in Liu's model such that:

  - for each vertex i in V, there are two vertices i_1, i_2 in V' and an edge (i_1, i_2) in E',
  - for each edge (i, j) in E, there is an edge (i_2, j_1) in E'
  - the weights are given by $\tau_{i_1} = \sum_{(j,i)\in E} d_j + m_i + d_i$ and $\tau_{i_2} = d_i$.

  We also verify that for a given processing order of the vertices of V, by processing nodes in the same order and each i_2 right after i_1, the memory usage in T' after each i_1 is the same as the memory usage in T during the processing of i, while the memory after i_2 corresponds to the memory after completing i.

Finally, note that on task graphs that are not trees, our new model considers that a task i with several successors $j_1, j_2, \ldots$ produces one different data for each of them $(d_{i,j_1}, d_{i,j_2}, \ldots)$.

2.4 Problem definitions

We consider a computing system composed of a main memory, in which the data of the task currently processed must reside, and a secondary storage, such as a disk. As we have seen in the previous chapter, the order chosen to process the tasks plays a key role in determining which amount of main memory or I/O volume is needed for a complete execution of a task graph. More precisely, here are the two main problems that are addressed in this part:

MinMemory: Determine the minimum amount of main memory that is required to execute a task graph without any access to secondary memory, as well as a schedule of the tasks that achieves this amount of memory.

MinIO: Given the size M of the main memory, determine the minimum I/O volume that is required to execute a task graph as well as a corresponding schedule of the tasks and a schedule of the I/O operations.

In the following, we generally assume that tasks are processed sequentially, one after the other (except in Chapter 6 and in Section 5.3 of Chapter 5).

2.5 Adaptation of previous algorithms on trees

For the sake of completeness, we present below a modified version of the three algorithms on trees reported in the previous chapter, namely PostorderMinMem, LiuMinMem and PostorderMinIO. The adaptation to the dataflow model is most of the time straightforward. For LiuMinMem, we choose to detail the decomposition of a schedule into segments, so that one could easily implement it from this description. Note that the "sort" statement is actually a "MergeSort", as outlined by Liu in its original algorithm [65], as segments within a specific sub-schedule π_i are already sorted. In the following chapters, when we refer to one of these algorithms, this always denotes its version for the dataflow model.

Algorithm 6: PostorderMinMem (T) (dataflow model)

Data: $T = (V, E, d, m)$: tree in the dataflow model
Result: (σ, P): postorder schedule and its minimal peak memory
if $V = \{u\}$ **then return** $((u), d_u + m_u)$
Let r be the root of T, $c_1, c_2 \dots c_k$ its children and $T_1, T_2, \dots T_k$ the corresponding subtrees
for $i = 1$ *to* k **do**
  $(\sigma_i, P_i) \leftarrow$ PostorderMinMem (T_i)
Sort the k subtrees such that $P_1 - d_{c_1} \geq \cdots \geq P_k - d_{c_k}$
$\sigma \leftarrow$ Concatenate $\sigma_1, \sigma_2, \dots, \sigma_k$ and add r at the end

$$P \leftarrow \max\left(d_r + m_r + \sum_{j=1}^{k} d_{c_j}, \max_{1 \leq j \leq k} \sum_{\ell=1}^{j-1} d_{c_\ell} + P_j\right)$$

return (σ, P)

Algorithm 7: LiuMinMem (T) (dataflow model)

Data: $T = (V, E, d, m)$: tree in the dataflow model
Result: σ: schedule with minimal peak memory
if $V = \{u\}$ **then return** (u)
Let r be the root of T and $T_1, T_2, \dots T_k$ its subtrees
for $i = 1$ *to* k **do**
  $\sigma_i \leftarrow$ LiuMinMem (T_i)
  $v_0^i \leftarrow 0$, $s \leftarrow 1$, $j \leftarrow 1$
  while $s < \mathit{NumberOfNodes}(T_i)$ **do**

$$H_j^i \leftarrow \max_{t \geq s}\left(d_{\sigma_i(t)} + m_{\sigma_i(t)} + \sum_{k \in \mathit{Children}(\sigma_i(t))} d_k\right)$$

    Let h_j^i be the largest index t achieving the previous maximum

$$V_j^i = \min_{t \geq h_j^i}\left(d_{\sigma_i(t)}\right)$$

    Let v_j^i be the largest index t achieving the previous minimum
    NB: we have identified in σ_i the jth segment at indices $[v_{j-1}^i + 1, v_j^i]$, with hill H_j^i and valley V_j^i
    $s \leftarrow v_j^i + 1$, $j \leftarrow j + 1$
$L \leftarrow$ sort all $(H_j^i, h_j^i, V_j^i, v_j^i)$ quadruplets by non-increasing $H_i^j - V_i^j$
$\sigma \leftarrow \emptyset$
foreach $(H_j^i, h_j^i, V_j^i, v_j^i) \in L$ **do**
  Add $(\sigma_i(v_{j-1}^i + 1), \dots, \sigma_i(v_j^i))$ at the end of σ
Add the root r at the end of σ
return σ

Algorithm 8: PostorderMinIO (T, M) (dataflow model)

Data: $T = (V, E, d, m)$: tree in the dataflow model, M: memory bound

Result: (σ, S): postorder schedule with minimal I/O and its storage requirement

if $V = \{u\}$ **then return** $((u), d_u + m_u)$

Let r be the root of T, $c_1, c_2 \dots c_k$ its children and $T_1, T_2, \dots T_k$ the corresponding subtrees

for $i = 1$ *to* k **do**

  $(\sigma_i, S_i) \leftarrow$ PostorderMinIO (T_i)

  $A_i \leftarrow \min(S_i, M)$

Sort the k subtrees such that $A_1 - d_{c_1} \geq \dots \geq A_k - d_{c_k}$

$\sigma \leftarrow$ Concatenate $\sigma_1, \sigma_2, \dots, p_k$ and add r at the end

$$S \leftarrow \max\left(d_i + m_i + \sum_{j=1}^{k} d_k, \max_{1 \leq j \leq k} \sum_{l=1}^{j-1} d_l + S_j\right)$$

return (σ, S)

Chapter 3

More on Peak Memory and I/O Volume of Task Trees

In this chapter, we present some results that directly complement the work described in Section 1.4 on the memory-aware scheduling of task trees. We first focus on the MinMemory problem (Sections 3.1 and 3.2) and then move to the MinIO problem (Section 3.3).

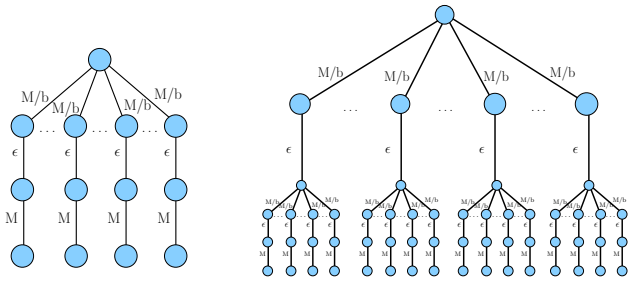
3.1 Postorder traversals for memory minimization

We focus here on postorder traversals, as presented in Chapter 1. Postorder traversals treat subtrees one after the other, and thus are simpler to implement and offer easier memory management. Besides, their use is natural when trying to reduce the peak memory: by processing subtrees one after each other, the number of data that stay in memory is reduced. When data sizes are almost homogeneous, this also minimizes the memory footprint of a traversal (in particular, postorder traversals are dominant for unit-weight trees). However, postorder traversals are not optimal on heterogeneous trees, which motivates the use of LiuMinMem to compute the optimal traversal.

The optimal traversal is more complex to implement than a postorder, as it requires an involved runtime management to suspend the processing on some subtree while another subtree is being processed. Besides, finding the optimal traversal has a time complexity in $O(n^2)$ (n being the number of nodes) while finding the best a postorder only requires $O(n \log n)$ time. The natural question is then: is it worth implementing such a complex traversal to reduce the peak memory?

We investigated this question both theoretically and experimentally and first proved that postorder traversals may be much worse than optimal traversals, as outlined in the following theorem.

Theorem 3.1. *Given any arbitrarily large integer K, there exist trees for which the best postorder traversal peak memory is at least K times the minimum peak memory required to traverse the tree.*



(a) One level. (b) Two levels.

Figure 3.1: First levels of the graph for the proof of Theorem 3.1. 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 3.1a. All branches are identical and all tasks have no temporary data ($m_i = 0$ for all i). 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 $M_{\text{min}} = M + b\varepsilon$. Now replace each leaf by a copy of the harpoon graph, as shown in Figure 3.1b. The value of M_{min} becomes $M + (2b-1)\varepsilon$, while a postorder traversal requires $M_{PO} = M + \varepsilon + 2(b-1)M/b$. Replacing the leaves again with the harpoon graph for L times, postorder requires $M_{PO} = M + \varepsilon + L(b-1)M/b$ while $M_{\text{min}} = L(b-1)\varepsilon + M + \varepsilon$. Thus, for any ratio K, there exists L such that when iterating the process L times, $M_{PO}/M_{\text{min}} > K$. □

This negative result seems a serious issue since postorder traversals are widely used in practice. We thus decided to test how postorders behave on actual trees. As outlined previously, tree-shaped task graphs with large memory requirement especially happen during the factorization of sparse matrices through direct multifrontal methods, as for example in the MUMPS [5, 6] or QR-MUMPS [3] softwares. Thus, we investigated the task trees corresponding to the elimination tree of actual sparse matrices from the University of Florida Sparse Matrix Collection[1] (see [C19] for more details on elimination trees and the data set).

The detailed results are given in Table 3.1. They show that postorder reaches the optimal peak memory in 95.8% of the cases. While its maximum

[1] http://www.cise.ufl.edu/research/sparse/matrices/

Fraction of suboptimal postorder traversals	4.2%
Maximum increase in memory of postorder compared to optimal	18%
Average increase in memory of postorder compared to optimal	1%

Table 3.1: Statistics on peak memory cost of the best postorder traversal compared to the optimal traversal on actual elimination trees.

peak memory increase is non negligible (18%), in average, its increase in peak memory is hardly noticeable. This means that except for very particular matrices, a postorder traversal is sufficient to reach near-optimal memory performance.

3.2 Another optimal algorithm on trees

We presented in Chapter 1 the LiuMinMem algorithm that computes a tree traversal with minimum peak memory, as well as its adaptation to the proposed dataflow model of Chapter 2. Liu's algorithm performs a recursive bottom-up traversal of the tree, and at each node, combines the optimal traversals built for all subtrees. The combination requires a sophisticated multi-way merging algorithm in order to reach the $O(n^2)$ complexity.

In this section, we introduce TopDownMinMem, another exact algorithm which proceeds top-down and searches for the best reachable state in the tree with a given memory bound. While the worst-case complexity of TopDownMinMem is the same as Liu's exact algorithm, it runs faster in practical cases resulting from multifrontal methods as we outlined below. Contrarily to the previous algorithm designed for in-trees, the proposed algorithm works on out-trees. We first introduce out-trees and present their equivalence with in-trees before presenting the algorithm and its performance.

3.2.1 Top-down vs. bottom-up orientation of the trees

For the description of the TopDownMinMem algorithm, we consider out-trees, that is, rooted trees such that each edge is oriented with its backward end facing the root. In such a tree, each node has a single input (its parent) and possibly several outputs (its children). From an in-tree T, we denote by $\overline{T}$ the out-tree obtained by reversing the orientation of all edges. From a schedule σ of T, we consider the permutation $\overline{\sigma}$ which orders nodes in the reverse order of σ:

$$\overline{\sigma}(t) = n - \sigma(t) + 1.$$

The following lemma states the equivalence of σ on T and $\overline{\sigma}$ on $\overline{T}$.

Lemma 3.1. *If σ is a valid schedule of T of peak memory M, then $\overline{\sigma}$ is a valid schedule of $\overline{T}$ with peak memory M.*

This result is easily deduced by checking that the set of processed nodes after step $\sigma(t)$ in T is exactly the set of unprocessed nodes after step $\overline{\sigma}(n-t)$ in $\overline{T}$, and that the memory used during step $\sigma(t)$ in T is the same as the memory used as step $\overline{\sigma}(n-t+1)$ in $\overline{T}$.

This allows to consider trees in a top-down process for computing a peak-memory minimizing schedule: once an optimal schedule σ is computed for the out-tree T, the reverse schedule $\overline{\sigma}$ is optimal for the in-tree $\overline{T}$. We use this transformation twice: in the design of the TopDownMinMem algorithm and when decomposing series-parallel graphs into out-trees and in-trees in Chapter 4.

3.2.2 The TopDownMinMem algorithm

The TopDownMinMem algorithm is based on an advanced tree exploration routine: the Explore algorithm. For the sake of clarity, we present here simplified versions of Explore and TopDownMinMem with a larger complexity, and then detail the improvements needed to decrease it.

The Explore algorithm requires a tree T, a node i to start the exploration, and an amount of available memory M_{avail}. With these parameters, the algorithm computes the state with minimal memory consumption that can be reached. If the whole tree can be processed, then the minimal reachable 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 data of their root nodes are still stored in memory. The Explore algorithm outputs the cut with minimal memory occupation, as well as a traversal to reach this state from node i with the provided memory.

When called on a node i, the algorithm first checks if the current node can be executed. If not, the algorithm stops and returns an empty traversal. 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 (or the cut is empty), then the algorithm outputs the current cut.

The TopDownMinMem algorithm uses the Explore algorithm to check whether the tree can be processed using a given memory. It performs a binary search between trivial lower and upper bounds to compute the minimum memory needed to traverse a tree, as well as a valid traversal.

Algorithm 9: EXPLORE (T, i, M)

Input: tree T, root i of subtree to explore, available memory M
Output: furthest reachable minimum cut L, traversal Tr
if *node i is a leaf and* $m_i + d_i \leq M$ **then**
  return $(\emptyset, [i])$
if $m_i + d_i + \sum_{j \in Children(i)} d_j > M$ **then**
  return $(\emptyset, [\,])$
$L \leftarrow Children(i)$
$Tr \leftarrow [i]$
continue $\leftarrow$ *true*
while *continue is true and* $L \neq \emptyset$ **do**
  continue $\leftarrow$ *false*
  $L' \leftarrow \emptyset$
  while $L \neq \emptyset$ **do**
    $j \leftarrow$ remove first element of L
    $(L_j, Tr_j) \leftarrow$ EXPLORE$(T, j, M - \sum_{k \in L \setminus \{j\}} d_k,)$
    if $Tr_j \neq \emptyset$ *and* $\sum_{k \in L_j} d_k \leq d_j$ **then**
      $L' \leftarrow L' \cup L_j$ `/* replace` j `by` L_j `*/`
      $Tr \leftarrow Tr \oplus Tr_j$ `/* append` Tr_j `to the end of` Tr `*/`
      continue $\leftarrow$ *true*
    else
      $L' \leftarrow L' \cup \{j\}$ `/* keep` j `*/`
  $L \leftarrow L'$
return (L, Tr)

Algorithm 10: TOPDOWNMINMEM (T)

Input: tree T
Output: minimum memory M to process the tree, traversal Tr
1 $M_{\text{LB}} \leftarrow \max_{i \in T} MemReq(i)$ `/* lower bound */`
2 $M_{\text{UB}} \leftarrow \sum_{i \in T} d_i + \max_{i \in T} m_i$ `/* upper bound */`
3 **while** $M_{\text{LB}} + 1 < M_{\text{UB}}$ **do**
4 $M \leftarrow \lfloor (M_{\text{LB}} + M_{\text{UB}})/2 \rfloor$
5 $(L, Tr) \leftarrow$ EXPLORE$(T, root, M)$
6 **if** $L = \emptyset$ **then**
7 $M_{\text{UB}} = M$
8 **else**
9 $M_{\text{LB}} = M$
10 $(L, Tr) \leftarrow$ EXPLORE$(T, root, M_{\text{LB}})$
11 **return** (M_{LB}, Tr)

As presented, the EXPLORE and TOPDOWNMINMEM algorithms have a large complexity as the same exploration is performed several times. Its running time can be improved as follows:

- The EXPLORE algorithm may also return a lower bound on the additional memory needed to process each of the subtrees in the cut; the minimum incremental memory can be used to speed up the binary search.
- To avoid recomputing the exploration from the root when the available memory is too small, we may add the best cut reached up to now and the corresponding traversal as input parameters to EXPLORE.

In [C19] we used these optimizations to derive an algorithm whose complexity is in $O(n^2)$.

3.2.3 Performance of TopDownMinMem

The optimized TOPDOWNMINMEM algorithms has the same worst-case complexity as the LIUMINMEM algorithm, i.e., $O(n^2)$. We compared the actual running times of both algorithms, as well as the one of the POSTORDERMINMEM algorithm (of worst-case complexity $O(n \log n)$) on assembly trees coming from actual matrices (see [C19] for details). Note that all three algorithms have been implemented in highly optimized C++ versions.

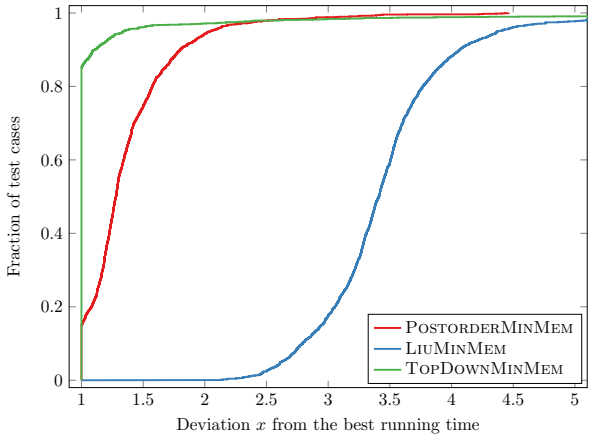


Figure 3.2: Performance profiles for comparing the running time of the three algorithms for the Minmem problem on the assembly trees.

The results are presented in Figure 3.2 as a performance profile [32], which plots to the cumulative distribution of the running time of each algorithm divided by the best running time for the considered tree. It gives the fraction of the cases where a specific algorithm has a running time within some deviation x of the best running time. Therefore, the higher the fraction, the faster the algorithm. We observe on Figure 3.2 that TOPDOWN-

MinMem is the fastest algorithm in 80% of the cases, and clearly outperforms LiuMinMem. Although LiuMinMem and TopDownMinMem exhibit the same worst-case complexity, the former sorts certain segments of the tree and combines them using a sophisticated multi-way merging algorithm. This sort operation is the computational core of the method. On the contrary, TopDownMinMem does not use sorting. This can make a significant difference: if, for example, for each task i, $d_i \geq \sum_{j \in Children(i)} d_j$, TopDownMinMem will run in time $O(n)$ for a tree of n nodes. In this same case, LiuMinMem is likely to be slower, as it will repeatedly sort the segments. Note that TopDownMinMem is also faster than PostorderMinMem on most trees as it does not require to sort nodes.

In the rest of the chapter, we get back to the classical case of in-trees and bottom-up traversals.

3.3 Results and open problems for MinIO

We now consider the MinIO problem, as defined in Section 2.4: we assume that the primary storage (e.g. the memory) available for the computation has a fixed, limited size M. In some cases, this is not sufficient to process the whole tree, which means that the minimum peak memory as computed by one of the previous optimal algorithms is larger than M. In this case, we are compeled to use the secondary storage (e.g, the disk), but we want to minimize the total amount of communication, i.e., read or write operations from and to the secondary storage, as they are usually much more costly than accesses to the primary storage. We distinguish two cases depending on whether whole data have to be written to disk or if it is possible to write only part of a data. As detailed below, the algorithm PostorderMinIO of Agullo et al. [4] presented in Section 1.4.3 considers the latter case.

3.3.1 MinIO without splitting data

In this variant, we consider that only whole data can be written to disk: when some memory has to be freed for new computations, we have to select a subset of the data currently in the memory that we want to transfer to the disk. We call this variant of the problem MinIOAtomic.

Contrarily to MinMemory, the MinIOAtomic 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 data should be transferred into secondary memory at each step. More precisely, the following three variants of the problem are NP-complete.

Theorem 3.2. *Given a tree T with n 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.

Note that (iii) is the original MinIOAtomic 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). The proof proceeds using the same reduction from the 2-Partition problem [42] for all problems: on the tree built for this reduction, all traversals are indeed postorder traversals, and finding an efficient I/O scheme is equivalent to finding a good partition of the integers.

3.3.2 MinIO with paging

We consider here the variant of the MinIO problem when it is possible to split data that reside in memory, and write only part of it to the disk if needed. This is for instance what is done using *paging*: the main memory containing all data is divided in same-size *pages*, which can be individually moved from/to secondary storage when needed. Since all modern computer systems implement paging, it is natural to consider this variant when minimizing the I/O volume. This is the variant considered in Section 1.4.3, for which the PostorderMinIO has been proposed (we come back to this algorithm below).

A solution to this problem is described both by an ordering σ of the tasks, and a I/O function f_{IO}, which states which amount $f_{IO}(i)$ of a task T_i output data should be written to disk: $f_{IO}(i) = x$ means that x among d_i units of the output data of task i are written to disk (then we assume they are written as soon as task i completes). Note that we do not need to clarify which part of the data is written to disk, as our cost function only depends on the total volume. Besides, we assume that when $f_{IO}(i) \neq 0$, the *write* operation on the output data of task i is performed right after task i completes (and produces the data), and the *read* operation is performed just before the use of this data by task i's parent, as any other I/O scheme would use more memory at some time step for the same I/O volume.

Contrarily to the previous variant which forbids splitting data, the complexity of the MinIO problem with paging is still open. However, we gather here a few recent results on this problem.

First, given an ordering σ of the tasks and a memory bound M, one can prove that the I/O function f_{IO} following the *Furthest in the Future* policy achieves the smallest amount of I/O for σ. The *Furthest in the Future* policy is defined as follows: during the execution of σ, whenever the

memory exceeds the limit M, the data which is (partially) evicted from the memory and written to disk is the one which will be used the furthest in the future, i.e., whose processing comes last in the schedule σ. This result is similar to Belady's rule which states the optimality of the offline MIN cache replacement policy [12, 60] that evicts from the cache the data which is used the latest. It was already stated for postorder traversals in [2], and we generalized it in [R1]. It allows to describe a solution in a more compact form, as the I/O function can be deduced from the ordering of the nodes. Note that the converse is also true: from an I/O function f_{IO} which corresponds to an optimal solution (σ, f_{IO}), we may reconstruct σ by expanding nodes as explain below (see Figure 3.3) and applying a memory-minimizing scheduling algorithm.

Minimizing memory as a first step to minimizing I/O

A natural idea to minimize the I/O volume is to first consider a traversal that minimizes the peak memory (using LiuMinMem or TopDownMinMem), and to transform it into a traversal for the memory-bounded case by adding I/O operations following the Furthest in the Future policy, in the hope that it will result in a small I/O volume. Unfortunately, this strategy has no guarantee on the resulting I/O volume: we are able to problem instances on which any memory-minimizing algorithm will produce k times as many I/Os as the optimal traversal for I/O minimization, where k is a parameter linked to the depth of the tree [R1]. By making k as large as we want, we prove that any peak-memory minimizing strategy is not constant-factor competitive in the MinIOPaging problem.

Best postorder traversal

We presented in Section 1.4.3 the algorithm PostorderMinIO, introduced by Agullo et al. [4], which computes the postorder traversal that minimizes the I/O volume when data can be partly written to disk. Unfortunately, it may also be far from reaching the optimal I/O volume. More specifically, we prove that there exist problem instances on which PostorderMinIO performs arbitrarily more I/O than the optimal I/O amount. This is trivially true as on some instances, the optimal traversal, which is not a postorder traversal, processes the whole tree within the bounded memory M, while any postorder traversal needs a memory larger than M and thus performs some I/O. Moreover, we have produced problem instances where the optimal traversal requires 1 I/O, but where PostorderMinIO induces an I/O volume in $\Omega(nM)$.

Postorder traversals may be arbitrarily bad on general trees, however we have shown that they are optimal on homogeneous trees, as outlined by the following result.

Theorem 3.3. *Consider a tree where all edge weights are equal to 1 and all node weights equal to 0 ($d_i = 1, m_i = 0$ for all i), then* PostorderMinIO *outputs an optimal traversal for the* MinIO *problem.*

Note that we assume that the memory bound M is integer, as well as all amount of I/Os $f_{IO}(i)$, so that this results holds for both variants of the problem, with or without paging. This theorem generalizes a result of Sethi and Ullman [79] for homogeneous binary trees.

ILP formulation and heuristics In order to produce efficient solutions for the general problem (MinIOPaging with heterogeneous data sizes), we have exhibited an ILP formulation of the problem which allows to compute the optimal solution for small trees (up to around 30 nodes). It requires $O(n^2)$ variables and $O(n^3)$ constraints.

For trees of larger sizes, we have designed a heuristic that is based on a peak-memory minimizing algorithm such as LiuMinMem or TopDownMinMem. It first computes a peak-memory minimizing traversal using one of these algorithms. When used with a limited memory M, this traversal induces some I/Os. We select one node i which is (partly) written to disk and expand it as illustrated on Figure 3.3. After expansion, node i is replaced by three nodes: i_1 represents the computation of node i, i_2 represents the write operation, and i_3 represents the read operations. The characterics of the new nodes are chosen to represent the amount of memory used during and after the read/write operations.

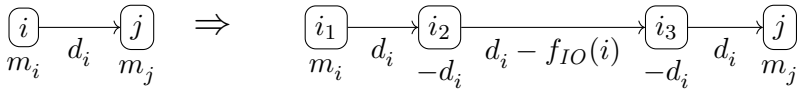


Figure 3.3: Expansion of a node to force an I/O operation

Once a chosen node with I/O is expanded, the heuristic computes again a peak-memory minimizing traversal of the expanded tree. The interest of expanding a node is the following: once we have decided to expand a given node and to write some data on disk, the memory freed by this operation is known to the peak-memory minimizing algorithm, which can take advantage of this information to schedule more nodes without incurring new I/Os.

The FullRecExpand heuristic, described in Algorithm 11, applies this expansion mechanism recursively starting from leaves, up to the root of the tree, and ensures that it produces an expanded tree and a traversal which never exceeds the memory bound M. Note that a node may be expanded several times, so it is not possibly to polynomially bound the complexity of this heuristic. Thus, we propose a simpler variant, denoted by RecExpand, which limits the while loop of Algorithm 11 to two iterations. In this variant,

Algorithm 11: FullRecExpand (G, r, M)

Input: tree G, root of exploration r
Output: expanded tree G_r with peak memory not larger than M
foreach *child* i *of* r **do**
  $G_i \leftarrow$ FullRecExpand(G, i, M)
$G_r \leftarrow$ tree formed by the root r and the G_i subtrees
while LiuMinMem(G_r) *uses a memory larger than* M **do**
  $f_{IO} \leftarrow$ I/O function obtained from LiuMinMem(G_r) using the Furthest in the Future policy
  $i \leftarrow$ node for which $f_{IO}(i) > 0$ whose parent is scheduled the latest in LiuMinMem(G_r)
  modify G_r by expanding node i according to $f_{IO}(i)$
return G_r

the resulting tree G_r might need I/Os to be executed. The final schedule is computed as in FullRecExpand by running LiuMinMem on G_r.

The different strategies previously studied were tested through simulations on both synthetic and actual trees coming from the multifrontal factorization of sparse matrices. For each tree, we set the memory bound as the mean between the minimum memory needed to process each node of the tree, and the minimum memory for which some I/O are necessary. On the actual trees, we had to discarded some trees for which I/O are not needed, that is, trees where the minimum memory peak is equal to the minimum memory needed to process each node.

The results obtained on synthetic trees are presented in Figure 3.4 as performance profiles (see Section 3.2). We first notice that PostorderMinIO is much worse that the others: in 75%, it results in a deviation from the best larger than 100% , meaning that it produces at least twice as many I/Os as the expansion heuristics. LiuMinMem performs better, but is outperformed by the heuristics, while the polynomial heuristic RecExpand is only slightly worse than the non-polynomial FullRecExpand.

We observed the same trends on actual trees, however the absolute difference among heuristics are much smaller: these trees are easier to schedule than the synthetic ones. We also compared the performance of the heuristics to the optimal obtained with the ILP on small synthetic trees (30 nodes). It shows that RecExpand and FullRecExpand are optimal in 99% of the cases and only a few percent away otherwise.

3.4 Conclusion of the chapter

In this chapter, we have thoroughly studied the problem of memory-aware scheduling of task trees. We proved that postorder traversals lead to an

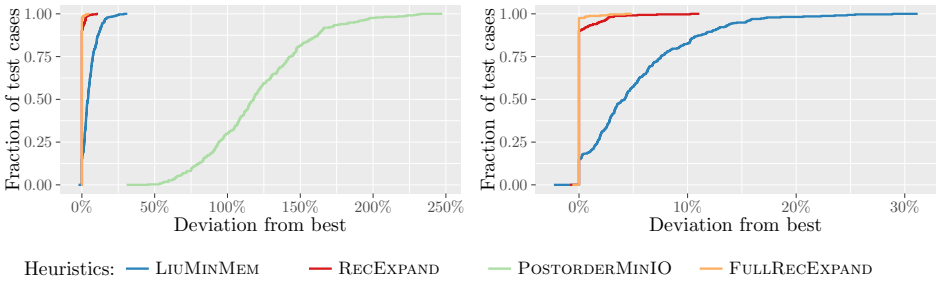


Figure 3.4: Performance profiles of the heuristics on the synthetic trees (right: zoom on the left part of the graph, not including PostorderMinIO).

arbitrarily larger peak-memory than optimal traversals. We presented a recursive top-down peak-memory minimizing algorithm which is usually faster than Liu's algorithm. We also revisited the I/O minimization problem: we proved that the atomic variant of the problem was NP-complete, and proposed both an ILP and efficient heuristics for the non-atomic variant. Despite our effort, the complexity of this variant is still open, even if we conjecture that it is NP-complete.

Note on the material presented in this chapter. The first part of this work was done during the PhD of Mathias Jacquelin, co-advised by Yves Robert and in collaboration with Bora Uçar. It was presented at IPDPS 2011 [C19]. More recently, we revisited the MinIO problem during the PhD of Bertrand Simon, co-advised with Frédéric Vivien, in collaboration with Samuel McCauley; a first version was presented at the APDCM workshop of IPDPS 2017 [W13] and an extended version has been submitted to Journal of Scheduling.

Chapter 4

Minimizing Peak Memory of Series-Parallel Task Graphs

In this chapter, we present an optimal algorithm for minimizing peak memory for series-parallel task graphs. This algorithm extends Liu's algorithm dedicated to trees [65].

4.1 Introduction on series-parallel task graphs

Our focus here is on a certain class of applications whose graphs are series-parallel; these applications have already receive some attention in the scheduling literature [25, 38, 70] as they represent an important class of scientific computing applications.

Series-parallel graphs may be defined as follows (see for example [37]).

Definition 4.1. *A two-terminal series-parallel graph, or SP-graph, G with terminals s and t is recursively defined to be either:*

Base case: *A graph with two vertices s and t, and an edge (s, t).*

Series composition: *The* series *composition of two SP-graphs G_1 with terminals s_1, t_1 and G_2 with terminals s_2, t_2 formed by identifying $s = s_1$, $t = t_2$ and $t_1 = s_2$;*

Parallel composition: *The* parallel *composition of two SP-graphs G_1 with terminals s_1, t_1 and G_2 with terminals s_2, t_2 formed by identifying $s = s_1 = s_2$ and $t = t_1 = t_2$.*

The vertices s and t are called source *and* target *of the graph.*

Note that series-parallel graphs can be recognized and decomposed into a tree of series and parallel combinations in linear time [85]. We consider series-parallel graphs whose vertices and edges have weights, as described by

the memory-aware dataflow task graph model in Chapter 2. We are interesting into computing the minimal amount of main memory to completely process the graph (MinMemory problem).

In order to solve the peak memory minimization problem for SP-graphs, we first propose a solution for a more restricted family of SP-graphs, namely parallel-chain graphs as defined below.

Definition 4.2. *A* chain *is a two-terminal series-parallel graph obtained without using any parallel composition. A* parallel-chain graph *is a two-terminal series-parallel graph obtained by the parallel compositions of a number of chains.*

A sample parallel-chain graph is shown in Figure 4.1.

4.2 Optimal algorithm for parallel-chain graphs

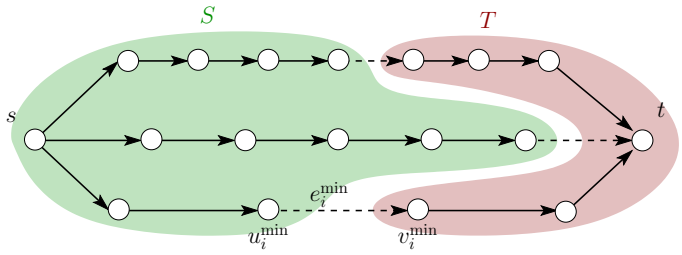


Figure 4.1: Sample parallel-chain graph and its decomposition for Lemma 4.1. If the dashed edges have minimum weight in the corresponding chains, an optimal traversal can be found by first ordering the vertices in the set S and then the vertices in the set T.

The main idea to minimized peak memory for parallel-chain graphs is to remove one edge from each chain, so as to disconnect the graph into one in-tree (with edges oriented towards the root) and one out-tree (with edges oriented towards the leaves). Then, we can reuse Liu's algorithm to compute an optimal traversal for these two trees. The following lemma states that if the removed edges are of minimal weight in each chain, it is possible to first schedule all the vertices that are *before* this minimal cut, and then all the vertices *after* the cut, without increasing the peak memory.

Lemma 4.1. *Let G be a parallel-chain graph. For each chain C_i of this graph, let $e_i^{\min} = (u_i^{\min}, v_i^{\min})$ be an edge of C_i with minimum weight. Let S be the set of ancestors of the $u_i^{\min}$'s, including them and T be the set of successors of the $v_i^{\min}$'s, including them. Then, there exists a schedule σ such that:*

- *σ schedules all tasks of S before any task of T;*

- *σ reaches the minimum peak memory.*

Proof. We consider a parallel-chain graph G and a peak memory optimal traversal γ of G. We consider the (S,T) partition of the nodes as in the lemma. We transform γ into a schedule which obeys the first constraint of the lemma, and prove that it does not increase the peak memory. To this goal, we first schedule all nodes of S with respect to their relative order in γ, and then schedule all nodes of T likewise. Let σ be the obtained schedule. It is easy to check that σ is a valid schedule, i.e., it respects precedence constraints as γ does. By construction, σ fulfills the first condition of the lemma.

We now prove that σ does not require more memory than γ. Let k be a node in some chain C_i. We first consider that k is in set S. We denote by t_1 (respectively t_2) the step when k is scheduled in γ (resp. σ): $\gamma(t_1) = \sigma(t_2) = k$. Just before step t_1, the set of data in memory contains the incoming edge of node k plus exactly one edge (a_j, b_j) of each other chain $j \neq i$. If $a_j \in S$, then the same edge lies in memory when k is schedule in σ. Otherwise ($a_j \in T$), the edge of chain j in memory during the processing of k in schedule σ is $e_i^{\min}$. Since this is the edge of minimum weight of the whole chain, in particular $d_{e_i^{\min}} \leq d_{a_j,b_j}$. Thus, the data in memory while k is processed is not larger in σ than in γ. In the case when k is in T we can similarly prove the same result. Hence the peak memory of σ is not larger than that of γ. □

Note that the (S,T) partition considered in the lemma is indeed a *topological cut*: there is no edge $(i,j) \in E$ with $i \in T$ and $j \in S$.

Thanks to this result, we know that there exists an optimal schedule which orders first vertices from S, and then vertices from T. Assume for a moment that the weight of all $e_i^{\min}$ edges is zero ($d_{e_i^{\min}} = 0$). Then, it is as if the graph was disconnected, and we have two separate trees to schedule. T is an in-tree, and Liu's algorithm can compute an optimal schedule σ for it. S is an out-tree, so that the mirror tree $\overline{S}$ as defined in Section 3.2.1 is an in-tree: if γ is the optimal schedule computed by Liu's algorithm for $\overline{S}$, the reversed schedule of γ, noted $\overline{\gamma}$, is optimal for S. Then, $(\overline{\gamma}, \sigma)$ is an optimal schedule of the whole graph. This approach can be generalized to parallel-chain graphs with non-zero weights on the minimal edges, as stated in Algorithm 12. The main idea is to zero-out the weight of all edges in the cut, by subtracting $d_{e_i^{\min}}$ from all edges of chain i. Then, the same quantity $d_{e_i^{\min}}$ is added to each vertex weight m_k for each node k of the chain (except s and t). By setting $C = \sum_{i=1}^{q} d_{e_i^{\min}}$, it is easy to verify that for any schedule σ, the memory footprint during the execution of a node or after its execution in the modified graph is the same memory as in the original graph minus C. Thus, any optimal schedule for the modified graph is an optimal schedule for the original graph.

Algorithm 12: ParallelChainMinMem(PC, Cut)

Input: $PC = (V, E, m, d)$: parallel-chain graph in the dataflow model
Input: Cut: edge set of a topological cut of minimum weight
Output: σ: schedule with minimal peak memory
Let $C_1, \ldots, C_q$ be the chains of PC
for $i = 1$ *to* q **do**
  Let $e_i^{\min}$ be the edge of Cut in chain C_i
  Remove edge $e_i^{\min}$ from E
  foreach *edge* e *of chain* i *except* $e_i^{\min}$ **do** $d_e \leftarrow d_e - d_{e_i^{\min}}$
  foreach *node* k *in chain* i *except* s *and* t **do** $m_k \leftarrow m_k + d_{e_i^{\min}}$
Consider the two trees T_{out}, T_{in} made by disconnecting PC
reversing all edges in T_{out}
$\gamma \leftarrow$ LiuMinMem $(\overline{T_{out}})$
$\sigma \leftarrow$ LiuMinMem (T_{in})
return $(\overline{\gamma}, \sigma)$

4.3 Optimal algorithm for series-parallel graphs

The optimal algorithm for general series-parallel graphs follows the same ideas as the one for parallel-chain graphs, but performs the transformation recursively, following the series-parallel decomposition of the graph. It outputs both an optimal schedule for peak-memory minimization and a topological cut of the graph with minimum weight.

The base case of the algorithm considers a graph with a single edge, and outputs the unique schedule along with the unique topological cut. In the general case, G is a series or parallel composition of two smaller series-parallel graphs G_1 and G_2. We first recursively compute two optimal schedules σ_1 and σ_2 and their corresponding topological cuts (S_1, T_1) and (S_2, T_2) for G_1 and G_2. If G is a series composition, the final optimal schedule is obtained through a simple concatenation of σ_1 and σ_2. We select among the two topological cuts one with minimum weight. In case of parallel composition, we first transform subgraphs G_1 and G_2 into chains using the linear arrangement based on schedules σ_1 and σ_2, as defined below.

Definition 4.3 (Linear arrangement)**.** *The* ***linear arrangement*** *of a vertex- and edge-weighted graph $G = (V, E, m, d)$ according to the topological ordering σ of its vertices, is the chain graph $C = (V, E^{(C)}, m^{(C)}, d^{(C)})$ with the same set of vertices, such that:*

- *The chain follows the σ ordering: $(i, j) \in E^{(C)}$ iff $\sigma(j) = \sigma(i) + 1$;*
- *Vertex-weights are kept unchanged: $m_i^{(C)} = m_i$;*

- *Edge weights span the whole interval from the position of their source vertex in the linear arrangement from the one of their destination vertex:*

$$d_{i,j}^{(C)} = \sum_{\substack{(k,l)\in E \text{ s.t.} \\ \sigma(k)\leq\sigma(i) \\ \sigma(l)\geq\sigma(j)}} d_{k,l}$$

The graph obtained by replacing G_1 and G_2 by C_1 and C_2 is a parallel-chain graph with two chains. We consider the topological-cut (S, T) obtained by merging the cuts of both subgraphs. We then apply the ParallelChainMinMem algorithm presented above to compute an optimal schedule. This is summarized in Algorithm 13.

Algorithm 13: SeriesParallelMinMem (G)

Input: $G = (V, E, m, d)$: series-parallel graph in the dataflow model
Output: σ: schedule and (S, T) minimum topological cut
if $V = \{i, j\}$ *and* $E = \{(i, j)\}$ **then**
  $(S, T) \leftarrow (\{i\}, \{j\})$
  $\sigma \leftarrow (i, j)$
  return $\sigma, (S, T)$
else
  G is the composition of two subgraphs: G_1 and G_2
  $\sigma_1, (S_1, T_1) \leftarrow$ SeriesParallelMinMem(G_1)
  $\sigma_2, (S_2, T_2) \leftarrow$ SeriesParallelMinMem(G_2)
  if G *is the series composition of* G_1 *and* G_2 **then**
    Let (S, T) be a cut among $(S_1, T_1 \cup G_2)$ and $(G_1 \cup S_2, T_2)$ with smallest weight
    $\sigma \leftarrow \sigma_1 \oplus \sigma_2$ `/* concatenation of both subschedules */`
    return $\sigma, (S, T)$
  else
    G is the parallel composition of G_1 and G_2
    Let C_1 be the linear arrangement of G_1 according to σ_1
    Let C_2 be the linear arrangement of G_2 according to σ_2
    $(S, T) \leftarrow (S_1 \cup S_2, T_1 \cup T_2)$
    $\sigma \leftarrow$ ParallelChainMinMem $(C_1 \cup C_2, (S, T))$
    return $\sigma, (S, T)$

Contrarily to relative simplicity of the algorithm, its proof of optimality is complex and involved. This is not surprising as it was also the case for Liu's algorithm on trees [65], which extended an already involved algorithm [89]. Our proof, available in [J17], introduces a new graph model with only vertex weights, and extends the relation on schedules proposed by Liu for trees.

4.4 Conclusion of the chapter

In this chapter, we designed an optimal algorithm for peak-memory minimization on series-parallel graphs. The algorithm builds on Liu's optimal algorithm on trees, both because it consists in transforming any series-parallel graphs into a combination of trees, but also as the proof is a generalization of the concepts used in Liu's optimality proof. As in Liu's algorithm, there is a large discrepancy between the relative simplicity of the algorithm, and the complexity of the proof (omitted here).

Series-parallel graphs are well-known examples of graph with small treewidth, and it is thus reasonable to wonder whether there exists an optimal algorithm for graphs with bounded treewidths. However, given the complexity of the proof for series-parallel graphs, it seems unlikely that an easy generalization could exist.

The use of the proposed algorithm is limited to task graphs structured as series-parallel computations, which already constitue an important class of scientific computing applications. However, it may serve as a heuristic to schedule general graphs: any graph may be turned into series-parallel graphs, for example using the SP-ization process of [46] as proposed in [24]. This will probably lead to suboptimal schedules, because some unnecessary synchronization points are added to the graph during this process. The use of such a strategy for the parallel processing of task graphs under limited memory is still under investigation.

Note on the material presented in this chapter. The work presented in this chapter was first started during the internship of Thoma Lambert, and then continued with Enver Kayaaslan, both co-advised with Bora Uçar. It has been published in the TCS journal [J17].

Chapter 5

Scheduling task graphs on hybrid platforms with limited memories

5.1 Adaptation to hybrid platforms

Modern computing platforms are frequently heterogeneous: a typical node is composed of a multi-core processor equipped with a dedicated accelerator, such as a GPU or a Xeon Phi. These two computational units (cores and accelerator) are strongly heterogeneous. To complicate matters, each unit comes with its dedicated memory. Altogether, such an architecture with two computational resources and two memory types, which we call a dual memory system hereafter, leads to new challenges when scheduling scientific workflows on such platforms.

This chapter is devoted to study the effect of such a dual memory system on the problem of task graph scheduling with limited memory. Namely, we would like to efficiently schedule a task graph, that is, minimize its execution time, when tasks have different running time on both computing resources and when the memory of both resources is limited. There is little hope to derive efficient and guaranteed solutions, as this problem is a combination of two independent NP-complete problems:

- Scheduling task graphs, in particular on a hybrid platform, is known to be challenging ([57] proposes a 6-approximation algorithm);
- Traversing task graphs to minimize memory is a hard problem, as outlined on Chapter 1 (Section 1.2.1).

We tackle the problem in two different ways:

- On the theoretical side, we study a simplified version of the problem, where the graph is a tree, tasks can only run on their favorite resource and the objective is to minimize both peaks of memory usage (Section 5.2).

- On the practical side, we propose efficient heuristics for the general problem as well as an ILP formulation that is suitable for small problem sizes (Section 5.3).

5.2 Tree traversals with pre-assigned tasks

We consider in this chapter that we have two different processing units at our disposal, such as a CPU and a GPU. For the sake of generality, we designate them by a color (namely *blue* and *red*). We first consider a variant of the general problem where:

- Each task in the workflow is best suited to a given resource type (say a core or a GPU), and is *colored* accordingly.
- The task-graph is an in-tree, as studied in Chapter 3.
- We temporary forget performance issues and focus on peak memory minimization. Since two type of memory are involved, this is already a bi-criteria problem.

To execute a task of a given color, all data needed for this task (inputs, output and temporary data) must fit within the corresponding memory. As the workflow tree is traversed, tasks of different colors are processed, and capacity constraints on both memory types must be met. In addition, when a child of a task has a different color than its parent, say for example that a blue task has a red child, a communication from the blue memory to the red memory must be scheduled before the red child can be processed. All these constraints require to carefully orchestrate the scheduling of the tasks, as well as the communications between memories, in order to minimize the maximum amount of each memory that is needed throughout the tree traversal.

5.2.1 Application model

We consider applications modeled by tree-shaped task graphs, as in Chapter 3. However, we simplify the dataflow model of Chapter 2 by considering that all temporary data have size zero ($m_i = 0$ for all i). When this is not the case, we add a fictitious leaf child j to node i such that $d_j = m_i$, and set m_i to 0. This way, the memory needed for the processing of node i is preserved by scheduling node j right before node i.

We let $color(i) \in \{red, blue\}$ represent the memory type of task i. If $color(i) = red$, then i is a computational node which operates in the *red* memory, which it uses to load its input data, execute its program and produce its output for its parent. Similarly, if $color(i) = blue$, then i is a computational node which operates in the *blue* memory. Each communication from one memory to the other is achieved through a communication node, which is uncolored. Hence, there are three types of nodes in the tree, *red* or *blue* computational nodes (or tasks), and uncolored communication

nodes. For each data dependence between two tasks assigned to different memories, the output data of the source task need to be loaded from one memory into the other; this is done via a communication node. Thus, in the model, the tree $\mathcal{T}$ does not contain direct edges between *blue* and *red* nodes; memory loads from one memory to the other occur only when processing a communication node. An example of a bi-colored tree is presented in Figure 5.1.

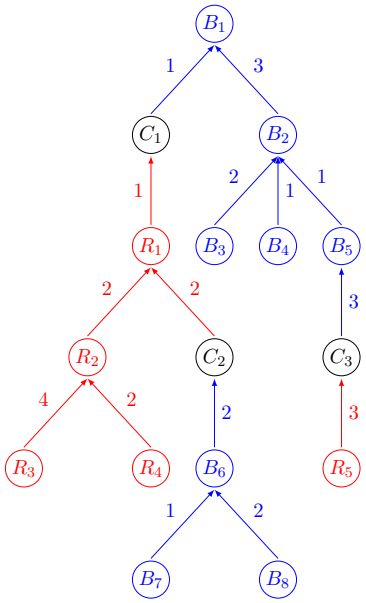


Figure 5.1: An example of bi-colored tree.

Whenever a colored task is processed, the amount of corresponding memory changes as detailled in the (uncolored) dataflow memory model described in Chapter 2. Communication tasks are the only ones that affect both memories simultaneously: when a data of size d_i is sent from the *blue* to the *red* memory, the data is first allocated to the *red* memory (which increases its usage) and then removed from the *blue* memory at the end of the communication.

It is important to stress that a communication node need not be processed right after the execution of its parent. The only constraint is that its processing must precede the execution of its unique child. This flexibility in the schedule severely complicates the search for efficient solutions.

As explained in Chapter 2, a schedule for a bi-colored tree is simply given by a permutation σ of the node, or traversal of the tree, which associates to its step t the (computation of communication) node $\sigma(t)$ that is processed at this step. Such a definition of a schedule forbids any parallelism in the tree, which is not a problem as we do not consider performance issues such as makespan minimization (for now).

We extend the *PeakMemory* notation from the uncolored memory model to bi-colored trees: $BluePeakMem(\sigma, \mathcal{T})$ (resp. $RedPeakMem(\sigma, \mathcal{T})$) is the maximum amount of *blue* (resp. *red*) memory needed for the traversal σ of $\mathcal{T}$. We also consider $OptBluePeakMem(\mathcal{T})$ (resp. $OptRedPeakMem(\mathcal{T})$) as the minimum amount of *blue* (resp. *red*) memory need to process the whole tree when the other memory is unbounded. Note that each of this peak memories can be computed in polynomial time: for computing $OptBluePeakMem(\mathcal{T})$, we zero out all edge weights between two computational red nodes, as well as edges between communication and computational red nodes, and then apply an optimal algorithm for uncolored trees, such as LiuMinMem.

5.2.2 Problem complexity and inapproximability

Our first result shows that considering two memories instead of a single one renders the problem NP-complete.

Theorem 5.1. *Given a tree $\mathcal{T}$ with n nodes, and two fixed memory amounts M_{red} and M_{blue}, finding whether there exists a traversal σ of the tree such that $BluePeakMem(\sigma, \mathcal{T}) \leq M_{blue}$ and $RedPeakMem(\sigma, \mathcal{T}) \leq M_{red}$ is NP-complete.*

The proof of this results, available in [J13], consists in a reduction from the 2-partition problem. In order to study approximations on the bi-criteria problem, we consider the uncolored tree $\mathcal{T}_{\text{unco}}$ obtained from a tree $\mathcal{T}$ by removing all colors: *blue*/*red* computational nodes become all black computational nodes, and communication nodes are replaced by computational nodes with the same inputs/output. The following lemma shows the relation between the *blue*, *red* and uncolored peak memories.

Lemma 5.1. *Given a bi-colored tree $\mathcal{T}$ and a traversal σ of $\mathcal{T}$,*

$$RedPeakMem(\sigma, \mathcal{T}) + BluePeakMem(\sigma, \mathcal{T}) \geq PeakMemory(\mathcal{T}_{\text{unco}}).$$

The proof of this lemma (available in [J13]) considers a traversal σ for the colored tree $\mathcal{T}$ and studies how it translates to $\mathcal{T}_{\text{unco}}$. At each step, it is easy to check that the amount of uncolored memory used is equal to the amount of *blue* memory plus the amount of *red* memory used by σ in $\mathcal{T}$ at the precise same step. Hence, the peak memory of σ in $\mathcal{T}_{\text{unco}}$ is smaller than or equal to $RedPeakMem(\sigma, \mathcal{T}) + BluePeakMem(\sigma, \mathcal{T})$ (as both peaks may not happen simultaneously), and has to be larger than or equal to the minimum $PeakMemory(\mathcal{T}_{\text{unco}})$, which gives the result.

Thanks to this lemma, it is possible to prove that there exists no scheduling algorithm that can simultaneously approximate both *red* and *blue* minimum memories within constant factors, as stated in the following theorem. Since the (usually unfeasible) point of the Pareto diagram with coordinates

($OptBluePeakMem(\mathcal{T})$, $OptRedPeakMem(\mathcal{T})$) is sometimes called the *Zenith* in multi-objective optimization [35], this result amounts to proving that there exists no *Zenith* (or simultaneous) approximation.

Theorem 5.2. *Given two constants α and β, there exists no algorithm that is both an α-approximation for blue memory peak minimization and a β-approximation for red memory peak minimization, when scheduling bi-colored trees.*

To prove this result, for any value of α and β, we recursively build a tree $\mathcal{T}$ such that both $OptBluePeakMem(\mathcal{T})$ and $OptRedPeakMem(\mathcal{T})$ are small constants, while $PeakMemory(\mathcal{T}_{\text{unco}}) = \Theta(n)$, where n is the number of nodes in the tree. Thanks to the previous lemma, for a sufficiently large value of n, we know that either the *blue* or the *red* peak memories with exceed the approximation ratio.

In [J13], we also examine the behavior of postorder traversals (as studied in Section 3.1 for uncolored trees) on bi-colored trees. Interestingly, there exists a single postorder traversal which minimizes both the *blue* and *red* peak memories (among all postorders), and it is possible to compute it in polynomial time. Unfortunately, such a traversal is not very efficient. Consider for example the *blue* node B_1 in Figure 5.1: it has two children a communication node C_1 from the *red* memory and another *blue* node B_2. In a postorder traversal, if we decide to compute the C_1 subtree first, then the communication will take place right before we start the subtree rooted in B_2, thus increasing the amount of *red* memory during its processing. On the contrary, if we consider a more flexible definition of postorder traversals, where communication nodes can be postponed before the data is really needed (e.g., right before the processing of B_1), then computing the best relaxed postorder is NP-complete.

Overall, all previous results show that going from one memory to two memories induced a large gap in complexity, and that there is little hope to find guaranteed algorithms, even without considering performance issues.

5.3 Task graph scheduling with bounded memories

We now come back to the general problem targeted in this chapter, that is, computing efficient schedules for general task graphs $G = (V, E)$ on hybrid platforms with limited memory.

We consider a dual-memory heterogeneous platform with P_1 identical processors which share the first (*blue*) memory and with P_2 identical processors which share the second (*red*) memory, as illustrated on Figure 5.2.

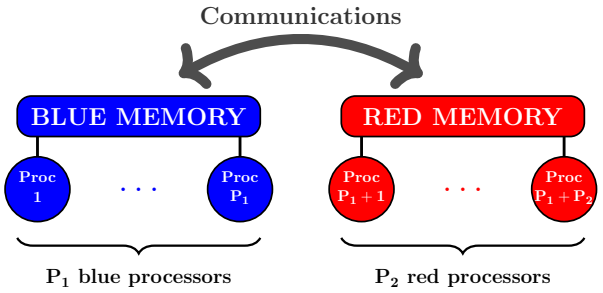


Figure 5.2: Description of the dual-memory platform.

A task may be processed either on a *blue* or on a *red* processor, but its processing time depends on the processor's color: a task i requires a processing time $w_i^{(blue)}$ on one of the *blue* processors and a time of $w_i^{(red)}$ on one of the *red* processors.

Each dependence $(i, j) \in E$ corresponds to the communication of the corresponding data, which is instantaneous if tasks i and j are executed on processors that belong to the same memory. Otherwise, the data produced by task i and needed as input by task j has to be sent from one memory to the other. This transfer takes $c_{i,j}$ time units.

At every time step, each memory should contain the data needed for the processing of the tasks currently executed on processors of the corresponding color (input data, output data and temporary data) as well as the data produced earlier (or received) but not yet consumed by another task (or not yet sent to the other memory). Both memory are limited, and $M^{(blue)}$ and $M^{(red)}$ denote the bounds on the *blue* and the *red* memories. Our objective is to minimize the total processing time, or makespan.

5.3.1 Integer Linear Programming formulation

Following successful attempts to derive an Integer Linear Programming (ILP) formulation for several variants of the task graph scheduling problems, such as [87, 27], we have proposed an ILP for this problem. Memory constraints are naturally expressed in a non-linear way: we want to sum the weights of all data currently in memory, which corresponds to finding all data that (i) are already processed and (ii) are in the considered memory (blue or red), which leads to the product of two decision variables in the ILP. Fortunately, these products can be linearized by adding new variables, using the technique presented in [87, 27]. We end up with an ILP counting 44 different types of constraints (see [W10] for details). For an arbitrary task graph with $|V| = n$ nodes and $|E| = m$ edges, the ILP has $O(m^2 + mn)$ variables and constraints.

Thanks to this ILP, we are able to provide an optimal solution for small and medium instances (using state-of-the-art solvers, it optimally solves

problems with $n = 30$ tasks) and thus, to compare the heuristics presented below with the optimal schedule, to evaluate their absolute quality.

5.3.2 Heuristics

Given the complexity of minimizing the makespan under memory constraints, we propose two heuristics in this section, MemHEFT and MemMinMin. The key idea is to add memory awareness to the design of traditional scheduling heuristics. We briefly present the two memory-aware heuristics as well as an experimental evaluation through simulations and we refer the interested reader to [W10] for more details.

The MemHEFT algorithm

MemHEFT is based on HEFT (Heterogeneous Earliest Finish Time) [84]. The HEFT algorithm is highly competitive and widely used to schedule task graphs on heterogeneous platforms with a low time complexity. HEFT has two major phases: a *task prioritizing phase* for computing the priorities of all tasks, and a *processor selection phase* for allocating each task (in the order of their priorities) to their best processor, defined as the one which minimizes the task finish time. The MemHEFT algorithm follows the same pattern as HEFT.

Task prioritizing phase. This phase is the same as in HEFT and requires the priority of each task to be set with the upward rank value, $rank(i)$, which is based on mean computation and mean communication costs:

$$\forall i \in V,\ rank(i) = \frac{w_i^{(red)} + w_i^{(blue)}}{2} + \max_{j \in Succ(i)} \{rank(j) + \frac{c_{i,j}}{2}\}$$

where $Succ(i)$ denote the immediate successors of i in the graph. The task list is generated by sorting the tasks by non-increasing order of $rank(i)$. Tie-breaking is done randomly.

Processor color selection phase. To select the processor where a task is computed, HEFT computes its Earliest Starting Time (EST) on each possible processor, and select the one with smallest EST. In our model, there are only two processor types, *blue* and *red*, hence each selected task will be mapped on one of two candidates, namely the *blue* and *red* processors with earliest available time. This is why the processor selection phase is renamed as the *processor color selection phase*.

Computing the earliest starting time $EST(i, c)$ of a task i on a processor color c is done in three steps, to take all constraints into account:

- We first compute the $resource\_EST(i, c)$, the earliest time when a processor of color c is available for computation.

- Then we compute $precedence\_EST(i, c)$, the earliest time when the input data of task i may be available in the memory of color c, that is, after predecessors and potential data transfered have completed.
- Finally, we keep trace of the memory consumption of our schedule to ensure that it does not violate the memory constraints. Thus, the algorithm maintains a function $free\_mem(c, t)$ that represents the amount of memory of color c available at time t in the partial schedule constructed so far. For each color c, this is a stair-case function that can be stored as a list of $(t, free\_mem(c, t))$ values. Thanks to this information, we are able to compute $memory\_EST(i, c)$ as the earliest time when there is enough memory for the input, temporary and output data of task i in the memory of color c. Note that the output data of processed tasks that serves as input of data not scheduled yet have to be kept in memory. Thus, depending on the partial schedule and the memory bound, it may happen that there is never enough memory to process i on processors of color c. In that case, $memory\_EST(i, c)$ is set to infinity.

Finally, the EST of task i on processors of color c is set to the maximum of these three quantities. A task is then scheduled on the processor color with smallest EST. Then, communication potentially needed to transfer its input data are scheduled to start as late as possible. If for some task i $memory\_EST(i, c) = \infty$, then MemHEFT fails to schedule the graph within the memory bounds.

The MemMinMin algorithm

The MemMinMin algorithm is the memory-aware counterpart of the MinMin heuristic [19]. It does not include a task prioritizing phase but dynamically decides the order in which tasks are mapped onto resources. Indeed, at each step, MemMinMin maintains the set *available_tasks* representing the tasks whose predecessors have already been scheduled. Then it selects the task $i_{\min}$ in *available_tasks* and the color $c_{\min}$ in $\{red, blue\}$ that minimizes $EFT(i, c) = EST(i, c) + w_i^{(c)}$, where $EST(i, c)$ is defined as in the MemHEFT algorithm using the current partial schedule.

For a task graph with $|V| = n$ nodes and $|E| = m$ edges, both heuristics have a worst-case complexity of $O(n^2(n + m))$.

5.3.3 Experimental evaluation through simulations

The algorithms presented above were evaluated by simulation on different datasets:

- Two sets of synthetic task graphs, the first one with small graphs (30 tasks) and the second one with larger graphs (1000 tasks).

- Task graphs coming from linear algebra operations (LU factorization and Cholesky decomposition), where task running times as well as data transfer times where measured on an actual hybrid CPU/GPU platform.

For simplicity, we assume that both *blue* and *red* memories have the same memory bound: $M^{(blue)} = M^{(red)} = M^{(bound)}$.

First, we compute for each task graph G the makespan $Makespan_{HEFT}$ returned by the memory-oblivious HEFT algorithm and its maximum usage of each memory $BluePeakMem(HEFT, G)$ and $RedPeakMem(HEFT, G)$. The idea is that the classical HEFT algorithm will not be able to schedule the graph on a platform with less than these amounts of *blue* and *red* memory. It is also clear that for larger amounts of *blue* and *red* memory, MemHEFT will take exactly the same decisions as HEFT. We consider

$$M^{(bound)} = \alpha \times max(BluePeakMem(HEFT, G), RedPeakMem(HEFT, G)).$$

If $\alpha \geq 1$, the performance of MemHEFT will be the same as that of HEFT. Figure 5.3 reports the performances of MemHEFT and MemMinMin with $\alpha \in [0, 1]$ (denoted normalized memory). Plain lines show the ratio of the average makespan of our heuristics, and of the solution returned by the ILP, over the makespan of HEFT. The average is computed over all task graphs successfully scheduled within the given memory bounds (to be read on the left scale). Dotted lines show the fraction of task graphs in the whole data set that the algorithms manage to schedule with the given memory bounds (to be read on the right scale).

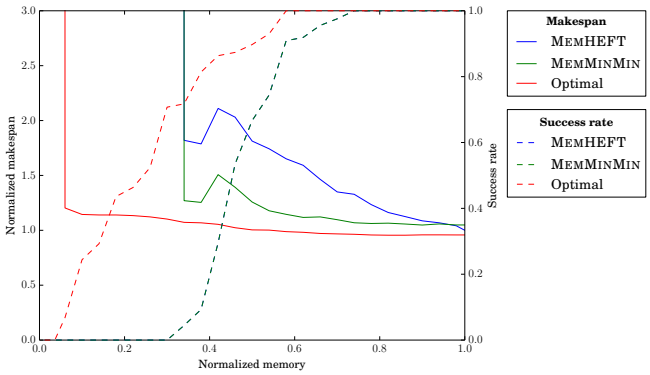


Figure 5.3: Results for small synthetic task graphs.

Results on small synthetic graphs We see that MemHEFT and MemMinMin are really close to the optimal makespan when large amounts of memory are available. MemMinMin provides better results with a makespan overhead smaller than 50% w.r.t. HEFT, even when memory becomes critical. The dotted lines for MemHEFT and MemMinMin in Figure 5.3 are

indistinguishable, which means that both heuristics roughly fail on the same instances when memory becomes critical. MemHEFT and MemMinMin both fail to provide a feasible schedule when the memory bounds is smaller to 35% of the amount required by HEFT. However, the ILP shows that there exists a feasible schedule for approximately 70% of the cases with this memory bound. Our heuristics can provide a feasible schedule for every task graph in the data set when the memory bound is greater than 75% of the amount required by HEFT, whereas, in theory, every task graph can be scheduled down to 60% of this amount.

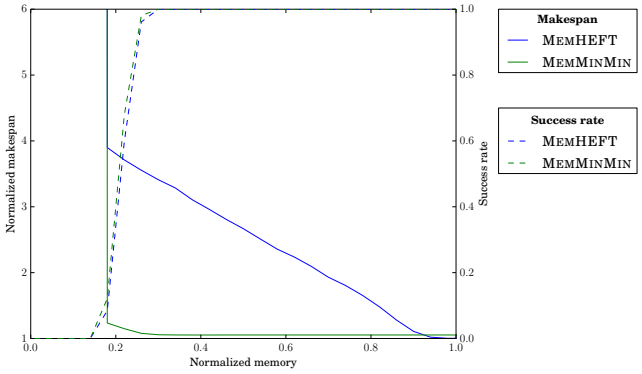


Figure 5.4: Results for large synthetic task graphs.

Results on large synthetic graphs The same experimental procedure has been applied to the large random task graphs, except that the optimal schedule cannot be computed in reasonable time anymore. The average relative makespan of our heuristics are depicted in Figure 5.4. We see that both MemHEFT and MemMinMin succeed to schedule all the task graphs in the data set with only 30% of the memory required by the classical HEFT algorithm. The average makespan of the schedules returned by MemHEFT decreases almost linearly with the amount of available memory. Furthermore, for large amounts of memory, MemHEFT provides slightly better results, while MemMinMin is clearly the best heuristic when memory is critical. MemMinMin provides only a 20% makespan overhead compared to HEFT while using 5 times less memory.

Results on linear algebra graphs We provide results for numerical algebra sets corresponding to a 13×13 tiled matrix. Figure 5.5 depicts the results for LU factorization, whereas Figure 5.6 deals with Cholesky factorization. MemMinMin seems to be the best heuristic when large amounts of memory are available. For both applications, MemHEFT has a 10% makespan overhead compared to MemMinMin when large amounts of memory are available, but it requires far less memory to provide a feasible schedule.

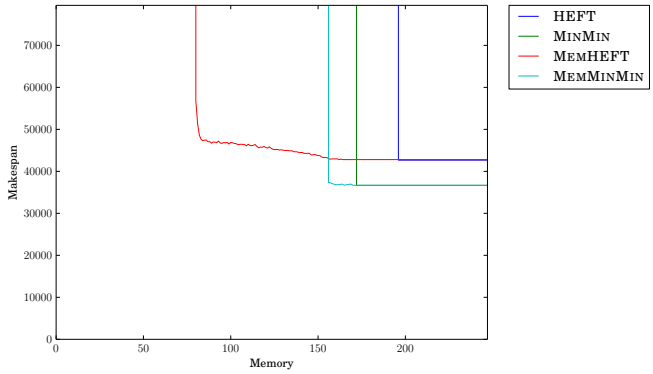


Figure 5.5: Results on the task graph of a 13×13 tiled LU factorization.

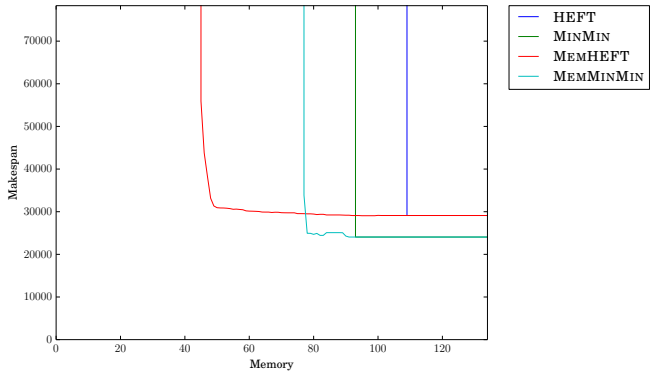


Figure 5.6: Results on the task graph of a 13×13 tiled Cholesky decomposition.

Indeed, Figure 5.5 shows that MemMinMin fails to schedule the LU factorization when each memory does not have enough space to store 155 tiles. However, MemHEFT can still provide a feasible schedule with half available memory. This comes from the fact that in numerical algebra task graphs, a lot of non critical tasks are released early in the process and will eventually be immediately scheduled by MemMinMin, thereby filling up the memory. On the contrary, MemHEFT will focus on the critical path of the graph. Actually MemHEFT fails when $M^{(bound)} \approx 85$ which approximately corresponds to the amount needed to store all the $13 \times 13 = 169$ tiles of the matrix on both memories. Since Cholesky factorization is performed on the lower half of the matrix (94 tiles), the results for the Cholesky factorization lead to similar conclusions.

Overall, both memory-aware heuristics achieve quite satisfactory tradeoffs. In most cases, they are able to drastically reduce the amount of memory needed by HEFT or MinMin, at the price of a relatively small increase in execution time. For small graphs, their absolute performance is close to the optimum as soon as half the memory required by HEFT is available.

5.4 Conclusion of the chapter

In this chapter, we have extended the problem of minimizing the peak memory of a task graph to hybrid platforms with two types of memory, each one being shared by a number of processors. On the theoretical side, we proved that the bi-criterion problem of minimizing both memory types for a task tree was NP-complete, even without makespan consideration and when tasks may only execute on a single type of resource. For the general problem on graphs, we proposed the adaptation of two classical task graph scheduling heuristics to the memory constrained setting, as well as an ILP formulation of the problem.

In the last part of the chapter, we considered the parallel processing of a task graph with limited shared memory. In the next chapter, we will thoroughly study this problem on trees, and especially establish the complexity of the bi-criterion makespan/memory problem.

Note on the material presented in this chapter. The work presented in this chapter was carried out during the PhD of Julien Herrmann, coadvised with Yves Robert. The complexity study on trees was first published in Europar 2013 [C23] and an extended study was published in JPDC [J13]. The designed of memory-aware scheduling strategies for general task graphs was presented at the APCM workshop of IPDPS 2014 [W10].

Chapter 6

Memory-Aware Parallel Tree Processing

In the last section of the previous chapter, we considered the parallel execution of a tree on a hybrid platform. Apart from the specificity of the hybrid model, this asks the question of the complexity of the following bi-criterion problem: how to schedule a task tree on a parallel platform to optimize both the performance and the memory footprint. In this chapter, we focus on this problem, more specifically to minimize the makespan (or total completion time) and the peak memory, in a shared memory environment. Note that in this chapter, we consider only sequential tasks, that is, tasks that run on a single processor. We first assess the complexity of the bi-objective problem, then we move to the design of scheduling heuristics to minimize the makespan in the case where the memory is bounded.

6.1 Complexity of the bi-objective problem

In this section we relate some complexity and inapproximation results of the bi-objective makespan/memory problem. We start by formally defining the problem.

6.1.1 Problem model and objectives

Our goal here is to come up with a simple model to study the trade-offs between memory and makespan when scheduling a tree of tasks. We consider an application described by a tree-shaped task graph in the dataflow model of Chapter 2. In addition, each task i is provided with a computational weight w_i, which corresponds to the task processing time on one processor.

The considered computing platform is made of p identical processors sharing a single memory. Tasks may only execute on a single processor, that is, we do not consider parallel tasks. Any sequential optimal schedule for

peak memory minimization is obviously an optimal schedule for peak memory minimization on a platform with any number p of processors. Therefore, memory minimization on parallel platforms is only meaningful in the scope of multi-criteria approaches that consider trade-offs between the following two objectives:

Makespan: the classical makespan, or total execution time, which corresponds to the time-span between the beginning of the execution of the first leaf task and the end of the processing of the root task.

Peak memory: the maximal amount of memory needed for the computation.

There is a necessary tradeoff between the two objectives: computing many tasks in parallel, and thus starting many subtrees simultaneously, is necessary to reduce the makespan, but is likely to increase the memory usage, and thus the peak memory.

6.1.2 NP-completeness of the bi-objective problem

With the previous two objectives, the decision version of our problem can be stated as follows.

Definition 6.1 (BiObjectiveParallelTreeScheduling)**.** *Given a tree-shaped task graph T with memory sizes and task execution times, p processors, and two bounds $B_{C_{\max}}$ and B_{mem}, is there a schedule of the task graph on the processors whose makespan is not larger than $B_{C_{\max}}$ and whose peak memory is not larger than B_{mem}?*

This problem is obviously NP-complete. Indeed, when there are no memory constraints ($B_{mem} = \infty$) and when the task tree does not contain any inner node, that is, when all tasks are either leaves or the root, then our problem is equivalent to scheduling independent tasks on a parallel platform which is an NP-complete problem as soon as tasks have different execution times [63]. Conversely, minimizing the makespan for a tree of same-size tasks can be solved in polynomial-time when there are no memory constraints [53]. In this section, we consider the simplest variant of this problem. We assume that all input data have the same size ($d_i = 1$ for each task i) and no extra memory is needed for computation ($m_i = 0$ for each task i). Furthermore, we assume that the processing of each task takes one unit time: $w_i = 1$ for each task i. We call this variant of the problem the *pebble-game model* since it perfectly corresponds to the pebble-game problems introduced above: the weight $d_i = 1$ corresponds to the pebble one must put on node i to process it; this pebble must remain there until the parent of task i has been completed, as it is used as the input of the parent of i. Processing a node is done in unit time.

The following theorem proves that, even in this simple variant, the introduction of memory constraints (a limit on the number of pebbles) makes the problem NP-hard. The detailed proof, available in [J15], consists in a complex reduction from the 3-PARTITION problem.

Theorem 6.1. *The BiObjectiveParallelTreeScheduling problem is NP-complete in the pebble-game model (i.e., with* $\forall i, d_i = w_i = 1, m_i = 0$*).*

As the bi-objective problem is NP-complete, it is natural to wonder whether approximation algorithms can be designed. The next theorem shows that it is not possible to approximate both the minimum makespan and the minimum peak memory with constant factors. As outlined in the previous chapter, this is equivalent to saying that there is no *zenith* approximation.

Theorem 6.2. *For any given constants* α *and* β*, there does not exist any algorithm for the pebble-game model that is both an* α*-approximation for makespan minimization and a* β*-approximation for peak memory minimization when scheduling tree-shaped task graphs.*

The proof of this result (available in [J15]) relies on the following lemma, valid for any tree-shaped task graph, which provides lower bounds for the makespan of any schedule.

Lemma 6.1. *For any schedule* S *on* p *processors with a peak memory* M*, we have the following lower bound on the makespan* $C_{\max}$*:*

$$M \times C_{\max} \geq \sum_{i=1}^{n} \left(m_i + d_i + \sum_{j \in Children(i)} d_j \right) w_i$$

In the pebble-game model, it can be rewritten as $M \times C_{\max} \geq 2n - 1$*.*

To obtain this bound, we consider the function $mem(t)$ representing the memory occupation of the schedule S at time t, and the area below its curve: $\int_0^{+\infty} mem(t)dt$. A simple upper bound on this area is $M \times C_{\max}$. Besides, a task i requires an amount of memory $m_i + d_i + \sum_{j \in Children(i)} d_j$ during a time w_i: it contributes for $(m_i + d_i + \sum_{j \in Children(i)} d_j)w_i$ to this area, so that the sum of these quantities is a lower bound on the area, which gives the result.

Theorem 6.2 states that there is no constant approximation ratios for both makespan and peak memory, i.e., approximation ratios independent of the number of processors p. The next result proposes a refined version which analyzes algorithms whose approximation ratios may depend on the number p of processors in the platform.

Theorem 6.3. *When scheduling tree-shaped task graphs in the pebble-game model on a platform with $p \geq 2$ processors, any algorithm that achieves both an $\alpha(p)$-approximation for makespan minimization and a $\beta(p)$-approximation for peak memory minimization verifies*

$$\alpha(p)\beta(p) \;\geq\; \frac{2p}{\lceil \log(p) \rceil + 2}.$$

For makespan-optimal algorithms ($\alpha(p) = 1$), the bound on the peak memory can be increased to $\beta(p) \geq p-1$. Both results are obtained through careful, painstaking analysis of a particular task tree (see [J15]).

6.2 Heuristics for the bi-objective problem

Given the complexity of optimizing the makespan and memory at the same time, we have investigated heuristics and we propose two classes of algorithms. The intention is that the proposed algorithms cover a wide range of use cases, where the optimization focus ranges from the makespan to the required memory. The first class of heuristics consists in splitting the tree into p subtrees, which are then scheduled with a memory-optimizing sequential algorithm. Hence, its focus is more on the memory side. The second class of heuristics extend list scheduling heuristics, which are known to be efficient for makespan minimization.

6.2.1 Processing subtrees in parallel

A natural idea to process a tree in parallel is arguably to split it into subtrees, to process each of these subtrees with a sequentially memory-optimal algorithm (see Section 3.2) and to have these sequential executions happen in parallel. The underlying idea is to assign to each processor a whole subtree in order to enable as much parallelism as there are processors, while allowing to use a single-processor memory-optimal traversal on each subtree. Algorithm 14 outlines such an algorithm, using Algorithm 15 for splitting T into subtrees.

Algorithm 14: ParSubtrees (T, p)

Split tree T into q subtrees ($q \leq p$) and a set of remaining nodes, using SplitSubtrees (T, p).
Concurrently process the q subtrees, each using a memory minimizing algorithm, such as LiuMinMem.
Sequentially process the set of remaining nodes, using a memory minimizing algorithm.

In this approach, q subtrees of T, $q \leq p$, are processed in parallel. Each of these subtrees is a maximal subtree of T. In other words, each of these

subtrees includes all the descendants (in T) of its root. The nodes not belonging to the q subtrees are processed sequentially. These are the nodes where the q subtrees merge, the nodes included in subtrees that were produced in excess (if more than p subtrees were created), and the ancestors of these nodes. An alternative approach, as discussed below, is to process all produced subtrees in parallel, assigning more than one subtree to each processor when $q > p$. The advantage of Algorithm 14 is that we can construct a splitting into subtrees that minimizes its makespan, as stated below in Lemma 6.2.

As w_i is the computation weight of node i, W_i denotes the total computation weight (i.e., sum of weights) of all nodes in the subtree rooted in i, including i. SplitSubtrees uses a node priority queue PQ in which the nodes are sorted by non-increasing W_i. $head(PQ)$ returns the first node of PQ, while $popHead(PQ)$ also removes it. $PQ[i]$ denotes the i-th element in the queue.

SplitSubtrees starts with the root of the entire tree and continues splitting the subtree with largest weight W until this subtree is a leaf node. The execution time of the parallel part of ParSubtrees is that of the largest of the q subtrees of the splitting, hence $W_{head(PQ)}$ for the solution found by SplitSubtrees. Splitting subtrees that are smaller than the largest leaf ($W_j < \max_{i \in T} w_i$) cannot decrease the parallel time, but only increase the sequential time. More generally, given any splitting s of T into subtrees, the best execution time for s with ParSubtrees is achieved by choosing the p largest subtrees for the parallel step. This can be easily derived, as swapping a large tree included in the sequential part with a smaller tree included in the parallel part cannot increase the total execution time. Hence, the value $C_{\max}(s)$ computed on Line 13 of SplitSubtrees is the makespan that would be obtained by ParSubtrees on the splitting computed so far. At the end of algorithm SplitSubtrees, the splitting which yields the smallest makespan is selected.

The following lemma (see proof in [J15]) establishes the makespan-optimality of SplitSubtrees, provided that the resulting subtrees are processed by ParSubtrees, i.e., the largest p subtrees are processed in parallel and then, all remaining nodes are scheduled.

Lemma 6.2. SplitSubtrees *returns a splitting of T into subtrees that results in the makespan-optimal processing of T with* ParSubtrees.

We also proved that ParSubtrees is a p-approximation for both peak memory and makespan minimization, and that the bound is tight for the makespan. This is why we also consider an optimized version, called ParSubtreesOptim, which allocates all produced subtrees to the p processors instead of only p subtrees. This can be done by ordering the subtrees by non-increasing total weight and allocating each subtree in turn to the processor with the lowest total weight. Each of the parallel processors executes

Algorithm 15: SplitSubtrees (T, p)

foreach *node* i **do** compute W_i (the total processing time of the tree rooted at i)

Initialize priority queue PQ with the tree root
$seqSet \leftarrow \emptyset$
$Cost(0) = W_{root}$
$s \leftarrow 1$ `/* splitting rank */`
while $W_{head(PQ)} > w_{head(PQ)}$ **do**
  $node \leftarrow popHead(PQ)$ `/* Remove` $PQ[1]$ `*/`
  $seqSet \leftarrow seqSet \cup node$
  Insert all children of *node* into priority queue PQ
  $p' \leftarrow \min(p, |PQ|)$
  $LargestSubtrees[s] \leftarrow \{PQ[1], \ldots, PQ[p']\}$
  `/* All nodes not in` *LargestSubtrees* `will be processed sequentially. */`
  $C_{\max}[s] = W_{PQ[1]} + \sum_{i \in seqSet} w_i + \sum_{i=p'+1}^{|PQ|} W_{PQ[i]}$
  $s \leftarrow s + 1$
Select subtree set $LargestSubtrees[s_{\min}]$ such that $C_{\max}[s_{\min}] = \min_{t=0}^{s-1} C_{\max}[t]$ (break ties in favor of smaller t) to be processed in parallel

its subtrees sequentially. Note that this optimization should improve the makespan, but it will likely worsen the peak memory usage. Indeed, we proved that it does not have an approximation ratio with respect to memory usage.

The complexity of both variants is dominated by the cost of computing a peak-memory optimal traversal of the subtrees, and is thus $O(n^2)$.

6.2.2 List-scheduling heuristics

ParSubtrees is a high-level algorithm employing sequential memory-optimizing algorithms. An alternative, explored in this section, is to design algorithms that directly work on the tree in parallel. One of the strong points of list scheduling algorithms is that they are $(2 - \frac{1}{p})$-approximation algorithms for makespan minimization [47].

Algorithm 16 outlines a generic list scheduling, driven by node finish time events. At each event at least one node has finished so at least one processor is available for processing nodes. Each available processor is given the respective head node of the priority queue. The priority of nodes is given by the total order O, a parameter to Algorithm 16.

Algorithm 16: ListScheduling(T, p, O)

```
Insert leaves in priority queue PQ according to order O
eventSet ← {0}                                  /* ascending order */
while eventSet ≠ ∅ do                        /* event: node finishes */
    t ← popHead(eventSet)
    NewReadyNodes ← set of nodes whose last children completed at
     time t
    Insert nodes from NewReadyNodes in PQ according to order O
    P ← available processors at time t
    while P ≠ ∅ and PQ ≠ ∅ do
        proc ← popHead(P)
        node ← popHead(PQ)
        Assign node to proc
        eventSet ← eventSet ∪ finishTime(node)
```

From this skeleton of a list scheduling algorithm, we derive two heuristics:

Heuristic ParInnerFirst. From the study of the sequential case, is is known that a postorder traversal, while not optimal for all instances, provides good results for memory minimization [C19]. Our intention in this heuristic is to extend the principle of postorder traversal to the parallel processing. Thus, the priority queue uses the following ordering O: 1) inner nodes, in an arbitrary order; 2) leaf nodes ordered according to a given postorder traversal. Although any postorder may be used to order the leaves, it makes heuristic sense to choose the optimal sequential postorder computed by PostorderMinMem, so that peak memory is minimized (this is what is done in the experimental evaluation below). We do not further define the order of inner nodes because it has absolutely no impact. Indeed, because we target the processing of trees, the processing of a node makes at most one new inner node available, and the processing of this new inner node can start right away on the processor that freed it by completing the processing of its last un-processed child.

Heuristic ParInnerFirst. Contrarily to the previous heuristic, which tries to rely on the memory-friendly properties of *postorder* traversals, we here focus solely on makespan minimization. In tree-shaped task graphs, an inner node depends on all the nodes in the subtree it defines. Therefore, it makes heuristic sense to process the deepest nodes first to reduce any possible waiting time. For the parallel processing of a tree, the most meaningful definition of the depth of a node i is the w-weighted length of the path from i to the root of the tree, including w_i. A deepest node in the tree is a deepest node in a critical path of the

tree. Thus, we consider O wich orders nodes according to their depths. In case of ties, inner nodes have priority over leaf nodes, and remaining ties are broken according to an optimal sequential postorder.

The complexity of both variants is $O(n \log n)$, which corresponds both to the complexity of building O and of managing PQ with n insertions/deletions in time $O(\log n)$. Also, both variants are not approximation algorithms with respect to peak memory minimization.

6.2.3 Experimental comparison

The four previous heuristics have been tested trough simulations on task trees coming from the multifrontal factorization of actual sparse matrices (see [J15] for details).

Table 6.1: Proportions of scenarii when heuristics reach best (or close to best) performance, and average memory (normalized with the optimal memory) and makespan (normalized with the classical makespan lower bound).

Heuristic	Best memory	Within 5% of best memory	Normalized memory	Best makespan	Within 5% of best makespan	Normalized makespan
ParSubtrees	81.1 %	85.2 %	2.34	0.2 %	14.2 %	1.40
ParSubtreesOptim	49.9 %	65.6 %	2.46	1.1 %	19.1 %	1.33
ParInnerFirst	19.1 %	26.2 %	3.79	37.2 %	82.4 %	1.07
ParDeepestFirst	3.0 %	9.6 %	4.13	95.7 %	99.9 %	1.04

The comparison of the previous heuristics is summarized in Table 6.1. It presents the fraction of the cases where each heuristic reaches the best memory (respectively makespan) among all heuristics, or when its memory (resp. makespan) is within 5% of the best one. It also shows the average normalized memory and makespan, where the memory is normalized with the optimal sequential memory, and the makespan is normalized using the classical lower bound (maximum of average work and critical path). For each scenario (consisting in a tree and a number of processors), the memory obtained by each heuristic is normalized by the optimal (sequential) memory, and the makespan is normalized using a classical lower bound, since makespan minimization is NP-hard even without memory constraint. The lower bound is the maximum between the total processing time of the tree divided by the number of processors, and the maximum weighted critical path.

Table 6.1 shows that ParSubtrees and ParSubtreesOptim are the best heuristics for memory minimization. On average they use less than 2.5 times the amount of memory required by the optimal sequential traversal, when ParInnerFirst and ParDeepestFirst respectively need 3.79 and 4.13 times this amount of memory. ParInnerFirst and ParDeepestFirst perform best for makespan minimization, having makespans very close on average to the best achieved ones, which is consistent with their

2-approximation ratio for makespan minimization. Furthermore, given the critical-path-oriented node ordering, we can expect that ParDeepestFirst makespan is close to optimal. ParDeepestFirst outperforms ParInnerFirst for makespan minimization, at the cost of a noticeable increase in memory. ParSubtrees and ParSubtreesOptim may be better trade-offs, since they use (on average) almost only half the memory of ParDeepestFirst for an average increase of 35% in makespan.

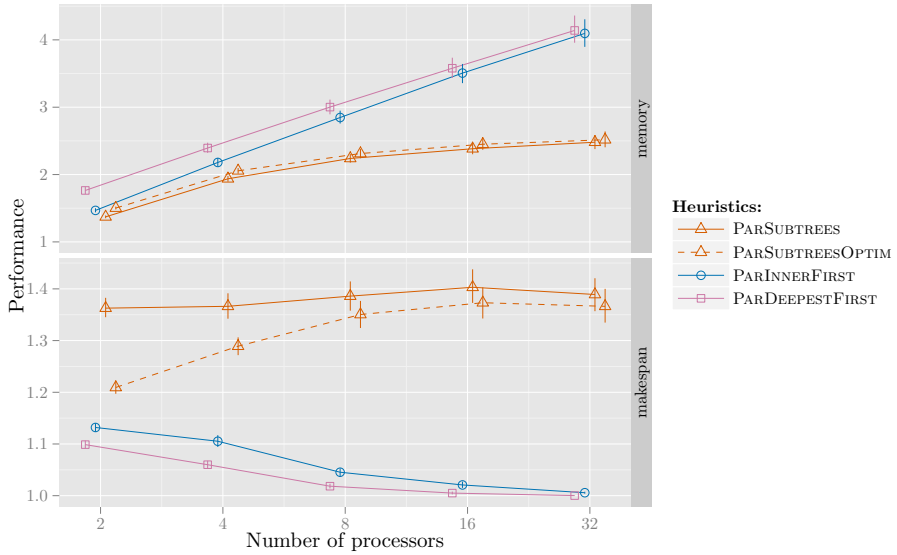


Figure 6.1: Performance (makespan and memory) to the respective lower bounds for the heuristics, excluding the trees with extreme performance. Vertical bars represent 95% confidence intervals.

Figure 6.1 presents the evolution of the performance of these heuristics with the number of processors. The figure displays average normalized makespan and memory, and vertical bars represent 95% confidence intervals for the mean. On this figure, we plot the results for all 608 trees except 76 of them, for which the results are so different that it does not make sense to compute average values anymore (see [J15] for details and results on these outliers). Note that on both figures, values of the different plots are slightly offset on the X axis for better readability. Figure 6.1 shows that ParDeepestFirst and ParInnerFirst have a similar performance evolution, just like ParSubtrees and ParSubtreesOptim. The performance gap between these two groups, both for memory and makespan, increases with the number of processors. With a large number of processors, ParDeepestFirst and ParInnerFirst are able to decrease the normalized makespan (at the cost of an increase of memory), while ParSubtrees has an almost constant normalized makespan with the number of processor.

6.3 Memory-bounded tree scheduling

In this section, we transpose the result and heuristic of the previous section to a realistic scenario: given a parallel platform with a limited shared memory, how to use it efficiently to process a task tree? In other words, given a memory bound M, schedule the task tree so that the memory usage never exceeds M and the makespan is minimized. We present three solutions to this problem, of increasing complexity. The first two approches rely on some simplifying assumptions on the tree that we first present.

6.3.1 Simplifying assumptions

To design our memory-constrained heuristics, we make two simplifying assumptions. First, the considered trees do not have any temporary data, that is, $m_i = 0$ for all tasks i. To still be able to deal with general trees, we may transform any tree with temporary data as follows: we add a new leaf child i' to each task i with $d_{i'} = m_i$ and $w_{i'} = 0$, before setting m_i to zero. This new child may be scheduled right before its parent node and accounts for the temporary data of i.

The second assumption, called the *reduction tree* assumption, considers that the output data of a task is always larger than (or equal to) its inputs data: $d_i \leq \sum_{j \in Children(i)} d_j$. This reduction property is very useful, as it implies that executing an inner node never increases the amount of memory needed to store the resulting data. Again, we may transform general tree to enforce this reduction property by adding fictitious tasks: each task i is given a new child i'' with $d_{i''} = \max(0, d_i - \sum_{j \in Children(i)} d_j)$ and $m_{i''} = 0$. However, contrarily to the removal of temporary data, the new tree may have a larger peak memory than the original one. Removing this strong assumption is the subject of the last (but more complex) strategy presented below.

6.3.2 Memory constrained list-scheduling heuristics

We first propose a memory-constrained version of the list scheduling skeleton presented in the previous section. To achieve this, we modify Algorithm 16 into Algorithm 17 (the code common to both algorithms is shown in light blue and the new code is printed in black). In order to guarantee a bounded peak memory, we check the amount of memory used before processing a leaf (as we know that inner nodes never increase the resulting memory). If the current memory plus the size of the leaf's output data is larger than M, the assignment is stopped until some more memory is freed. Thus, we may deliberately keep some processors idle when there are available tasks, contrarily to what happens in pure list schedules.

Algorithm 17 may be executed with any sequential node ordering O and any memory bound M as long as the peak memory usage of the sequential schedule following node order O is no larger than M. From Algorithm 17 we design two new heuristics, ParInnerFirstMemLimit and ParDeepestFirstMemLimit. ParInnerFirstMemLimit uses for the order O an optimal sequential postorder with respect to peak memory usage. For ParDeepestFirstMemLimit, nodes are ordered by non-increasing depths and, in case of ties, inner nodes have priority over leaf nodes, and remaining ties are broken according to a given optimal sequential postorder.

Algorithm 17: ListSchedulingWithMemoryLimit(T, p, O, M)

```
Insert leaves in priority queue PQ according to order O
eventSet ← {0}                                  /* ascending order */
M_used ← 0                                      /* amount of memory used */
while eventSet ≠ ∅ do                           /* event: node finishes */
    t ← popHead(eventSet)
    NewReadyNodes ← set of nodes whose last predecessor completed
      at time t
    Insert nodes from NewReadyNodes in PQ according to order O
    P ← available processors at time t
    Done ← nodes completed at time t
    M_used ← M_used − Σ_{j∈Done} Σ_{k∈Children(j)} d_k
    while P ≠ ∅ and PQ ≠ ∅ do
        c ← head(PQ)
        if |Children(c)| > 0 or M_used + d_c ≤ M then
            M_used ← M_used + d_c
            proc ← popHead(P)
            node ← popHead(PQ)
            Assign node to proc
            eventSet ← eventSet ∪ finishTime(node)
        else
            P ← ∅
```

Given that the tree follows the previous two assumptions, we are able to bound the peak memory using the modified list scheduling algorithm, as outlined by the following theorem.

Theorem 6.4. *The peak memory requirement of Algorithm 17 for a reduction tree without temporary data processed with a memory bound M and a node order O is at most $2M$, if $M \geq M_{seq}$, where M_{seq} is the peak memory usage of the corresponding sequential algorithm with the same node order O.*

Thus, any scheduling algorithm based on Algorithm 17 may exceed the memory bound by a factor at most two. This factor comes from the fact that during the processing of inner nodes, both outputs and inputs must be loaded, which may result in at most twice the amount of memory needed for the input data. In practice, if the memory bound M is hard, we need to call Algorithm 17 with $M/2$ and with a node order whose sequential peak memory is most $M/2$. This limits the scenarios where these heuristics can be used.

6.3.3 Memory booking heuristic

Contrarily to the two previous heuristics, we now describe a heuristic which aims at satisfying an achievable memory bound M in the strong sense, that is, never uses more than a memory M. This heuristic also relies on the two simplifying properties presented in Section 6.3.1.

To achieve such a goal, we want to ensure that whenever an inner node i becomes ready, there is enough memory to process it. Therefore, we book in advance some memory for its later processing. Our goal is to book as little memory as possible, and to do so as late as possible. The algorithm then relies on a sequential postorder schedule, denoted PO: for any node k in the task graph, $PO(k)$ denotes the step at which node k is executed under PO. Let j be the last child of i to be processed. If the total size of the input data of j is larger than (or equal to) d_i, then only that last child will book some memory for node i. In this case (part of) the memory that was used to store the input data of j will be used for d_i. If the total size of the input data of j is smaller than d_i, then the second to last child of i will also have to book some memory for d_i, and so on. The following recursive formula states the amount of memory $Contrib[j]$ a child j has to book for its parent i:

$$Contrib[j] = \min \left(\sum_{k \in Children(j)} d_k, d_i - \sum_{\substack{j' \in Children(i) \\ PO(j') > PO(j)}} Contrib[j'] \right)$$

If j is a leaf, it may also have to book some memory for its parent. However, the behavior for leaves is quite different than for inner nodes. A leaf node cannot transfer some of the memory used for its input data (because it does not have any) to its parent for its parent output data. Therefore, the memory booked by a leaf node may not be available at the time of the booking. However, this memory will eventually become available (after some inner nodes are processed); booking the memory prevents the algorithm from starting the next leaf if it would use too much memory: this ensures that the algorithm completes the processing without violating the

memory bound. The contribution of a leaf j for its parent i is:

$$Contrib[j] = d_i - \sum_{\substack{j' \in Children(i) \\ PO(j') > PO(j)}} Contrib[j']$$

Note that the value of *Contrib* for each node can be computed before starting the algorithm, in a simple tree traversal. Using these formulas, we are able to guarantee that enough memory is booked for each inner node i:

$$\sum_{j \in Children(i)} Contrib[j] = d_i.$$

Using these definitions, we design a new heuristic, MemBookingInnerFirst, which is described in Algorithm 18. In this algorithm, $Booked[i]$ denotes the amount of memory currently booked for the processing of an inner node i. We make use of a new notation: we denote by $Ancestors(i)$ the set of nodes on the path from i to the root node (excluding i itself).

The following results states that given an achievable memory M and a compatible postorder traversal, MemBookingInnerFirst succeeds to process the whole tree without exceeding the memory bound (see complete proof in [J15]).

Theorem 6.5. MemBookingInnerFirst *called with on a reduction tree without temporary data with a postorder PO, and a memory bound M not smaller than the peak memory of the sequential traversal defined by PO, processes the whole tree with memory M.*

6.3.4 Refined activation scheme

All previous memory-guaranteed heuristics make heavy use of the two simplifying assumptions, i.e., that the tree involves no temporary data and satisfies the reduction property. However, these properties are usually not satisfied in actual task trees. The transformation used to cope with the reduction property is particularly inconvenient, as it may increase the sequential peak memory of the tree, and may prevent its processing with limited memory. This is why we present another scheduling algorithm that does not rely on these assumptions.

This algorithm is based on the simple activation strategy proposed by Agullo et al. [3] to ensure that a parallel traversal of a task tree will process the whole tree without running out of memory. The strategy is summarized in Algorithm 19. The first step is to compute a memory-friendly postorder traversal, for example using PostorderMinMem. This postorder traversal serves as an order for task activation and is denoted by AO (activation order). As previously, it requires that the memory bound M is not smaller than the sequential peak memory of the activation order. At each step of

Algorithm 18: MemBookingInnerFirst (T, p, PO, M)

Input: tree T, number of processors p, postorder PO, memory limit M (not smaller than the peak memory of the sequential traversal defined by PO)

foreach *task* i **do** $Booked[i] \leftarrow 0$
$M_{used} \leftarrow 0$
while *the whole tree is not processed* **do**
  Wait for an event (task completion or $t = 0$)
  foreach *finished non-leaf task* j **do**
    $M_{used} \leftarrow M_{used} - \sum_{k \in Children(j)} d_k$
    $Booked[parent(j)] \leftarrow Booked[parent(j)] + Contrib[j]$
  NewReadyTasks $\leftarrow$ tasks whose last children completed at event
  Insert tasks from *NewReadyTasks* in PQ according to order O
  WaitForNextTermination $\leftarrow$ *false*
  while *WaitForNextTermination* = *false* ***and*** *there is an available processor* P_u ***and*** PQ *is not empty* **do**
    $j \leftarrow pop(PQ)$
    if j *is an inner node and* $M_{used} + d_j \leq M$ **then**
      $M_{used} \leftarrow M_{used} + d_j$
      $Booked[j] \leftarrow 0$
      Make P_u process j
    else if j *is a leaf* ***and*** $M_{used} + d_j + \sum_{k \notin Ancestors(j)} Booked[k] \leq M$ **then**
      $M_{used} \leftarrow M_{used} + d_j$
      $Booked[parent(j)] \leftarrow Booked[parent(j)] + Contrib[j]$
      Make P_u process j
    else
      $push(j, PQ)$
      WaitForNextTermination $\leftarrow$ *true*

the algorithm, it first activates as many tasks as possible, given M, where the activation of a task i consists in allocating all the memory needed for this tasks, i.e., $m_i + d_i$. Then, only tasks that are both activated and whose dependency constraints are satisfied (i.e., all predecessors in the tree are already processed) are available for execution. Another scheduling heuristic may be used to choose which tasks among the available ones are executed: we denote by EO the order giving the priority of the tasks for execution.

Algorithm 19: ACTIVATION(T, p, AO, EO, M)

```
M_Booked ← 0, Activated ← ∅
while the whole tree is not processed do
    Wait for an event (task completion or t = 0)
    // 1. Free the memory booked by j
    foreach just finished task j do
        M_Booked ← M_Booked − m_j − Σ_{k∈Children(j)} d_k
    // 2. Activate new tasks
    while true do
        Remove the first task i from AO
        if M_Booked + m_i + d_i ≤ M then
            M_Booked ← M_Booked + m_i + d_i
            Put i in Activated
        else Put i back to AO, break
    // 3. Process available and activated tasks
    while there is an idle processor P_u and an available task in
      Activated do
        Remove the task i from Activated which is available and has
          maximal priority in EO, make P_u process i
```

This simple procedure is efficient to schedule task trees without exceeding the available memory. However, it may book too much memory, and thus limit the available parallelism in the tree. Consider for example a chain of tasks $T_1 \to T_2 \to T_3$. Algorithm 19 will first book $m_1 + d_1$ for task T_1, then $m_2 + d_2$ for T_2 and finally $m_3 + d_3$ for T_3 (assuming all this memory is available). However, no two tasks of this chain can be scheduled simultaneously because of their precedence order. Thus, it is not necessary to book m_1, m_2 and m_3 at the same time, nor is it necessary to book memory for d_1 and d_3 simultaneously: the memory used for T_1 can be later reused for the processing of T_2 and T_3. By booking memory in a very conservative way, this heuristic may prevent tasks from other branches to be available for computation, and thus delay the processing of these tasks.

We thus propose a new algorithm, named REFINEDACTIVATION, which combines the idea of the activation procedure from [3] with the booking

strategy of the previous heuristic. Similarly to Algorithm 19, we rely on the activation of tasks, following an activation AO which is guaranteed to complete the whole tree within the prescribed memory in the case of a sequential processing. However, activating a task does not correspond here to booking the exact memory $m_i + d_i$ needed for this task: some of this memory may be transfered by some of its descendants in the tree, and if needed, we only book the exact amount of memory that is missing. The core idea of the algorithm is the following: when a task completes its execution, we want (i) to reuse the memory that is freed for one of its ancestors and (ii) perform these transfers of booked memory as late as possible. More precisely, the memory freed by a completed task j will only be transfered to one of its ancestors i if (a) all the descendants of i have enough memory to be executed (that is, they are activated), and (b) if this memory is necessary and cannot be obtained from another descendant of i that will complete its execution later. Finally, an execution order EO states which of the activated and available tasks should be processed whenever a processor is available.

The general framework of the resulting RefinedActivation strategy (presented in details in [C28]) follows the one of Algorithm 19. Step 1 (releasing the memory of task j) is replaced by a careful distribution of the memory of task j to its ancestors, while Step 2 (task activation) checks if the memory already booked for the task by its descendants at previous iterations plus the available memory is sufficient to activate a task. Both procedures are complex, especially when one takes care to limit the running time of the algorithm through the use of special data structures. Note that thanks to the refined booking strategy, the activation order AO does not need to be a postorder, but can be any topological order of the tree.

Theorem 6.6. *Given a task tree with n tasks and maximal height H, a memory bound M, an activation order AO whose sequential peak memory is not larger than M and an execution order EO,* RefinedActivation *processes the whole tree without exceeding the memory bound M. Its time complexity is in $O(n(H + \log n))$.*

The dependency of the running time in H, the height of the tree, is explained by the fact that at each task termination, the algorithm has to scan its ancestor to distribute the released memory. On very deep trees, this may cause a large running time, as outlined below in the experiments, where we also discuss how to avoid this problem.

6.3.5 Experimental comparisons of the heuristics

On reduction trees without temporary data

We first present the results on simulations performed on trees which satisfy the simplifying assumptions. The original trees used for these simulations,

originating from the multifrontal factorization of sparse matrices, are first transformed into reduction trees without temporary data as described in Section 6.3.1. For each tree, we then compute the postorder traversal that minimizes the peak memory M_{seq}. The different heuristics are then tested with a factor of this minimal memory, which we call below normalized memory bound. We only plot an average result when a given strategy was able to schedule at least 95% of the trees within the memory bound. It may well happen that some heuristic is be able to cope with some memory bound: ParInnerFirstMemLimit needs at least twice as much memory as the best sequential postorder, and ParDeepestFirstMemLimit needs twice as much memory as the sequential deepest first traversal, which is usually less memory friendly than a postorder. For each heuristic and each tree, we compute the normalized makespan as the makespan divided by the classical lower bound (maximum between the critical path and the average work).

Figure 6.2 presents the results of these simulations. It also includes two variants ParInnerFirstMemLimitOptim and ParDeepestFirstMemLimitOptim which are slitghly more aggressive when starting leaves, but with the same memory guarantee as ParInnerFirstMemLimit and ParDeepestFirstMemLimit. This figure shows that when the memory is very limited ($M < 2M_{seq}$), MemBookingInnerFirst is the only heuristic that can be run, and it achieves reasonable makespans. For a less strict memory bound ($2M_{seq} \leq M < 5M_{seq}$ or $2M_{seq} \leq M < 10M_{seq}$ depending on the number of processors), ParInnerFirstMemLimit is able to process the tree, and achieves better makespans, especially when M is large. Finally, when memory is abundant, ParDeepestFirstMemLimit is the best among all heuristics.

Results on general trees

We now move to the evaluation of RefinedActivation. We performed similar simulations as in the previous case, and compared it to both the existing Activation strategy and our previous heuristics MemBookingInnerFirst. We only present results for 8 processors, as different numbers of processors show similar trends. We used the same memory bound based on the best sequential postorder traversal. The makespan was normalized using the maximum between the classical bound and the new bound derived in Lemma 6.1. The best sequential postorder was used as the activation order AO and execution order EO for both Activation and RefinedActivation.

Figure 6.3 plots the average normalized makespan of all strategies for various memory constraints. We notice that for a memory bound twice the minimum memory, RefinedActivation is 1.4 times faster than Activation on average. However, even this particular speedup spans a wide interval (between 1 and 6) due to the large heterogeneity of the actual trees.

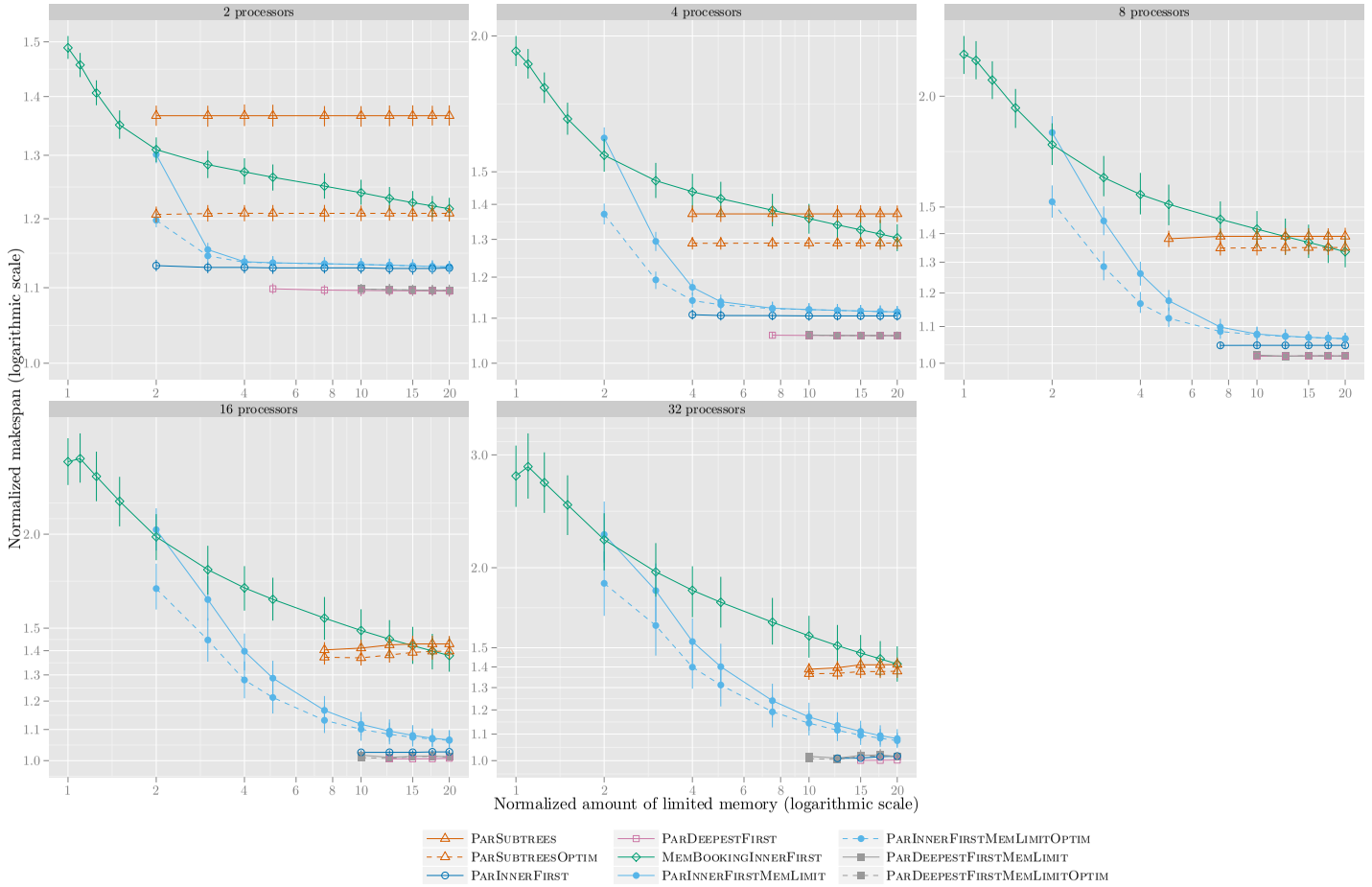


Figure 6.2: Memory and makespan performance of memory-constrained heuristics (logarithmic scale).

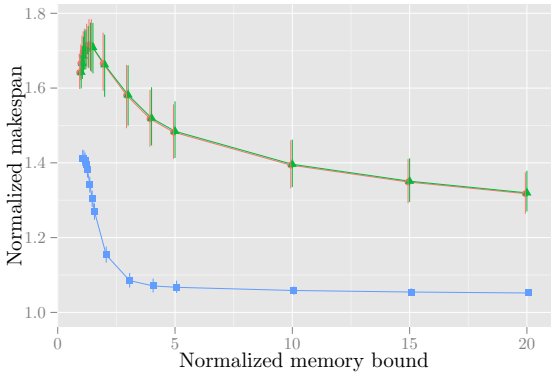


Figure 6.3: Makespan of actual trees depending on the memory bound (Red: Activation, Green: MemBookingInnerFirst, Blue: RefinedActivation).

Note that Activation and MemBookingInnerFirst give very similar results: this is explained by the fact that MemBookingInnerFirst first transforms the trees before applying a smart booking strategy: on these trees, adding fictitious edges has the same effect than booking to much memory (as Activation does) and hinders the benefit of the booking strategy. We also note that RefinedActivation is able to take advantage of very scarce memory conditions: as soon as the available memory increases from its minimum value, its makespan quickly drops and is within 10% of the lower bound for three times the minimum memory, leaving very little room for better algorithms.

Different activation and execution orders have been tested: critical path, optimal sequential traversal for peak memory minimization, and other postorders. We notice that it only slightly changes the results of both Activation and RefinedActivation (by a few percents), and that in general, using the critical path as an execution order allows to reach the best makespan.

Figure 6.4 presents the time needed to schedule the trees by all algorithms, which allows to experimentally verify the time complexity of RefinedActivation exhibited in Section 6.3.4. We notice that for height up to 1,000 nodes, all algorithms need at most a few microseconds per node in the tree. Above this threshold, the scheduling time of RefinedActivation grows and can be as large as a few tens of seconds for very large and very high trees. This problem could be avoided by implementing a limit in the ancestor exploration when distributing the memory freed by a node. However, we notice that the trees on which RefinedActivation allows to speedup the processing are wide and short, and that it both gives poor

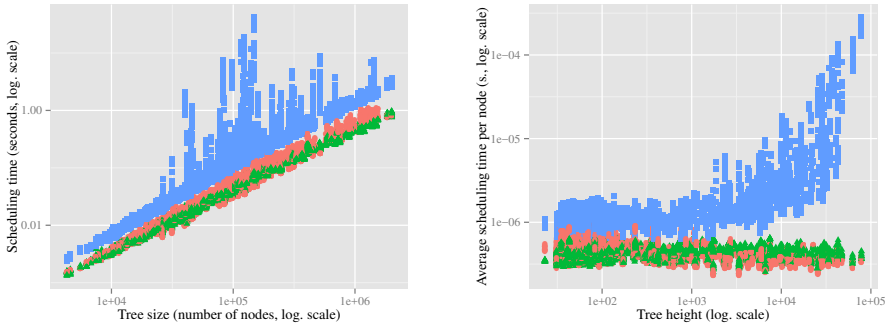


Figure 6.4: Running times of the heuristics on assembly trees (same legend as Figure 6.3)

speedups and long running times on high and thin trees. Depending on the height and width, it is possible to predict which scheduling algorithm will perform best.

6.4 Conclusion of the chapter

In this chapter, we have discussed the problem of scheduling task trees on parallel computing platforms with bounded shared memory. We have studied the complexity of the bi-criterion problem, and shown that it was NP-complete and has no approximation with respect to both the optimal makespan without memory constraint and the optimal memory. We presented a large variety of heuristic solutions to schedule task trees on such platforms, from simple list-scheduling strategies to more involved algorithms that can deal with a hard memory bound.

A major limitation of the work presented here is the fact that each task is sequential. While it is well justified to start by a simple task model to study the complexity of the problem, such a limitation is more serious when simulating the execution of large task trees, and especially the ones we used from sparse linear algebra: the tasks at the top of the tree are usually very large and would benefit from some data parallelism. There are two answers to these limitations. Firstly, the heuristics designed in this chapter are totally agnostic about the task processing times, and most of them rely on an external scheduling policy (such as the task execution order EO) for scheduling available tasks. Thus, one could easily plug a parallel task scheduler in most heuristics. Secondly, we have also studied the problem of modeling and scheduling trees of parallel tasks in other studies that do not deal with memory considerations. We first revisited the mostly theoretical model proposed by Prasanna and Musicus in [73], and then we analyzed scheduling heuristics for parallel task trees in a more realistic two-

segment rooftop model [C27, J18]. These complementary studies are not further described in the present document as they depart from its main focus: memory-aware algorithms.

Note on the material presented in this chapter. The study of the bi-criterion makespan-memory problem was initiated during the sabbatical stay of Oliver Sinnen, from University of Auckland, in our team in 2012, in collaboration with Frédéric Vivien. After presenting a first version of this work at the IPDPS'2013 conference [C22], we continued on this subject with Lionel Eyraud-Dubois and finally published the complete study including the heuristics for the bi-criterion problem and for reduction trees with bounded memory in the ACM TOPC journal in 2015 [J15]. We later re-opened the subject to take advantage of the activation strategy during the internship of Clément Brasseur, co-advised with Guillaume Aupy, to design the last memory-bounded algorithm, presented at IPDPS'2017 [C28].

Part II

Minimizing data movement for matrix computations

Foreword

In the following chapters, we present some contributions that are are not related to peak memory or I/O minimization, but to a closely related subject: minimizing the amount of data movement when performing computations on a distributed platform. The objective is then to take advantage of data locality when scheduling tasks, or to carefully distribute the data before the computation. Each of the following three chapters study a particular problem related to data movement.

Chapter 7

Matrix product for memory hierarchy

7.1 Introduction

In this chapter, we come back to the classical matrix product problem that we have encountered in the introductory chapter. We saw that when the size M of the available memory is limited, using a blocked algorithm reduces the amount of data movement and even reaches the lower bound $O(n^3/\sqrt{M})$, for square matrices of size n. The precise lower bound obtained in Section 1.3.1 writes $\Theta(n^3/\sqrt{8M})$. It was later improved into $\Theta(n^3\sqrt{27/8M})$ by Pineau et al. [33] for parallel 2D algorithm, and again improved by Langou [59] to $\Theta(2n^3/\sqrt{M})$ for both 2D and 3D algorithms, which is the largest possible bound as it is achieved by existing algorithms. These bounds apply to any system with a fast and bounded main storage (memory, cache, etc.) and an infinite but slower secondary storage (disk, memory, etc.).

These studies consider a single level of limited memory, and aim at minimizing the amount of data movement between this memory and a secondary storage which is large enough to contain all the data needed for the computation (the whole three matrices). However, modern platforms involve a hierarchy of memory and caches, from large and slow memories to fast and limited caches. We consider the problem of minimizing the amount of data movement on a multicore processor described by a simple memory hierarchy, made of two levels of caches, as illustrated on Figure 7.1.

Matrix multiplication has extensively been studied on parallel architectures. Two well-known parallel versions are Cannon's algorithm [20] and the ScaLAPACK outer product algorithm [14]. 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

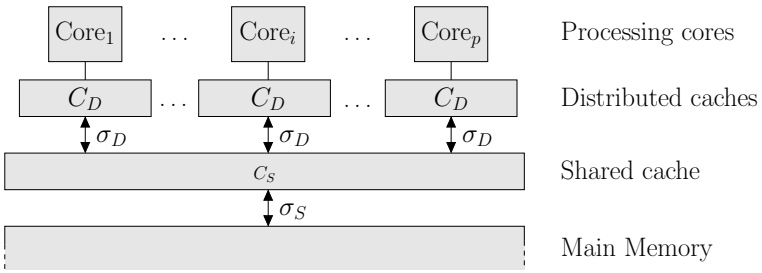


Figure 7.1: Multicore architecture model.

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, such algorithms are not well suited for multicore architectures, where data access is made through a hierarchy of caches. We need to take further advantage of data locality to minimize data movement.

7.2 Platform 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 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, as shown in Figure 7.1. 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: $C_S \geq p \times C_D$. Our caches are also "fully associative", and can therefore store any data from main memory.

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 its distributed cache or in the shared cache, the same mechanism applies. When the data is in the distributed cache of another core, this remote copy is furthermore invalidated for cache coherence. Rather than trying to model this complex behavior, we assume in the following an *ideal cache model* [40]: 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 [40] 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 $2L$ and 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.

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 [14]. 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.

7.3 Objectives

The key point to performance in a multicore architecture is efficient data reuse. Thus, we aim at designing algorithms that minimize data movement but still fully use the available computing cores. 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 consider the bi-objective problem that aims at minimizing 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 data access time T_{data} required for data movement. With the previously introduced bandwidth, 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.

7.4 Lower bounds

The lower bounds on data movement seen in the introduction can be adapted to our hierarchy of caches. We first define the communication-to-computation ratio, or CCR, for both the shared and distributed caches. Let $comp(c)$ be the amount of computation performed by core c and M_S the number of cache misses in the shared cache. Then, the CCR for the shared cache can be computed as:

$$CCR_S = \frac{M_S}{\sum_c comp(c)}.$$

We consider the lower bound $O(2mnz/\sqrt{M})$ on data movement as improved by Langou [59], and identify the limited memory of size M to the shared cache of size C_S. We obtain the following lower bound on the CCR:

$$CCR_S \geq \frac{2}{\sqrt{C_S}}.$$

In case of the distributed caches, we first apply the result to a single core c, with cache size C_D:

$$CCR_c = \frac{M_D(c)}{comp(c)} \geq \frac{2}{\sqrt{C_D}}.$$

We define the overall distributed CCR as the average of the CCR_c of all cores. We assume that all computing cores are fully used so that the overall amount of computation (mnz operations) is equally balanced among all cores: $comp(c) = mnz/p$. Therefore, the bound on the distributed CCR is

$$CCR_D \geq \frac{2}{\sqrt{C_D}}.$$

7.5 Algorithms

We propose three algorithms to minimize (i) the shared cache misses, (ii) the distributed cache misses and (iii) the data access time T_{data} defined above. The following algorithms are adapted from a previous strategy by Pineau et al. [33], called Maximum Reuse algorithm, for minimizing communication when performing a matrix product on a master-worker platform with limited memory.

In the blocked algorithm presented in Section 1.1.2, 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 $O(mnz/\sqrt{M})$ for a memory of size M but it does not use the memory optimally. The Maximum Reuse Algorithm [33] proposes a more efficient memory allocation: it splits the available memory into $1 + \mu + \mu^2$ blocks, storing a square block $C_{i_1 \dots i_2, j_1 \dots j_2}$ of size μ^2 of matrix C, a fraction of row $B_{k, j_1 \dots j_2}$ of size μ of matrix B and one element $A_{i,k}$ of matrix A (with $i_i \leq i \leq i_2$, $1 \leq k \leq z$ and $j_1 \leq j \leq j_2$). This enables the computation of $C_{i, j_1 \dots j_2} += A_{i,k} \times B_{k, j_1 \dots j_2}$. 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 adapt the Maximum Reuse Algorithm to multicore architectures, we must take into account both cache levels. Depending on our objective, we adapt the previous data allocation scheme so as to fit with the shared cache, with the distributed caches, or with both. 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.

7.5.1 Minimizing the shared cache misses

To minimize the number of shared cache misses, we adapt the Maximum Reuse Algorithm with parameter λ. A square block C_{block} of size λ^2 of C is allocated in the shared cache, together with a fraction of a row of λ elements of B and one element of A. Then, the row of C_{block} is distributed and computed by the different cores. This is described in details in Algorithm 20, and the memory layout is depicted in Figure 7.2.

In this algorithm, the whole matrix C is loaded once 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 of $M_S = mn + 2mnz/\lambda$ shared cache misses. For large matrices, this leads to a shared cache CCR of $2/\lambda$, which is close to the lower bound.

Algorithm 20: SharedOpt

```
for Step = 1 to (m×n)/λ² do
    Load a new block C_block (of size λ × λ) from C in the shared cache
    for k = 1 to z do
        Load a fraction of row B_row (of size λ) from row z of B in the
         shared cache
        Distribute B_row to the distributed caches
        for l = 1 to λ do
            foreach core c in parallel do
                Load the element a = A[l, k] in the shared and
                 distributed cache
                Load a fraction of row C_row (of size λ/p) from C_block in
                 the distributed cache
                Compute the new contribution:
                 C_row ← C_row + a × B_row
                Write back C_row to the shared cache
    Write back the block C_block to main memory
```

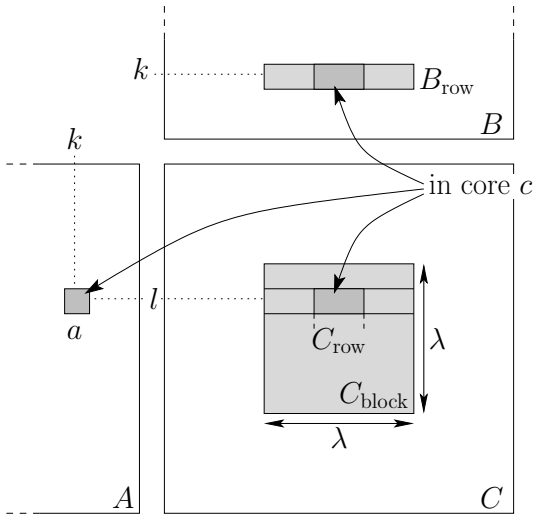


Figure 7.2: Data layout for Algorithm 20.

7.5.2 Minimizing distributed cache misses

Our 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 cores work on different blocks of C. Thanks to the constraint $p \times C_D \leq C_S$, we know that the shared cache has the capacity to store all necessary data. The overall number of distributed cache misses on a core will then be $M_D = \frac{1}{p}(mn + 2mnz/\mu)$ (see [55] for details). For large matrices, this leads to a distributed cache CCR of $2/\mu$, which is close to the lower bound.

7.5.3 Data access time

To get a tradeoff between minimizing the number of shared cache and distributed cache misses, we now aim at minimizing $T_{\text{data}} = \frac{M_S}{\sigma_S} + \frac{M_D}{\sigma_D}$. 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 fraction of the M_D cache misses 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 data distribution is illustrated on Figure 7.3 and the sketch of the algorithm, detailed in [55], is the following:

1. A block from C of size $\alpha \times \alpha$ is loaded in the shared cache. Its size satisfies $p \times \mu^2 \leq \alpha^2 \leq \lambda^2$. 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 cyclicly 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 cache.
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,

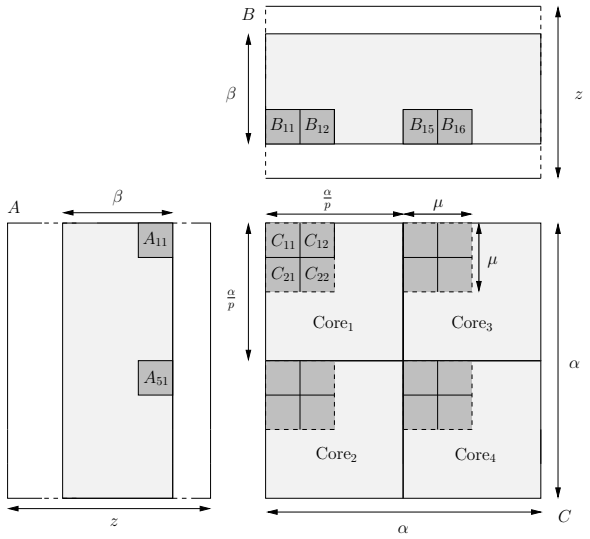


Figure 7.3: Data distribution of matrices A, B and C: light gray blocks reside in shared cache, dark gray blocks are distributed among distributed caches ($\alpha = 8, \mu = 2,\ p^2 = 4$).

another $\mu \times \mu$ block of C residing in shared cache is distributed among every distributed cache, 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.

With this algorithm, we get: $T_{\text{data}} = \frac{1}{\sigma_S}(mn + \frac{2mnz}{\alpha}) + \frac{1}{\sigma_D}(\frac{mnz}{p\beta} + \frac{2mnz}{p\mu})$. Together with the constraint $2\alpha \times \beta + \alpha^2 \leq C_D$, this allows us to compute the best value for parameters α and β, depending on the ratio σ_S/σ_D (see [55] for details).

7.6 Performance evaluation

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 [14] and Toledo [83] 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.

However, we also implemented and tested our algorithms on real hardware to evaluate the impact of real 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.

7.6.1 Evaluation through simulations

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, ...).

We implemented a simple simulator, which is fully described in [55]. It implements two data replacement policies: the classical *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;

We have implemented two reference algorithms: (i) *Outer Product*, the algorithm in [14], for which we organize cores as a (virtual) processor torus and distribute square blocks of data elements to be updated among them; and (ii) *Equal*, an algorithm inspired by the blocked algorithm (Algorithm 2) of Section 1.1.2 from [83], which uses a simple equal-size memory scheme: one third of distributed caches is equally allocated to each loaded matrix sub-block. As this algorithm deals with a single cache level, we used two versions of it: *Shared Equal* for shared cache optimization, and *Distributed Equal* for distributed cache optimization. We also implemented the three algorithms proposed above:

- *Shared Opt.* which minimizes the shared caches misses,
- *Distributed Opt.* which minimizes the distributed cache misses and
- *Tradeoff* which minimizes the data access time.

All tests are made on square matrices ($m = n = z$).

LRU vs IDEAL replacement policies

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 7.4 shows the total number of shared cache misses for Shared Opt., in function of the matrix dimension. 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 very close, thereby experimentally validating the prediction of [40]. Similar results are obtained for Shared

Equal. Furthermore, the same conclusions hold for Distributed Opt. and Distributed Equal, see [55]. Note that Outer Product is insensitive to cache policies.

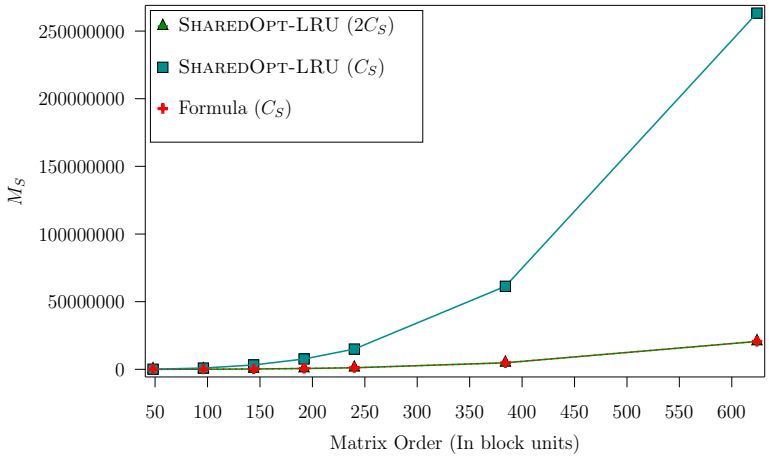


Figure 7.4: Impact of LRU policy on the number of shared cache misses M_S with $C_S = 977$.

This leads us to run the following 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 uses entire cache sizes (C_S and/or C_D) as a parameter for the algorithms
- The **LRU-50%** setting, which relies on a LRU cache data replacement policy, but uses only one half of cache sizes as a parameter for the algorithms. The other half is used by the LRU policy as kind of an automatic prefetching buffer.

Performance through simulations

We tested the number of cache misses of several variants of the algorithms presented above, both on the shared cache, the distributed caches and in terms of access time. We report here only a summary of the results and refer to [55] for a complete discussion.

Figure 7.5 depicts the number of shared cache misses as well as the lower bound $m^3\sqrt{27/8C_S}$ from [33][1]. We see that Shared Opt. performs significantly better than Outer Product and Shared Equal for the LRU-50% policy. Under the IDEAL policy, it is closer to the lower bound, but this

[1]The better bound of $2m^3/\sqrt{C_S}$ of [59] is posterior to this work.

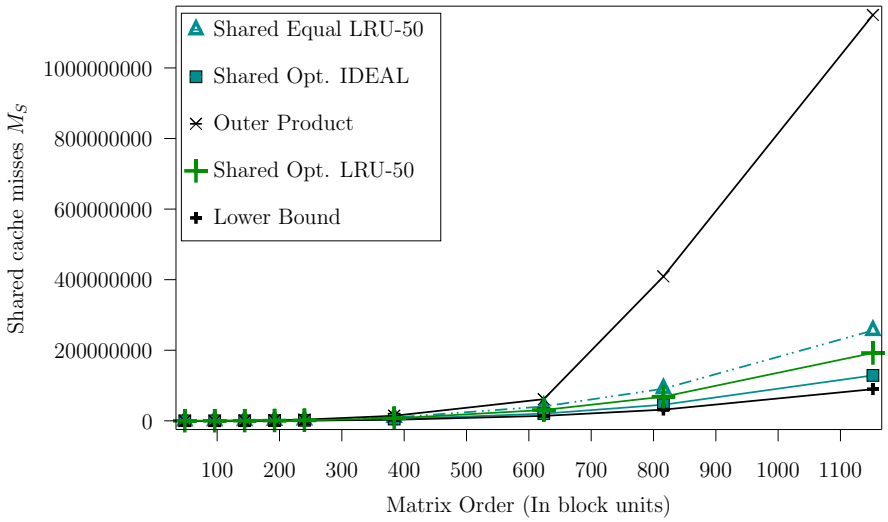


Figure 7.5: Shared cache misses M_S in function of matrix order.

latter setting is not realistic. When tested for distributed cache misses, the Distributed Opt. performs significantly better than the others, as expected. The simulations using the access time T_{data} show that TradeOff offers the best performance, although Shared Opt. is sometimes very close.

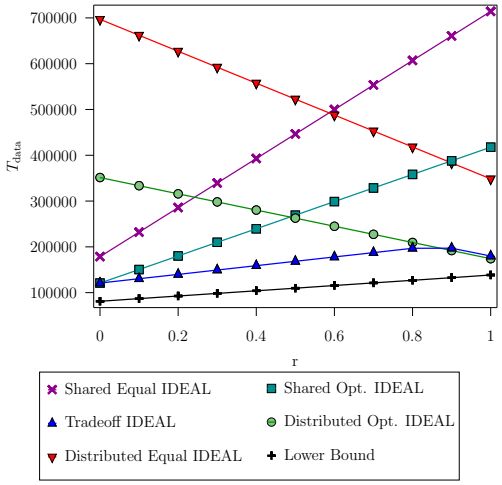


Figure 7.6: Cache bandwidth impact on T_{data} in function of $r = \sigma_S/(\sigma_S + \sigma_D)$.

Figure 7.6 shows the impact on the relative cache bandwidth on the access time T_{data}. We see that Tradeoff performs best, and still offers the best performance even after distributed misses have become predominant. When the latter event occurs, plots cross over: Shared Opt. and Distributed Opt. achieve the same T_{data}. We also point out that when $r = 0$, Tradeoff

achieves almost the same T_{data} than Shared Opt., while when $r = 1$, it ties Distributed Opt.

7.6.2 Performance evaluation on actual multicore platforms

In addition to the previous simulations, the three algorithms proposed above have been implemented on a real multicore CPU as well as on a GPU. The goal of these experiments is to evaluate the impact of real hardware caches, which are not fully associative in practice and to measure the influence of the cost of cache misses on the execution time.

Performance evaluation on CPUs

The proposed algorithms as well as the Outer Product and the variants of Equal have been implemented and compared to vendor libraries (MKL and GotoBLAS2) on a quad-core processor[2]. We summarize here the results, which are fully described in [55].

When comparing the time needed to perform a matrix product, as in Figure 7.7, most of the times, all implemented algorithms offer very similar performance and are only able to reach 89% of the vendor libraries. Considering the fact that libraries are really low-level implementations which have required a huge effort to develop for each specific architecture, whereas we aimed at design higher-level strategies, this result is not surprising nor discouraging.

We have also measured the number of cache misses induced by each algorithms and by the libraries. The gap between the algorithms minimizing C_S and the libraries is smaller, our algorithms even create less caches misses than the libraries in the case of big matrices. However, most of the cache misses caused by the libraries are automatically prefetched by the processor contrarily to our algorithms which have a more irregular access pattern.

Since on such CPUs, it is hard to precisely predict and thus control the cache behavior, our algorithms are unable to outperform tailored vendor libraries.

Performance evaluation on GPUs

We also adapted our algorithms to the architecture of GPUs: GPUs have several level of memory, including on-board RAM memory as well as on-chip memory, which is order of magnitude faster than its RAM. We have taken into account this speed heterogeneity when adapting our algorithms to this architecture. The main idea is to consider the on-board RAM memory as the shared cache and on-chip memory as the distributed caches. We have also

[2]Note that this study dates back to 2009, which explains the old hardware and the small number of cores.

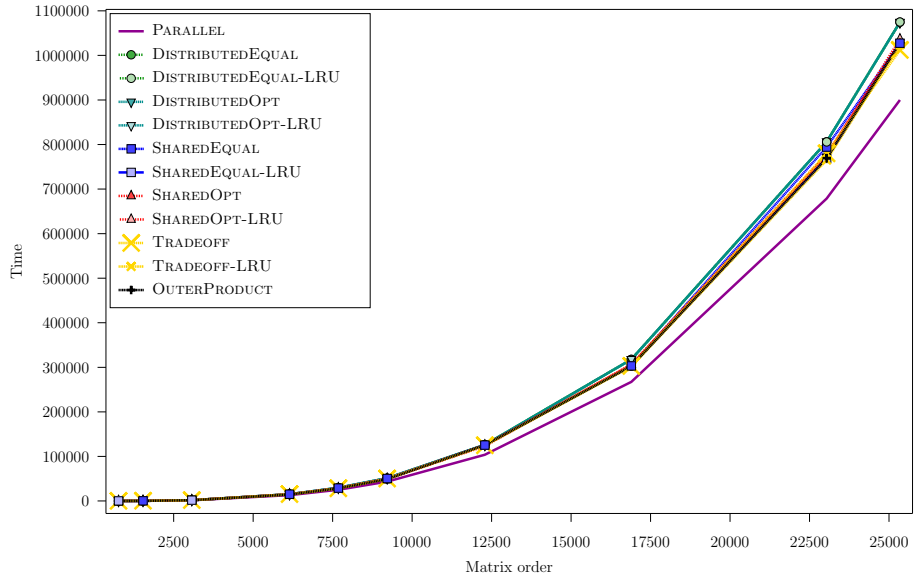


Figure 7.7: Running times of the algorithms on a CPU compared to the GotoBLAS2 library.

slightly modified its design to handle a non-square grid of $p \times q$ processors, to reserve additional memory in the shared memory in order to overlap PCIE transfers between the host and the GPU with computations, and to use a large number of threads to fully use the GPU processing power.

We compared our algorithms to the vendor CUBLAS library on a GeForce GTX 285 GPU, embedding 240 cores and 2GB global memory. As depicted in Figure 7.8, our algorithms are slightly better for two matrix dimensions, and worse by up to 40% in four other cases. This is explained by the use of different kernels in CUBLAS depending of the matrix size: some optimized kernels make use of GPU-specific hardware features, such as texture units, which we ignore, and they are thus able to largely outperform our algorithms. The specific kernel which is outperformed by our algorithms does not make use of these features.

As in the CPU experiments, we have also measured the number of cache misses, and show that CUBLAS experiences on average 1.4 more shared cache misses and between 1.9 and 3.4 more distributed cache misses.

7.7 Conclusion of the chapter

In this study, we proposed a simple model of cache hierarchy for multicore architectures, and we extended the known lower bound for matrix multiplication to this new model. We proposed several matrix multiplication algorithms designed to reduce several data movement metrics: shared cache

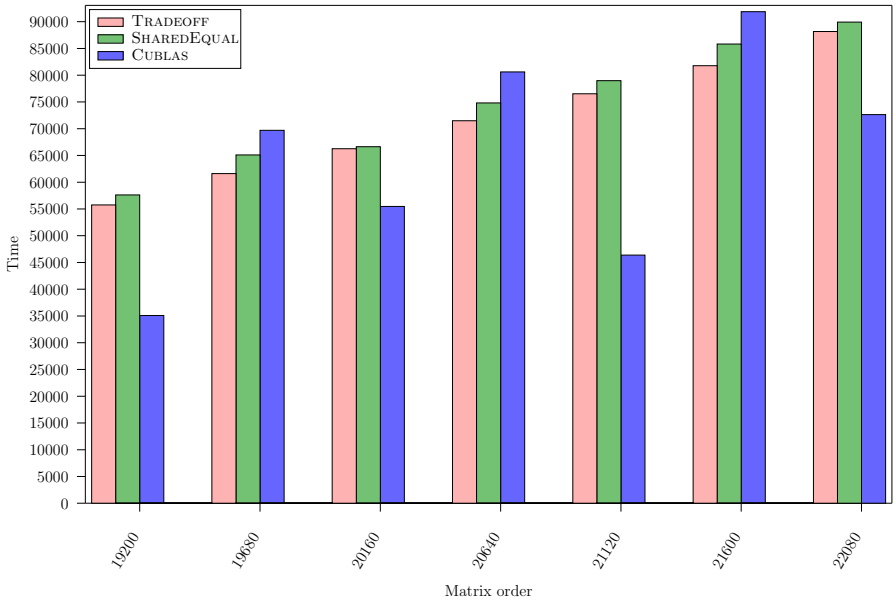


Figure 7.8: Running times on GTX285

misses, distributed cache misses and data access time. We proved by simulations that the proposed algorithms succeed to significantly reduce data movement in the proposed model, before moving to real implementation on both CPUs and GPUs. When compared to low-level vendor libraries, our analytical and high-level approach is rarely able to decrease the processing time of a matrix product. However, it is generally able to generate less cache misses. Cache misses are not always the key to performance, due to complex hardware prefetching, especially for modern CPUs. However, we stress out that reducing data movement is also important for reducing the energy consumption of the processor.

Note on the material presented in this chapter. This study was performed in the beginning of the PhD thesis of Mathias Jacquelin. It was presented at the ICPP conference [C16] and the complete study is available in his PhD manuscript [55].

Chapter 8

Data redistribution for parallel computing

In the previous chapters, we have mainly studied the case where the amount of memory was limited, and sometimes not sufficient to store all the data needed for a given computation. In the present chapter, we slightly change the perspective and focus on the data layout for parallel computing platforms: even when the amount of memory is sufficient to store all necessary data, the way the data is distributed across processors impacts the performance of the computation. An improper data layout may be an obstacle to obtaining peak performance. We study here the cost of the data redistribution and the tradeoff between the additional data movement of the redistribution and the cost of computing with improper data distribution.

8.1 Introduction

In parallel computing systems, data locality has a strong impact on application performance. To achieve good locality, a redistribution of the data may be needed between two different phases of the application, or even at the beginning of the execution, if the initial data layout is not suitable for performance. This happens for example when the input data of some computation has been produced by another application, which uses a different data layout, or when this data was acquired by sensors (as depicted later in Figure 8.3).

On the contrary, most scientific applications, and in particular the ones relying on linear algebra kernels, require a regular data distribution to reach their optimal performance. For such 2D data, on of the most common data distribution, which we will use in the following examples, is the block distribution. Consider for instance a square matrix $A = (a_{ij})_{0 \leq i,j < n}$ of size n, and a grid of $p \times p$ processors, the block distribution of the matrix would

allocate block (i, j), containing elements $(a_{k,\ell})$, where $ri \leq k < (r+1)i$ and $rj \leq \ell < (r+1)j$, to the processor of coordinates (i, j).

If we have to perform some computation requiring a block distribution on data which have initially been randomly distributed, we have two options:

(i) Leave the data in place, and run the computation with suboptimal performance,

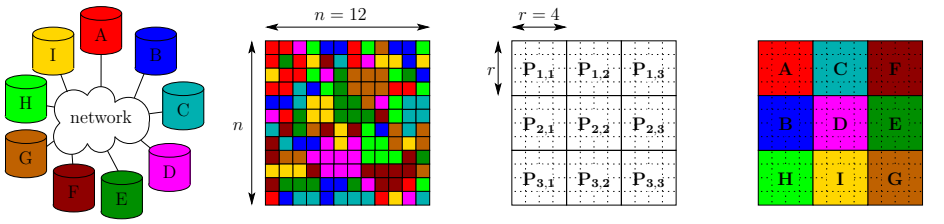
(ii) Redistribute the data in a block distribution before performing the computation.

In the case of a redistribution, we also have to choose which block distribution to choose, that is, which processor will own which block, in order to minimize the amount of data movement during the redistribution.

8.2 Problem modeling

Consider a set of N data items (numbered from 0 to $N-1$) distributed onto P processors (numbered from 0 to $P-1$).

Definition 8.1 (Data distribution)**.** *A* data distribution $\mathcal{D}$ *defines the mapping of the elements onto the processors: for each data item* x*,* $\mathcal{D}(x)$ *is the processor holding it.*



(a) processors holding the data (b) intial data distribution (c) target data partition (d) final data distribution

Figure 8.1: Example of matrix redistribution with $N = 12 \times 12$ data blocks and $P = 3 \times 3$ processors. Each color in the data distributions corresponds to a processor, e.g., all red data items reside on processor A.

Figure 8.1b depicts an initial random data distribution of the N=144 tiles of a square 12×12 matrix on the $P = 9$ processors illustrated on Figure 8.1a. On this example, the goal is to obtain a square block distribution of 3×3 blocks, each block being of size 4×4, as shown on Figure 8.1c. Modern computing platforms are equipped with interconnection switches and routing mechanisms mapping the most usual interconnection graphs onto the physical network with reduced (or even negligible) dilation and contention. In the previous example, the 3×3 2D grid will be virtual, i.e., an overlay topology mapped into the physical topology, forcing the interconnection switch to emulate a 2D-grid. Hence, the layout of the processors in the grid is completely flexible. Figure 8.1d depicts such a possible block

distribution of the data. To account for the $P!$ possible distributions that are suitable, we come up with the following definition of a data partition: a data partition states the general shape of the desired data distribution (Figure 8.1c), but not the precise location of each processor.

Definition 8.2 (Data partition)**.** *A* data partition $\mathcal{P}$ *associates to each data item x an index $\mathcal{P}(x)$ ($0 \leq \mathcal{P}(x) \leq P-1$) so that two data items with the same index reside on the same processor (not necessarily processor $\mathcal{P}(x)$). The* j^{th} component *of the data partition $\mathcal{P}$ is the subset of the data items x such that $\mathcal{P}(x) = j$.*

It is straightforward to see that a data distribution $\mathcal{D}$ defines a single corresponding data partition $\mathcal{P} = \mathcal{D}$. However, a given data partition does not define a unique data distribution. On the contrary, any of the $P!$ permutations of $\{0, \ldots, P-1\}$ can be used to map a data partition onto the processors.

Definition 8.3 (Compatible distribution)**.** *A data distribution $\mathcal{D}$ is* compatible *with a data partition $\mathcal{P}$ if and only if there exists a permutation of processors σ of $\{0, \ldots, P-1\}$ such that for each data item x, $\mathcal{D}(x) = \sigma(\mathcal{P}(x))$.*

One of our goals is to assess the complexity of the problem of finding the best processor mapping for a given initial data distribution and a target data partition. This amounts to determining the processor assignment that minimizes the cost of redistributing the data according to the partition. We use two criteria from the literature to compute the cost of a redistribution.

Total volume. This is the total amount of data that is sent through the network during the redistribution. Formally, given an initial data distribution $\mathcal{D}_{ini}$ and a target distribution $\mathcal{D}_{tar}$, for $0 \leq i, j \leq P-1$, let $q_{i,j}$ be the number of data items that processor i must send to processor j: $q_{i,j}$ is the number of data items x such that $\mathcal{D}_{ini}(x) = i$ and $\mathcal{D}_{tar}(x) = j$. The total communication volume of the redistribution is defined as $\mathit{RedistVol}(\mathcal{D}_{ini} \rightarrow \mathcal{D}_{tar}) = \sum_{i,j} q_{i,j}$.

This metric is suitable in the case where the platform is not dedicated. Minimizing this volume makes it less likely to disrupt the other applications running on the platform, and is expected to decrease network contention, hence redistribution time. Conceptually, this is equivalent to assuming that the network is a bus, globally shared by all computing resources.

Number of parallel steps. We consider here that the platform is dedicated to the application, and several communications can take place in parallel, provided that they involve different processor pairs. This corresponds to the one-port bi-directional model used in [49, 52]. We define a (parallel) step of the redistribution as a set of unit-size communications (one data

item each) such that all senders are different, and all receivers are different (the set of senders of the set of receivers are not necessary disjoint). With this definition, a processor can send and receive a data item at the same time but can not send (respectively receive) a data item to (respectively from) more than one processor during the same communication step. Given an initial data distribution $\mathcal{D}_{ini}$ and a target distribution $\mathcal{D}_{tar}$, we define $RedistSteps(\mathcal{D}_{ini} \rightarrow \mathcal{D}_{tar})$ as the minimal number of parallel steps that are needed to perform the redistribution.

8.3 Redistribution strategies minimizing communications

We first deal with the problem of finding a redistribution strategy which is optimal for one of the two communication criteria exposed above: given a data partition $\mathcal{P}_{tar}$ and an initial data distribution $\mathcal{D}_{ini}$, find one target distribution $\mathcal{D}_{tar}$ among all possible $P!$ compatible target distributions that minimizes the cost of the redistribution, either expressed in total volume or number of parallel steps.

8.3.1 Redistribution minimizing the total volume

We present here an algorithm that optimally solves the problem of the redistribution for the *total volume* metric.

Theorem 8.1. *Given an initial data distribution $\mathcal{D}_{ini}$ and target data partition $\mathcal{P}_{tar}$, Algorithm 21 computes a data distribution $\mathcal{D}_{tar}$ compatible with $\mathcal{P}_{tar}$ such that $RedistVol(\mathcal{D}_{ini} \rightarrow \mathcal{D}_{tar})$ is minimized, and its complexity is $O(NP^2 + P^3)$.*

Proof. Finding a redistribution that minimizes the total volume amounts to finding a one-to-one perfect matching between each component of the target data partition and the processors, so that the total volume of communications is minimized. Algorithm 21 builds the complete bipartite graph where the two sets of vertices represents the P processors and the P components of the target data partition. Each edge (i, j) of this graph is weighted with the amount of data that processor P_i would have to receive if matched to component j of the data partition. Computing the weight of the edges can be done with complexity $O(NP^2)$. The complexity of finding a minimum-weight perfect matching in a bipartite graph with n vertices and m edges is $O(n(m + n \log n))$ (see Corollary 17.4a in [76]). Here n=P and m=P^2, hence the overall complexity of Algorithm 21 is $O(NP^2 + P^3)$. □

Algorithm 21: BestDistribForVolume

Data: Initial data distribution $\mathcal{D}_{ini}$ and target data partition $\mathcal{P}_{tar}$
Result: a data distribution $\mathcal{D}_{tar}$ compatible with the given data partition, such that $RedistVol(\mathcal{D}_{ini} \to \mathcal{D}_{tar})$ is minimized

$A \leftarrow \{0, \ldots, P-1\}$ (set of processors)
$B \leftarrow \{0, \ldots, P-1\}$ (set of data partition indices)
$G \leftarrow$ complete bipartite graph (V, E) where $V = A \cup B$
for *edge* (i, j) *in* E **do**
  $weight(i, j) \leftarrow |\{x \text{ s.t. } \mathcal{P}_{tar}(x) = j \text{ and } \mathcal{D}_{ini}(x) \neq i\}|$
$\mathcal{M} \leftarrow$ minimum-weight perfect matching of G
for $(i, j) \in \mathcal{M}$ **do**
  for x s.t. $\mathcal{P}_{tar}(x) = j$ **do** $\mathcal{D}_{tar}(x) \leftarrow i$
return $\mathcal{D}_{tar}$

8.3.2 Redistribution minimizing the number of parallel steps

The second metric is the number of parallel communications steps in the bidirectional one-port model. Note that this objective is quite different from the total communication volume: consider for instance a processor which has to send and/or receive much more data than the others; all the communications involving this processor will have to be performed sequentially, creating a bottleneck.

Theorem 8.2. *Given an initial data distribution $\mathcal{D}_{ini}$ and target data partition $\mathcal{P}_{tar}$, Algorithm 22 computes a data distribution $\mathcal{D}_{tar}$ compatible with $\mathcal{P}_{tar}$ such that $RedistSteps(\mathcal{D}_{ini} \to \mathcal{D}_{tar})$ is minimized, and its complexity is $O(NP^2 + P^{\frac{9}{2}})$.*

Proof. First, given an initial data distribution $\mathcal{D}_{ini}$ and a target distribution $\mathcal{D}_{tar}$, we define s_i (respectively r_i) as the total number of data items that processor i must send (resp. receive) during the redistribution. We have

$$s_i = \sum_{j \neq i} q_{i,j} \quad \text{and} \quad r_i = \sum_{j \neq i} q_{j,i}.$$

Thanks to König's theorem (see Theorem 20.1 in [76]) stating that the edge-coloring number of a bipartite multigraph is equal to its maximum degree, we can compute the number of parallel steps as follows (see also [31]):

$$RedistSteps(\mathcal{D}_{ini} \to \mathcal{D}_{tar}) = \max_{0 \leq i \leq P-1} max(s_i, r_i).$$

Algorithm 22 builds the complete bipartite graph G where the two sets of vertices represent the P processors and the P components of $\mathcal{P}_{tar}$. Each edge (i, j) of the complete bipartite graph is weighted with the maximum

between the amount $r_{i,j}$ of data that processor i would have to receive if matched to component j of the data partition, and the amount of data that it would have to send in the same scenario. A one-to-one matching between the two sets of vertices whose maximal edge weight is minimal represents an redistribution strategy that minimizes the number of parallel steps. We denote by $\mathcal{M}_{opt}$ such a matching and m_{opt} its maximal edge weight. Since there are P processors and P components in $\mathcal{P}_{tar}$, the one-to-one matching $\mathcal{M}_{opt}$ is a matching of size P.

Algorithm 22 prunes an edge with maximum weight from G until it is not possible to find a matching of size P, and it returns the last matching of size P. We denote by $\mathcal{M}_{ret}$ this matching and m_{ret} its maximum edge weight. Using a proof by contradiction, we first assume that $m_{ret} > m_{opt}$. Then matching $\mathcal{M}_{opt}$ only contains edges with weight strictly smaller than m_{ret}. Since Algorithm 22 prunes edges starting from the heaviest ones, these edges are still in G when Algorithm 22 returns $\mathcal{M}_{ret}$. Thus we can remove the edges with maximal weight m_{ret} in $\mathcal{M}_{ret}$ and still have a matching of size P. This contradicts the stopping condition of Algorithm 22. Thus $m_{ret} = m_{opt}$ and the matching returned by Algorithm 22 is an optimal solution.

Again, computing edge weights can be done with complexity $O(NP^2 + P^2)$. Algorithm 22 uses the Hopcroft–Karp algorithm [51] to find a matching of maximum cardinality from a bipartite graph $G = (V, E)$ in time $O(|E|\sqrt{|V|})$. There are no more than P^2 iterations in the while loop, and Algorithm 22 has a worst-case complexity of $O(NP^2 + P^{\frac{9}{2}})$. □

8.4 Coupling redistribution and computation

We now move to the problem that arises when we have to perform some computation on data that are not well distributed. Should we first redistribute the data, using one of the above algorithms, and pay an extra cost for this data movement, or should we stick to the current distribution and bear with the slow down of our numerical kernel ?

8.4.1 Problem complexity

Of course, the answer to this question highly depends on the type of computation, whether its performance depends a lot on data locality or not. Formally, given an initial data distribution $\mathcal{D}_{ini}$, we aim at executing some computational kernel whose cost $T_{comp}(\mathcal{P}_{tar})$ depends upon the data partition $\mathcal{P}_{tar}$ that will be selected. Note that this computational kernel has the same execution cost for any distribution $\mathcal{D}_{tar}$ compatible with $\mathcal{P}_{tar}$, because of the symmetry of the target platform. However, the redistribution cost from $\mathcal{D}_{ini}$ to $\mathcal{D}_{tar}$ depends upon $\mathcal{D}_{tar}$. We define the total cost as the sum of the time of the redistribution and of the computation. Letting τ_{comm} denote the

Algorithm 22: BestDistribForSteps

Data: Initial data distribution $\mathcal{D}_{ini}$ and target data partition $\mathcal{P}_{tar}$
Result: A data distribution $\mathcal{D}_{tar}$ compatible with the given data partition so that $RedistSteps(\mathcal{D}_{ini} \to \mathcal{D}_{tar})$ is minimized

$A \leftarrow \{0, \ldots, P-1\}$ (set of processors)
$B \leftarrow \{0, \ldots, P-1\}$ (set of data partition indices)
$G \leftarrow$ complete bipartite graph (V, E) where $V = A \cup B$
for *edge* (i, j) *in* E **do**
  $r_{i,j} \leftarrow |\{x \text{ s.t. } \mathcal{P}_{tar}(x) = j \text{ and } \mathcal{D}_{ini}(x) \neq i\}|$
  $s_{i,j} \leftarrow |\{x \text{ s.t. } \mathcal{P}_{tar}(x) \neq j \text{ and } \mathcal{D}_{ini}(x) = i\}|$
  $weight(i, j) \leftarrow \max(r_{i,j}, s_{i,j})$
$\mathcal{M} \leftarrow$ maximum cardinality matching of G (using the Hopcroft–Karp Algorithm)
while $|\mathcal{M}| = P$ **do**
  $\mathcal{M}_{save} \leftarrow \mathcal{M}$
  Suppress all edges of G with maximum weight
  $\mathcal{M} \leftarrow$ maximum cardinality matching of G (using the Hopcroft–Karp Algorithm)
return $\mathcal{M}_{save}$

time to perform a communication, the time to execute the redistribution is either $RedistVol(\mathcal{D}_{ini} \to \mathcal{D}_{tar}) \times \tau_{comm}$ or $RedistSteps(\mathcal{D}_{ini} \to \mathcal{D}_{tar}) \times \tau_{comm}$, depending upon the communication model.

Note that computing $T_{comp}(\mathcal{P}_{tar})$ for any target data partition $\mathcal{P}_{tar}$ is realistic only for very simple computational kernels. The following theorem, proved in [J16], consider such a simple kernel, namely the 1D-stencil, and shows that even for such a simple kernel, finding the distribution that minimizes the total time is NP-complete, for both communication criteria.

Theorem 8.3. *Consider a 1D 2-point stencil kernel. Given a number of processors P, elementary communication and computation times τ_{comm} and τ_{calc}, a number of steps K, an initial data distribution $\mathcal{D}_{ini}$, finding a partition $\mathcal{P}_{tar}$ and a distribution $\mathcal{D}_{tar}$ compatible with $\mathcal{P}_{tar}$, such that either $T_{total}(\mathcal{D}_{ini}, \mathcal{D}_{tar}) = RedistVol(\mathcal{D}_{ini} \to \mathcal{D}_{tar}) \times \tau_{comm} + T_{comp}(\mathcal{P}_{tar})$ or $T_{total}(\mathcal{D}_{ini}, \mathcal{D}_{tar}) = RedistSteps(\mathcal{D}_{ini} \to \mathcal{D}_{tar}) \times \tau_{comm} + T_{comp}(\mathcal{P}_{tar})$ is minimal is NP-complete.*

The complex proof of this result, as well as the complete definition of the stencil kernel, is available in [J16].

8.4.2 Experiments

The algorithms designed in Section 8.3 find the optimal target distribution according to different models for the redistribution time. These algorithms

may be sub-optimal for minimizing the total processing time when it takes the processing of an arbitrary application into account. We proved in the previous section that there is no polynomial-time optimal algorithm to minimize this total processing time (unless P=NP) even for a simple application like the 1D-stencil algorithm, which motivates the use of low-complexity sub-optimal heuristics. The following experiments show that the redistribution algorithms introduced above are good enough to provide performance improvements for real-life applications.

The experiments have been conducted on a multicore cluster using two application, a simple 1D-stencil kernel and a more compute-intensive dense linear algebra routine, namely the QR factorization. Both applications have been implemented on top of the PaRSEC runtime [18, 17].

The PaRSEC runtime deals with computational threads and MPI communications. It allows the user to define the initial distribution of the data onto the platform, as well as the target distribution for the computations. Data items are first moved from their initial data distribution to the target data distribution. Then computations take place, and finally data items are moved back to their initial position. It is important to stress that the PaRSEC runtime will overlap the initial communications due to the redistribution with the processing of the computational kernel (either 1D-stencil or QR), so that the total execution time does not strictly obey the simplified model of the previous sections. However, choosing a good data partition (leading to an efficient implementation of the computational kernel), and an efficient compatible data distribution (leading to fewer communications during the redistribution) is still important to achieve high performance.

Experiments have been conducted on *Dancer*, a small cluster hosted at the Innovative Computing Laboratory (University of Tennessee, Knoxville). This cluster has 16 multi-core nodes, each equipped with 8 cores, and an InfiniBand 10G interconnection network (see details in [J16]).

We have tested the following four strategies. In the *owner computes* strategy, the data items are not moved and the computational kernel is applied on the initial distribution. In the other strategies, we redistribute the data items towards three target distributions, each compatible with the canonical data partition $\mathcal{P}_{can}$ (specific to the target application): (i) the distribution $\mathcal{D}_{can} = \mathcal{P}_{can}$ with the original (arbitrary) labeling of the processors; (ii) the distribution that minimizes the volume of communications $\mathcal{D}_{vol}$ (computed by Algorithm 21); and (iii) the distribution that minimizes the number of redistribution steps $\mathcal{D}_{steps}$ (computed by Algorithm 22).

Results on 1D-stencil

A simple stencil application has been implemented on top of PaRSEC. In our experiments, each data item consists in a block of 1.6×10^6 double-precision floats. These items are distributed on the $P = 16$ processors according to

a random balanced distribution. The canonical data partition consists in assigning data item i to component $\lfloor iP/N \rfloor$.

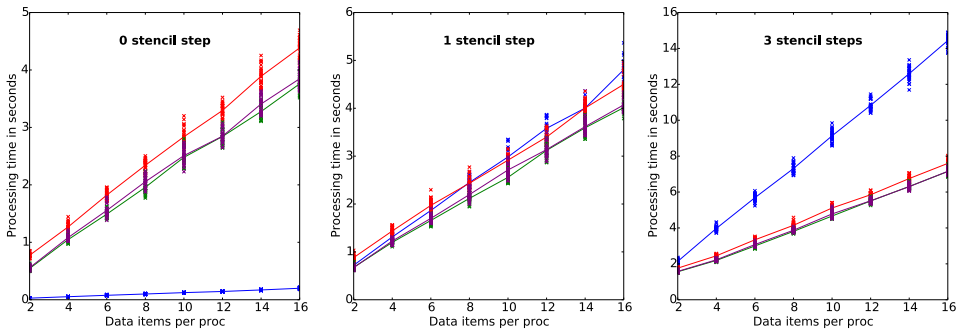


Figure 8.2: Processing time of the stencil application for small number of stencil steps, in the case where $\tau_{comm}/\tau_{calc} = 1$.

Some results for the 1D-stencil kernel are depicted on Figure 8.2 for the case where communications and computations are equally costly (the average time to send one data item is equal to the average time of the update kernel). We notice that in the case no iteration is performed, the *owner-compute* strategy has a processing time close to zero which corresponds to the overhead of the ParSEC runtime. However, as soon as more than one stencil steps are performed, it is outperformed by the strategies relying on data redistributions. Without any stencil iteration, we notice that both $\mathcal{D}_{vol}$ and $\mathcal{D}_{steps}$ provide a 20% improvement over the $\mathcal{D}_{can}$. This improvement, although still present, is less visible when the number of iteration increases.

Results on the QR factorization

The QR factorization is a widely used linear algebra algorithm for solving linear systems and linear least squares problems. To optimize performance, the matrix is usually stored in tiled form, and we use these tiles as our data items. Contrarily to the 1D-stencil, the QR factorization is a complex workflow, and it is thus not easy to predict its processing time on a given data partition. However, some distributions are known to be well-suited. A widely-used data partition consists of mapping the tiles onto the processors following a 2D block cyclic partition [22]. The P processors (numbered from 0 to $P-1$) are arranged in a $p \times q$ grid where $p \times q = P$. Matrix tile $A_{i,j}$ is then mapped onto processor $(i \mod p) \times p + (j \mod q)$. We choose this data partition to be our target partition $\mathcal{P}_{tar}$, which defines the canonical distribution $\mathcal{D}_{can}$ as well as the $\mathcal{D}_{vol}$ and $\mathcal{D}_{steps}$ as detailed above.

A highly optimized version of QR implemented on top of PaRSEC is available in DPLASMA [15]. We have modified this implementation to deal with different data distributions. We use two initial data distributions:

- *SkewedSet*: Matrix tiles are first distributed following an arbitrary 2D block cyclic distribution (used as reference) and, then, half of the tiles are randomly moved onto another processor, so that the workload is likely to be unbalanced. Our optimal redistribution is likely to find the 2D block cyclic distribution used as reference and move only half of the tiles, while the redistribution towards the arbitrary distribution $\mathcal{D}_{can}$ can potentially move all of them.

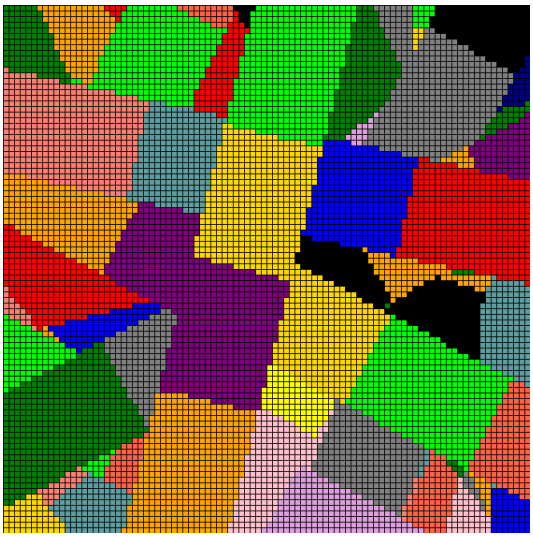


Figure 8.3: Example of data distribution after its acquisition by satellites. Each color corresponds to a processor storing the data.

- *ChunkSet*: This distribution set comes from an Earth Science application [81]. Astronomy telescopes collect data over days of observations and process them into a 2D or 3D coordinate system, which is usually best modeled as a matrix. Then, linear algebra routines such as QR factorization must be applied to the resulting matrix. The collected data are stored on a set of processors in a round-robin manner, ensuring spacial locality of data that are sampled at close time-steps. If a certain region of Earth is observed twice, the latest data overwrites the previous one. We generated a set of initial distributions fitting the telescope behavior. Figure 8.3 depicts the data distribution of a matrix in chunkset where matrix tiles of the same color are initially stored on the same processor.

The results of these experiments are summarized in Figure 8.4, which shows the average improvement in total processing time of the redistribution strategies over the *owner-compute* policy. Complete results are detailed in [J16]. We conclude that redistributing towards a suitable data partition for the QR factorization leads to significant improvement, compared to not

n	canonical	$\mathcal{D}_{vol}$	$\mathcal{D}_{steps}$
16	41.9%	39.5%	43.4%
34	64.1%	67.7%	66.4%
52	65.8%	70.5%	71.2%
70	70.8%	72.7%	71.4%
88	70.8%	72.6%	72.4%

(a) Results for *SkewedSet*.

n	canonical	$\mathcal{D}_{vol}$	$\mathcal{D}_{steps}$
16	27.0%	28.1%	28.1%
34	20.6%	25.5%	22.1%
52	13.6%	25.8%	26.2%
70	12.7%	14.5%	4.8%
88	12.0%	15.7%	13.4%

(b) Results for *ChunkSet*.

Figure 8.4: Improvements in the processing time of the QR factorization compared to the *owner-compute* strategy (with n^2 matrix tiles).

redistributing the data as with the *owner computes* strategy. While any redistribution to a suitable partition (such as $\mathcal{D}_{can}$) usually already reduces the completion times, better redistributions such as the $\mathcal{D}_{vol}$ or $\mathcal{D}_{steps}$ sometimes allow to improve performance even further, and largely reduce the volume of communication, especially for the *SkewedSet* dataset.

8.5 Conclusion of the chapter

We have studied in this chapter the problem of redistributing the data before a computational kernel. We have shown how to optimally redistribute the data for a given target data partition, for two cost metrics, the total volume of communications and the number of parallel redistribution steps. We have also proved that finding the optimal data partition and redistribution scheme to minimize the completion time of a 1D-stencil is NP-complete. Altogether, these results lay the theoretical foundations of the data partition problem on modern computers.

Admittedly, the platform model used in this study will only be a coarse approximation of actual parallel performance, because state-of-the-art runtimes use intensive prefetching and overlap communications with computations. Therefore, experimental validation of the algorithms on a multicore cluster have been presented for a 1D-stencil kernel and a dense linear algebra routine. The proposed redistribution strategies lead to better performance in all cases, and the improvement is significant when the initial data distribution is not well-suited for the computational kernel.

Note on the material presented in this chapter. This study was performed during the PhD thesis of Julien Herrmann, co-advised with Yves Robert, in collaboration with Thomas Hérault and George Bosilca from the Innovative Computing Laboratory (University of Tennessee, Knoxville). A first version of this work was presented at the ISPDC conference [C25] and the complete study was published in PARCO [J16].

Chapter 9

Dynamic scheduling for matrix computations

9.1 Introduction

As in the previous chapter, we focus here on data movement for distributed platforms. We consider simple data-parallel applications consisting of many independant tasks with input data, and we consider their processing on a distributed computing platform. This corresponds to the classical master-worker scenario when all initial input data reside on some centralized data storage [74, 13]. It is also the basis of popular frameworks such as MapReduce [28], which allows users without particular knowledge in parallel computing to harness the power of large parallel machines. In MapReduce, a large computation is broken into small tasks that run in parallel on multiple machines, and scales easily to very large clusters of inexpensive commodity computers. MapReduce is a very successful example of dynamic schedulers, as one of its crucial feature is its inherent capability of handling hardware failures and processing speed heterogeneity, thus hiding this complexity to the programmer, by relying on on-demand allocations and the on-line detection of nodes that perform poorly (in order to re-assign tasks that slow down the process).

While the scheduling community has proposed a large number of static schedulers, i.e., algorithms that take allocation decisions prior to the computation based on predicted task and data movement durations, dynamic schedulers are usually preferred for MapReduce-like frameworks as they do not rely on such accurate estimations. However, their performance are usually not guaranteed. In this chapter, we study the master-worker scheduling problem for matrix computations, for which data dependencies take the form of a two-dimensional or a three-dimensional grid. We study the performance of dynamic schedulers for nodes with heterogeneous processing capabilities.

Our objective is to take advantage of data locality in order to limit the amount of input data movement.

9.2 Problem statement

We consider here a master-worker scheduling problem with complex but structured data dependencies. While the classical task distribution problem focuses on independant tasks that depends only on their own input data, MapReduce has been used for more complex operations, such as linear algebra computations [77, 23, 21]. We first focus here two-dimensional dependencies, which are best exemplify by the outer product of two vectors. Given two vectors a and b of size n, our objective is to compute all products $a_i b_j$ for $1 \leq i, j \leq n$. For performance issues, each element of a and b usually represents in fact a block of elements, so that the elementary product is a block outer-product. We call task $T_{i,j}$ the elementary product $a_i b_j$ for $1 \leq i, j \leq n$.

We target heterogeneous platforms consisting of p processors $P_1, \ldots, P_p$, where the speed of processor P_i, i.e., the number of elementary products that P_k can do in one time unit, is given by s_k. We will also denote by rs_k is relative speed $rs_k = \frac{s_k}{\sum_i s_i}$. Note that the randomized strategies proposed below are agnostic to processor speeds, but they are demand driven, so that a processor with a twice larger speed will request work twice faster.

In the following, we assume that a master processor P_0 originally owns the a and b inputs and coordinates the work distribution: it is aware of which a and b blocks are replicated on the computing nodes and decides which new blocks are sent, as well as which tasks are assigned to the nodes. After completion of their allocated tasks, computing nodes simply report to the master processor, requesting for new tasks. We also assume that data movement and computation can be overlapped. This can be achieved with dynamic strategies by uploading a few blocks in advance at the beginning of the computations and then to request work as soon as the number of blocks to be processed becomes smaller than a given threshold.

This kernel do not induce dependency among its tasks, however the initial input data must be replicated on the processors so that they can all take part in the computation. Our objective is to minimize the overall amount of data movement, that is, the total number of elements of a and b sent by the master node, under the constraint that a perfect load-balancing should be achieved among resources allocated to the outer-product computation. For the outer-product case, we assume that it is not necessary to gather all output results on a single node, and thus we let aside output data movement. We will revoke this assumption when considering the matrix-matrix product in Section 9.8.

9.3 Lower bound and static solutions

In a very optimistic setting, each processor is dedicated to computing a "square" area of $M = ab^t$, whose area is proportional to its relative speed, so that all processors finish their work at the same instant. In this situation, the amount of data sent to P_k is proportional to the half perimeter of this square of area $n^2 r s_k$. This gives the following lower bound on the total amount of data movement:

$$LB = 2n \sum_k \sqrt{r s_k} = 2n \sum_k \sqrt{\frac{s_k}{\sum_i s_i}},$$

Note that this lower bound is not expected to be achievable (consider for instance the case of 2 heterogeneous processors). Indeed, the best known static algorithm (based on a complete knowledge of all relative speeds) has an approximation ratio of 7/4 [11]. As outlined in the introduction, such an allocation mechanism is not practical in our context, since our aim is to rely on more dynamic runtime strategies, but can be used as a comparison basis.

9.4 Dynamic data distribution strategies

One of the simplest strategy to allocate computational tasks to processors is to distribute tasks at random: whenever a processor is ready, some task $T_{i,j}$ is chosen uniformly at random among all available tasks and is allocated to the processor. The data necessary to this task that is not yet on the processor, that is one or two of the a_i and b_j are then sent by the master. We denote this strategy by RandomOuter. Another simple option is to allocate tasks in lexicographical order of indices (i, j) rather than randomly. This strategy will be denoted as SortedOuter.

Algorithm 23: DynamicOuter

while *there are unprocessed tasks* **do**
- Wait for a processor P_k to finish its tasks
- $I \leftarrow \{i$ such that P_k owns $a_i\}$
- $J \leftarrow \{j$ such that P_k owns $b_j\}$
- Choose $i \notin I$ and $j \notin J$ uniformly at random
- Send a_i and b_j to P_k
- Allocate all tasks of $\{T_{i,j}\} \cup \{T_{i,j'}, j' \in J\} \cup \{T_{i',j}, i' \in I\}$ that are not yet processed to P_k and mark them processed

Both previous algorithms are expected to induce a large amount of data movement because of data replication. Indeed, in these algorithms, the

data already present on a processor P_k requesting for some work is not taken into account when allocating a new task. To improve data re-use, we propose a data-aware strategy, denoted DynamicOuter, in Algorithm 23: when a processor P_k receives a new pair of blocks (a_i, b_j), all possible products $a_i b_{j'}^t$ and $a_{i'} b_j^t$ are also allocated to P_k, for all data blocks $a_{i'}$ and $b_{j'}$ that have already been transmitted to P_k in previous steps. Note that the DynamicOuter scheduler is not computationally expensive: it is sufficient to maintain a set of unknown a and b data (of size $O(n)$) for each processor, and to randomly pick an element of this set when allocating new blocks to a processor P_k.

We have compared the performance of the three previous schedulers through simulations on Figure 9.1, where the amount of data movement is normalized by the previous lower bound. Processor speeds are chosen uniformly in the interval $[10, 100]$, which means a large degree of heterogeneity. Each point in this figure and the following ones is the average over 10 or more simulations. The standard deviation is always very small, typically smaller than 0.1 for any point, and never impacts the ranking of the strategies.

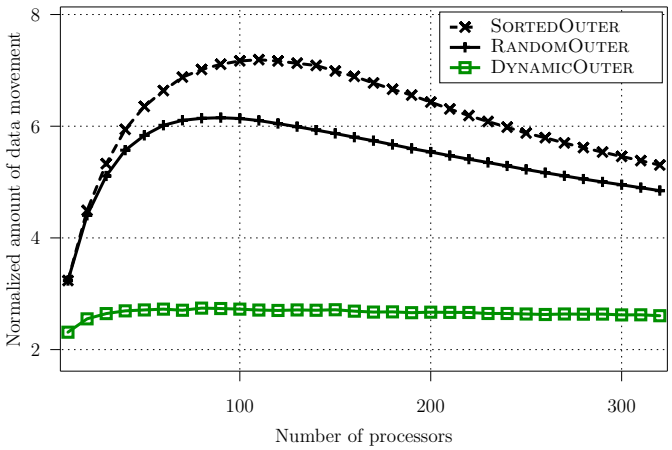


Figure 9.1: Comparison of random and data-aware dynamic strategies, for vectors of size $n = 100$.

Our DynamicOuter allocation scheme suffers some limitation: when the number of remaining blocks to compute is small, the proposed strategy is inefficient as it may send a large number of a and b blocks to a processor P_k before it is able to process one of the last few available tasks. Thus, we propose an improved version DynamicOuter2Phases in Algorithm 24: when the number of remaining tasks becomes smaller than a given threshold, we switch to the basic randomized strategy: any available task $T_{i,j}$ is allocated to a requesting processor, without taking data locality into account. The corresponding data a_i and b_j are then sent to P_k if needed.

Algorithm 24: DynamicOuter2Phases

while *the number of processors is larger than the threshold* **do**
- Wait for a processor P_k to finish its tasks
- $I \leftarrow \{i$ such that P_k owns $a_i\}$
- $J \leftarrow \{j$ such that P_k owns $b_j\}$
- Choose $i \notin I$ and $j \notin J$ uniformly at random
- Send a_i and b_j to P_k
- Allocate all tasks of $\{T_{i,j}\} \cup \{T_{i,j'}, j' \in J\} \cup \{T_{i',j}, i' \in I\}$ that are not yet processed to P_k and mark them processed

while *there are unprocessed tasks* **do**
- Wait for a processor P_k to finish its tasks
- Choose randomly an unprocessed task $T_{i,j}$
- **if** P_k *does not hold* a_i **then** send a_i to P_k
- **if** P_k *does not hold* b_j **then** send b_j to P_k
- Allocate $T_{i,j}$ to P_k

As illustrated on Figure 9.2, for a well chosen number of tasks processed in the second phase, this new strategy allows to further reduce the amount of data movement. However, this requires to accurately set the threshold, depending on the size of the matrix and the relative speed of the processors. If too many tasks are processed in the second phase, the performance is close to the one of RandomOuter. On the contrary, if too few tasks are processed in the second phase, the behavior becomes close to DynamicOuter. The optimal threshold corresponds here to a few percent of tasks being processed in the second phase. In the following, we present an analysis of the DynamicOuter2Phases strategy that both allows to predict its performance and to optimally set the threshold, so as to minimize the amount of data movement.

9.5 Analysis and optimization

We present here an analytical model for DynamicOuter2Phases, which allows us to tune the parameters of this dynamic strategies depending on input parameters to optimize its performance. We only present the sketch of the analysis, which is completely described in [C26].

In what follows, we assume that n, the size of vectors a and b, is large and we consider a continuous dynamic process whose behavior is expected to be close to the one of DynamicOuter2Phases. We concentrate on processor P_k. At each step, DynamicOuter2Phases chooses to send one data block of a and one data block of b, so that P_k knows the same number y of data blocks of a and b. We denote by $x = y/n$ the ratio of elements of a and

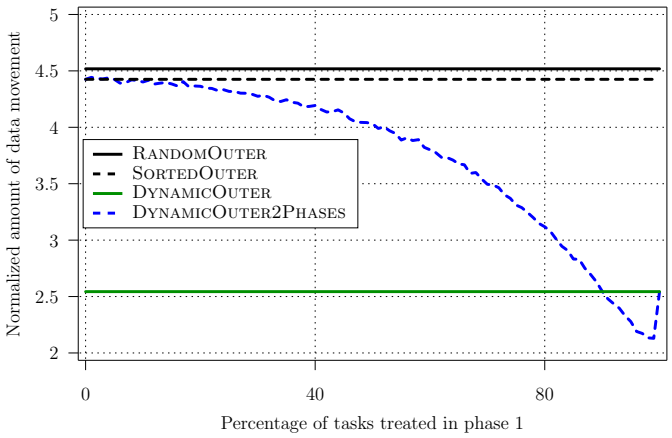


Figure 9.2: Amount of data movement of DynamicOuter2Phases and comparison to the other schedulers for different thresholds (for a given distribution of computing speeds with 20 processors and $n = 100$).

b that are available on P_k at a given time step of the process and by $t_k(x)$ the time step where processor P_k owns such a fraction x . We concentrate on a basic step of DynamicOuter2Phases during which the fraction of data blocks of both a and b known by P_k goes from x to $x + \delta x$. In fact, since DynamicOuter2Phases is a discrete process and the ratio known by P_k goes from $x = y/n$ to $x + 1/n = (y + 1)/n$. Under the assumption that n is large, we assume that we can approximate the randomized discrete process by the continuous process described by the corresponding Ordinary Differential Equation on expected values. We do not provide a complete proof of convergence, which is probably out of reach, but rather rely on the simulation results provided below.

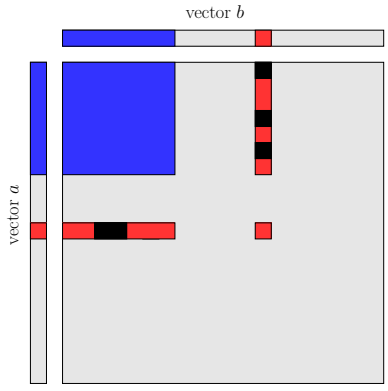


Figure 9.3: Illustration of the dynamics of DynamicOuter.

Let us remark that during the execution of DynamicOuter2Phases, tasks $T_{i,j}$ are greedily computed as soon as a processor knows the corre-

sponding data blocks of a_i and b_j. Therefore, at time $t_k(x)$, all tasks $T_{i,j}$ such that P_k knows data blocks a_i and b_j have been processed and there are x^2n^2 such tasks. Note also that those tasks may have been processed either by P_k or by another processor P_ℓ since processors compete to process tasks. Indeed, since data blocks of a and b are possibly replicated on several processors, then both P_k and P_ℓ may know at some point both a_i and b_j.

Figure 9.3 depicts the computational domain during the first phase of DynamicOuter2Phases from the point of view of a given processor P_k (rows and columns have been reordered for the sake of clarity). The top-left square (in blue) corresponds to value of a and b that are known by P_k, and all corresponding tasks have already been processed (either by P_k or by another processor). The remaining "L"-shaped area (in grey) corresponds to tasks $T_{i,j}$ such that P_k does not hold either the corresponding value of a, or the corresponding value of b, or both. When receiving a new value of a and b (in red), P_k is able to process all the tasks (in red) from the two corresponding row and column. Some elements from this row and this column may be already processed (in black).

We consider the fraction $g_k(x)$ of tasks $T_{i,j}$ in the previously described "L"-shaped area that have not been computed yet. We assume that the distribution of unprocessed tasks in this area is uniform, and we claim that this assumption is valid for a reasonably large number of processors. Our simulations below show that this leads to a very good accuracy. Based on the estimation of the number of tasks being processed by P_k and by other processors during time interval $[t_k(x), t_k(x) + \delta x]$, we are able to write a differential equation on this fraction. Solving this differential equation leads to $g_k(x) = (1 - x^2)^{\alpha_k}$, where $\alpha_k = \frac{\sum_{i \neq k} s_i}{s_k}$.

This allows us to estimate the number of tasks that have already been processed among the new tasks that a processors P_k is able to process when receiving new a and b elements (the black squares in the red stripes on Figure 9.3). Then, we are able to estimate the time needed by P_k to complete the red tasks, which leads to an expression of $t_k(x)$:

$$t_k(x) \sum_i s_i = n^2(1 - (1 - x^2)^{\alpha_k + 1}).$$

Above equations well describe the dynamics of DynamicOuter2Phases as long as it is possible to find blocks of a and b that enable to compute enough unprocessed tasks. We now have to decide when it is beneficial to switch to the other strategy which randomly picks an unprocessed task. In order to decide when to switch from one strategy to the other, we introduce an additional parameter β.

As presented above, a lower bound on the amount of data received by P_k (if perfect load balancing is achieved) is given by $LB = 2n \sum_k \sqrt{rs_k}$. We will switch from the DynamicOuter to the RandomOuter strategy when

the fraction of tasks $x_k^2 n^2$ for which P_k owns the input data is approximately β times what it would have computed optimally, that is, when x_k^2 is close to $\beta \frac{s_k}{\sum_i s_i} = \beta r s_k$, for a value of β that is to be determined. For the sake of the analysis, it is important that we globally define the instant at which we switch to the random strategy, and that it does not depend on the processor P_k. In order to achieve this, we look for x_k^2 as

$$x_k^2 = (\beta r s_k - \alpha r s_k^2)$$

and we search α such that $t_k(x_k)$ does not depend on k at first order in $1/rs_k$, where rs_k is of order $1/p$ and p is the number of processors.

We first prove that if $\alpha = \beta^2/2$, then

$$t_k(x_k) \sum_i s_i = n^2(1 - e^{-\beta}(1 + o(rs_k))).$$

One remarkable characteristics of the above result is that it does not depend (at least up to order 2) on k anymore. Otherwise stated, at time $T = \frac{n^2}{\sum_i s_i}(1 - e^{\beta})$, each processor P_k has received $\sqrt{\beta r s_k}(1 - \beta r s_k/4)n$ data, to be compared with the lower bound on the amount of data received by processor P_k: $\sqrt{rs_k}n$.

Using these results, it is possible to compute the overall amount of data movement induced by the both the first and the second phase of the algorithm, which is equal to:

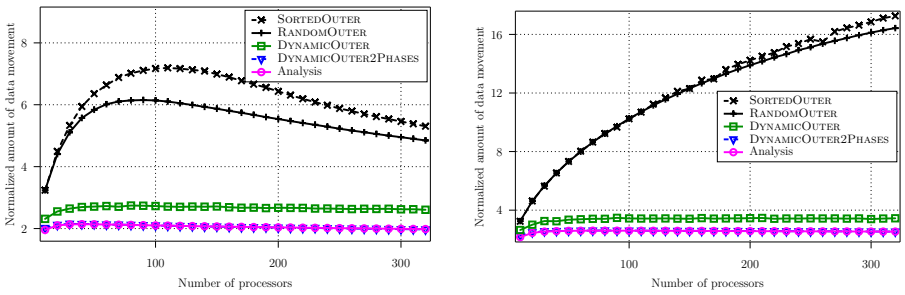
$$\left(\sqrt{\beta} + \frac{\beta^{3/2} \sum_k rs_k^{3/2}}{4 \sum_k rs_k^{1/2}} + e^{-\beta} n^2 \frac{1 - \sqrt{\beta} \sum_k rs_k^{3/2}}{\sum_k rs_k^{1/2}}\right) \times LB.$$

Therefore, in order to minimize the overall amount of data movement, we numerically determine the value of β that minimizes the above expression and then switch between Phases 1 and 2 when $e^{-\beta}n^2$ tasks remain to be processed.

9.6 Evaluation through simulations

We have performed simulations to study the accuracy of the previous theoretical analysis, that is a priori valid only for large values of p and n, and to show how it is helpful to compute the threshold for DynamicOuter2Phases. Details on these simulations are available in [C26].

Figure 9.4 presents the results for two different vector sizes n (the corresponding number of tasks is n^2). In both figures, the analysis is extremely close to the performance of DynamicOuter2Phases (which makes them indistinguishable on the figures) and proves that our analysis succeed to accurately model our dynamic strategy, even for relatively small values of p and n. Moreover, we can see in Figure 9.4b that it is even more crucial to



(a) $n = 100$ (b) $n = 1000$

Figure 9.4: Amount of data movement of all outer-product strategies for two vector sizes.

use a data-aware dynamic scheduler when n is large, as the ratio between the amount of data movement of simple random strategies (RandomOuter and SortedOuter) and the one of dynamic data-aware schedulers (such as DynamicOuter2Phases) can be very large.

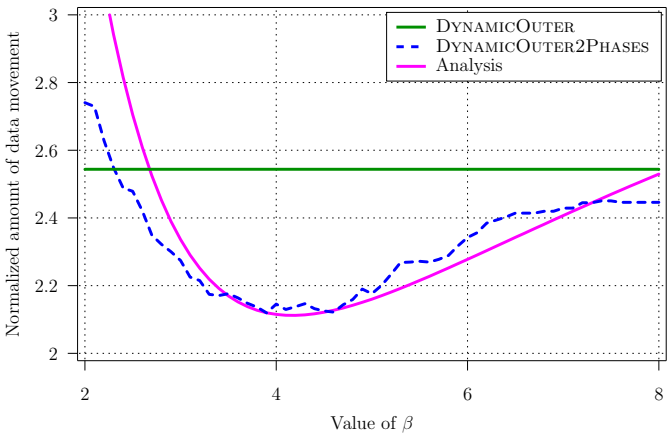


Figure 9.5: Amount of data movement of DynamicOuter2Phases and its analysis for varying value of the β parameter which defines the threshold.

Our second objective is to show that the theoretical analysis that we propose can be used in order to accurately compute the threshold of DynamicOuter2Phases, i.e., that the β parameter computed earlier is close to the best one. To do this, we compare the amount of data movement DynamicOuter2Phases for various values of the β parameter. Figure 9.5 shows the results for 20 processors and $n = 100$. This is done for a single and arbitrary distribution of computing speeds, as it would make no sense to compute average values for different distributions since they would lead to different optimal values of β. This explains the irregular performance graph for DynamicOuter2Phases. This figure shows that in the domain of in-

terest, i.e., for $3 \leq \beta \leq 6$, the analysis correctly fits to the simulations, and that the value of β that minimizes the analysis (here $\beta = 4.17$) lies in the interval of β values that minimize the amount of data movement of DynamicOuter2Phases. To compare to Figure 9.2, this corresponds to 98.5% of the tasks being processed in the first phase.

Impact of speed heterogeneity. In [C26], we report simulations performed to test whether the previous results and the relative performance of all strategies vary with the degree of heterogeneity: we considered different heterogeneity distributions as well as dynamically changing processor speeds. In all cases, the analysis of DynamicOuter2Phases and the relative performance of the algorithms in almost unchanged.

9.7 Runtime estimation of β

In order to estimate the β parameter in the DynamicOuter2Phases strategy, it seems necessary to know the processing speed, as β depends on $\sum_k \sqrt{s_k/\sum_i s_i}$. However, we have noticed a very small deviation of β with the speeds. For example, in Figure 9.5, the value of β computed when assuming homogeneous speeds (4.1705) is very close to the one computed for heterogeneous speeds (4.1679).

For a large range of n and p values (namely, p in $[10, 1000]$ and $n \in [\max(10, \sqrt{p}), 1000]$), for processor speeds in $[10, 100]$, the optimal value for β goes from 1 to 6.2. However, for fixed values of n and p, the deviations among the β values obtained for different speed distributions is at most 0.045 (with 100 tries). Our idea is to approximate β with β_{hom} computed using a homogeneous platform with the same number of processors and with the same matrix size. The relative difference between β_{hom} and the average β of the previous set is always smaller than 5%. Moreover, the error on the amount of data movement predicted by the analysis when using homogeneous speeds instead of the actual ones is at most 0.1%.

This proves that even if our previous analysis ends up with a formula for β that depends on the computing speeds, in practice, only the knowledge of the matrix size and of the number of processors are actually needed to define the threshold β. Our dynamic scheduler DynamicOuter2Phases is thus totally agnostic to processor speeds.

9.8 Extension to matrix-matrix multiplication.

We have extended the previous study to three-dimensional computational domains, which correspond for example to the computation of the product of two matrices $C = AB$. The basic computation step is a task $T_{i,j,k}$ corresponding to the update $C_{i,j} \leftarrow C_{i,j} + A_{i,k}B_{k,j}$. To perform such a task, a

processor has to receive the input data from A and B and to send the resulting contribution of C back to the master at the end of the computation. As previously, this elementary product is usually performed using blocked data for better performance: each input/output data is a square matrix of size l^2.

As previously, our objective is to minimize the amount of data movement by taking advantage of the elements of A, B and C that have already been allocated to a processor P_u when assigning a new task to P_u. Note that while A and B elements must be distributed by the master before the computation, C elements are sent back to the master at the end of the computation. Then, the master computes the final results by adding the different contributions. This computational load is much smaller than computing the products $T_{i,j,k}$ and is neglected.

We have proposed an adaptation of the DynamicOuter strategy into DynamicMatrix as follows. We ensure that at each step, each processor P_u owns a square of the A,B, C matrices (depicted in blue on Figure 9.6) corresponding to a sub-cube of the computational domain (in grey on the figure). More precisely, there exist sets of indices I, J and K such that P_u owns all values $A_{i,k}$, $B_{k,j}$, $C_{i,j}$ for $i \in I$, $j \in J$ and $k \in K$, so that it is able to compute all corresponding tasks $T_{i,j,k}$. When a processor becomes idle, instead of sending a single element of A or B, we choose a tuple (i, j, k) of new indices (with $i \notin I$, $j \notin J$ and $k \notin K$) and allocate to P_u all the data needed to extend the sets I, J, K with (i, j, k). This corresponds to

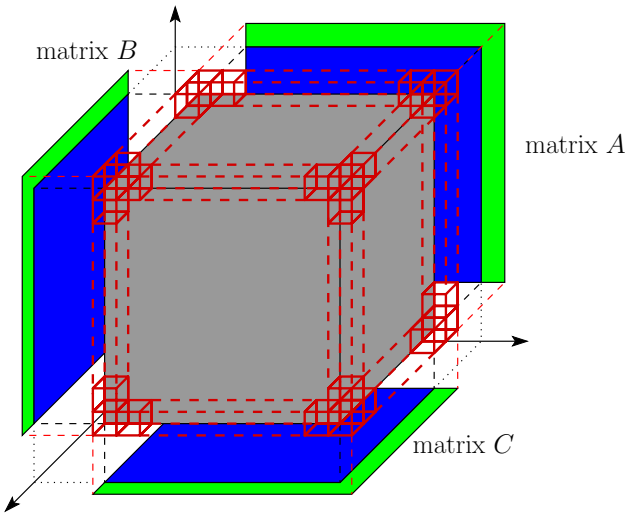


Figure 9.6: One step of the DynamicMatrix algorithm.

sending $2 \times (2|I| + 1)$ elements of A and B before the computation and $2|I| + 1$ elements of C back to the master after the computation (in green

on Figure 9.6).). Processor P_u is then allocated all the unprocessed tasks that can be done with the new data (in red on Figure 9.6).

As in the case of the outer product, when the number of remaining elementary products to be processed becomes small, such a strategy turns out to be inefficient. We therefore introduce the DynamicMatrix2Phases strategy that switches from the DynamicMatrix strategy to the random distribution strategy when the number of unprocessed tasks becomes smaller than a threshold. We also adapt the previous analysis, which allows to compute the optimal value of this threshold in order to minimize the amount data movement. Similarly, we show that our analysis succeeds in optimizing the threshold, which leads to a much reduced amount of data movement compared to other dynamic strategies (see Figure 9.7).

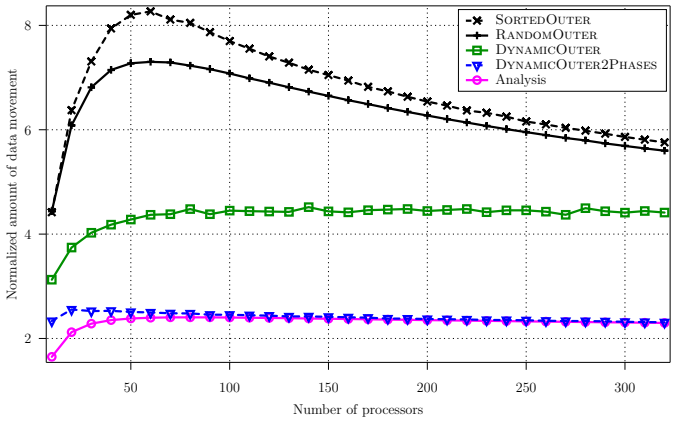


Figure 9.7: Data movement of all strategies for matrix-matrix product of size $n = 40$ ($n^3 = 64,000$ tasks).

9.9 Conclusion of the chapter

In this chapter, we have focused on dynamic strategies to allocate tasks whose dependance with their input data follows a two-dimensional or three-dimensional grid, which appears for instance with the outer-product and matrix-product operations. We have proposed strategies that take advantage of the data already distributed to minimize the amount of data movement. Our analysis considers that the behavior of this strategy ressembles a continuous process when the number of tasks is large and thus can be studied using differential equations. This allows to compute the threshold defining when to switch to a pure random strategy, which is more efficient to distribute the last remaining tasks. We provided simulations showing that the analysis succeeds in computing such a threshold and that the resulting amount of data movement is reduced compared to other dynamic strategies.

An interesting remaining question is to know if our analysis can be better justified using theoretical tools such as mean field theory [43]. However, given the complexity of our distribution process, it is not clear if this goal is achievable. Extending this type of analysis to more complex dynamic schedulers and data dependency patterns would also be very useful, given the practical and growing importance of dynamic runtime schedulers.

Note on the material presented in this chapter. This work was done in collaboration with Olivier Beaumont (Inria Bordeaux) and presented at the HPDC'2014 conference [C26].

Chapter 10

Conclusion and perspectives

In this document, I have presented several contributions on memory-aware algorithms and data movement optimizations. Most of them (Part I) deal with task graphs, and how to better schedule them to optimize memory and I/O usage. They revisit and largely extend the algorithms for trees presented in Chapter 1 to a number of different scenarios: larger class of graphs (series-parallel), processing on parallel platforms or hybrid CPU-GPU machines. This was also an opportunity to express all known algorithms in this area in a common, simple but expressive model, proposed in Chapter 2. The other contributions (Part II) deal with different problems of data layout and data distribution for matrix computations. Their target is less focused, but each of them looks at an important problem: how to deal with memory hierarchies (Chapter 7), how and when to redistribute data before a computation (Chapter 8) and how to dynamically distribute data on a heterogeneous computing platform (Chapter 9).

In all these contributions, we have tried to avoid the pitfall of being too theoretical or too applied: we made sure that the model used was close to the behavior of actual software codes running on actual machines, so that the results have a practical application, but also that the obtained results were general enough to survive technological changes and/or could be applied to other scheduling problems. The memory-aware dataflow model proposed in Chapter 2 is for instance a good compromise between theory and practice because: (i) it can be used to model a lot of actual applications in a straightforward way, and (ii) it is general enough to be useful in some other scheduling contexts where storage of temporary data is important.

Fortunately or not, the problems tackled in this document are not likely to be overcome by new computer design or technologies: if memory sizes keep increasing, the ratio between memory bandwidth and computation rate is always shrinking, which makes optimization for memory and I/O still a crucial concern. In addition, the work on memory-aware algorithms presented in this document may be extended in several directions:

- Numerous challenges are still to be addressed, as computers seem to become more heterogeneous than ever with the use of different types of memory such as non-volatile RAM or high-bandwidth MCDRAM, in addition to the now common use of computing accelerators such as GPUs.

- For now, we have mainly focused on the shared-memory case (except in Chapter 5). There is a need to design memory-aware algorithms for distributed platforms, where the available memory is scattered on different nodes. In this setting, there is a natural tradeoff between using all the available memory (as it was a single shared memory) to process more tasks, and avoiding data movement among cores. The hard part will be to come up with a model that is expressive enough, taking into account the platform topology and the data movement costs, and where most problems are still tractable.

- In many scientific applications, the task graph is not determined before the execution but is only discovered at runtime, as it depends on the data itself. Besides, runtime schedulers usually avoid exposing all the graph at the beginning of the computation to limit their scheduling time, even when processing deterministic applications. Thus, it would be interesting to design memory-aware algorithms that can deal with dynamically uncovered task graphs. We cannot expect to have the same optimality results, however a guarantee that the processing will not exceed a given memory limit would be very helpful.

- On a shorter term perspective, I would like to implement the algorithms proposed in this document, or maybe a more "applied" version of these algorithms in a real runtime scheduler. This is the reason for a starting collaboration with the StarPU team [8]. We first need to reduce the runtime complexity of some algorithms and to make them more dynamic so they can be embedded in such a runtime system.

On a longer term, I plan to continue focusing on scheduling for modern computing platforms. As stated above, their heterogeneity keeps increasing, with both hybrid cores and different memories, and they are getting even more distributed, which asks for taking communications into account when scheduling applications. My objective is still to design "usable" scheduling algorithms, that is, possibly dynamic algorithms, which do not depend on hard-to-instantiate platform characteristics, and with limited complexity. Covering the full path from theoretical studies to implementations in actual schedulers is the ultimate goal. Of course, it can only be reached on special cases and by collaborating with experts in applications and runtime schedulers.

Appendix A

Bibliography

[1] Alok Aggarwal, Jeffrey Vitter, et al. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] Emmanuel Agullo. *On the Out-Of-Core Factorization of Large Sparse Matrices*. PhD thesis, École normale supérieure de Lyon, France, 2008.

[3] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, and Florent Lopez. Implementing multifrontal sparse solvers for multicore architectures with sequential task flow runtime systems. *ACM Trans. Math. Softw.*, 43(2):13, 2016.

[4] Emmanuel Agullo, Abdou Guermouche, and Jean-Yves L'Excellent. Reducing the I/O volume in sparse out-of-core multifrontal methods. *SIAM Journal on Scientific Computing*, 31(6):4774–4794, 2010.

[5] 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.

[6] 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.

[7] Lars Arge, Michael T. Goodrich, Michael J. Nelson, and Nodari Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *Proceedings of the 20th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2008)*, pages 197–206, 2008.

[8] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on

heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[9] Grey Ballard, Dulceneia Becker, James Demmel, Jack J. Dongarra, Alex Druinsky, Inon Peled, Oded Schwartz, Sivan Toledo, and Ichitaro Yamazaki. Communication-avoiding symmetric-indefinite factorization. *SIAM J. Matrix Analysis Applications*, 35(4):1364–1406, 2014.

[10] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Minimizing communication in numerical linear algebra. *SIAM J. Matrix Analysis Applications*, 32(3):866–901, 2011.

[11] Olivier Beaumont, Vincent Boudet, Fabrice Rastello, and Yves Robert. Partitioning a square into rectangles: Np-completeness and approximation algorithms. *Algorithmica*, 34(3):217–239, 2002.

[12] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[13] Veeravalli Bharadwaj, Debasish Ghose, and Thomas G. Robertazzi. Divisible load theory: A new paradigm for load scheduling in distributed systems. *Cluster Computing*, 6(1), January 2003.

[14] 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, 1997.

[15] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Herault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, Piotr Luszczek, Asim YarKhan, and Jack Dongarra. Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA. In *12th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC'11)*, 2011.

[16] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Herault, and Jack J Dongarra. PaRSEC: Exploiting heterogeneity for enhancing scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013.

[17] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. DAGuE: A generic distributed DAG engine for high performance computing. In *16th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'11)*, 2011.

[18] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. DAGuE: A generic distributed DAG engine for high performance computing. *Parallel Computing*, 38(1):37–51, 2012.

[19] T.D. Braun, H.J. Siegel, N. Beck, L.L. Bölöni, M. Maheswaran, A.I. Reuther, J. P. Robertson, M.D. Theys, B. Yao, D. Hensgen, and R. F. Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 61(6):810–837, 2001.

[20] L. E. Cannon. *A cellular computer to implement the Kalman filter algorithm.* PhD thesis, Montana State University, 1969.

[21] Matteo Ceccarello and Francesco Silvestri. Experimental evaluation of multi-round matrix multiplication on mapreduce. In *Proceedings of the Seventeenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 119–132, 2015.

[22] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley. ScaLAPACK: a portable linear algebra library for distributed memory computers — design issues and performance. *Computer Physics Communications*, 97(1–2):1–15, 1996.

[23] Paul G Constantine and David F Gleich. Tall and skinny QR factorizations in mapreduce architectures. In *Proceedings of the second international workshop on MapReduce and its applications*, pages 43–50. ACM, 2011.

[24] Gennaro Cordasco, Rosario De Chiara, and Arnold L. Rosenberg. Assessing the computational benefits of area-oriented dag-scheduling. In *Euro-Par 2011 Parallel Processing - 17th International Conference*, pages 180–192, 2011.

[25] Gennaro Cordasco and Arnold L. Rosenberg. On scheduling series-parallel dags to maximize area. *Int. J. Found. Comput. Sci.*, 25(5):597–622, 2014.

[26] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauser, Eunice E. Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 1–12, 1993.

[27] Tatjana Davidović, Leo Liberti, Nelson Maculan, and Nenad Mladenović. Towards the optimal solution of the multiprocessor scheduling problem with communication delays. In *In MISTA Proceedings*, 2007.

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

[29] James Demmel. Communication-avoiding algorithms for linear algebra and beyond. In *27th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2013, Cambridge, MA, USA, May 20-24, 2013*, page 585, 2013.

[30] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. Communication-optimal parallel and sequential QR and LU factorizations. *SIAM J. Scientific Computing*, 34(1), 2012.

[31] Frédéric Desprez, Jack Dongarra, Antoine Petitet, Cyril Randriamaro, and Yves Robert. Scheduling block-cyclic array redistribution. *IEEE Trans. Parallel Distributed Systems*, 9(2):192–205, 1998.

[32] Elizabeth D. Dolan and Jorge J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213, 2002.

[33] Jack Dongarra, Jean-Francois Pineau, Yves Robert, Zhiao Shi, and Frédéric Vivien. Revisiting matrix product on master-worker platforms. *Int. J. Found. Comput. Sci.*, 19(6):1317–1336, 2008.

[34] Maciej Drozdowski. Scheduling parallel tasks – algorithms and complexity. In Joseph Leung, editor, *Handbook of Scheduling*. Chapman and Hall/CRC, 2004.

[35] Pierre-Francois Dutot, Krzyztof Rzadca, Erik Saule, Denis Trystram, et al. Multiobjective scheduling. In Yves Robert and Frédéric Vivien, editors, *Introduction to Scheduling*. CRC Press, 2010.

[36] Venmugil Elango, Fabrice Rastello, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. On characterizing the data access complexity of programs. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 567–580, 2015.

[37] David Eppstein. Parallel recognition of series-parallel graphs. *Information and Computation*, 98(1):41 – 55, 1992.

[38] Lucian Finta, Zhen Liu, Ioannis Mills, and Evripidis Bampis. Scheduling uet-uct series-parallel graphs on two processors. *Theoretical Computer Science*, 162(2):323–340, 1996.

[39] Philippe Flajolet, Jean-Claude Raoult, and Jean Vuillemin. The number of registers required for evaluating arithmetic expressions. *Theoretical Computer Science*, 9(1):99–125, 1979.

[40] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *FOCS'99, the 40th IEEE Symposium on Foundations of Computer Science*, pages 285–298. IEEE Computer Society Press, 1999.

[41] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99*, pages 285–298, 1999.

[42] M. R. Garey and D. S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W.H. Freeman and Co, London (UK), 1979.

[43] Nicolas Gast, Bruno Gaujal, and Jean-Yves Le Boudec. Mean field for Markov Decision Processes: from Discrete to Continuous Optimization. *IEEE Transactions on Automatic Control*, 57(9):2266 – 2280, 2012.

[44] Thierry Gautier, Xavier Besseron, and Laurent Pigeon. Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, PASCO '07, pages 15–23, New York, NY, USA, 2007. ACM.

[45] John R. Gilbert, Thomas Lengauer, and Robert Endre Tarjan. The pebbling problem is complete in polynomial space. *SIAM J. Comput.*, 9(3):513–524, 1980.

[46] Arturo González-Escribano, Arjan J. C. van Gemund, and Valentín Cardeñoso-Payo. Mapping unstructured applications into nested parallelism. In *High Performance Computing for Computational Science - VECPAR*, pages 407–420, 2002.

[47] R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, XLV(9):1563–1581, 1966.

[48] Susan L Graham, Marc Snir, Cynthia A Patterson, et al. *Getting up to speed: The future of supercomputing*. National Academies Press, 2005.

[49] L. Hollermann, T. S. Hsu, D. R. Lopez, and K. Vertanen. Scheduling problems in a practical allocation model. *J. Combinatorial Optimization*, 1(2):129–149, 1997.

[50] 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.

[51] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matching in bipartite graphs. *SIAM Journal of Computing*, 2(4):225–231, 1973.

[52] T. S. Hsu, J. C. Lee, D. R. Lopez, and W. A. Royce. Task allocation on a network of processors. *IEEE Trans. Computers*, 49(12):1339–1353, 2000.

[53] T.C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9:841–848, 1961.

[54] Dror Ironya, Sivan Toledo, and Alexander Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel Distributed Computing*, 64(9):1017–1026, 2004.

[55] Mathias Jacquelin. *Memory-aware algorithms : from multicores to large scale platforms. (Algorithmes orientés mémoire : des processeurs multi-cœurs aux plates-formes à grande échelle)*. PhD thesis, École normale supérieure de Lyon, France, 2011.

[56] Richard M. Karp and Vijaya Ramachandran. Parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pages 869–942. The MIT Press, 1990.

[57] Safia Kedad-Sidhoum, Florence Monna, and Denis Trystram. Scheduling tasks with precedence constraints on hybrid multi-core machines. In *International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 27–33, 2015.

[58] Chi-Chung Lam, Thomas Rauber, Gerald Baumgartner, Daniel Cociorva, and P. Sadayappan. Memory-optimal evaluation of expression trees involving large objects. *Computer Languages, Systems & Structures*, 37(2):63–75, 2011.

[59] Julien Langou. Communication lower bounds for matrix-matrix multiplication. Talk at the Dagstuhl Seminar 15281. Slides available online: http://materials.dagstuhl.de/files/15/15281/15281.JulienLangou.Slides.pdf, 2015.

[60] Mun-Kyu Lee, Pierre Michaud, Jeong Seop Sim, and DaeHun Nyang. A simple proof of optimality for the MIN cache replacement policy. *Inf. Process. Lett.*, 116(2):168–170, 2016.

[61] Thomas Lengauer. Black-white pebbles and graph separation. *Acta Informatica*, 16(4):465–475, 1981.

[62] Thomas Lengauer and Robert E Tarjan. Asymptotically tight bounds on time-space trade-offs in a pebble game. *Journal of the ACM*, 29(4):1087–1130, 1982.

[63] J. K. Lenstra, A. H. G. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1:343–362, 1977.

[64] Joseph W. H. Liu. On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Trans. Math. Software*, 12(3):249–264, 1986.

[65] Joseph W. H. Liu. An application of generalized tree pebbling to sparse matrix factorization. *SIAM J. Algebraic Discrete Methods*, 8(3):375–395, 1987.

[66] Lynn H Loomis and Hassler Whitney. An inequality related to the isoperimetric inequality. *Bulletin of the American Mathematical Society*, 55(10):961–962, 1949.

[67] Friedhelm Meyer auf der Heide. A comparison of two variations of a pebble game on graphs. *Theoretical Computer Science*, 13(3):315–322, 1981.

[68] Lynette I Millett and Samuel H Fuller. *The Future of Computing Performance:: Game Over or Next Level?* National Academies Press, 2011.

[69] Burkhard Monien and Ivan Hal Sudborough. Min cut is NP-complete for edge weighted trees. *Theoretical Computer Science*, 58(1):209–229, 1988.

[70] Clyde L. Monma and Jeffrey B. Sidney. Sequencing with series-parallel precedence constraints. *Mathematics of Operations Research*, 4(3):215–224, 1979.

[71] OpenMP Architecture Review Board. OpenMP application program interface, version 4.0. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, July 2013.

[72] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. Hierarchical task-based programming with StarSs. *IJHPCA*, 23(3):284–299, 2009.

[73] G. N. Srinivasa Prasanna and Bruce R. Musicus. Generalized multiprocessor scheduling and applications to matrix computations. *IEEE TPDS*, 7(6):650–664, 1996.

[74] Sartaj Sahni and George Vairaktarakis. The master-slave scheduling model. In J. Leung, editor, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. Chapman and Hall/CRC, 2004.

[75] John E. Savage. *Models of Computation: Exploring the Power of Computing*. Addison-Wesley, 1997.

[76] A. Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*, volume 24 of *Algorithms and Combinatorics*. Springer-Verlag, 2003.

[77] Sangwon Seo, Edward J Yoon, Jaehong Kim, Seongwook Jin, Jin-Soo Kim, and Seungryoul Maeng. Hama: An efficient matrix computation with the mapreduce framework. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 721–726. IEEE, 2010.

[78] Ravi Sethi. Complete register allocation problems. In *Proceedings of the 5th Annual ACM Symposium on Theory of Computing (STOC'73)*, pages 182–195, New York, NY, USA, 1973. ACM Press.

[79] Ravi Sethi and J.D. Ullman. The generation of optimal code for arithmetic expressions. *J. ACM*, 17(4):715–728, 1970.

[80] John Shalf, Sudip S. Dosanjh, and John Morrison. Exascale computing technology challenges. In *9th International conference on High Performance Computing for Computational Science - VECPAR 2010*, pages 1–25, 2010.

[81] Michael Stonebraker, Jennie Duggan, Leilani Battle, and Olga Papaemmanouil. SciDB DBMS research at M.I.T. *IEEE Data Eng. Bull.*, 36(4):21–30, 2013.

[82] Arthur N Strahler. Hypsometric (area-altitude) analysis of erosional topography. *Geological Society of America Bulletin*, 63(11):1117–1142, 1952.

[83] Sivan Toledo. A survey of out-of-core algorithms in numerical linear algebra. In *External Memory Algorithms and Visualization*, pages 161–180. American Mathematical Society Press, 1999.

[84] H. Topcuoglu, S. Hariri, and M. Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distributed Systems*, 13(3):260–274, 2002.

[85] Jacobo Valdes, Robert Endre Tarjan, and Eugene L. Lawler. The recognition of series parallel digraphs. *SIAM J. Comput.*, 11(2):298–313, 1982.

[86] Peter van Emde Boas and Jan van Leeuwen. Move rules and trade-offs in the pebble game. In *Theoretical Computer Science 4th GI Conference*, pages 101–112. Springer, 1979.

[87] Sarad Venugopalan and Oliver Sinnen. Optimal linear programming solutions for multiprocessor scheduling with communication delays. In *ICA3PP (1)*, pages 129–138, 2012.

[88] Jeffrey Scott Vitter. External memory algorithms. In *Handbook of massive data sets*, pages 359–416. Springer, 2002.

[89] Mihalis Yannakakis. A polynomial algorithm for the min-cut linear arrangement of trees. *Journal of the ACM (JACM)*, 32(4):950–988, 1985.

Appendix B

Personal publications

B.1 International journals

[J1] Olivier Beaumont, Arnaud Legrand, Loris Marchal, and Yves Robert. Scheduling strategies for mixed data and task parallelism on heterogeneous clusters. *Parallel Processing Letters*, 13(2):225–244, 2003.

[J2] Arnaud Legrand, Loris Marchal, and Yves Robert. Optimizing the steady-state throughput of scatter and reduce operations on heterogeneous platforms. *J. Parallel and Distributed Computing*, 65(12):1497–1514, 2005.

[J3] Olivier Beaumont, Arnaud Legrand, Loris Marchal, and Yves Robert. Steady-state scheduling on heterogeneous clusters. *Int. J. of Foundations of Computer Science*, 16(2):163–194, 2005.

[J4] Olivier Beaumont, Arnaud Legrand, Loris Marchal, and Yves Robert. Pipelining broadcasts on heterogeneous platforms. *IEEE Trans. Parallel Distributed Systems*, 16(4):300–313, 2005.

[J5] Loris Marchal, Yang Yang, Henri Casanova, and Yves Robert. Steady-state scheduling of multiple divisible load applications on wide-area distributed computing platforms. *Int. Journal of High Performance Computing Applications*, 20(3):365–381, 2006.

[J6] Olivier Beaumont, Loris Marchal, and Yves Robert. Complexity results for collective communications on heterogeneous platforms. *Int. Journal of High Performance Computing Applications*, 20(1):5–17, 2006.

[J7] Loris Marchal, Veronika Rehn, Yves Robert, and Frédéric Vivien. Scheduling algorithms for data redistribution and load-balancing on master-slave platforms. *Parallel Processing Letters*, 17(1):61–77, 2007.

[J8] Olivier Beaumont, Larry Carter, Jeanne Ferrante, Arnaud Legrand, Loris Marchal, and Yves Robert. Centralized versus distributed schedulers for multiple bag-of-task applications. *IEEE Trans. Parallel Distributed Systems*, 19(5):698–709, 2008.

[J9] Anne Benoit, Loris Marchal, Jean-François Pineau, Yves Robert, and Frédéric Vivien. Scheduling concurrent bag-of-tasks applications on heterogeneous platforms. *IEEE Transactions on Computers*, 59(2):202–217, 2010.

[J10] Sékou Diakité, Loris Marchal, Jean-Marc Nicod, and Laurent Philippe. Practical steady-state scheduling for tree-shaped task graphs. *Parallel Processing Letters*, 21(4):397–412, 2011.

[J11] Tudor David, Mathias Jacquelin, and Loris Marchal. Scheduling streaming applications on a complex multicore platform. *Concurrency and Computation: Practice and Experience*, 24(15):1726–1750, 2012.

[J12] Anne Benoit, Louis-Claude Canon, and Loris Marchal. Non-clairvoyant reduction algorithms for heterogeneous platforms. *Concurrency and Computation: Practice and Experience*, 27(6):1612–1624, 2015.

[J13] Julien Herrmann, Loris Marchal, and Yves Robert. Memory-aware tree traversals with pre-assigned tasks. *J. Parallel Distrib. Comput.*, 75:53–66, 2015.

[J14] Thomas Lambert, Loris Marchal, and Bora Uçar. Comments on the hierarchically structured bin packing problem. *Information Processing Letters*, 115(2):306–309, 2015.

[J15] Lionel Eyraud-Dubois, Loris Marchal, Oliver Sinnen, and Frédéric Vivien. Parallel scheduling of task trees with limited memory. *ACM Transactions on Parallel Computing*, 2(2):36, July 2015.

[J16] Julien Herrmann, George Bosilca, Thomas Hérault, Loris Marchal, Yves Robert, and Jack Dongarra. Assessing the cost of redistribution followed by a computational kernel: Complexity and performance results. *Parallel Computing*, 52:20, 2016.

[J17] Enver Kayaaslan, Thomas Lambert, Loris Marchal, and Bora Uçar. Scheduling series-parallel task graphs to minimize peak memory. *Theoretical Computer Science*, 2017.

[J18] Loris Marchal, Bertrand Simon, Oliver Sinnen, and Frédéric Vivien. Malleable task-graph scheduling with a practical speed-up model. *Transactions on Parallel and Distributed Systems*, 29(6):1357–1370, 2018.

B.2 Book chapters

[B1] Olivier Beaumont and Loris Marchal. Steady-state scheduling. In *Introduction to Scheduling*, pages 159–186. Chapman and Hall/CRC Press, 2009.

[B2] Anne Benoit, Loris Marchal, Yves Robert, and Frédéric Vivien. Algorithms and scheduling techniques for clusters and grids. In Wolfgang Gentzsch, Lucio Grandinetti, and Gerhard Joubert, editors, *Advances in Parallel Computing, vol.18: High Speed and Large Scale Scientific Computing*, pages 27–51. IOS Press, 2009.

[B3] Anne Benoit, Loris Marchal, Yves Robert, Bora Uçar, and Frédéric Vivien. Scheduling for large-scale systems. In Teofilo F. Gonzalez, Jorge Diaz-Herrera, and Allen Tucker, editors, *Computing Handbook, Third Edition: Computer Science and Software Engineering*, pages 59: 1–33. CRC Press, 2014.

B.3 International conference proceedings

[C1] Henri Casanova, Arnaud Legrand, and Loris Marchal. Scheduling distributed applications: the simgrid simulation framework. In *IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 138–145, 2003.

[C2] Olivier Beaumont, Arnaud Legrand, Loris Marchal, and Yves Robert. Complexity results and heuristics for pipelined multicast operations on heterogeneous platforms. In *International Conference on Parallel Processing (ICPP)*, pages 267–274, 2004.

[C3] Olivier Beaumont, Arnaud Legrand, Loris Marchal, and Yves Robert. Pipelining broadcasts on heterogeneous platforms. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[C4] Olivier Beaumont, Loris Marchal, and Yves Robert. Scheduling divisible loads with return messages on heterogeneous master-worker platforms. In *International Conference on High Performance Computing (HiPC)*, pages 498–507, 2005.

[C5] Loris Marchal, Pascale Primet, Yves Robert, and Jingdi Zeng. Optimizing network resource sharing in grids. In *IEEE Global Telecommunications Conference (GlobeCom)*, 2005.

[C6] Olivier Beaumont, Loris Marchal, and Yves Robert. Broadcast trees for heterogeneous platforms. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

[C7] Loris Marchal, Yang Yang, Henri Casanova, and Yves Robert. A realistic network/application model for scheduling divisible loads on large-scale platforms. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

[C8] Olivier Beaumont, Arnaud Legrand, Loris Marchal, and Yves Robert. Independent and divisible tasks scheduling on heterogeneous star-schaped platforms with limited memory. In *13th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 179–186, 2005.

[C9] Loris Marchal, Pascale Primet, Yves Robert, and Jingdi Zeng. Optimal bandwidth sharing in grid environment. In *15th International Symposium on High Performance Distributed Computing (HPDC)*, pages 144–155, 2006.

[C10] Olivier Beaumont, Larry Carter, Jeanne Ferrante, Arnaud Legrand, Loris Marchal, and Yves Robert. Centralized versus distributed schedulers for multiple bag-of-task applications. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[C11] Olivier Beaumont, Anne-Marie Kermarrec, Loris Marchal, and Etienne Riviére. Voronet: A scalable object network based on voronoi tessellations. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2007.

[C12] Loris Marchal, Veronika Rehn, Yves Robert, and Frédéric Vivien. Scheduling and data redistribution strategies on star platforms. In *15th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 288–295, 2007.

[C13] Matthieu Gallet, Loris Marchal, and Frédéric Vivien. Allocating series of workflows on computing grids. In *14th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 48–55, 2008.

[C14] Matthieu Gallet, Loris Marchal, and Frédéric Vivien. Efficient scheduling of task graph collections on heterogeneous resources. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2009.

[C15] Sékou Diakité, Loris Marchal, Jean-Marc Nicod, and Laurent Philippe. Steady-state for batches of identical task graphs. In *15th International Euro-Par Conference*, pages 203–215, 2009.

[C16] Mathias Jacquelin, Loris Marchal, and Yves Robert. Complexity analysis and performance evaluation of matrix product on multicore archi-

tectures. In *International Conference on Parallel Processing (ICPP)*, pages 196–203, 2009.

[C17] Javier Celaya and Loris Marchal. A fair decentralized scheduler for bag-of-tasks applications on desktop grids. In *10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 538–541, 2010.

[C18] Anne Benoit, Loris Marchal, Oliver Sinnen, and Yves Robert. Mapping pipelined applications with replication to increase throughput and reliability. In *22nd International Symposium on Parallel and Distributed Computing (SBAC-PAD)*, 2010.

[C19] Mathias Jacquelin, Loris Marchal, Yves Robert, and Bora Uçar. On optimal tree traversals for sparse matrix factorization. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 556–567, 2011.

[C20] Franck Cappello, Mathias Jacquelin, Loris Marchal, Yves Robert, and Marc Snir. Comparing archival policies for BlueWaters. In *International Conference on High Performance Computing (HiPC'2011)*, 2011.

[C21] Olivier Beaumont, Nicolas Bonichon, Lionel Eyraud-Dubois, and Loris Marchal. Minimizing weighted mean completion time for malleable tasks scheduling. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 273–284, 2012.

[C22] Loris Marchal, Oliver Sinnen, and Frédéric Vivien. Scheduling tree-shaped task graphs to minimize memory and makespan. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 839–850, 2013.

[C23] Julien Herrmann, Loris Marchal, and Yves Robert. Model and complexity results for tree traversals on hybrid platforms. In *19th International Euro-Par Conference*, pages 647–658, 2013.

[C24] Olivier Beaumont, Hubert Larchevêque, and Loris Marchal. Non linear divisible loads: There is no free lunch. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 863–873, 2013.

[C25] Thomas Hérault, Julien Herrmann, Loris Marchal, and Yves Robert. Determining the optimal redistribution for a given data partition. In *IEEE 13th International Symposium on Parallel and Distributed Computing, (ISPDC)*, pages 95–102, 2014.

[C26] Olivier Beaumont and Loris Marchal. Analysis of dynamic scheduling strategies for matrix multiplication on heterogeneous platforms. In *23rd International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 141–152, 2014.

[C27] Abdou Guermouche, Loris Marchal, Bertrand Simon, and Frédéric Vivien. Scheduling Trees of Malleable Tasks for Sparse Linear Algebra. In *International European Conference on Parallel and Distributed Computing (Euro-Par 2015)*, 2015.

[C28] Guillaume Aupy, Clément Brasseur, and Loris Marchal. Dynamic Memory-Aware Task-Tree Scheduling. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017.

[C29] Louis-Claude Canon, Loris Marchal, and Frédéric Vivien. Low-Cost Approximation Algorithms for Scheduling Independent Tasks on Hybrid Platforms. In *International European Conference on Parallel and Distributed Computing (Euro-Par 2017)*, 2017.

[C30] Anne Benoit, Changjiang Gou, and Loris Marchal. Memory-aware tree partitioning on homogeneous platforms. In *26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, 2018. Short paper.

[C31] Loris Marchal, Hanna Nagy, Bertrand Simon, and Frédéric Vivien. Parallel scheduling of DAGs under memory constraints. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018.

B.4 International workshop proceedings

[W1] Arnaud Legrand, Loris Marchal, and Yves Robert. Optimizing the steady-state throughput of scatter and reduce operations on heterogeneous platforms. In *6th Workshop on Advances in Parallel and Distributed Computational Models (APDCM, workshop of IPDPS)*, 2004.

[W2] Olivier Beaumont, Arnaud Legrand, Loris Marchal, and Yves Robert. Steady-state scheduling on heterogeneous clusters: why and how? In *6th Workshop on Advances in Parallel and Distributed Computational Models (APDCM, workshop of IPDPS)*, 2004.

[W3] Olivier Beaumont, Arnaud Legrand, Loris Marchal, and Yves Robert. Assessing the impact and limits of steady-state scheduling for mixed task and data parallelism on heterogeneous platforms. In *HeteroPar (workshop of Euro-Par)*, pages 296–302, 2004.

[W4] Olivier Beaumont, Loris Marchal, Veronika Rehn, and Yves Robert. FIFO scheduling of divisible loads with return messages under the one-port model. In *15th Heterogeneous Computing Workshop (HCW, workshop of IPDPS)*, 2006.

[W5] Jack DiGiovanna, Loris Marchal, Prapaporn Rattanatamrong, Ming Zhao, Shalom Darmanjian, Babak Mahmoudi, Justin Sanchez, José Príncipe, Linda Hermer-Vazquez, Renato Figueiredo, and José Fortes. Towards real-time distributed signal modeling for brain machine interfaces. In *Proceedings of Dynamic Data Driven Application Systems (workshop of ICCS)*, volume 4487 of *LNCS*, pages 964–971, 2007.

[W6] Anne Benoit, Loris Marchal, Jean-François Pineau, Yves Robert, and Frédéric Vivien. Offline and online scheduling of concurrent bags-of-tasks on heterogeneous platforms. In *10th Workshop on Advances in Parallel and Distributed Computational Models (APDCM, workshop of IPDPS)*, 2008.

[W7] Anne Benoit, Loris Marchal, Jean-François Pineau, Yves Robert, and Frédéric Vivien. Resource-aware allocation strategies for divisible loads on large-scale systems. In *18th Heterogeneity in Computing Workshop (HCW, workshop of IPDPS)*, 2009.

[W8] Matthieu Gallet, Mathias Jacquelin, and Loris Marchal. Scheduling complex streaming applications on the cell processor. In *Workshop on Multithreaded Architectures and Applications (MTAAP, workshop of IPDPS)*, 2010.

[W9] Anne Benoit, Louis-Claude Canon, and Loris Marchal. Non-clairvoyant reduction algorithms for heterogeneous platforms. In *HeteroPar (workshop of Euro-Par)*, pages 270–279, 2013.

[W10] Julien Herrmann, Loris Marchal, and Yves Robert. Memory-aware list scheduling for hybrid platforms. In *23rd Heterogeneity in Computing Workshop (HCW, workshop of IPDPS)*, pages 689–698, 2014.

[W11] Emmanuel Agullo, Olivier Beaumont, Lionel Eyraud-Dubois, Julien Herrmann, Suraj Kumar, Loris Marchal, and Samuel Thibault. Bridging the Gap between Performance and Bounds of Cholesky Factorization on Heterogeneous Platforms. In *24th Heterogeneity in Computing Workshop (HCW, workshop of IPDPS)*, 2015.

[W12] Fouad Hanna, Loris Marchal, Jean-Marc Nicod, Laurent Philippe, Veronika Rehn-Sonigo, and Hala Sabbah. Minimizing Rental Cost for Multiple Recipe Applications in the Cloud. In *25th Heterogeneity in Computing Workshop (HCW, workshop of IPDPS)*, pages 28–37, 2016.

[W13] Loris Marchal, Samuel Mccauley, Bertrand Simon, and Frédéric Vivien. Minimizing I/Os in Out-of-Core Task Tree Scheduling. In *19th Workshop on Advances in Parallel and Distributed Computational Models (APDCM, workshop of IPDPS)*, 2017.

[W14] Olivier Beaumont, Thomas Lambert, Loris Marchal, and Bastien Thomas. Data-locality aware dynamic schedulers for independent tasks with replicated inputs. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2018, Vancouver, BC, Canada, May 21-25, 2018*, pages 1206–1213, 2018.

B.5 Theses

[O1] Loris Marchal. *Communications collectives et ordonnancement en régime permanent sur plates-formes hétérogènes.* PhD thesis, École Normale Supérieure de Lyon, France, October 2006.

B.6 Research reports (not published elsewhere)

[R1] Loris Marchal, Samuel Mccauley, Bertrand Simon, and Frédéric Vivien. Minimizing I/Os in Out-of-Core Task Tree Scheduling. Research Report RR-9025, INRIA, February 2017.