Memory-aware Algorithms and Scheduling Techniques for Matrix Computattions

Julien Herrmann

▶ **To cite this version:**

Julien Herrmann. Memory-aware Algorithms and Scheduling Techniques for Matrix Computattions. Distributed, Parallel, and Cluster Computing [cs.DC]. Ecole normale supérieure de lyon - ENS LYON, 2015. English. NNT : 2015ENSL1043 . tel-01241485

HAL Id: tel-01241485

https://theses.hal.science/tel-01241485

Submitted on 10 Dec 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

N[o] attribué par la bibliothèque: 2015ENSL1043

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

THÈSE

en vue d'obtenir le grade de

Docteur de l'Université de Lyon, délivré par l'École Normale Supérieure de Lyon

Spécialité : Informatique

au titre de l'École Doctorale Informatique et Mathématiques

présentée et soutenue publiquement le 25 Novembre 2015 par

Julien HERRMANN

Memory-aware Algorithms and Scheduling Techniques for Matrix Computations

Directeur de thèse :	Yves	ROBERT
Co-encadrant de thèse :	Loris	MARCHAL

Devant la commission d'examen formée de :

Luc	GIRAUD	*Examinateur*
Pierre	MANNEBACK	*Examinateur*
Loris	MARCHAL	*Co-encadrant*
Yves	ROBERT	*Directeur*
Oliver	SINNEN	*Rapporteur*
Denis	TRYSTRAM	*Rapporteur*

Contents

Introduction **i**

I Linear Algebra **1**

1 Mixing LU and QR factorization algorithms **3**
1.1 Introduction 3
1.2 Hybrid LU-QR algorithms 5
1.2.1 LU step 6
1.2.2 QR step 7
1.2.3 LU step variants 8
1.2.4 Comments 9
1.3 Robustness criteria 10
1.3.1 Max criterion 10
1.3.2 Sum criterion 11
1.3.3 MUMPS criterion 12
1.3.4 Extending the MUMPS criterion to tiles 13
1.3.5 Complexity 14
1.4 Implementation 15
1.5 Experiments 17
1.5.1 Experimental framework 17
1.5.2 Results for random matrices 20
1.5.3 Results for different processor grid sizes 22
1.5.4 Results for special matrices 23
1.5.5 Results for varying the condition number 25
1.5.6 Assessment of the three criteria 26
1.6 Related work 26
1.6.1 LUPP 26
1.6.2 LU NoPiv 26
1.6.3 LU IncPiv 27
1.6.4 CALU 27
1.6.5 Summary 27
1.7 Conclusion 27

2 Cholesky factorization on heterogeneous platforms 29
2.1 Introduction 29
2.2 Context 30
2.2.1 Cholesky factorization 30
2.2.2 Multiprocessor scheduling 32
2.3 Tools and libraries 33
2.3.1 StarPU runtime system 33
2.3.2 Chameleon dense linear algebra library 34
2.3.3 Simgrid simulation engine 34
2.4 Lower bounds 35
2.4.1 Linear Programming formulation 35
2.4.2 Constraint Programming formulation 36
2.4.3 Upper bounds on performance 37
2.5 Experiments and Results 38
2.5.1 Schedulers 38
2.5.2 Experimental Setup 38
2.5.3 Results 38
2.6 Discussion 46
2.6.1 dmda vs dmdas scheduler 46
2.6.2 Mapping from Constraint Programming solution 47
2.6.3 Constraint Programming schedule in actual execution 47
2.7 Conclusion 47

II Scheduling Under Memory Constraints 49

3 Memory-aware list scheduling for hybrid platforms 51
3.1 Introduction 51
3.2 Related work 53
3.2.1 Task graph scheduling 53
3.2.2 Pebble game and its variants 53
3.2.3 Scheduling with memory constraints 54
3.2.4 Hybrid computing 54
3.3 Model and framework 54
3.3.1 Flow and resources constraints 54
3.3.2 Memory constraints 56
3.3.3 Optimization problem 58
3.4 Tree traversal with pre-assigned task 59
3.4.1 Complexity results on trees 59
3.4.2 Depth-first traversals 66
3.4.3 Heuristics 69
3.4.4 Experiments 70
3.5 General problem 73
3.5.1 ILP formulation 75
3.5.2 Heuristics 78
3.5.3 Simulation results 81
3.6 Conclusion 86

4 Assessing the cost of redistribution **89**
4.1 Introduction 89
4.2 Related work 92
4.2.1 Communication model 92
4.2.2 General data redistribution 92
4.2.3 Array redistribution 93
4.3 Model and framework 93
4.3.1 Definitions 93
4.3.2 Cost of a redistribution 94
4.3.3 Optimization problems 95
4.4 Redistribution 96
4.4.1 Total communication volume 96
4.4.2 Number of parallel communication steps 96
4.4.3 Evaluation of optimal vs. arbitrary redistributions 98
4.5 Coupling redistribution and stencil computations 100
4.5.1 Application model 101
4.5.2 Complexity 102
4.6 Experiments 108
4.6.1 Setup 109
4.6.2 Stencil 109
4.6.3 QR factorization 110
4.7 Conclusion 116

III Auto-adjoint computations **117**

5 Optimal multistage algorithms for adjoint computation **119**
5.1 Introduction 119
5.2 Framework 120
5.2.1 The AC problem 120
5.2.2 Algorithm model 122
5.3 Solution of $\textsc{Prob}_\infty(l, c_m, w_d, r_d)$ 123
5.3.1 Properties of an optimal algorithm 123
5.3.2 Optimal algorithm 130
5.4 Simulations 136
5.4.1 Stumm and Walther's algorithm ($\text{SWA}^\star(l, c_m, w_d, r_d)$) 136
5.4.2 Simulation setup 137
5.4.3 Simulation results 137
5.5 Conclusion and future work 139

Conclusion **141**

Bibliography **145**

Publications **153**

Introduction

The processing power of computers has always been inextricably linked to the characteristics of their memory. From the early 1940's, many different technologies have been designed to store information. Built between 1943 and 1945, the world's first electronic digital computer, the ENIAC, was composed of twenty *vacuum tube accumulators*, each of them being able to store one 10-digit decimal number. Vacuum tubes had a considerable power consumption and were frequently overheating, leaving the ENIAC non-functional most of the time. Instead of storing data in individual bits, the next generation of computer memory, *Delay line memory*, was turning electrical pulses into sound waves, sending them through a medium that slowed them down. It significantly increased the storage capacity and the reliability of the computational systems. Delay line memory was sequential-access, as opposed to modern random-access memory, and required power to maintain the information. In the late 1940's, *magnetic-core memory* was the first non-volatile random-access memory and remained the main form of computer memory from the mid 1950's to the mid 1970's. It has been gradually replaced by *semiconductor-based memory*, that is still used as primary storage in nowadays computers. The cost of this memory has led computer designers to introduce a secondary storage that is cheaper but slower and not directly accessible by the CPU. Since their introduction in 1956, *hard disk drives* are usually used as secondary storage in modern computers. With the rise of storage capacity, the memory architecture has become more and more convoluted.

In the mid 1960's, the concept of *Virtual Memory* was introduced to provide the ability for operating systems to deal with this complex hierarchies of memory. The need to properly manage the memory usage of computation systems increased with the introduction of *multiprocessor* architectures. The Cray X-MP computer, released in 1982, was one of the first *shared-memory parallel vector processors*. It housed two CPUs embedded within a single machine that could concurrently access the same shared-memory, for a peak system performance of 400 millions Floating-point Operations Per Second (400 Megaflops). In the race for performance, the number of processors per machine kept on growing but faced technical issues. Instead of endlessly adding computational units within a single computer, multiprocessor machines were interconnected to form computer clusters. In such systems, the processors hosted on different machines work together following a *distributed memory* pattern, significantly complicating the memory architecture. In the 1990's, supercomputers with thousands of processors began to appear and ASCI Red was the first system to break through the 1 Teraflops barrier in 1996. Twelve years later, the IBM Roadrunner was the first supercomputer to reach Petascale performance. As of today, the Tianhe-2 supercomputer, released in 2013, is ranked number one in the TOP500 list of supercomputers [1] at the speed of 33.86 Petaflops in HPL Linpack benchmark [3]. Pushed by the increasing need of processing power by scientific applications, reaching the Exascale performance mark has become a common goal for computer designers.

Recently, on 29 July 2015, U.S. President Obama, established the National Strategic Computing Initiative (NSCI) to accelerate the development of an Exascale system [5]. It is an effort "to create a cohesive, multi-agency strategic vision and federal investment strategy in *High Performance Computing*

(HPC)". Such HPC systems are essential in science, engineering, technology, and industry. They are used in a wide range of computationally intensive tasks, like large data analysis or simulation of complex physical systems. Many scientific applications rely on supercomputers, such as the simulation of seismic activity, evolving climate, brain activity, nuclear fusion, among others. Most of these applications use subroutines to perform basic *matrix computations*. Developing efficient implementations of these linear algebra subroutines has become a crucial objective since the mid 1970's. In 1979, a specification for these common kernel operations using scalars and vectors was published: the level-1 *Basic Linear Algebra Subroutines* (BLAS). With the advent of vector processors, BLAS was augmented with level-2 kernel operations that concerned vector-matrix operations, such as vector-matrix multiplication, or linear triangular system resolution. Level-2 BLAS was designed to get the most of vector processors that were operating on one-dimensional arrays of data. In 1987, level-3 BLAS introduced kernel operations that concerned matrix-matrix operations, such as matrix-matrix multiplication, or dense linear system solvers. These level-3 BLAS operations partition the matrices into blocks, keeping data manipulations localized, allowing for a better usage of the *cache*. Many linear algebra libraries are available. Among them, we can highlight LAPACK [2], released in 1992, that provides a well-tuned BLAS implementation to exploit the caches on modern cache-based architectures. LAPACK is written in FORTRAN and is the successor of LINPACK that is still used as a benchmark for modern supercomputers. Indeed, linear algebra problems are computationally intensive jobs and represent a good way to compare the processing power of HPC systems. The TOP500 list ranks the supercomputers twice a year based on their performance on solving a dense linear system using an LU factorization with partial pivoting and a 64 bit floating point arithmetic. Optimizing these linear algebra libraries for tomorrow's supercomputers is one of the main concern in the HPC community.

To built the next generation of supercomputers, new scientific breakthroughs are required to cope with extreme scale computing. In February 2014, the Advanced Scientific Computing Advisory Committee (ASCAC) published the top ten Exascale research challenges [6]. This list suggests that a good memory management in those more and more complex architectures is a sine-qua-non condition to achieve the development of an Exascale system. Indeed, there is a growing disparity between the speed of the CPU and the main memory throughput, mainly due to the limited communication bandwidth. From 1986 to 2000, processor speed improved at an average annual rate of 55% while memory speed only improved at 10% [109]. One of the main consequences is that the processors may found themselves lacking in work and waiting for data to arrive from memory. To limit the number of memory accesses, processors are connected to an ever-increasing amount of high-speed cache memory. With the increasing size of the applications, multi-processor systems tend to provide a separate dedicated memory for each processor. This is the *Non-Uniform Memory Access* (NUMA) architecture. On most supercomputers, processors are equipped with accelerators, such as GPUs, FPGAs or the more recent Xeon Phi. The CPU offloads compute-intensive portions of the code to these coprocessors, increasing the general performance of the system. Altogether with the increasing complexity of memory architectures and the general heterogeneity at every level, modern supercomputers offer an intricate challenge for software designers. Such systems are hard to model and lead to numerous algorithmic problems.

In this thesis, we focus on designing memory-aware algorithms and new scheduling techniques suited to modern storage architectures, for various applications. In the different chapters, we take data communications and memory usage into account, while designing new algorithms for HPC systems. We also exploit the heterogeneity of these systems to improve the performance of generic matrix computations. Different approaches can be taken, and we classify them into three main parts in this thesis. In Chapters 1 and 2, we design new linear algebra solvers that squeeze the most out of future HPC architectures. We also improve existing schedulers to better deal with the heterogeneity of modern HPC architectures. In Chapters 3 and 4, we opt for a more general technique by adding memory-awareness

to generic scheduling techniques. Indeed, the ordering of the computations and the mapping of the tasks onto the computational resources deeply impact the memory usage while computing a workflow application. Finally, in Chapter 5, we target another scientific application that deal with large data files. In this context, the memory is extremely limited and it can be necessary to recompute some tasks whose output has not be stored because of memory constraints. Throughout the thesis, we delve into methods to improve memory management, using numerous techniques and targeting various scientific applications. The main contributions of the different chapters are sketched in the following paragraphs.

Chapter 1: Mixing LU and QR factorization algorithms to design high-performance dense linear algebra solvers

In this first chapter, we exhibit a new numerical algorithm to solve dense linear systems. Finding the solution to a linear system and linear algebra problems in general, are a common issue in the HPC community. The HPL Linpack benchmark takes algebra routines performance to compare the computational power of supercomputers. To solve a linear system, one usually factors the matrix using a LU or a QR factorization. These two methods have both advantages and drawbacks. The generic LU factorization is highly parallel and involves only broadcasts of information that are efficient on modern distributed architectures. At the same time, the accuracy of the computed solution may be subject to deterioration depending upon the condition number of the initial matrix. The QR factorization is twice as costly as the LU factorization and is slightly less parallel. It involves pipelining communications when factorizing the panel. However, the computed solution is more accurate than when using a LU factorization. Here, we design an hybrid LU-QR algorithm that alternates LU and QR factorization steps to take advantage of the best characteristics of each method. The choice of which routine to run at each step is conducted by a stability criteria that will detect, before the actual computation of the step, whether performing an LU factorization step is safe from a stability point of view, or whether we should rather perform a QR step. We propose different stability criteria based on the numerical properties of the panel, and implement them in the PaRSEC runtime. The PaRSEC runtime is well-suited to exploit the parallelism in large clusters of multiprocessors machines. It deals with task scheduling and communications, allowing us to concentrate only on algorithm design. We run an extensive set of experiments to analyse both performance and stability results compared to state-of-the-art algorithms. The experiments are run on a comprehensive set of random matrices, and ill-conditioned matrices coming from the Higham's Matrix Computation Toolbox. The work presented in this chapter has been done in collaboration with Jack Dongarra and Mathieu Faverge from the University of Tennessee, Knoxville, USA, and Julien Langou and Bradley R. Lowery from the University of Colorado, Denver, USA.

Chapter 2: Bridging the gap between experimental performance and theoretical bounds for the Cholesky factorization on heterogeneous platforms

While the previous chapter was dealing with clusters of multiprocessor machines, in the second chapter of this thesis, we target shared-memory multiprocessors accelerated with GPUs. In this context, specific schedulers has to be designed to exploit the heterogeneity of the platform. The StarPU runtime takes care of mapping tasks efficiently onto the computational resources, using well-known generic dynamic scheduling policies from the literature, while optimizing data transfers using prefetching and overlapping, in particular. Although those purely dynamic schedulers are efficient in practice, they are not guaranteed to perform optimally, mostly due to their local vision of the workflow at any moment. In this chapter, we explore how adding static rules to the purely dynamic schedulers implemented in StarPU can improve their performance. We focus on another classical dense linear algebra routine, namely the

Cholesky factorization. We introduce theoretical bounds on the best achievable performance on a fully heterogeneous platform made of CPUs and GPUs. We investigate how adding general information of the problem can help bridging the gap between these bounds and the actual performance. This work has been conducted in collaboration with Emmanuel Agullo, Olivier Beaumont, Lionel Eyraud-Dubois, Suraj Kumar, and Samuel Thibault from INRIA Bordeaux, Talence, France.

Chapter 3: Memory-aware list scheduling for hybrid platforms

In this chapter, we study the complexity of scheduling task graph workflows whose tasks require large I/O files. We target a heterogeneous architecture with two resource types, each with a different memory, such as a multicore node equipped with a dedicated accelerator (FPGA or GPU). Each task has an unrelated processing time for either resource and can be processed only if all its input and output files can be stored in the corresponding memory. The amount of used memory of each type at a given execution step strongly depends upon the ordering in which the tasks are executed, and upon when communications between both memories are scheduled. The objective is to determine an efficient schedule taking into account the memory constraints for each type. First, we establish the complexity of minimizing memory usage when scheduling tree-shaped workflows whose tasks are already mapped on the resources. Even for this simplified version of the problem, we provide NP-completness proofs and inapproximability results. Then, we design memory-aware heuristics that provide schedules achieving good completion times, without violating the memory constraints. These heuristics are evaluated on a comprehensive set of graphs, including random graphs as well as graphs arising in the context of matrix factorizations.

Chapter 4: Assessing the cost of redistribution followed by a computational kernel

In linear algebra applications, when dealing with a distributed computational platform, the data layout usually respects the *owner-computes* distribution. It means that each block of data is updated by the processor that hosts it. This strategy requires that data blocks are distributed onto the computational resources in a workload-balanced manner. If the initial data distribution is not suitable for the computational kernel, a data redistribution may be required before starting the computation. Usually, for a given application, a specific data distribution is known to be optimal, or close to optimal. Most of the time, this data distribution is not unique. In particular, if all the processors are identical, a processor permutation will not change the performance of a given data distribution. In this chapter, we find the optimal data partition among all the optimal (or close to optimal) ones, that minimizes the cost of redistribution, given an initial data distribution. We also show the NP-hardness of the problem to find the optimal data partition and processor permutation that minimize the cost of redistribution followed by a simple computational kernel. Our new redistribution algorithms are experimentally validated using the PaRSEC environment on dense linear algebra routines. The work presented in this chapter has been done with the help of Thomas Hérault from the University of Tennessee, Knoxville, USA.

Chapter 5: Optimal multistage algorithms for adjoint computation

In this last chapter, we target another scientific application, namely adjoint computation. In this context, the tasks graphs to schedule deal with large I/O files and provide no parallelism across tasks. To cope with the memory constraints, some output files have to be deleted and recomputed later when needed. The objective of the scheduling algorithm is to determine which output files to store, in order to minimize the total processing time of the task graph. As in chapter 3, we deal with a bi-memory model. But this time, memories are different. One of them has no size constraint, but it takes a significant time to write

and read data from it, while the other one is limited in volume but provides a free writing/reading cost. We design the first optimal algorithm in the literature to solve this problem, and compare its performance with the state-of-the-art heuristics. This work has been conducted in collaboration with Guillaume Aupy, another PhD student of my research team, and Paul Hovland from the Argonne National Laboratory, Illinois, USA.

Part I

Linear Algebra

Chapter 1

Mixing LU and QR factorization algorithms to design high-performance dense linear algebra solvers

As said in the introduction of this thesis, optimizing the linear algebra libraries is a main concern for the HPC community. Many scientific applications rely on the numerical algorithms provided by these libraries. It is thus important to design new numerical algorithm that exploit the complex hierarchical architecture of the next generation of supercomputers. One of the main application of the linear algebra libraries is to solve a dense linear systems of the form $Ax = b$. It is indeed the benchmark use to access the computational power of modern supercomputers. To solve a dense linear system $Ax = b$, one usually use either an LU factorization or a QR factorization of the matrix A. LU elimination steps can be very efficiently parallelized, and are twice as cheap in terms of floating-point operations, as QR steps. However, LU steps are not necessarily stable, while QR steps are always stable. In this first chapter, we introduce hybrid LU-QR algorithms for solving dense linear systems. Throughout a matrix factorization, these algorithms dynamically alternate LU with local pivoting and QR elimination steps based upon some robustness criterion. The hybrid algorithms execute a QR step when a robustness criterion detects some risk for instability, and they execute an LU step otherwise. The choice between LU and QR steps must have a small computational overhead and must provide a satisfactory level of stability with as few QR steps as possible. In this chapter, we introduce several robustness criteria and we establish upper bounds on the growth factor of the norm of the updated matrix incurred by each of these criteria. In addition, we describe the implementation of the hybrid algorithms through an extension of the PaRSEC software to allow for dynamic choices during execution. Finally, we analyze both stability and performance results compared to state-of-the-art linear solvers on parallel distributed multicore platforms. A comprehensive set of experiments shows that hybrid LU-QR algorithms provide a continuous range of trade-offs between stability and performances.

1.1 Introduction

Consider a dense linear system $Ax = b$ to solve, where A is a square tiled-matrix, with n tiles per row and column. Each tile is a block of n_b-by-n_b elements, so that the actual size of A is $N = n \times n_b$. Here, n_b is a parameter tuned to squeeze the most out of arithmetic units and memory hierarchy. To solve the linear system $Ax = b$, with A a general matrix, one usually applies a series of transformations, pre-multiplying A by several elementary matrices. There are two main approaches: *LU factorization*, where one uses lower unit triangular matrices, and *QR factorization*, where one uses orthogonal Householder

matrices. To the best of our knowledge, this chapter is the first study to propose a mix of both approaches during a single factorization. The LU factorization update is based upon matrix-matrix multiplications, a kernel that can be very efficiently parallelized, and whose library implementations typically achieve close to peak CPU performance. Unfortunately, the efficiency of LU factorization is hindered by the need to perform partial pivoting at each step of the algorithm, to ensure numerical stability. On the contrary, the QR factorization is always stable, but requires twice as many floating-point operations, and a more complicated update step that is not as parallel as a matrix-matrix product. Tiled QR algorithms [27, 28, 90] greatly improve the parallelism of the update step since they involve no pivoting but are based upon more complicated kernels whose library implementations requires twice as many operations as LU.

The main objective of this chapter is to explore the design of *hybrid* algorithms that would combine the low cost and high CPU efficiency of the LU factorization, while retaining the numerical stability of the QR approach. In a nutshell, the idea is the following: at each step of the elimination, we perform a *robustness* test to know if the diagonal tile can be stably used to eliminate the tiles beneath it using an LU step. If the test succeeds, then go for an elimination step based upon LU kernels, without any further pivoting involving sub-diagonal tiles in the panel. Technically, this is very similar to a step during a block LU factorization [37]. Otherwise, if the test fails, then go for a step with QR kernels. On the one extreme, if all tests succeed throughout the algorithm, we implement an LU factorization without pivoting. On the other extreme, if all tests fail, we implement a QR factorization. On the average, some of the tests will fail, some will succeed. If the fraction of the tests that fail remains small enough, we will reach a CPU performance close to that of LU without pivoting. Of course the challenge is to design a test that is accurate enough (and not too costly) so that LU kernels are applied only when it is numerically safe to do so.

Implementing such a hybrid algorithm on a state-of-the-art distributed-memory platform, whose nodes are themselves equipped with multiple cores, is a programming challenge. Within a node, the architecture is a shared-memory machine, running many parallel threads on the cores. But the global architecture is a distributed-memory machine, and requires MPI communication primitives for inter-node communications. A slight change in the algorithm, or in the matrix layout across the nodes, might call for a time-consuming and error-prone process of code adaptation. For each version, one must identify, and adequately implement, inter-node versus intra-node kernels. This dramatically complicates the task of the programmers if they rely on a manual approach. We solve this problem by relying on the PaRSEC software [23, 22, 20], so that we can concentrate on the algorithm and forget about MPI and threads. Once we have specified the algorithm at a task level, the PaRSEC software will recognize which operations are local to a node (and hence correspond to shared-memory accesses), and which are not (and hence must be converted into MPI communications). Previous experiments show that this approach is very powerful, and that the use of a higher-level framework does not prevent our algorithms from achieving the same performance as state-of-the-art library releases [42].

However, implementing a hybrid algorithm requires the programmer to implement a *dynamic* task graph of the computation. Indeed, the task graph of the hybrid factorization algorithm is no longer known statically (contrarily to a standard LU or QR factorization). At each step of the elimination, we use either LU-based or QR-based tasks, but not both. This requires the algorithm to dynamically fork upon the outcome of the robustness test, in order to apply the selected kernels. The solution is to prepare a graph that includes both types of tasks, namely LU and QR kernels, to select the adequate tasks on the fly, and to discard the useless ones. We have to join both potential execution flows at the end of each step, symmetrically. Most of this mechanism is transparent to the user. We discuss this extension of PaRSEC in more detail in Section 1.4.

The major contributions of this chapter are the following:

- The introduction of new LU-QR hybrid algorithms;

- The design of several robustness criteria, with bounds on the induced growth factor;
- The extension of PaRSEC to deal with dynamic task graphs;
- A comprehensive experimental evaluation of the best trade-offs between performance and numerical stability.

The rest of the chapter is organized as follows. First we explain the main principles of LU-QR hybrid algorithms in Section 1.2. Then we describe robustness criteria in Section 1.3. Next we detail the implementation within the PaRSEC framework in Section 1.4. We report experimental results in Section 1.5. We discuss related work in Section 1.6. Finally, we provide concluding remarks and future directions in Section 1.7.

1.2 Hybrid LU-QR algorithms

In this section, we describe hybrid algorithms to solve a dense linear system $Ax = b$, where $A = (A_{i,j})_{(i,j)\in[\![1..n]\!]^2}$ is a square tiled-matrix, with n tiles per row or column. Each tile is a block of n_b-by-n_b elements, so that A is of order $N = n \times n_b$.

The common goal of a classical one-sided factorization (LU or QR) is to *triangularize* the matrix A through a succession of elementary transformations. Consider the first step of such an algorithm. We partition A by block such that $A = \begin{pmatrix} A_{11} & C \\ B & D \end{pmatrix}$. In terms of tile, A_{11} is 1-by-1, B is $(n-1)$-by-1, C is 1-by-$(n-1)$, and D is $(n-1)$-by-$(n-1)$. The first block-column $\begin{pmatrix} A_{11} \\ B \end{pmatrix}$ is the *panel* of the current step.

Traditional algorithms (LU or QR) perform the same type of transformation at each step. The key observation of this chapter is that any type of transformation (LU or QR) can be used for a given step independently of what was used for the previous steps. The common framework of a step is the following:

$$\begin{pmatrix} A_{11} & C \\ B & D \end{pmatrix} \Leftrightarrow \begin{pmatrix} \textit{factor} & \textit{apply} \\ \textit{eliminate} & \textit{update} \end{pmatrix} \Leftrightarrow \begin{pmatrix} U_{11} & C' \\ 0 & D' \end{pmatrix}. \tag{1.1}$$

First, A_{11} is *factored* and transformed in the upper triangular matrix U_{11}. Then, the transformation of the factorization of A_{11} is *applied* to C. Then A_{11} is used to *eliminate* B. Finally D is accordingly *updated*. Recursively factoring D' with the same framework will complete the factorization to an upper triangular matrix.

For each step, we have a choice for an LU step or a QR step. The operation count for each kernel is given in Table 1.1.

	LU step, var A1		QR step	
factor A	$2/3$	GETRF	$4/3$	GEQRT
eliminate B	$(n-1)$	TRSM	$2(n-1)$	TSQRT
apply C	$(n-1)$	TRSM	$2(n-1)$	TSMQR
update D	$2(n-1)^2$	GEMM	$4(n-1)^2$	UNMQR

Table 1.1: Computational cost of each kernel. The unit is n_b^3 floating-point operations.

Generally speaking, QR transformations are twice as costly as their LU counterparts. The bulk of the computations take place in the update of the trailing matrix D. This obviously favors LU *update* kernels. In addition, the LU *update* kernels are fully parallel and can be applied independently on the

Algorithm 1: Hybrid LU-QR algorithm

for $k = 1$ *to* n **do**
Factor: Compute a factorization of the diagonal tile: either with LU partial pivoting or QR;
Check: Compute some robustness criteria (see Section 1.3) involving only tiles $A_{i,k}$, where $k \leq i \leq n$, in the elimination panel;
Apply, Eliminate, Update:
if *the criterion succeeds* **then**
Perform an LU step;
else
Perform a QR step;

Algorithm 2: Step k of an LU step - var (A1)

Factor: $A_{k,k} \leftarrow GETRF(A_{k,k})$;
for $i = k + 1$ *to* n **do**
Eliminate: $A_{i,k} \leftarrow TRSM(A_{k,k}, A_{i,k})$;
for $j = k + 1$ *to* n **do**
Apply: $A_{k,j} \leftarrow SWPTRSM(A_{k,k}, A_{k,j})$;
for $i = k + 1$ *to* n **do**
for $j = k + 1$ *to* n **do**
Update: $A_{i,j} \leftarrow GEMM(A_{i,k}, A_{k,j}, A_{i,j})$;

$(n-1)^2$ trailing tiles. Unfortunately, LU updates (using GEMM) are stable only when $\|A_{11}^{-1}\|^{-1}$ is larger than $\|B\|$ (see Section 1.3). If this is not the case, we have to resort to QR kernels. Not only these are twice as costly, but they also suffer from enforcing more dependencies: all columns can still be processed (*apply* and *update* kernels) independently, but inside a column, the kernels must be applied in sequence.

The hybrid *LU-QR Algorithm* uses the standard 2D block-cyclic distribution of tiles along a virtual p-by-q cluster grid. The 2D block-cyclic distribution nicely balances the load across resources for both LU and QR steps. Thus at step k of the factorization, the panel is split into p *domains* of approximately $\frac{n-k+1}{p}$ tile rows. Domains will be associated with physical memory regions, typically a domain per node in a distributed memory platform. Thus an important design goal is to minimize the number of communications across domains, because these correspond to nonlocal communications between nodes. At each step k of the factorization, the domain of the node owning the diagonal tile $A_{k,k}$ is called the *diagonal domain*.

The hybrid *LU-QR Algorithm* applies LU kernels when it is numerically safe to do so, and QR kernels otherwise. Coming back to the first elimination step, the sequence of operations is described in Algorithm 1.

1.2.1 LU step

We assume that the criterion validates an LU step (see Section 1.3). We describe the variant (A1) of an LU step given in Algorithm 2.

The kernels for the LU step are the following:

Algorithm 3: Step k of the HQR factorization

for $i = k+1$ *to* n **do**
  $elim(i, eliminator(i,k), k)$;

- *Factor:* $A_{k,k} \leftarrow GETRF(A_{k,k})$ is an LU factorization with partial pivoting: $P_{k,k}A_{k,k} = L_{k,k}U_{k,k}$, the output matrices $L_{k,k}$ and $U_{k,k}$ are stored in place of the input $A_{k,k}$.
- *Eliminate:* $A_{i,k} \leftarrow TRSM(A_{k,k}, A_{i,k})$ solves in-place, the upper triangular system such that $A_{i,k} \leftarrow A_{i,k}U_{k,k}^{-1}$ where $U_{k,k}$ is stored in the upper part of $A_{k,k}$.
- *Apply:* $A_{k,j} \leftarrow SWPTRSM(A_{k,k}, A_{i,k})$ solves the unit lower triangular system such that $A_{k,j} \leftarrow L_{k,k}^{-1}P_{k,k}A_{k,j}$ where $L_{k,k}$ is stored in the (strictly) lower part of $A_{k,k}$.
- *Update:* $A_{i,j} \leftarrow GEMM(A_{i,k}, A_{k,j}, A_{i,j})$ is a general matrix-matrix multiplication $A_{i,j} \leftarrow A_{i,j} - A_{i,k}A_{k,j}$.

In terms of parallelism, the factorization of the diagonal tile is followed by the $TRSM$ kernels that can be processed in parallel, then every $GEMM$ kernel can be processed concurrently. These highly parallelizable updates constitute one of the two main advantages of the LU step over the QR step. The second main advantage is halving the number of floating-point operations.

During the *factor* step, one variant is to factor the whole diagonal domain instead of only factoring the diagonal tile. Considering Algorithm 2, the difference lies in the first line: rather than calling $GETRF(A_{k,k})$, thereby searching for pivots only within the diagonal tile $A_{k,k}$, we implemented a variant where we extend the search for pivots across the *diagonal domain* (the *Apply* step is modified accordingly). Working on the diagonal domain instead of the diagonal tile increases the smallest singular value of the factored region and therefore increases the likelihood of an LU step. Since all tiles in the diagonal domain are local to a single node, extending the search to the diagonal domain is done without any inter-domain communication. The stability analysis of Section 1.3 applies to both scenarios, the one where $A_{k,k}$ is factored in isolation, and the one where it is factored with the help of the diagonal domain. In the experimental section, we will use the variant which factors the diagonal domain.

1.2.2 QR step

If the decision to process a QR step is taken by the criterion, the LU decomposition of the diagonal domain is dropped, and the factorization of the panel starts over. This step of the factorization is then processed using orthogonal transformations. Every tile below the diagonal (matrix B in Equation (1.1)) is zeroed out using a triangular tile, or eliminator tile. In a QR step, the diagonal tile is factored (with a GEQRF kernel) and used to eliminate all the other tiles of the panel (with a TSQRT kernel) The trailing submatrix is updated, respectively, with UNMQR and TSMQR kernels. To further increase the degree of parallelism of the algorithm, it is possible to use several eliminator tiles inside a panel, typically one (or more) per domain. The only condition is that concurrent elimination operations must involve disjoint tile pairs (the unique eliminator of tile $A_{i,k}$ will be referred to as $A_{eliminator(i,k),k}$). Of course, in the end, there must remain only one non-zero tile on the panel diagonal, so that all eliminators except the diagonal tile must be eliminated later on (with a TTQRT kernel on the panel and TTMQR updates on the trailing submatrix), using a reduction tree of arbitrary shape. This reduction tree will involve inter-domain communications. In our hybrid LU-QR algorithm, the QR step is processed following an instance of the generic hierarchical QR factorization HQR [42] described in Algorithms 3 and 4.

Each elimination $elim(i, eliminator(i,k), k)$ consists of two sub-steps: first in column k, tile (i,k) is zeroed out (or killed) by tile $(eliminator(i,k), k)$; and in each following column $j > k$, tiles (i,j)

Algorithm 4: Elimination $elim(i, eliminator(i,k), k)$

(a) With TS kernels

$A_{eliminator(i,k),k} \leftarrow GEQRT(A_{eliminator(i,k),k})$;
$A_{i,k}, A_{eliminator(i,k),k} \leftarrow TSQRT(A_{i,k}, A_{eliminator(i,k),k})$;
for $j = k+1$ *to* $n-1$ **do**
  $A_{eliminator(i,k),j} \leftarrow UNMQR(A_{eliminator(i,k),j}, A_{eliminator(i,k),k}$;
  $A_{i,j}, A_{eliminator(i,k),j} \leftarrow TSMQR(A_{i,j}, A_{eliminator(i,k),j}, A_{i,k})$;

(b) With TT kernels

$A_{eliminator(i,k),k} \leftarrow GEQRT(A_{eliminator(i,k),k})$;
$A_{i,k} \leftarrow GEQRT(A_{i,k})$;
for $j = k+1$ *to* $n-1$ **do**
  $A_{eliminator(i,k),j} \leftarrow UNMQR(A_{eliminator(i,k),j}, A_{eliminator(i,k),k}$;
  $A_{i,j} \leftarrow UNMQR(A_{i,j}, A_{i,k}$;
$A_{i,k}, A_{eliminator(i,k),k} \leftarrow TTQRT(A_{i,k}, A_{eliminator(i,k),k})$;
for $j = k+1$ *to* $n-1$ **do**
  $A_{i,j}, A_{eliminator(i,k),j} \leftarrow TTMQR(A_{i,j}, A_{eliminator(i,k),j}, A_{i,k})$;

and $(eliminator(i,k), j)$ are updated; all these updates are independent and can be triggered as soon as the elimination is completed. The algorithm is entirely characterized by its elimination list, which is the ordered list of all the eliminations $elim(i, eliminator(i,k), k)$ that are executed. The orthogonal transformation $elim(i, eliminator(i,k), k)$ uses either a TTQRT kernel or a TSQRT kernel depending upon whether the tile to eliminate is either triangular or square. In our hybrid *LU-QR Algorithm*, any combination of reduction trees of the HQR algorithm described in [42] is available. It is then possible to use an intra-domain reduction tree to locally eliminate many tiles without inter-domain communication. A unique triangular tile is left on each node and then the reductions across domains are performed following a second level of reduction tree.

1.2.3 LU step variants

In the following, we describe several other variants of the LU step.

1.2.3.1 Variant (A2)

It consists of first performing a QR factorization of the diagonal tile and proceeds pretty much as in (A1) thereafter.

- *Factor:* $A_{k,k} \leftarrow GEQRF(A_{k,k})$ is a QR factorization $A_{k,k} = Q_{k,k}U_{k,k}$, where $Q_{k,k}$ is never constructed explicitly and we instead store the Householder reflector $V_{k,k}$. The output matrices $V_{k,k}$ and $U_{k,k}$ are stored in place of the input $A_{k,k}$.
- *Eliminate:* $A_{i,k} \leftarrow TRSM(A_{k,k}, A_{i,k})$ solves in-place the upper triangular system such that $A_{i,k} \leftarrow A_{i,k}U_{k,k}^{-1}$ where $U_{k,k}$ is stored in the upper part of $A_{k,k}$.
- *Apply:* $A_{k,j} \leftarrow ORMQR(A_{k,k}, A_{i,k})$ performs $A_{k,j} \leftarrow Q_{k,k}^T A_{k,j}$ where $Q_{k,k}^T$ is applied using $V_{k,k}$ stored in the (strictly) lower part of $A_{k,k}$.
- *Update:* $A_{i,j} \leftarrow GEMM(A_{i,k}, A_{k,j}, A_{i,j})$ is a general matrix-matrix multiplication $A_{i,j} \leftarrow A_{i,j} - A_{i,k}A_{k,j}$.

The *Eliminate* and *Update* steps are the exact same as in (A1). The (A2) variant has the same data dependencies as (A1) and therefore the same level of parallelism. A benefit of (A2) over (A1) is that if the criterion test decides that the step is a QR step, then the factorization of $A_{k,k}$ is not discarded but rather used to continue the QR step. A drawback of (A2) is that the *Factor* and *Apply* steps are twice as expensive as the ones in (A1).

1.2.3.2 Variants (B1) and (B2)

Another option is to use the so-called *block LU factorization* [37]. The result of this formulation is a factorization where the U factor is block upper triangular (as opposed to upper triangular), and the diagonal tiles of the L factor are identity tiles. The *Factor* step can either be done using an LU factorization (variant (B1)) or a QR factorization (variant (B2)). The *Eliminate* step is $A_{i,k} \leftarrow A_{i,k}A_{k,k}^{-1}$. There is no *Apply* step. And the *Update* step is $A_{i,j} \leftarrow A_{i,j} - A_{i,k}A_{k,j}$.

The fact that row k is not updated provides two benefits: (i) $A_{k,k}$ does not need to be broadcast to these tiles, simplifying the communication pattern; (ii) The stability of the LU step can be determined by considering only the growth factor in the Schur complement of $A_{k,k}$. One drawback of (B1) and (B2) is that the final matrix is not upper triangular but only block upper triangular. This complicates the use of these methods to solve a linear system of equations. The stability of (B1) and (B2) has been analyzed in [37].

We note that (A2) and (B2) use a QR factorization during the *Factor* step. Yet, we still call this an LU step. This is because all four LU variants mentioned use the Schur complement to update the trailing sub-matrix. The mathematical operation is: $A_{i,j} \leftarrow A_{i,j} - A_{i,k}A_{k,k}^{-1}A_{k,j}$,. In practice, the *Update* step for all four variants looks like $A_{i,j} \leftarrow A_{i,j} - A_{i,k}A_{k,j}$, since $A_{k,k}^{-1}$ is somehow applied to $A_{i,k}$ and $A_{k,j}$ during the preliminary *update* and *eliminate* steps. The Schur update dominates the cost of an LU factorization and therefore all variants are more efficient than a QR step. Also, we have the same level of parallelism for the update step: embarrassingly parallel. In terms of stability, all variants would follow closely the analysis of Section 1.5.4. We do not consider further variants (A2), (B1), and (B2) in this chapter, since they are all very similar, and only study Algorithm 2, (A1).

1.2.4 Comments

1.2.4.1 Solving systems of linear equations

To solve systems of linear equations, we augment A with the right-hand side b to get $\tilde{A} = (A, b)$ and apply all transformations to $\tilde{A}$. Then an N-by-N triangular solve is needed. This is the approach we used in our experiments. We note that, at the end of the factorization, all needed information about the transformations is stored in place of A, so, alternatively, one can apply the transformations on b during a second pass.

1.2.4.2 No restriction on N

In practice, N does not have to be a multiple of n_b. We keep this restriction for the sake of simplicity. The algorithm can accommodate any N and n_b with some clean-up codes, which we have written.

1.2.4.3 Relation with threshold pivoting

The *LU-QR Algorithm* can be viewed as a tile version of standard threshold pivoting. Standard threshold pivoting works on 1-by-1 tiles (scalars, matrix elements). It checks if the diagonal element passes a

certain threshold. If the diagonal element passes the threshold then elimination is done without pivoting using this diagonal element as the pivot. If the diagonal element does not pass the threshold, then standard pivoting is done and elimination is done with largest element in absolute value in the column. Our Hybrid LU algorithm can be seen as a variant with tiles. Our algorithm checks if the diagonal tile passes a certain threshold. If the diagonal tile passes the threshold then elimination is done without pivoting using this diagonal tile as the pivot. If the diagonal tile does not pass the threshold, then a stable elimination is performed. The fact that our algorithm works on tiles as opposed to scalars leads to two major differences. (1) New criteria for declaring a diagonal tile as being safe had to develop. (2) In the event when a diagonal tile is declared not safe, we need to resort to a tile QR step.

1.3 Robustness criteria

The decision to process an LU or a QR step is done dynamically during the factorization, and constitutes the heart of the algorithm. Indeed, the decision criteria has to be able to detect a potentially "large" stability deterioration (according to a threshold) due to an LU step before its actual computation, in order to preventively switch to a QR step. As explained in Section 1.2, in our hybrid LU-QR algorithm, the diagonal tile is factored using an LU decomposition with partial pivoting. At the same time, some data (like the norm of non local tiles belonging to other domains) are collected and exchanged (using a Bruck's all-reduce algorithm [26]) between all nodes hosting at least one tile of the panel. Based upon this information, all nodes make the decision to continue the LU factorization step or to drop the LU decomposition of the diagonal tile and process a full QR factorization step. The decision is broadcast to the other nodes not involved in the panel factorization within the next data communication. The decision process cost will depend on the choice of the criterion and must not imply a large computational overhead compared to the factorization cost. A good criterion will detect only the "worst" steps and will provide a good stability result with as few QR steps as possible. In this section, we present three criteria, going from the most elaborate (but also most costly) to the simplest ones.

The stability of a step is determined by the growth of the norm of the updated matrix. If a criterion determines the potential for an unacceptable growth due to an LU step, then a QR step is used. A QR step is stable as there is no growth in the norm (2-norm) since it is a unitary transformation. Each criterion depends on a threshold α that allows us to tighten or loosen the stability requirement, and thus influence the amount of LU steps that we can afford during the factorization. In Section 1.5.4, we experiment with different choices of α for each criterion.

1.3.1 Max criterion

LU factorization with partial pivoting chooses the largest element of a column as the pivot element. Partial pivoting is accepted as being numerically stable. However, pivoting across nodes is expensive. To avoid this pivoting, we generalize the criterion to tiles and determine if the diagonal tile is an acceptable pivot. A step is an LU step if

$$\alpha \times \|(A_{k,k}^{(k)})^{-1}\|_1^{-1} \geq \max_{i>k} \|A_{i,k}^{(k)}\|_1. \tag{1.2}$$

For the analysis we do not make an assumption as to how the diagonal tile is factored. We only assume that the diagonal tile is factored in a stable way (LU with partial pivoting or QR are acceptable). Note that, for the variant using pivoting in the diagonal domain (see Section 1.2.1), which is the variant we experiment with in Section 1.5, $A_{k,k}^{(k)}$ represents the diagonal tile after pivoting among tiles in the diagonal domain.

To assess the growth of the norm of the updated matrix, consider the update of the trailing sub-matrix. For all $i, j > k$ we have:

$$\begin{aligned}
\|A_{i,j}^{(k+1)}\|_1 &= \|A_{i,j}^{(k)} - A_{i,k}^{(k)}(A_{k,k}^{(k)})^{-1}A_{k,j}^{(k)}\|_1 \\
&\leq \|A_{i,j}^{(k)}\|_1 + \|A_{i,k}^{(k)}\|_1\|(A_{k,k}^{(k)})^{-1}\|_1\|A_{k,j}^{(k)}\|_1 \\
&\leq \|A_{i,j}^{(k)}\|_1 + \alpha\|A_{k,j}^{(k)}\|_1 \\
&\leq (1+\alpha)\max\left(\|A_{i,j}^{(k)}\|_1, \|A_{k,j}^{(k)}\|_1\right) \\
&\leq (1+\alpha)\max_{i \geq k}\left(\|A_{i,j}^{(k)}\|_1\right).
\end{aligned}$$

The growth of any tile in the trailing sub-matrix is bounded by $1 + \alpha$ times the largest tile in the same column. If every step satisfies (1.2), then we have the following bound:

$$\frac{\max_{i,j,k}\|A_{i,j}^{(k)}\|_1}{\max_{i,j}\|A_{i,j}\|_1} \leq (1+\alpha)^{n-1}.$$

The expression above is a growth factor on the norm of the tiles. For $\alpha = 1$, the growth factor of 2^{n-1} is an analogous result to an LU factorization with partial pivoting (scalar case) [67]. Finally, note that we can obtain this bound by generalizing the standard example for partial pivoting. The following matrix will match the bound above:

$$A = \begin{pmatrix} \alpha^{-1} & 0 & 0 & 1 \\ -1 & \alpha^{-1} & 0 & 1 \\ -1 & -1 & \alpha^{-1} & 1 \\ -1 & -1 & -1 & 1 \end{pmatrix}.$$

1.3.2 Sum criterion

A stricter criterion is to compare the diagonal tile to the sum of the off-diagonal tiles:

$$\alpha \times \|(A_{k,k}^{(k)})^{-1}\|_1^{-1} \geq \sum_{i>k}\|A_{i,k}^{(k)}\|_1. \tag{1.3}$$

Again, for the analysis, we only assume $A_{k,k}^{-1}$ factored in a stable way. For $\alpha \geq 1$, this criterion (and the Max criterion) is satisfied at every step if A is block diagonally dominant [67]. That is, a general matrix $A \in \mathbb{R}^{n\times n}$ is block diagonally dominant by columns with respect to a given partitioning $A = (A_{i,j})$ and a given norm $\|\cdot\|$ if:

$$\forall j \in [\![1, n]\!], \|A_{j,j}^{-1}\|^{-1} \geq \sum_{i\neq j}\|A_{i,j}\|.$$

Again we need to evaluate the growth of the norm of the updated trailing sub-matrix. For all $i, j > k$, we have

$$\begin{aligned}\sum_{i>k} \|A_{i,j}^{(k+1)}\|_1 &= \sum_{i>k} \|A_{i,j}^{(k)} - A_{i,k}^{(k)}(A_{k,k}^{(k)})^{-1}A_{k,j}^{(k)}\|_1 \\ &\leq \sum_{i>k} \|A_{i,j}^{(k)}\|_1 \\ &\quad + \|A_{k,j}^{(k)}\|_1 \|(A_{k,k}^{(k)})^{-1}\|_1 \sum_{i>k} \|A_{i,k}^{(k)}\|_1 \\ &\leq \sum_{i>k} \|A_{i,j}^{(k)}\|_1 + \alpha \|A_{k,j}^{(k)}\|_1.\end{aligned}$$

Hence, the growth of the updated matrix can be bounded in terms of an entire column rather than just an individual tile. The only growth in the sum is due to the norm of a single tile. For $\alpha = 1$, the inequality becomes

$$\sum_{i>k} \|A_{i,j}^{(k+1)}\|_1 \leq \sum_{i\geq k} \|A_{i,j}^{(k)}\|_1.$$

If every step of the algorithm satisfies (1.3) (with $\alpha = 1$), then by induction we have:

$$\sum_{i>k} \|A_{i,j}^{(k+1)}\|_1 \leq \sum_{i\geq 1} \|A_{i,j}\|_1,$$

for all i, j, k. This leads to the following bound:

$$\frac{\max_{i,j,k} \|A_{i,j}^{(k)}\|_1}{\max_{i,j} \|A_{i,j}\|_1} \leq n.$$

From this we see that the criteria eliminates the potential for exponential growth due to the LU steps. Note that for a diagonally dominant matrix, the bound on the growth factor can be reduced to 2 [67].

1.3.3 MUMPS criterion

In LU decomposition with partial pivoting, the largest element of the column is used as the pivot. This method is stable experimentally, but the seeking of the maximum and the pivoting requires a lot of communications in distributed memory. Thus in an LU step of the *LU-QR Algorithm*, the LU decomposition with partial pivoting is limited to the local tiles of the panel (i.e., to the diagonal domain). The idea behind the MUMPS criterion is to estimate the quality of the pivot found locally compared to the rest of the column. The MUMPS criterion is one of the strategies available in MUMPS although it is for symmetric indefinite matrices (LDL^T) [45], and Amestoy et al. [12] provided us with their scalar criterion for the LU case.

At step k of the *LU-QR Algorithm*, let $L^{(k)}U^{(k)}$ be the LU decomposition of the diagonal domain and $A_{i,j}^{(k)}$ be the value of the tile $A_{i,j}$ at the beginning of step k. Let $local_max_k(j)$ be the largest element of the column j of the panel in the diagonal domain, $away_max_k(j)$ be the largest element of the column j of the panel off the diagonal domain, and $pivot_k$ be the list of pivots used in the LU

decomposition of the diagonal domain:

$$local_max_k(j) = \max_{\substack{\text{tiles } A_{i,k} \text{ on the} \\ \text{diagonal domain}}} \max_l |(A_{i,k})_{l,j}|,$$

$$away_max_k(j) = \max_{\substack{\text{tiles } A_{i,k} \text{ off the} \\ \text{diagonal domain}}} \max_l |(A_{i,k})_{l,j}|,$$

$$pivot_k(j) = |U^{(k)}_{j,j}|.$$

$pivot_k(j)$ represents the largest local element of the column j at step j of the LU decomposition with partial pivoting on the diagonal domain. Thus, we can express the growth factor of the largest local element of the column j at step j as: $growth_factor_k(j) = pivot_k(j)/local_max_k(j)$. The idea behind the MUMPS criterion is to estimate if the largest element outside the local domain would have grown the same way. Thus, we can define a vector $estimate_max_k$ initialized to $away_max_k$ and updated for each step i of the LU decomposition with partial pivoting like $estimate_max_k(j) \leftarrow estimate_max_k(j) \times growth_factor_k(i)$. We consider that the LU decomposition with partial pivoting of the diagonal domain can be used to eliminate the rest of the panel if and only if all pivots are larger than the estimated maximum of the column outside the diagonal domain times a threshold α. Thus, the MUMPS criterion (as we implemented it) decides that step k of the *LU-QR Algorithm* will be an LU step if and only if:

$$\forall j, \alpha \times pivot_k(j) \geq estimate_max_k(j). \tag{1.4}$$

1.3.4 Extending the MUMPS criterion to tiles

In this section, we extend the MUMPS criterion [12, 45] to tiles. We did not implement this extension in software. We present the main idea here in the context of the max criterion. It is possible to adapt to the sum criterion as well.

The main idea is to maintain a local upper bound on the maximum norm of the tiles below the diagonal. The goal for MUMPS is to spare a search in the pivot column to see if the current pivot is acceptable. Our goal is to spare a search in the tile column to know if we are going to apply an LU step or a QR step. In both cases, if the criterion is satisfied, the search will not happen, hence (1) this avoids the communication necessary for the search, and (2) this avoids to synchronize the processes in the column. In other words, if the initial matrix is such that the criterion is always satisfied at each step of the algorithm, the synchronization of the panel will go away and a pipeline will naturally be instantiated.

Following our general framework, we have a matrix A partitioned in n tiles per row and column. At the start of the algorithm, all processors have a vector r of size n. If the process holds column j, $j = 1, \dots, n$, then it also holds r_j such that

$$r_j^{(1)} = \max_{i=2,\dots,n} \|A_{i,j}\|_1.$$

If the process does not hold column j, then it does not have (and will not need) a value for r_j. If

$$\alpha \times \|(A_{11}^{(1)})^{-1}\|_1^{-1} \geq \max_{i>1} \|A_{i,1}^{(1)}\|_1,$$

we know that an LU step will be performed according to the max criterion. (See Equation 1.2.) This condition is guaranteed if

$$\alpha \times \|(A_{11}^{(1)})^{-1}\|_1^{-1} \geq r_1^{(1)}.$$

We assume that this condition is satisfied and so an LU step is decided, so that our first step is an LU step. The pivot process can decide so without any communication or synchronization. An LU step therefore is initiated. Now, following the MUMPS criterion [12, 45], we update the vectors r as follows:

$$r_j^{(2)} = r_j^{(1)} + r_1^{(1)} \|(A_{11}^{(1)})^{-1}\|_1 \|A_{1j}^{(1)}\|_1, \quad j = 2, \ldots, n,$$

We note that $\|(A_{11}^{(1)})^{-1}\|_1^{-1}$ can be broadcast along with the L and U factors of $A_{11}^{(1)}$ and that $A_{1j}^{(1)}$ is broadcast for the update to all processes holding r_j, so all processes holding r_j can update without communicating r_j.

Now we see that

$$r_j^{(2)} \geq \max_{i=3,\ldots,n} \|A_{i,j}^{(2)}\|_1, \;\; j = 2, \ldots, n$$

Indeed, let $j = 2, \ldots, n$. Now, let $i = 3, \ldots, n$, we have

$$\begin{aligned}
\|A_{i,j}^{(2)}\|_1 &= \|A_{i,j}^{(1)} - A_{i,1}^{(1)} (A_{1,1}^{(1)})^{-1} A_{1,j}^{(1)}\|_1 \\
&\leq \|A_{i,j}^{(1)}\|_1 + \|A_{i,1}^{(1)}\|_1 \|(A_{1,1}^{(1)})^{-1}\|_1 \|A_{1,j}^{(1)}\|_1 \\
&\leq \left(\max_{i=2,\ldots,n} \|A_{i,j}^{(1)}\|_1\right) + \left(\max_{i=2,\ldots,n} \|A_{i,1}^{(1)}\|_1\right) \|(A_{1,1}^{(1)})^{-1}\|_1 \|A_{1,j}^{(1)}\|_1 \\
&\leq r_j^{(1)} + r_1^{(1)} \|(A_{1,1}^{(1)})^{-1}\|_1 \|A_{1,j}^{(1)}\|_1
\end{aligned}$$

Hence, for $j = 2, \ldots, n$, we have

$$\max_{i=3,\ldots,n} \|A_{i,j}^{(2)}\|_1 \leq r_j^{(1)} + r_1^{(1)} \|(A_{1,1}^{(1)})^{-1}\|_1 c_1^{(1)},$$

so that, as claimed,

$$\max_{i=3,\ldots,n} \|A_{i,j}^{(2)}\|_1 \leq r_j^{(2)}.$$

Therefore, at step 2, the process holding A_{22} can evaluate locally (without communication nor synchronization) the condition

$$\alpha \times \|(A_{2,2}^{(2)})^{-1}\|_1^{-1} \geq r_2^{(2)},$$

and decides whether an LU or a QR step is appropriate.

1.3.5 Complexity

All criteria require the reduction of information of the off-diagonal tiles to the diagonal tile. Criteria (1.2) and (1.3) require the norm of each tile to be calculated locally (our implementation uses the 1-norm) and then reduced to the diagonal tile. Both criteria also require computing $\|A_{k,k}^{-1}\|$. Since the LU factorization of the diagonal tile is computed, the norm can be approximated using the L and U factors by an iterative method in $O(n_b^2)$ floating-point operations. The overall complexity for both criteria is $O(n \times n_b^2)$. Criterion (1.4) requires the maximum of each column be calculated locally and then reduced to the diagonal tile. The complexity of the MUMPS criterion is also $O(n \times n_b^2)$ comparisons.

The Sum criterion is the strictest of the three criteria. It also provides the best stability with linear growth in the norm of the tiles in the worst case. The other two criteria have similar worst case bounds. The growth factor for both criteria are bound by the growth factor of partial (threshold) pivoting. The Max criterion has a bound for the growth factor on the norm of the tiles that is analogous to partial pivoting. The MUMPS criteria does not operate at the tile level, but rather on scalars. If the estimated growth factor computed by the criteria is a good estimate, then the growth factor is no worse than partial (threshold) pivoting.

1.4 Implementation

As discussed in section 1.1, we have implemented the *LU-QR Algorithm* on top of the PaRSEC runtime. There are two major reasons for this choice: (i) it allows for easily targeting distributed architectures while concentrating only on the algorithm and not on implementation details such as data distribution and communications; (ii) previous implementations of the HQR algorithm [42] can be reused for QR elimination steps, and they include efficient reduction trees to reduce the critical path of these steps. The other advantage of using such a runtime is that it provides an efficient *look-ahead* algorithm without the burden. This is illustrated by the figure 1.2 that shows the first steps of the factorization with the *LU-QR Algorithm*. QR STEPS (in green) and LU STEPS (in orange/red) are interleaved automatically by the runtime and panel factorization does not wait for the previous step to be finished before starting.

However, this choice implied major difficulties due to the parameterized task graph representation exploited by the PaRSEC runtime. This representation being static, a solution had to be developed to allow for dynamism in the graph traversal. To solve this issue, we decided to fully statically describe both LU and QR algorithms in the parameterized task graph. A layer of selection tasks has then been inserted between each iteration, to collect the data from the previous step, and to propagate it to the correct following step. These tasks, which do no computations, are only executed once they received a control flow after the criterion selection has been made. Thus, they delay the decision to send the data to the next elimination step until a choice has been made, in order to guarantee that data follow the correct path. These are the *Propagate* tasks on Figure 1.1. It is important to note, that these tasks do not delay the computations since no updates are available as long as the panel is not factorized. Furthermore, these tasks, as well as *Backup Panel* tasks, can receive the same data from two different paths which could create conflicts. In the PaRSEC runtime, tasks are created only when one of their dependencies is solved; then by graph construction they are enabled only when the previous elimination step has already started, hence they will receive their data only from the correct path. It also means that tasks belonging to the neglected path will never be triggered, and so never be created. This implies that only one path will forward the data to the *Propagate* tasks.

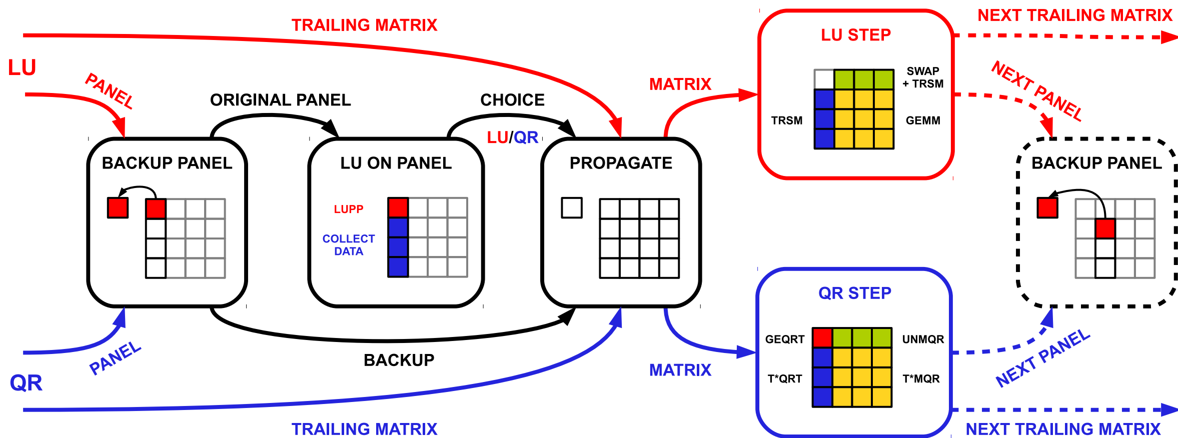


Figure 1.1: Dataflow of one step of the algorithm.

Figure 1.1 describes the connection between the different stages of one elimination step of the algorithm. These stages are described below:

Backup Panel This is a set of tasks that collect the tiles of the panel from the previous step. Since an LU factorization will be performed in-place for criterion computation, it is then necessary to backup the modified data in case the criterion fails the test on numerical accuracy. Only tiles from the current panel belonging to the node with the *diagonal* row are copied, and sent directly to the *Propagate* tasks in case a QR elimination step is needed. On other nodes, nothing is done. Then, all original tiles belonging to the panel are forwarded to the *LU On Panel* tasks.

LU On Panel Once the backup is done, the criterion is computed. Two kinds of work are performed in those tasks. On the first node, the U matrix related to this elimination step is computed. This can be done through an LU factorization with or without pivoting. We decided to exploit the multi-threaded recursive-LU kernel from the PLASMA library to enlarge the pivot search space while keeping good efficiency [43]. On all other nodes, the information required for the criterion is computed (see section 1.3). Then, an all-reduce operation is performed to exchange the information, so that everyone can take and store the decision in a local array. This all-reduce operation is directly implemented within the parameterized task graph with Bruck's algorithm [26] to optimize the cost of this operation. Once the decision is known by the nodes on the panel, it is stored in a global array by each process in order to give access to the information to every worker threads. The information is then broadcast by row to all other nodes such that everyone knows which kind of update to apply, and a control flow per process triggers all the local *Propagate* task which can now have access to the decision and release the correct path in the dataflow.

Propagate These tasks, one per tile, receive the decision from the previous stage through a control flow, and are responsible for forwarding the data to the computational tasks of the selected factorization. The tasks belonging to the panel (assigned to the first nodes) have to restore the data back to their previous state if QR elimination is chosen. In all cases, the backup is destroyed upon exit of these tasks.

We are now ready to complete the description of each step:

a) LU Step If the numerical criterion is met by the panel computation, the update step is performed. On the nodes with the *diagonal* row, a task per panel is generated to apply the row permutation computed by the factorization, and then, the triangular solve is applied to the *diagonal* to compute the U part of the matrix. The result is broadcasted per column to all other nodes and a block LU algorithm is used to performed the update. This means that the panel is updated with $TRSM$ tasks, and the trailing sub-matrix is updated with $GEMM$ tasks. This avoids the row pivoting between the nodes usually performed by the classical LU factorization algorithm with partial pivoting, or by tournament pivoting algorithms [61]. Here this exchange is made within a single node only.

b) QR Step If the numerical criterion is not met, a QR factorization has to be performed. Many solutions could be used for this elimination step. We chose to exploit the HQR method implementation presented in [42]. This allowed us to experiment with different kinds of reduction trees, so as to find the most adapted solution to our problem. The goal is to reduce the inter-nodes communications to the minimum while keeping the critical path short. In [42], we have shown that the FlatTree tree is very efficient for a good pipeline of the operations on square matrices, while Fibonacci, Greedy or BinaryTree are good for tall and skinny matrices because they reduce the length of the critical path. In this algorithm, our default tree (which we use in all of our experiments) is a hierarchical tree made of Greedy reduction trees inside nodes, and a Fibonacci reduction tree between the nodes. The goal is to perform as few QR steps as possible, so a Fibonacci tree between nodes has been chosen for

its short critical path and its good pipelining of consecutive trees if multiple QR steps are performed in sequence. Within a node, the GREEDY reduction tree is favored for similar reasons (See [42] for more details on the reduction trees). A two-level hierarchical approach is natural when considering multicore parallel distributed architectures, and those choices could be reconsidered according to the matrix size and numerical properties.

To implement the *LU-QR Algorithm* within the PARSEC framework, two extensions had to be implemented within the runtime. The first extension allows the programmer to generate data during the execution with the OUTPUT keywords. This data is then inserted into the tracking system of the runtime to follow its path in the dataflow. This is what has been used to generate the backup on the fly, and to limit the memory peak of the algorithm. A second extension has been made for the end detection of the algorithm. Due to its distributed nature, PARSEC detects the end of an algorithm by counting the remaining tasks to execute. At algorithm submission, PARSEC loops over all the domain space of each type of task of the algorithm and uses a predicate, namely *the owner computes* rule, to decide if a task is local or not. Local tasks are counted and the end of the algorithm, it is detected when all of them have been executed. As explained previously, to statically describe the dynamism of the *LU-QR Algorithm*, both LU and QR tasks exist in the parameterized graph. The size of the domain space is then larger than the number of tasks that will actually be executed. Thus, a function to dynamically increase/decrease the number of local tasks has been added, so that the *Propagate* tasks decrease the local counter of each node by the number of update tasks associated to the non selected algorithm.

The implementation of the *LU-QR Algorithm* is publicly available in the latest DPLASMA release (1.2.0).

1.5 Experiments

The purpose of this section is to present numerical experiments for the hybrid *LU-QR Algorithm*, and to highlight the trade-offs between stability and performance that can be achieved by tuning the threshold α in the robustness criterion (see Section 1.3).

1.5.1 Experimental framework

We used *Dancer*, a parallel machine hosted at the Innovative Computing Laboratory (ICL) in Knoxville, to run the experiments. This cluster has 16 multi-core nodes, each equipped with 8 cores, and an Infiniband interconnection network of 10GB/s bandwith (MT25208 cards). The nodes feature two Intel Westmere-EP E5606 CPUs at 2.13GHz. The system is running the Linux 64bit operating system, version 3.7.2-x86_64. The software was compiled with the Intel Compiler Suite 2013.3.163. BLAS kernels were provided by the MKL library and OpenMPI 1.4.3 has been used for the MPI communications by the PARSEC runtime. Each computational thread is bound to a single core using the HwLoc 1.7.1 library. If not mentioned otherwise, we will use all 16 nodes and the data will be distributed according to a 4-by-4 2D-block-cyclic distribution. In all our experiments, this distribution performs better than a 8-by-2 or a 16-by-1 distribution for our hybrid *LU-QR Algorithm*, as expected, since square grids are known to perform better for LU and QR factorization applied to square matrices [28]. The theoretical peak performance of the 16 nodes is 1091 GFLOP/sec.

For each experiment, we consider a square tiled-matrix A of size N-by-N, where $N = n \times n_b$. The tile size n_b has been fixed to 240 for the whole experiment set, because this value was found to achieve good performance for both LU and QR steps. We evaluate the backward stability by computing the

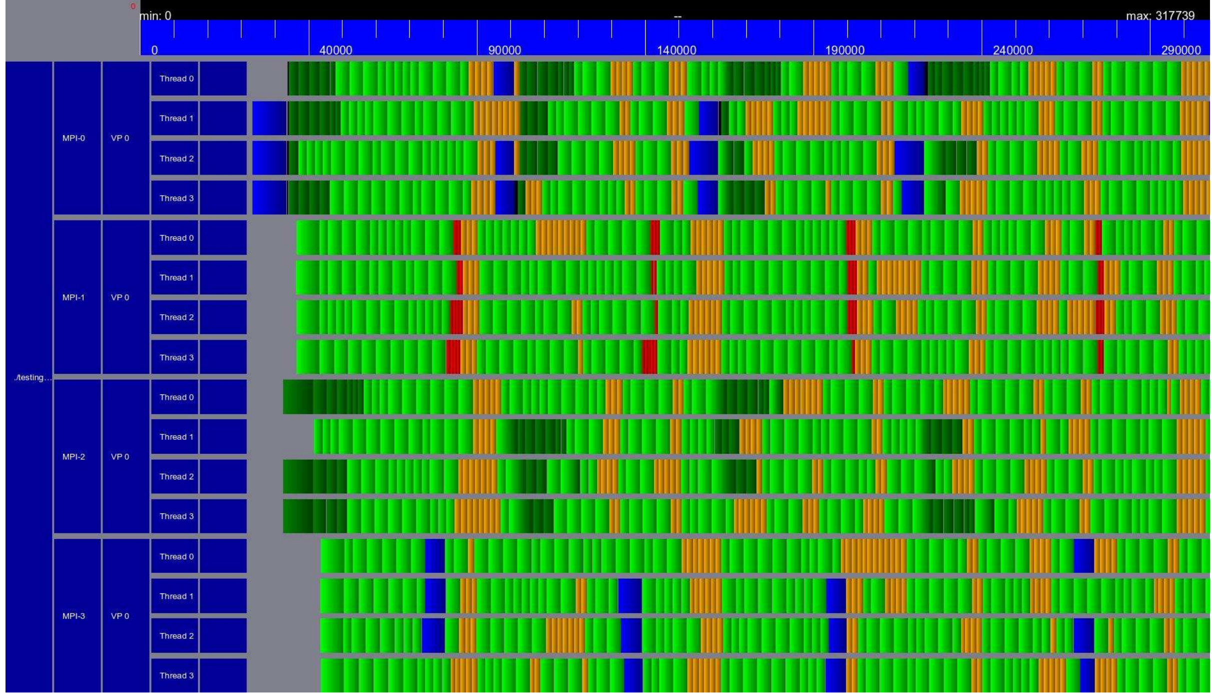


Figure 1.2: Execution trace of the first steps of the *LU-QR Algorithm* on a matrix of size $N = 5000$ with $n_b = 200$, and criterion that alternates between LU and QR steps. A grid of $2 - by - 2$ process with 4 threads each is used. The green tasks are the QR STEPS (dark for the panel factorization, and light for the trailing submatrix update); the orange and red tasks are the LU STEPS (red for the panel update and orange for the trailing submatrix update); the blue tasks are the LU panel factorizations performed at every step; and small black tasks, that do not show up on the figure because their duration is too small, are the criterion selections, and the *Propagate* tasks.

HPL3 accuracy test of the High-Performance Linpack benchmark [44]:

$$HPL3 = \frac{\|Ax - b\|_\infty}{\|A\|_\infty \|x\|_\infty \times \epsilon \times N},$$

where b is the right-hand side of the linear system, x is the computed solution and ϵ is the machine precision. Each test is run with double precision arithmetic. In all our experiments, the right-hand side of the linear system is generated using the *DPLASMA_dplrnt* routine. It generates a random matrix (or a random vector) with each element uniformly taken in $[-0.5, 0.5]$. For performance, we point out that the number of floating point operations executed by the hybrid algorithm depends on the number of LU and QR steps performed during the factorization. Thus, for a fair comparison, we assess the efficiency by reporting the *normalized* GFLOP/sec performance computed as

$$\text{GFLOP/sec} = \frac{\frac{2}{3}N^3}{\text{EXECUTION TIME}},$$

where $\frac{2}{3}N^3$ is the number of floating-point operations for LU with partial pivoting and EXECUTION TIME is the execution time of the algorithm. With this formula, QR factorization will only achieve half of the performance due to the $\frac{4}{3}N^3$ floating-point operations of the algorithm. Note that in all our experiments, the right-hand side b of the linear system is a vector. Thus, the cost of applying the transformations on b to solve the linear system is negligible, which is not necessarily the case for multiple right-hand sides.

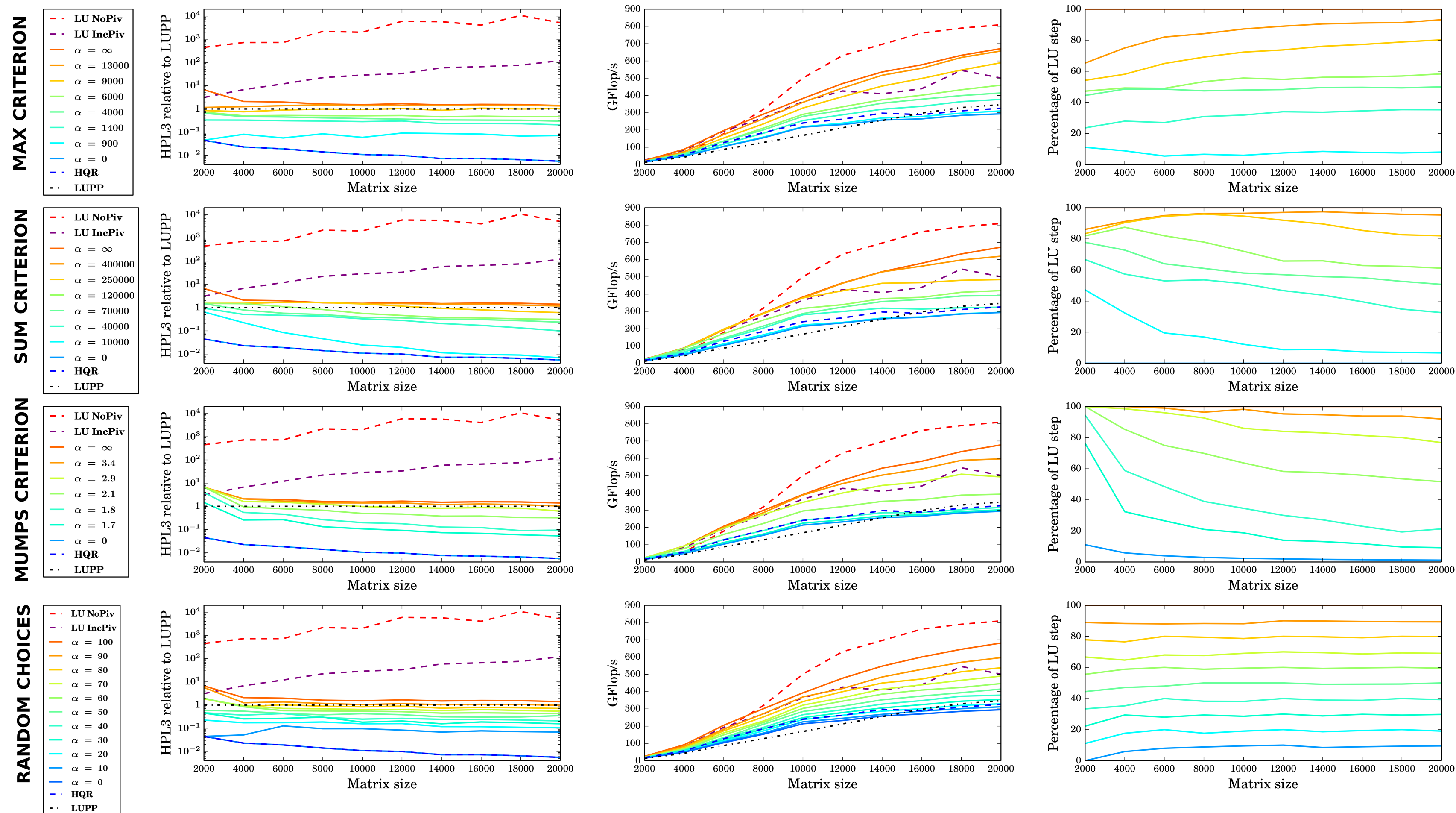


Figure 1.3: Stability, performance, and percentage of LU steps obtained by the three criteria and by random choices, for random matrices, on the Dancer platform (4x4 grid).

1.5.2 Results for random matrices

We start with the list of the algorithms used for comparison with the *LU-QR Algorithm*. All these methods are implemented within the PaRSEC framework:

- LU NoPiv, which performs pivoting only inside the diagonal tile but no pivoting across tiles (known to be both efficient and unstable)
- LU IncPiv, which performs incremental pairwise pivoting across all tiles in the elimination panel [28, 90] (still efficient but not stable either)
- Several instances of the hybrid *LU-QR Algorithm*, for different values of the robustness parameter α. Recall that the algorithm performs pivoting only across the diagonal domain, hence involving no remote communication nor synchronization.
- HQR, the Hierarchical QR factorization [42], with the same configuration as in the QR steps of the *LU-QR Algorithm*: GREEDY reduction trees inside nodes and FIBONACCI reduction trees between the nodes.

For reference, we also include a comparison with PDGEQRF: this is the LUPP algorithm (LU with partial pivoting across all tiles of the elimination panel) from the reference ScaLAPACK implementation [32].

Algorithm	α	Time	% LU steps	Fake GFLOP/sec	True GFLOP/sec	Fake % Peak Perf.	True % Peak Perf.
LU NoPiv		6.29	100.0	848.6	848.6	77.8	77.8
LU IncPiv		9.25	100.0	576.4	576.4	52.9	52.9
LUQR (MAX)	∞	7.87	100.0	677.7	677.7	62.1	62.1
LUQR (MAX)	13000	7.99	94.1	667.7	707.4	61.2	64.9
LUQR (MAX)	9000	8.62	83.3	619.0	722.2	56.8	66.2
LUQR (MAX)	6000	10.95	61.9	486.9	672.4	44.6	61.7
LUQR (MAX)	4000	12.43	51.2	429.0	638.4	39.3	58.5
LUQR (MAX)	1400	13.76	35.7	387.6	636.9	35.5	58.4
LUQR (MAX)	900	16.39	11.9	325.4	612.0	29.8	56.1
LUQR (MAX)	0	18.05	0.0	295.5	590.9	27.1	54.2
HQR		16.01	0.0	333.1	666.1	30.5	61.1
LUPP		15.30	100.0	348.6	348.6	32.0	32.0

Table 1.2: Performance obtained by each algorithm, for $N = 20,000$, on the Dancer platform (4×4 grid). We only show the results for the *LU-QR Algorithm* with the Max criterion. The other criteria have similar performance. In column Fake GFLOP/sec, we assume all algorithms perform $\frac{2}{3}N^3$ floating-point operations. In column True GFLOP/sec, we compute the number of floating-point operations to be $(\frac{2}{3}f_{LU} + \frac{4}{3}(1 - f_{LU}))N^3$, where f_{LU} is the fraction of the steps that are LU steps (column 4).

Figure 1.3 summarizes all results for random matrices. The random matrices are generated using the *DPLASMA_dplrnt* routine. The figure is organized as follows: each of the first three rows corresponds to one criterion. Within a row:

- the first column shows the relative stability (ratio of HPL3 value divided by HPL3 value for LUPP)
- the second column shows the GFLOP/sec performance
- the third column shows the percentage of LU steps during execution

The fourth row corresponds to a random choice between LU and QR at each step, and is intended to assess the performance obtained for a given ratio of LU vs QR steps. Plotted results are average values obtained on a set of 100 random matrices (we observe a very small standard deviation, less than 2%).

For each criterion, we experimentally chose a set of values of α that provides a representative range of ratios for the number of LU and QR steps. As explained in Section 1.3, for each criterion, the smaller the α is, the tighter the stability requirement. Thus, the numerical criterion is met less frequently and the hybrid algorithm processes fewer LU steps. A current limitation of our approach is that we do not know how to auto-tune the best range of values for α, which seems to depend heavily upon matrix size and available degree of parallelism. In addition, the range of useful α values is quite different for each

criterion.

For random matrices, we observe in Figure 1.3 that the stability of LU NoPiv and LU IncPiv is not satisfactory. We also observe that, for each criterion, small values of α result in better stability, to the detriment of performance. For $\alpha = 0$, *LU-QR Algorithm* processes only QR steps, which leads to the exact same stability as the HQR Algorithm and almost the same performance results. The difference between the performance of *LU-QR Algorithm* with $\alpha = 0$ and HQR comes from the cost of the decision making process steps (saving the panel, computing the LU factorization with partial pivoting on the diagonal domain, computing the choice, and restoring the panel). Figure 1.3 shows that the overhead due to the decision making process is approximately equal to 10% for the three criteria. This overhead, computed when QR eliminations are performed at each step, is primarily due to the backup/restore steps added to the critical path when QR is chosen. Performance impact of the criterion computation itself is negligible, as one can see by comparing performance of the random criterion to the MUMPS and Max criteria.

LU-QR Algorithm with $\alpha = \infty$ and LU NoPiv both process only LU steps. The only difference between both algorithms in terms of error analysis is that LU NoPiv seeks for a pivot in the diagonal tile, while *LU-QR Algorithm* with $\alpha = \infty$ seeks for a pivot in the diagonal domain. This difference has a considerable impact in terms of stability, in particular on random matrices. *LU-QR Algorithm* with $\alpha = \infty$ has a stability slightly inferior to that of LUPP and significantly better to that of LU NoPiv. When the matrix size increases, the relative stability results of the *LU-QR Algorithm* with $\alpha = \infty$ tends to 1, which means that, on random matrices, processing an LU factorization with partial pivoting on a diagonal domain followed by a direct elimination without pivoting for the rest of the panel is almost as stable as an LU factorization with partial pivoting on the whole panel. A hand-waving explanantion would go as follows. The main instabilities are proportional to the small pivots encountered during a factorization. Using diagonal pivoting, as the factorization of the diagonal tile proceeds, one is left with fewer and fewer choices for a pivot in the tile. Ultimately, for the last entry of the tile in position (n_b,n_b), one is left with no choice at all. When working on random matrices, after having performed several successive diagonal factorizations, one is bound to have encountered a few small pivots. These small pivots lead to a bad stability. Using a domain (made of several tiles) for the factorization significantly increases the number of choice for the pivot and it is not any longer likely to encounter a small pivot. Consequently diagonal domain pivoting significantly increases the stability of the *LU-QR Algorithm* with $\alpha = \infty$. When the local domain gets large enough (while being significanty less than N), the stability obtained on random matrices is about the same as partial pivoting.

When $\alpha = \infty$, our criterion is deactivated and our algorithm always performs LU step. We note that, when α is reasonable, (as opposed to $\alpha = \infty$,) the algorithm is stable whether we use a diagonal domain or a diagonal tile. However using a diagonal domain increases the chance of well-behaved pivot tile for the elimination, therefore using a diagonal domain (as opposed to a diagonal tile) increases the chances of an LU step.

Using random choices leads to results comparable to those obtained with the three criteria. However, since we are using random matrices in this experiment set, we need to be careful before drawing any conclusion on the stability of our algorithms. If an algorithm is not stable on random matrices, this is clearly bad. However we cannot draw any definitive conclusion if an algorithm is stable for random matrices.

Table 1.2 displays detailed performance results for the Max criterion with $N = 20,000$. In column Fake GFLOP/sec, we assume all algorithms perform $\frac{2}{3}N^3$ floating-point operations. In column True GFLOP/sec, we compute the number of floating-point operations to be $(\frac{2}{3}f_{LU} + \frac{4}{3}(1 - f_{LU}))N^3$, where f_{LU} is the fraction of the steps that are LU steps (column 4). For this example, we see that the *LU-QR Algorithm* reaches a peak performance of 677.7 GFLOP/sec (62.1% of the theoretical peak) when

every step is an LU step. Comparing *LU-QR Algorithm* with $\alpha = 0$ and HQR shows the overhead for the decision making process is 12.7%. We would like the true performance to be constant as the number of QR steps increases. For this example, we see only a slight decrease in performance, from 62.1% for $\alpha = \infty$ to 54.2% for $\alpha = 0$. HQR maintains nearly the same true performance (61.1%) as the *LU-QR Algorithm* with $\alpha = \infty$. Therefore, the decrease in true performance is due largely to the overhead of restoring the panel.

1.5.3 Results for different processor grid sizes

In Section 1.5.2, we presented the stability and performance results on a 4×4 grid of processors, which means that at each step of the *LU-QR Algorithm* the panel was split in 4 domains. The number of domains in the panel has a strong impact on the stability. In a LU step, increasing the number of domains will reduce the search area for pivots during the *factor* step. Conversely, the bigger the diagonal domain is, the larger the singular values of the factored region will be. Figure 1.4 displays the stability results for different sizes of processors grid.

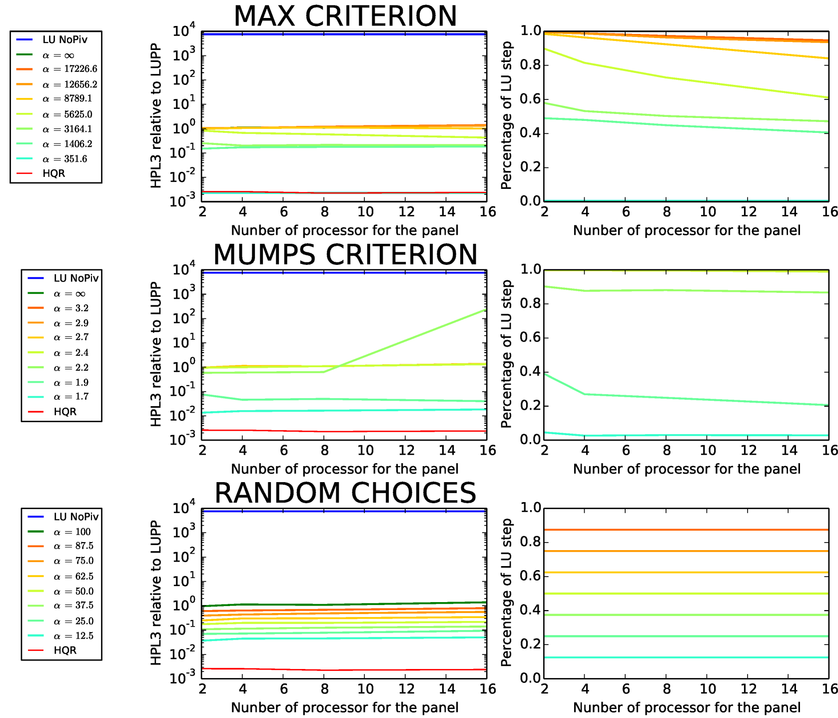


Figure 1.4: Stability and percentage of LU steps obtained by the Max criterion, the MUMPS criterion and by random choices, for random matrices of size 40000.

The first column shows the relative stability (ratio of HPL3 value divided by HPL3 value for LUPP) obtained by running the hybrid *LU-QR Algorithm* on a set of 5 random matrices of size $N = 40,000$. The second column shows the percentage of LU steps during execution. We can see that the relative stability remains quite constant when the number of domains in the panel increases. The alpha parameters sets the stability requirement for the factorization independently of the size of the platform. However, the percentage of LU steps for a fixed value of α decreases when the number of domain increases. To maintain this stability requirement when the size of the diagonal domain decreases, the *LU-QR Algorithm* has to perform more QR steps. For instance, for the Max criterion, and for $\alpha = 5625$, the *LU-QR Al-*

gorithm performs 90% of LU steps during the factorization when the panel is split in 2 domains. It can only perform 60% of LU steps to maintain the same stability requirement when the panel is split in 16 domains.

1.5.4 Results for special matrices

For random matrices, we obtain a good stability with random choices, almost as good as with the three criteria. However, as mentioned above, we should draw no definite conclusion. To highlight the need for a smart criterion, we tested the hybrid *LU-QR Algorithm* on a collection of matrices that includes several pathological matrices on which LUPP fails because of large growth factors. This set of special matrices described in Table 1.3 includes ill-conditioned matrices as well as sparse matrices, and mostly comes from the Higham's Matrix Computation Toolbox [67].

Figure 1.5 provides the relative stability (ratio of HPL3 divided by HPL3 for LUPP) obtained by running the hybrid *LU-QR Algorithm* on a set of 5 random matrices and on the set of special matrices. Matrix size is set to $N = 40,000$, and experiments were run on a 16-by-1 process grid. The parameter α has been set to 50 for the random criterion, $6,000$ for the Max criterion, and 2.1 for the MUMPS criterion (we do not report result for the Sum criterion because they are the same as they are for Max). Figure 1.5 considers LU NoPiv, HQR and the *LU-QR Algorithm*. The first observation is that using random choices now leads to numerical instability. The Max criterion provides a good stability ratio on every tested matrix (up to 58 for the RIS matrix and down to 0.03 for the Invhess matrix). The MUMPS criterion also gives modest growth factor for the whole experiment set except for the Wilkinson and the Foster matrices, for which it fails to detect some "bad" steps.

We point out that we also experimented with the Fiedler matrix from Higham's Matrix Computation Toolbox [67]. We observed that LU NoPiv and LUPP failed (due to small values rounded up to 0 and then illegally used in a division), while the Max and the MUMPS criteria provide HPL3 values ($\approx 5.16 \times 10^{-09}$ and $\approx 2.59 \times 10^{-09}$) comparable to that of HQR ($\approx 5.56 \times 10^{-09}$). This proves that our criteria can detect and handle pathological cases for which the generic LUPP algorithm fails.

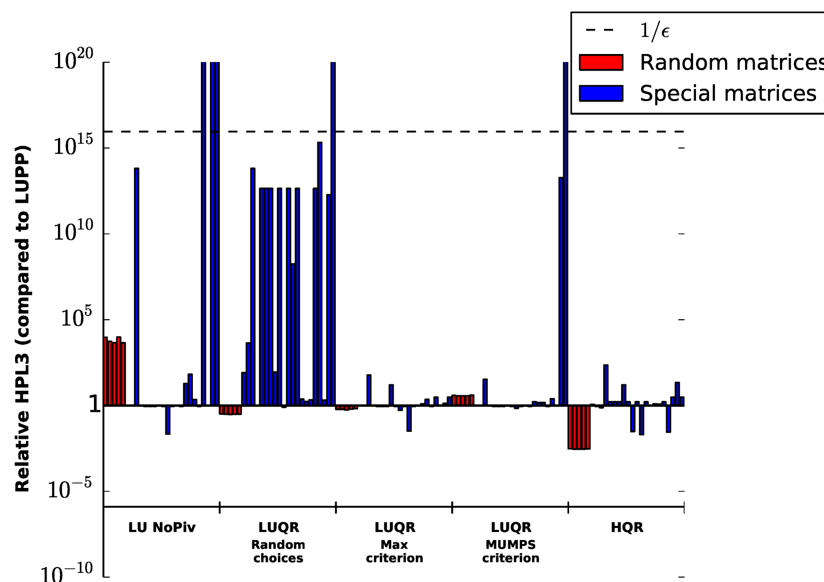


Figure 1.5: Stability on special matrices.

No.	Matrix	Description	Condition Number with $N = 4000$
1	house	Householder matrix, $A = eye(n) - \beta * v * v^H$	1
2	parter	Parter matrix, a Toeplitz matrix with most of singular values near Π. $A(i,j) = 1/(i-j+0.5)$.	4.8
3	ris	Ris matrix, matrix with elements $A(i,j) = 0.5/(n-i-j+1.5)$. The eigenvalues cluster around $-\Pi/2$ and $\Pi/2$.	4.8
4	condex	Counter-example matrix to condition estimators.	1.0×10^2
5	circul	Circulant matrix	2.2×10^4
6	hankel	Hankel matrix, $A = hankel(c,r)$, where $c = randn(n,1)$, $r = randn(n,1)$, and $c(n) = r(1)$.	3.8×10^4
7	compan	Companion matrix (sparse), $A = compan(randn(n+1,1))$.	4.1×10^6
8	lehmer	Lehmer matrix, a symmetric positive definite matrix such that $A(i,j) = i/j$ for $j \geq i$. Its inverse is tridiagonal.	1.7×10^7
9	dorr	Dorr matrix, a diagonally dominant, ill-conditioned, tridiagonal matrix (sparse).	1.6×10^{11}
10	demmel	$A = D * (eye(n) + 10 - 7 * rand(n))$, where $D = diag(1014 * (0 : n-1)/n)$.	9.8×10^{20}
11	chebvand	Chebyshev Vandermonde matrix based on n equally spaced points on the interval [0,1].	2.2×10^{19}
12	invhess	Its inverse is an upper Hessenberg matrix.	—
13	prolate	Prolate matrix, an ill-conditioned Toeplitz matrix.	9.4×10^{18}
14	cauchy	Cauchy matrix.	1.9×10^{21}
15	hilb	Hilbert matrix with elements $1/(i+j-1)$. $A = hilb(n)$.	7.5×10^{21}
16	lotkin	Lotkin matrix, the Hilbert matrix with its first row altered to all ones.	2.1×10^{23}
17	kahan	Kahan matrix, an upper trapezoidal matrix.	1.4×10^{27}
18	orthogo	Symmetric eigenvector matrix: $A(i,j) = sqrt(2/(n+1)) * sin(i * j * \pi/(n+1))$	1
19	wilkinson	Matrix attaining the upper bound of the growth factor of GEPP.	9.4×10^3
20	foster	Matrix arising from using the quadrature method to solve a certain Volterra integral equation.	2.6×10^3
21	wright	Matrix with an exponential growth factor when Gaussian elimination with Partial Pivoting is used.	6.5

Table 1.3: Special matrices in the experiment set.

1.5.5 Results for varying the condition number

In section 1.5.4, we presented stability results for a set of pathological matrices on which LUPP fails. Figure 1.6 provides the relative stability (ratio of HPL3 divided by HPL3 for LUPP) and the percentage of LU and QR steps obtained by the hybrid *LU-QR Algorithm* on matrices with varying condition numbers. The results have been obtained with Matlab (version 7.8.0) on matrices of size 1000, and for 100 domains per panel. The right-hand side of the linear system is a vector with each element randomly taken from a standard normal distribution (with a mean $\mu = 0$ and a standard deviation $\sigma = 1$). The matrix A was generated by a singular value decomposition. Orthogonal matrices were obtained via the Q-factor of a QR decomposition of matrices with each element randomly taken from a standard normal distribution. To obtain a condition number of κ, we let the maximum singular value be 1 and the smallest singular value be κ^{-1}. The base-10 logarithms of the singular values are linearly spaced. The diagonal matrix with the singular values on the diagonal is obtained in Matlab by diag(10.^(-linspace(0,log10(κ),1000))). Each point of the curves is an average of 10 runs.

We observe that the percentage of LU and QR steps and the relative backward error appear to be independent of the condition number of the matrix, and to depend only on the α parameter. The stability of our algorithm is governed by the quality of the diagonal tile during an LU step. The quality of the diagonal tile during an LU step does not depend on the condition number of the matrix, but it depends on the α parameter.

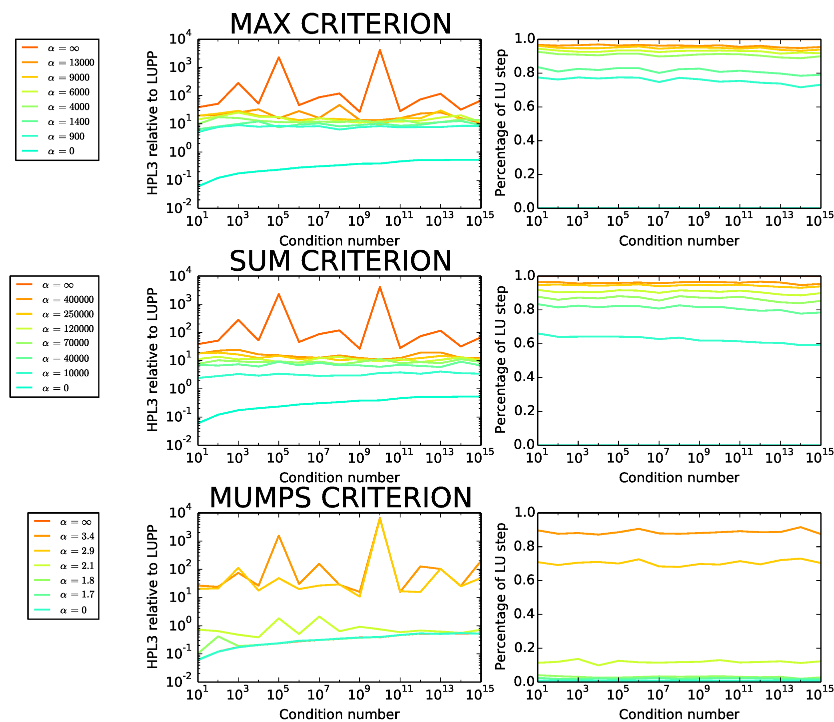


Figure 1.6: Stability and percentage of LU steps obtained by the Max criterion, the Sum criterion and the MUMPS criterion for random matrices of size 1000.

1.5.6 Assessment of the three criteria

With respect to stability, while the three criteria behave similarly on random matrices, we observe different behaviors for special matrices. The MUMPS criterion provides good results for most of the tested matrices but not for all. If stability is the key concern, one may prefer to use the Max criterion (or the Sum criterion), which performs well for all special matrices (which means that the upper bound of $(1+\alpha)^{n-1}$ on the growth is quite pessimistic).

With respect to performance, we observe very comparable results, which means that the overhead induced by computing the criterion at each step is of the same order of magnitude for all criteria.

The overall conclusion is that all criteria bring significant improvement over LUPP in terms of stability, and over HQR in terms of performance. Tuning the value of the robustness parameter α enables the exploration of a wide range of stability/performance trade-offs.

1.6 Related work

State-of-the-art QR factorizations use multiple eliminators per panel, in order to dramatically reduce the critical path of the algorithm. These algorithms are unconditionally stable, and their parallelization has been fairly well studied on shared memory systems [27, 90, 24] and on parallel distributed systems [42].

The idea of mixing Gaussian transformations and orthogonal transformations has been considered once before. Irony and Toledo [73] present an algorithm for reducing a banded symmetric indefinite matrix to diagonal form. The algorithm uses symmetric Gaussian transformations and Givens rotations to maintain the banded symmetric structure and maintain similar stability to partial symmetric pivoting.

The reason for using LU kernels instead of QR kernels is performance: (i) LU performs half the number of floating-point operations of QR; (ii) LU kernels relies on GEMM kernels which are very efficient while QR kernels are more complex and much less tuned, hence not that efficient; and (iii) the LU update is much more parallel than the QR update. So all in all, LU is much faster than QR (as observed in the performance results of Section 1.5). Because of the large number of communications and synchronizations induced by pivoting in the reference LUPP algorithm, *communication-avoiding* variants of LUPP have been introduced [36], but they have proven much more challenging to design because of stability issues. In the following, we review several approaches:

1.6.1 LUPP

LU with partial pivoting is not a communication-avoiding scheme and its performance in a parallel distributed environment is low (see Section 1.5). However, the LUPP algorithm is *stable in practice*, and we use it as a reference for stability.

1.6.2 LU NoPiv

The most basic communication-avoiding LU algorithm is LU NoPiv. This algorithm is stable for block diagonal dominant matrices [67, 37], but breaks down if it encounters a nearly singular diagonal tile, or loses stability if it encounters a diagonal tile whose smallest singular value is too small.

Baboulin et al. [16] propose to apply a random transformation to the initial matrix, in order to use LU NoPiv while maintaining stability. This approach gives about the same performance as LU NoPiv, since preprocessing and postprocessing costs are negligible. It is hard to be satisfied with this approach [16] because for any matrix which is rendered stable by this approach (i.e, LU NoPiv is stable), there exists a matrix which is rendered not stable. Nevertheless, in practice, this proves to be a valid approach.

1.6.3 LU IncPiv

LU IncPiv is another communication-avoiding LU algorithm [28, 90]. Incremental pivoting is also called *pairwise pivoting*. The stability of the algorithm [28] is not sufficient and degrades as the number of tiles in the matrix increases (see our experimental results on random matrices). The method also suffers some of the same performance degradation of QR factorizations with multiple eliminators per panel, namely low-performing kernels, and some dependencies in the update phase.

1.6.4 CALU

CALU [61] is a communication-avoiding LU. It uses tournament pivoting which has been proven to be *stable in practice* [61]. CALU shares the (good) properties of one of our LU steps: (i) low number of floating-point operations; (ii) use of efficient GEMM kernels; and (iii) embarrassingly parallel update. The advantage of CALU over our algorithm is essentially that it performs only LU steps, while our algorithm might need to perform some (more expensive) QR steps. The disadvantage is that, at each step, CALU needs to perform global pivoting on the whole panel, which then needs to be reported during the update phase to the whole trailing submatrix. There is no publicly available implementation of parallel distributed CALU, and it was not possible to compare stability or performance. CALU is known to be stable in practice [60, 40]. Performance results of CALU in parallel distributed are presented in [60]. Performance results of CALU on a single multicore node are presented in [40].

1.6.5 Summary

Table 1.4 provides a summary of key characteristics of the algorithms discussed in this section.

ALGORITHM	CA	KERNELS EFF. FOR UPDATE	PIPELINE	#FLOPS	STABLE
LU NoPiV	YES	GEMM-EFFICIENT	YES	1x	NOT AT ALL
LU IncPiv	YES	LESS EFFICIENT	YES	1x	SOMEWHAT
CALU	YES	GEMM-EFFICIENT	NO	1x	PRACTICALLY
LUQR (alpha) LU only	YES	GEMM-EFFICIENT	NO	1x	PRACTICALLY
LUQR (alpha) QR only	YES	LESS EFFICIENT	NO	2x	UNCONDITIONALLY
HQR	YES	LESS EFFICIENT	YES	2x	UNCONDITIONALLY
LUPP	NO	GEMM-EFFICIENT	NO	1x	PRACTICALLY

Table 1.4: A summary of key characteristics of each algorithm. CA in column 2 stands for *Communication Avoiding*. Other column titles are self-explanatory.

1.7 Conclusion

Linear algebra software designers have been struggling for years to improve the parallel efficiency of LUPP (LU with partial pivoting), the de-facto choice method for solving dense systems. The search for good pivots throughout the elimination panel is the key for stability (and indeed both NoPiv and IncPiv fail to provide acceptable stability), but it induces several short-length communications that dramatically decrease the overall performance of the factorization.

Communication-avoiding algorithms are a recent alternative which proves very relevant on today's archictectures. For example, in our experiments, our HQR factorization [42] based of QR kernels ends with similar performance as ScaLAPACK LUPP while performing 2x more floating-point operations, using slower sequential kernels, and a less parallel update phase. In this chapter, stemming from the

key observation that LU steps and QR steps can be mixed during a factorization, we present the *LU-QR Algorithm* whose goal is to accelerate the HQR algorithm by introducing some LU steps whenever these do not compromise stability. The hybrid algorithm represents dramatic progress in a long-standing research problem. By restricting to pivoting inside the diagonal domain, i.e., locally, but by doing so only when the robustness criterion forecasts that it is safe (and going to a QR step otherwise), we improve performance while guaranteeing stability. And we provide a continuous range of trade-offs between LU NoPiv (efficient but only stable for diagonally-dominant matrices) and QR (always stable but twice as costly and with less performance). For some classes of matrices (e.g., tile diagonally dominant), the *LU-QR Algorithm* will only perform LU steps.

This work opens several research directions. First, as already mentioned, the choice of the robustness parameter α is left to the user, and it would be very interesting to be able to auto-tune a possible range of values as a function of the problem and platform parameters. Second, there are many variants and extensions of the hybrid algorithm that can be envisioned. Several have been mentioned in Section 1.2, and many others could be tried. In particular, the tile extension of the MUMPS criterion looks promising and deserves to be implement in software during future work. Another goal would be to derive LU algorithms with several eliminators per panel (just as for HQR) to decrease the critical path, provided the availability of a reliable robustness test to ensure stability.

Chapter 2

Bridging the gap between experimental performance and theoretical bounds for the Cholesky factorization on heterogeneous platforms

Mainly linear algebra libraries rely on runtime systems to deal with resource allocation and low-level data management, such as MPI communications, shared-memory accesses... These runtimes allows developers to focus on their algorithms at a task-level. In the previous chapter, we designed a new dense linear solver relying on the dynamic scheduler provided by PaRSEC. In the purpose of increasing the performance of linear solvers, it is thus important to optimize the resource allocation of these runtime systems. In this chapter, we consider the problem of allocating and scheduling dense linear application on fully heterogeneous platform made of CPUs and GPUs. More specifically, we focus on the Cholesky factorization since it is the simplest factorization algorithm that exhibits the main problems encountered in heterogeneous scheduling. Indeed, the relative performance of CPU and GPU highly depends on the sub-routine: GPUs are for instance much more efficient to process regular kernels such as matrix-matrix multiplication rather than more irregular kernels such as matrix factorization. In this context, one solution consists in relying on dynamic scheduling and resource allocation mechanisms such as the ones provided by PaRSEC or StarPU. In this chapter we analyze the performance of dynamic schedulers based on both actual executions and simulations, and we investigate how adding static rules based on an offline analysis of the problem to their decision process can indeed improve their performance, up to reaching some improved theoretical performance bounds which we introduce.

2.1 Introduction

Linear algebra operations are the basis of many scientific operations. Our objective is to optimize the performance of one of them (namely the Cholesky decomposition) on a hybrid computing platform. The use of GPUs and other accelerators such as Xeon Phi are common ways to increase the computation power of computers at a limited cost. The large computing power available on such accelerators for regular computation makes them unavoidable for linear algebra operations. However, optimizing the performance of a complex computation on such a hybrid platform is very complex, and a manual optimization seems out of reach given the wide variety of hybrid configurations. Thus, several runtime systems have been proposed to dynamically schedule a computation on hybrid platforms, by mapping parts of the computation to each processing elements, either cores or accelerators. Among other successful projects,

we may cite StarPU [14] from INRIA Bordeaux (France), Quark [110] and PaRSEC [21] from ICL, Univ. of Tennessee Knoxville (USA), Supermatrix [30] from University of Texas (USA), StarSs [87] from Barcelona Supercomputing Center (Spain) or KAAPI [50] from INRIA Grenoble (France). Usually, the structure of the computation has to be described as a task graph, where vertices represent tasks and edges represent dependencies between them. Most of these tools enable, up to a certain extent, to schedule an application described as a task graph onto a parallel platform, by mapping individual tasks onto computing resources and by performing data movements between memories when needed.

There is an abundant literature on the problem of scheduling task graphs on parallel processors. This problem is known to be NP-complete [49]. Lower-bounds based either on the length of the critical path (the longest path from an entry vertex to an output vertex) or on the overall workload (assuming ideal parallelism) have been proposed, and simple list-scheduling algorithms are known to provide $2 - 1/m$-approximation on homogeneous platforms, at least when communication times are negligible [56]. Several scheduling heuristics have also been proposed, and among them the best-known certainly is HEFT [104], which inspired some dynamic scheduling strategies used in the above-mentioned runtimes. However, it remains a large gap between the theoretical lower-bounds and the actual performance of dynamic HEFT-like heuristics. Another way to assess the quality of a scheduling strategy is to compare the actual performance to the machine peak performance of the computing platform computed as the sum of the performance of its individual computational units. Rather than this machine peak performance which is known to be unreachable, one usually considers the GEMM peak obtained by running matrix multiplication kernels (GEMMs). For large matrices, the task-graph of a Cholesky factorization exhibits a sufficient amount of parallelism, and a sufficient number of GEMM calls for this bound to be reasonable. However, on small and medium size matrices, there is still a large gap between GEMM peak performance and the best-achievable Cholesky performance.

In this chapter, we optimize the dynamic scheduling of the Cholesky decomposition of a dense, symmetric, and positive-definite double-precision matrix A, into the product LL^T, where L is lower triangular and has positive diagonal elements, using one runtime system, StarPU, and provide better bounds to prove the quality of our schedules. The contributions of the chapter are:

- Better lower bounds on the processing time of a Cholesky factorization on a parallel hybrid platform;
- Better dynamic schedules, based not only on HEFT but also on an hybridization of static and dynamic task assignments;
- A very efficient schedule for a simple hybrid platform model, achieved by constraint programming.
- Numerous experiments and simulations to assess the performance of our schedules using the StarPU runtime.

Note that what is done here using StarPU could have been done with other runtimes, provided that we are able to control their mapping and scheduling policies. Similarly, we could have chosen another dense linear algebra factorization such as the QR or LU decompositions.

2.2 Context

2.2.1 Cholesky factorization

The Cholesky factorization (or Cholesky decomposition) is mainly used to solve a system of linear equations $Ax = b$, where A is a $N \times N$ symmetric positive-definite matrix, b is a vector, and x is the

Algorithm 5: Pseudocode of the tile Cholesky factorization

```
for k = 0 to n − 1 do
    A[k][k] ← POTRF(A[k][k]);
    for i = k + 1 to n − 1 do
        A[i][k] ← TRSM(A[k][k], A[i][k]);
    for j = k + 1 to n − 1 do
        A[j][j] ← SYRK(A[j][k], A[j][j]);
        for i = j + 1 to n − 1 do
            A[i][j] ← GEMM(A[i][k], A[j][k], A[i][j]);
```

unknown solution vector to be computed. Such systems often arise in physics applications, especially when looking for numerical solutions of partial differential equations, where A is positive-definite due to the nature of the modeled physical phenomenon. One way to solve such a linear system is first to compute the Cholesky factorization $A = LL^T$, where L (referred to as the Cholesky factor) is a $N \times N$ real lower triangular matrix with positive diagonal elements. The solution vector x can then be computed by solving the two following triangular systems: $Ly = b$ and $L^T x = y$.

To take advantage of modern highly parallel architectures, state-of-the-art numerical algebra libraries implement tiled Cholesky factorizations. The matrix $A = (A_{ij})_{0 \leq i,j \leq n}$ is divided into $n \times n$ tiles (or blocks) of $n_b \times n_b$ elements, and the tiled Cholesky algorithm can then be seen as a sequence of tasks that operate on small portions of the matrix. This approach greatly improves the parallelism of the algorithm and mostly involves BLAS3 kernels whose library implementations are really fast on modern architectures. The benefits of such an approach on parallel multicore systems have already been discussed in the past [63, 28, 89]. Following the BLAS and LAPACK terminology, the tiled algorithm for Cholesky factorization is based on the following set of kernel subroutines:

- *POTRF:* This LAPACK subroutine is used to perform the Cholesky factorization of a symmetric positive definite tile A_{kk} of size $n_b \times n_b$ producing a lower triangular tile L_{kk} of size $n_b \times n_b$.
- *TRSM:* This BLAS subroutine is used to apply the transformation computed by *POTRF* to a A_{ik} tile by means of a triangular system solving.
- *GEMM:* This BLAS subroutine computes a matrix-matrix multiplication of two tiles A_{ik}, A_{jk} and subtract the result to the tile A_{ij}. The old value of A_{ij} is overwritten by the new one.
- *SYRK:* This BLAS subroutine executes a symmetric rank-k update on a diagonal tile A_{kk}.

Note that no extra memory area is needed to store the L_{ij} tiles since they can overwrite the corresponding A_{ij} tiles from the original matrix. The tiled Cholesky algorithm can be written as in Algorithm 5.

In this sequential pseudocode, we can notice that some kernel subroutines depend on each other, while others can be processed in parallel. Such an algorithm is commonly represented by its task graph (or DAG) that depicts its actual dependencies. In this well established model, each vertex of the graph represents a call to one of the four kernel subroutines presented above. The edges between two task vertices represent a direct data dependency between tasks. Figure 2.1 depicts the task graph for the Cholesky decomposition of a 5×5 tiled matrix.

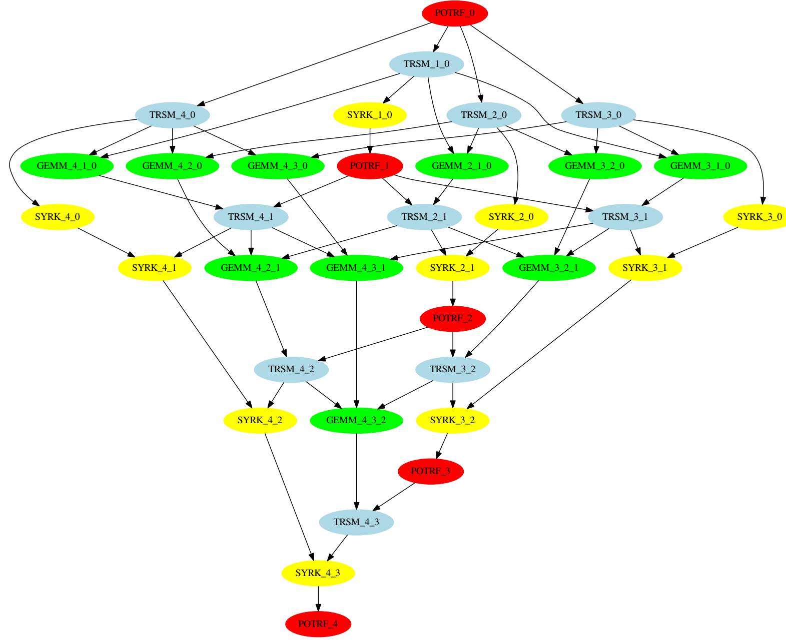


Figure 2.1: Task graph for the Cholesky decomposition of a 5×5 tiled matrix.

2.2.2 Multiprocessor scheduling

2.2.2.1 Static Mapping

In such an algorithm, the way the matrix tiles are distributed over the processors and the mapping of the tasks onto the computing resources have a strong impact on its performance and scalability. In the distributed context, the ScaLAPACK library [39] uses the standard 2D block-cyclic distribution of tiles along a virtual p-by-q homogeneous grid and the owner compute strategy. In this layout the p-by-q top-left tiles of the matrix are mapped topologically onto the processor grid and the rest of the tiles are distributed onto the processors in a round-robin manner. This layout has been incorporated in the High Performance Fortran standard [79]. Once the tiles of the matrix are distributed onto the processors, the mapping of the tasks is done following the owner-compute strategy: each subroutine overwriting a tile A_{ij} is executed on the processor hosting A_{ij}. The only freedom during the execution is the scheduling of the tasks inner a processor and the communications between processors. This layout ensures a good load balance for homogeneous computing resources and an equal memory usage between processors [39]. However, for heterogeneous resources, this layout is no longer an option, and dynamic scheduling is a widespread practice.

These ideas also make sense in a shared-memory environment in order to take advantage of locality. For instance the Plasma library provides an option for relying on such static schedules on multicore chips.

2.2.2.2 Dynamic Scheduling

Dynamic strategies have been developed in order to design methods flexible enough to cope with unpredictable performance of resources, especially in the context of real time systems, where on-line and adaptive scheduling strategies are required [31, 82]. More recently, the design of dynamic schedulers

received a lot of attention, since on modern heterogeneous and possibly shared systems, the actual prediction of either execution and communication times is very hard, thus justifying the design of ad-hoc tools that will be described in Section 2.3.

2.2.2.3 Task graph scheduling heuristics

As presented earlier, many scheduling heuristics have been proposed for DAGs since this problem is NP-complete. Most of these heuristics are list-scheduling heuristics: they sort tasks according to some criterion and then schedule them greedily. This makes them good candidates to be turned into dynamic scheduling heuristics. The best-known list-scheduling heuristic for DAGs on heterogeneous platforms is certainly HEFT [104]. It consists in sorting tasks by decreasing *bottom-level*, which is the weight of the longest path from a task to an exit task (a task without successors). In a heterogeneous environment, the weight of a task (or communication) is computed as the average computation (or communication) time over the whole platform. Then, each task is considered and scheduled on the resource on which it will finish the earliest. HEFT turns out to be an efficient heuristic for heterogeneous processors. Other approaches have been proposed to avoid data movement when taking communications into account, such as clustering tasks into larger granularity tasks before scheduling them [93].

2.3 Tools and libraries

For this study, we used the Chameleon [4] implementation of the Cholesky factorization, running on top of the StarPU runtime system. We performed real executions on the target platform, and we additionally used the Simgrid [29] simulator, in order to reduce the experimentation time, improve reproducibility of the experiments, and also be able to modify the execution platform.

2.3.1 StarPU runtime system

StarPU [14] is a runtime system aiming to allow programmers to exploit the computing power of the available CPUs and GPUs, while relieving them from the need to specifically adapt their programs to the target machine and processing units. The StarPU runtime supports a *task-based programming model*. Applications submit computational tasks, forming a task graph, with CPU and/or GPU implementations, and StarPU schedules these tasks and associated data transfers on available CPUs and GPUs. The data that a task manipulates is automatically transferred between the local memory of the accelerators and the main memory, so that application programmers are freed from the scheduling issues and technical details associated with these transfers. In particular, StarPU takes care of scheduling tasks efficiently, using well-known generic dynamic and task graphs scheduling policies from the literature (see Section 2.2.2), and optimizing data transfers using prefetching and overlapping, in particular. In addition, it allows scheduling experts, such as compiler or computational library developers, to implement custom scheduling policies in a portable fashion.

In this study, we specialize the StarPU scheduling algorithms to include a mixture of static and dynamic task assignments, based on the knowledge of the Cholesky task graph, to improve performance on small and medium size matrices. In the following, we call “small” a matrix with less than 10×10 tiles, “medium” a matrix with tile size between 10 and 20, and “large” a matrix with more than 20×20 tiles.

2.3.2 Chameleon dense linear algebra library

To cope with the increased degree of parallelism, a new class of dense linear algebra algorithms has been proposed, often referred as tile algorithms in the literature [28, 89]. These algorithms led to the design of new libraries in the past five years such as Plasma [28], Flame [72] and DPlasma [19]. Although both static and dynamic versions of the algorithms have been initially implemented, the dynamic codes are now predominant since they proved to provide more flexibility. These dynamic codes rely on runtime systems (Quark [110], Supermatrix [30], PaRSEC [21]) that have been specifically designed for the purpose of the numerical software (in the case of Plasma, Flame and DPlasma, respectively).

The advantage of relying on specialized runtime systems is that they can be optimized for both the numerical algorithm and the target architecture. On the other hand, designing and maintaining a runtime system is a highly time consuming task, which makes it difficult to design a fully-featured specialized runtime system. The Chameleon library is based on the Plasma tile algorithms and code but relies on the StarPU generic runtime system instead of the specialized Quark runtime system. One advantage is that it allows for handling heterogeneous architectures (whereas Plasma and Quark were initially designed for multicore chips). Another advantage when aiming at focusing on the impact of scheduling strategies is that it allows for running in simulation mode with the field-proven combination [99] of StarPU and Simgrid.

2.3.3 Simgrid simulation engine

Simgrid [29] is a versatile simulation toolkit initially designed to study the behavior of large-scale distributed systems like grids, clouds, or peer-to-peer systems. It builds on fluid network models that have been proven as a reasonable alternative to both simple analytic models and expensive, difficult-to-instantiate packet-level simulations.

The Simgrid version of StarPU [99] uses Simgrid to simulate the execution of an application within a single machine. The idea is to run the application normally, except that data transfers and computation kernel calls are replaced by a simple procedure accounting for the time they are expected to take, and gathered coherently by Simgrid. StarPU models each execution unit (CPUs and GPUs) by defining the time taken by each execution unit on each possible task/kernel [14]. It also models the PCI buses between them, using offline bus bandwidth measurements, and relies on Simgrid to compute the interferences on PCI buses between the different transfers.

The resulting simulated times are very close to actual measurements on the real platforms [99], and properly reproduce the various behaviors that can be observed for the various schedulers. This allows one to run experiments with the Simgrid version of StarPU, which provides several advantages:

- The time to simulate execution is reduced, since no actual computation or data transfer is done. The Simgrid simulator itself is not parallel, so the whole execution gets serialized, but several simulations can be run in parallel for e.g. various matrix sizes or schedulers, and one then gets all the results in parallel.
- The experiments do not depend on the availability of the platform, both in terms of quotas, and in terms of versions of the installed software, thus allowing reproducible experiments. This proved useful while performing the experimentation for this very article, since the platform became unavailable for a couple of weeks due to Air Conditioning issues.
- The platform can be modified, for instance to change the available PCI bandwidth, the execution times of the kernels, etc. In Section 2.5.3.2, we use this feature in order to build a virtual "related" heterogeneous platform.

2.4 Lower bounds

Performance results for linear algebra computation are often accompanied with an upper bound in terms of FLOP'S, in order to assess the achieved efficiency. Since the theoretical peak performance is usually unreachable, particularly with GPUs, the common bound being used is the performance of a simple matrix multiplication (GEMM) since this is the most efficient dense linear algebra operation, and thus providing a good hint of some achievable performance. This bound takes into account the heterogeneity of the platform by summing up the obtained GFLOPS on the various processing elements. It however does not take into account the *heterogeneity of the application*, which is particularly important for small and medium matrices, for which a fair amount of the tasks are not GEMMs but much less efficient tasks such as POTRFs, especially on accelerators.

We here propose much more accurate bounds that take into account both heterogeneity of the computation resources and of the application kernels, by taking as input the execution time of any kernel on any type of resource. They also to a certain extent take into account the task graph itself, in terms of task dependencies.

2.4.1 Linear Programming formulation

The lower bound computation is based on a relaxation of the scheduling problem, in which almost all precedence constraints are ignored. This formulation focuses on the number of tasks n_{rt} of each type t (GEMM, SYRK, TRSM, POTRF) which are executed on each resource type r (CPU, GPU, ...). From the Cholesky task graph, we know the number N_t of tasks of each type t that need to be performed, and from the platform we know the number M_r of processing elements of each type r available to schedule the tasks. For each task type t and resource type r, the calibration mechanisms inside StarPU provide the execution time T_{rt} of these tasks on this resource type. The basic area bound is obtained by solving the following linear problem:

$$
\begin{aligned}
& \text{minimize } l \text{ such that} & \\
\forall t, \quad & \sum_r n_{rt} = N_t & \text{(all } N_t \text{ tasks of type } t \text{ get executed)} \\
\forall r, \quad & \sum_t n_{rt} T_{rt} \le l \times M_r & \text{(resources of type } r \text{ complete their tasks)} \\
\forall r, t \quad & n_{rt} \in \mathbb{N}^+ &
\end{aligned}
$$

It is clear that the optimal value l^* of this linear program is a lower bound on the total execution time of the task graph, since any execution needs to execute all tasks. Ignoring the task graph precedences in this bound allows one to handle tasks of the same type with a couple of variables (one per resource type), instead of having one variable for each task in the graph, thus limiting the number of variables and reducing symmetries in the solution space. StarPU is able, without any input from the application beyond the normal task submission, to automatically generate this program and solve it on the fly, right after the application execution, which thus allows one to print this theoretical bound along the measured performance in the application output.

Due to the actual timings of the different task types, this linear program always decides that all POTRF tasks should be executed on CPUs, since all other task types make much more efficient use of the GPU resources. However, in practice all POTRF tasks are on the critical path of the Cholesky graph, and hence this implies that the resulting lower bound is too optimistic for small matrix sizes, since it does not take dependencies into account. This interesting feature of the Cholesky task graph to contain a

path with all n POTRF tasks can be used to strengthen the bound, without adding other variables in the linear program. In addition to the n POTRF tasks, this path contains $n-1$ of the $\frac{n\times(n-1)}{2}$ TRSM tasks, and $n-1$ of the $\frac{n\times(n-1)}{2}$ SYRK tasks. We can thus add the following constraint, which states that the execution time is necessarily larger than the time to execute all these tasks in sequence:

$$\sum_r n_{rP}T_{rP} + (n-1)\times T_T^* + (n-1)\times T_S^* \leq l$$

In this constraint, T_T^* and T_S^* denotes the fastest execution time of TRSM and SYRK tasks: we do not model exactly on which resources these TRSM and SYRK tasks are executed, and thus underestimate their completion times, ignoring which resource they actually run on[1]. The resulting lower bound is called the *mixed bound* in the rest of the chapter. This linear program has a very small number of variables and constraints (in particular, they are independent of the matrix size), and it can thus be solved very quickly.

2.4.2 Constraint Programming formulation

In addition to this lower bound computation, we have used a Constraint Programming formulation of the scheduling problem, in order to obtain good feasible solutions. These solutions provide both a comparison point for StarPU schedules and a limit for possible improvements of the lower bound. The formulation contains one boolean variable b_{ir} for each task i and each resource type r (only one can be true for a given task), and one integer variable s_i for each task i which represents the starting time of the task. The constraints are the following:

$$\begin{aligned}
& \text{minimize } l \text{ such that} & \\
\forall i,\quad & \text{OnlyOne}(b_{i1},\ldots,b_{iR}) & \text{(only one type of resource executes task } i\text{)}\\
\forall i,\quad & s_i + \sum_r b_{ir}T_{ir} \leq l & \text{(task } i \text{ completes)}\\
\forall r,\forall t,\quad & \sum_r b_{ir}T_{ir}\}| \leq M_r & \text{(at time } t \text{ the } M_r \text{ resources of type } r\\
& & \text{are executing at most } M_r \text{ tasks)}\\
\forall i\to j,\quad & s_i + \sum_r b_{ir}T_{ir} \leq s_j & \text{(dependency } i\to j \text{ is respected)}
\end{aligned}$$

We have implemented this constraint programming formulation using CP Optimizer v12.4. The first constraint is expressed using the `alternative` constraint, and the third constraint uses the concept of *cumulative functions* to express the number of tasks which use resources of type r at time t. The other constraints are simple linear constraints and are easily expressed. The solver explores the solution space with an exhaustive search and backtracking, using constraint propagation to reduce the search space as much as possible.

Furthermore, providing the result of a HEFT heuristic as an initial solution allows the solver to explore good solutions more rapidly. We let the solver optimize for 23 hours and keep the best solution

[1] It is possible to include additional variables to the linear program to have more precise values, but this does not provide a better bound unless we take more dependencies into account, which requires adding too many variables and constraints and makes the linear program intractable.

found in this duration. The obtained solutions are quite good compared to what is obtained with other heuristics, but the solver is unable to prove optimality.

Because it would otherwise be extremely costly to solve [2], this formulation does not take into account data transfers. With the usual platforms and the dense linear algebra operation being studied (the Cholesky factorization), data transfers are indeed not a concern: computation is dense enough for transfers to be largely overlapped with kernel computation.

2.4.3 Upper bounds on performance

Lower bounds on execution time also give upper bounds on the performance. Therefore, we have plotted different theoretical performance upper bounds of the Cholesky factorization in Figure 2.2, based on real execution timings of different tasks on the Mirage machine (described in section 2.5.2).

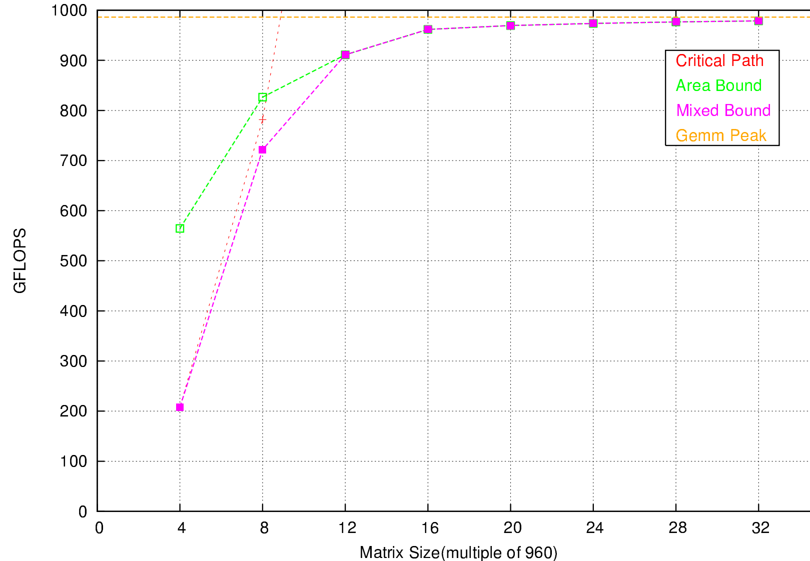


Figure 2.2: Heterogeneous theoretical performance upper bounds

The critical path bound is calculated based on the critical path of the Cholesky task graph. While calculating the critical path, we have taken into consideration the fastest execution time of each task among the different resources. The *area bound* and *mixed bound* calculations are based on the description given in mixed bound subsection 2.4.1. Since GEMM is the fastest kernel of the Cholesky factorization algorithm, we have also plotted the *GEMM Peak*. This plot shows that the *mixed bound* is the tightest upper bound among all upper bounds, and we will therefore compare the performance of our experiments only with the *mixed bound* in the experiment section.

The performance of the constraint programming bound described in section 2.4.2 will be discussed in section 2.5.3.3.

[2] We also have written a version of the constraint programming formulation which takes data transfer times into account but we could not obtain results at the scale of interest for this chapter.

2.5 Experiments and Results

2.5.1 Schedulers

We have experimented with a few schedulers of StarPU, which are representative of state-of-the-art heuristics.

The *random* scheduler assigns tasks randomly over all the computation resources. It uses an estimation of the relative performance of the resources to balance the randomness, so that GPUs will be assigned more tasks, according to their average acceleration ratio. This is thus representative of classical partitioning heuristics, which take into account the heterogeneity of the platform, but do not take into account the heterogeneity of the tasks.

The *dmda* (deque model data aware) and *dmdas* (deque model data aware sorted) schedulers use the minimum completion time heuristic to assign tasks to computational resources: each task is assigned to the processing resource which is estimated to complete it first, taking into account both the estimated computation time on the estimated target resource, and the possible required data transfer time. The difference between *dmda* and *dmdas* is that *dmdas* schedules tasks in order of their priorities, thus making it representative of the state-of-the-art HEFT heuristic [104, 14].

The Cholesky factorization is a structured application, so we can estimate some extra information in advance by analyzing the task graph with the help of different tools. These information could be an exact schedule, priorities for some specific tasks, scheduling of some tasks on a particular worker/resource type, etc. In the following section, we inject more or less of these extra information as static knowledge, to influence the scheduling decisions and get better performance.

2.5.2 Experimental Setup

We have used a machine called Mirage to run and simulate our experiments. It has 2 Hexa-core Westmere Intel® Xeon® X5650 processors and 3 Nvidia Tesla M2070 GPUs. In the actual execution, we used only 9 CPU cores of the mirage machine so that the remaining 3 CPU cores can be used to fully exploit the critical resource (GPUs) of the system. To make the performance comparable we stick to 9 CPU cores in all of our experiments.

2.5.3 Results

We have divided our experiments into two categories based on the types of configuration used. The first one is Homogeneous category where we have run and simulated the performance behavior with 9 homogeneous CPU cores and the second one is Heterogeneous category, where we used 9 CPU cores and 3 GPUs to run the tasks.

From the past literature we found that researchers are getting maximum performance in heterogeneous case with tile size equal to 960 [7], that is why we also kept the same tile size value throughout all our experiments.

For actual executions, we provide the average and standard deviation of 10 runs in the plots. In simulation mode, results are deterministic for all schedulers except for the *random scheduler* which relies on random allocation choices. The simulated plots therefore provide average and standard deviation values of 10 simulations with various seeds for the random scheduler.

2.5.3.1 Homogeneous case

For the homogeneous case, we provide the results of real execution runs of Cholesky factorization with the three different StarPU schedulers: random, dmda and dmdas.

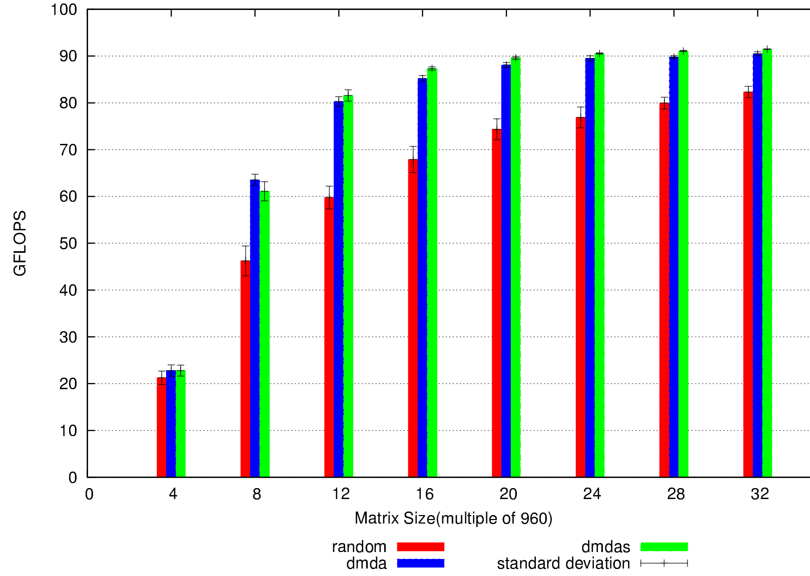


Figure 2.3: Homogeneous actual performance

From Figure 2.3, it is clear that the random scheduler does not perform well. This happens because it does not take into account the already assigned workload of the workers, and just selects a worker among all workers with equal probability. This shows that the scheduler needs to take scheduling decisions in some smart way. The other two schedulers which are based on data aware and early finish time strategies perform much better than the random scheduler. Figure 2.3 also shows that dmdas slightly under-performs compared to dmda for smaller number of tiles. This is due to the fact that dmdas is biased towards the longest path (path with more work) and chooses some tasks in the beginning which do not generate enough level of parallelism. But as time progresses, *dmdas* starts choosing tasks which releases a higher number of tasks, because these tasks would be the critical ones, which improves the overall performance of the execution.

We are also interested to know what the upper bound of the performance is, in order to determine how far these results are from that bound with different types of schedulers. Since actual executions add some runtime overhead and affect the performance, to mitigate this overhead we have compared the bound with simulated performance.

Figure 2.4 shows that the behavior is very similar to the original execution, with a slight increase in performance, since the communication cost was removed from the system. It also shows that the gap between *mixed bound* and achieved performance is significant for small matrices.

Table 2.1: GPUs relative performance

POTRF	TRSM	SYRK	GEMM
$\simeq 2\times$	$\simeq 11\times$	$\simeq 26\times$	$\simeq 29\times$

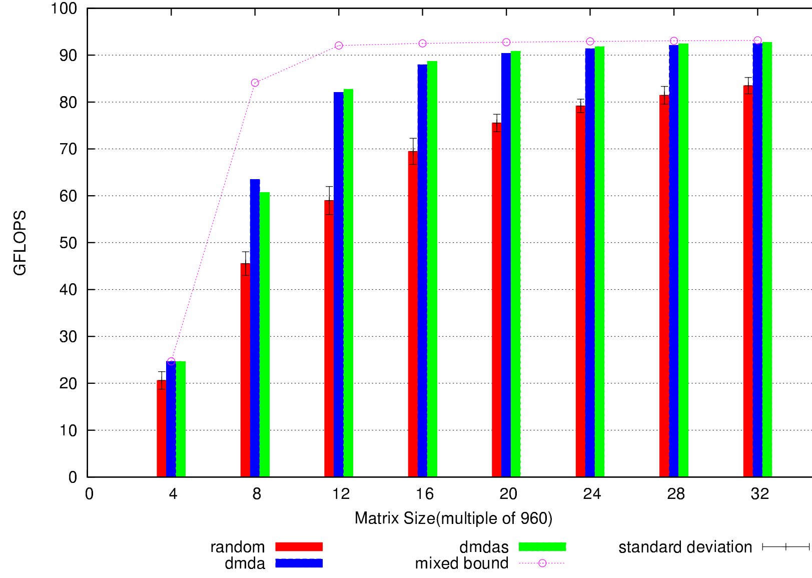


Figure 2.4: Homogeneous simulated performance

2.5.3.2 Heterogeneous Case

In this subsection, we consider all the processing units of the Mirage machine. 9 CPUs and 3 GPUs are used for the execution of tasks while the remaining 3 CPUs are used as drivers for the 3 GPUs. Table 2.1 shows the GPUs performance for each kernel with respect to CPUs performance, e.g.: GEMM is 29 times faster on GPU compared to CPU.

We divide our work into two parts. In the first part, we consider the impact of heterogeneity of resources by considering a heterogeneous platform with related performance. More specifically, we designed a fictitious hardware configuration, where execution time of each kernel on GPU is exactly K times faster than the CPU execution time, and we call this case the *heterogeneous related*. The common accelerator factor K is an average computed as follows :

$$K = \left(\frac{N_P * a_P + N_T * a_T + N_S * a_S + N_G * a_G}{\text{Total Number of Tasks}} \right)$$

where,
N_P: total number of POTRF tasks
a_P: acceleration factor of POTRF on GPU
N_T: total number of TRSM tasks
a_T: acceleration factor of TRSM on GPU
N_S: total number of SYRK tasks
a_S: acceleration factor of SYRK on GPU
N_G: total number of GEMM tasks
a_G: acceleration factor of GEMM on GPU

Here, the acceleration factor depends on the number of tasks and the number of tasks depends on the number of tiles. Therefore, we get different acceleration factors with different number of tiles.

In the second part of our work, we show the achieved performance with the actual hardware with the help of both actual and simulated executions, and we call this case the *heterogeneous unrelated* case.

We are using the *mixed bound* (as explained in Section 2.4.1) to compare the performance. The bounds do not take into account the communication constraints. Therefore, to be fair in the comparison we have used the simulated performance, where communication costs have been removed by modifying the platform file of our machine (one of the good features of the Simgrid version of the StarPU runtime system).

Heterogeneous related case Figure 2.5 shows the simulated performance with different schedulers on the fictitious heterogeneous platform. Here, we can see that the random scheduler performs very poorly because it assigns tasks randomly to the worker without knowing the already assigned workload of workers, which limits the number of ready tasks in the system, and introduces significant idle time on our critical resource (GPUs). We have also computed the *mixed bound* for this fictitious platform. The difference between simulated performance and *mixed bound* is once again significant for small and medium size matrices.

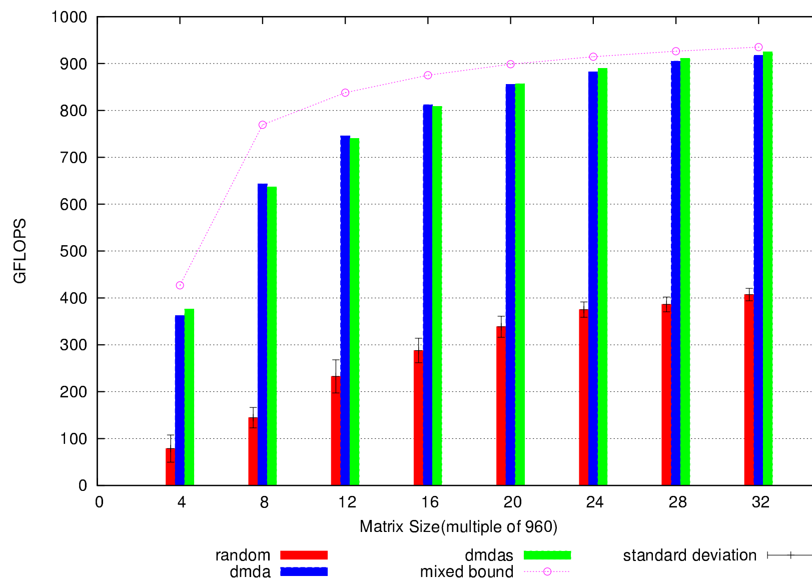


Figure 2.5: Heterogeneous related simulated performance

Heterogeneous unrelated case First we compare the performance of different schedulers in actual execution and then between simulated performance and mixed bound.

As shown in Figure 2.6, in actual executions, the *random* scheduler does not perform well because it is not taking data movement into account while making scheduling decisions: it assigns worker randomly for each task, which may select different resource types for data dependent tasks and result in lots of data movement from CPU memory to GPU memory and vice-versa. In addition, it is also not taking the affinity of tasks to resource (e.g.: GEMM/SYRK is more suitable to be executed on GPU) into account, which degrades the overall performance of the system. The other two schedulers perform comparatively better than the random scheduler because they take into account data transfers when assessing completion time in the HEFT-like scheduling strategies. Here we can also see that *dmda* outperforms *dmdas* performance for smaller matrices, for the same reason as for the homogeneous case (choosing the critical task versus tasks which generate high level of parallelism).

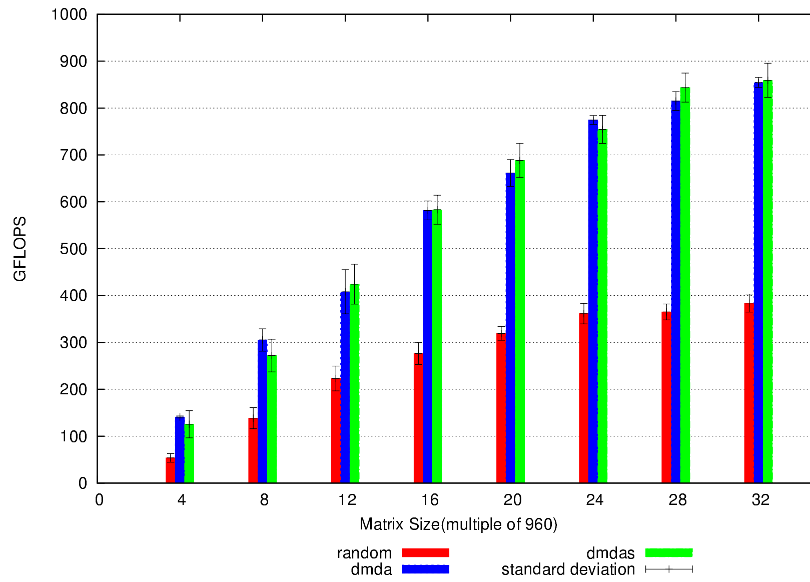


Figure 2.6: Heterogeneous unrelated actual performance

We are now again interested in determining how far we are from the peak performance of the application. Thus, we performed the simulation with different numbers of tiles. Figure 2.7 illustrates the comparison between bounds and achieved performance in simulation. Here we can also see that the performance difference between the best scheduler and the mixed bound is significant for small and medium size matrices.

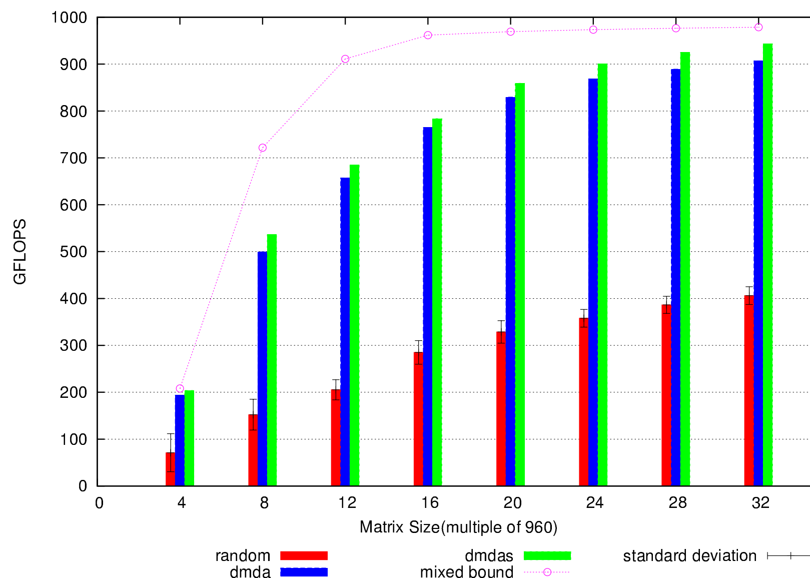


Figure 2.7: Heterogeneous unrelated simulated performance

Comparison between Heterogeneous related and unrelated case In order to determine the impact of heterogeneity of speed-up of tasks on performance, we present a comparison between related

and unrelated heterogeneous simulations. To this end, we scaled the mixed bound of the related case such that it perfectly matches with the mixed bound of the unrelated case, and also scaled all the performance values of the related case with the same factor. The obtained results are given in Figure 2.8, which can now be compared with the unrelated case of Figure 2.7.

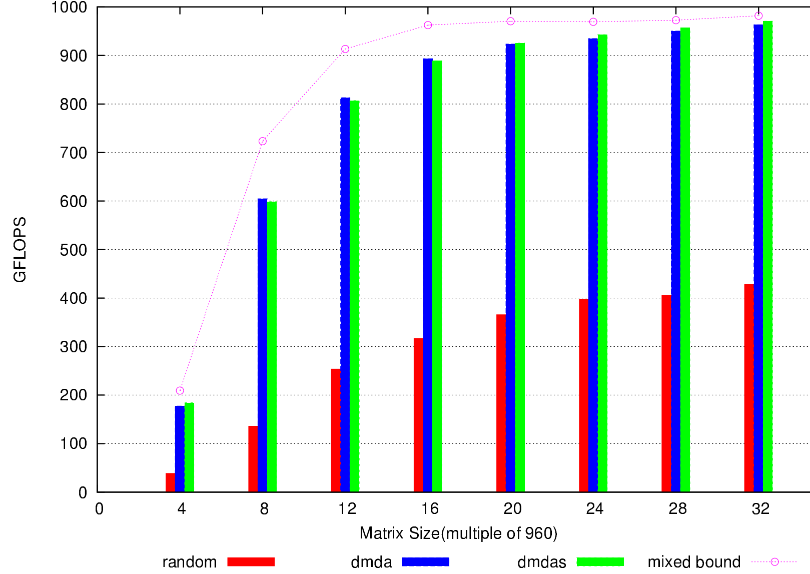


Figure 2.8: Heterogeneous related simulated *scaled performance*

Here we can see that unrelated speed-ups make the problem harder. That is why the gap between state-of-the-art schedulers performance and *mixed bound* is large in Figure 2.7 compared to Figure 2.8. Here, it is also clear that there is room for improvement in the case of small and medium size matrices in the heterogeneous case.

2.5.3.3 Scheduling with static knowledge

The significant gap between performances of StarPU schedulers and the theoretical bound (*mixed bound*) for small and medium size matrices in Figure 2.7 highlights the following things:

- Either the dynamic schedulers of StarPU return a scheduling that can be improved for small matrices;
- Or the theoretical bound is not tight enough;
- Or both.

Indeed, the *dmda* and *dmdas* schedulers take only dynamic decisions to map the ready tasks onto the processors depending on the state of resources and estimation of execution and communication times (also priorities among ready tasks in *dmdas*), without taking into account the overall task graph. These local choices may lead to bad decisions when the parallelism in the task graph is limited. We did some experiments to improve the overall performance (in simulation mode) with static information in the heterogeneous unrelated case.

Since GEMM and SYRK kernels are well suited to execute on GPUs, we enforced these kernels to be executed on GPUs as static information to the StarPU runtime system. This strategy improves the

performance a bit for some matrices in simulation but the performance improvement was not significant and the reason for this is that the StarPU schedulers (*dmda* and *dmdas*) already choose GPUs to execute most of the GEMM and SYRK kernels.

We also analyzed the solution of the *mixed bound* and noticed that a significant portion of the TRSM kernels were mapped onto CPUs. Analyzing traces generated by *dmda* and *dmdas* schedulers reveals that both policies allocate very few TRSMs on CPUs. Since the *mixed bound* does not take all dependencies into account, it is not clear which TRSM kernels should be executed on CPUs in order to improve the performance. On the Mirage machine, with real timings of tasks, we found that the critical path of the Cholesky factorization passes through the diagonal and second diagonal tiles (sequence of *POTRF* $\rightarrow$ *TRSM* $\rightarrow$ *SYRK* $\rightarrow$ *POTRF* ..... $\rightarrow$ *SYRK* $\rightarrow$ *POTRF*). Therefore, we have evaluated the performance in simulation while forcing all the TRSM kernels which are at least k ($1 \leq k <$ *Number of Tiles*) tiles away from the diagonal to be executed on the CPUs (see Figure 2.9) and plotted the best obtained performance in Figure 2.10.

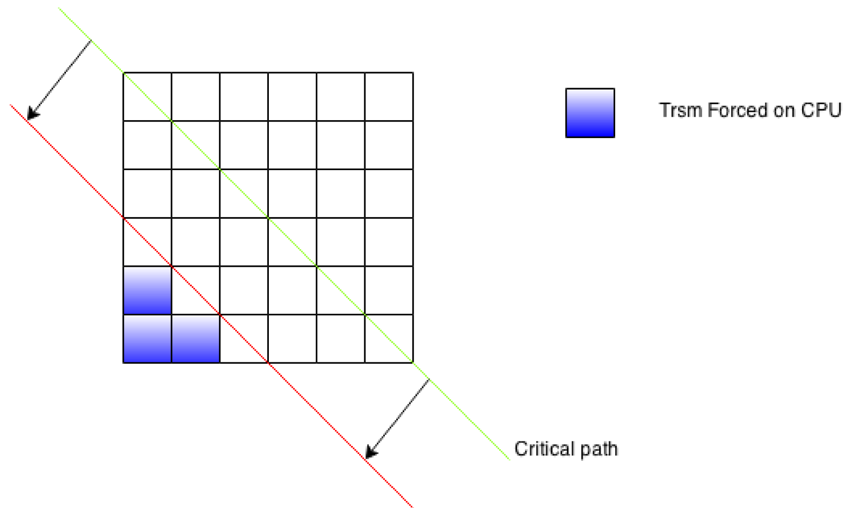


Figure 2.9: TRSMs forced on CPUs

Figure 2.10 shows that providing information about the TRSMs triangular structure statically allows one to achieve better performance than present state-of-the-art schedulers for small and medium size matrices.

We eventually used the constraint programming (CP) described in Section 2.4.2 to find an optimal solution and ran it for 23 hours, but unfortunately we did not manage to get an optimal solution, particularly for large matrix sizes, which produce a very large constraint program. Nevertheless, for reasonable matrix sizes, it provides good and feasible solutions in that span of time. Performance with feasible solution values were better than the values what we are getting with state-of-the-art schedulers for small and medium size matrices. We thus injected the exact schedule in the simulation and it improved the results achieved for small and medium matrix sizes.

Performance improvement obtained in simulation by injecting static information to scheduler motivated us to do some actual execution with static information. Therefore, We did some actual execution by injecting the triangular information, Force all TRSMs on CPUs which are k ($1 \leq k <$ *Number of Tiles*) tiles away from diagonal as static knowledge. Figure 2.11 shows the best obtained performance in actual execution among performance obtained with all possible values of k.

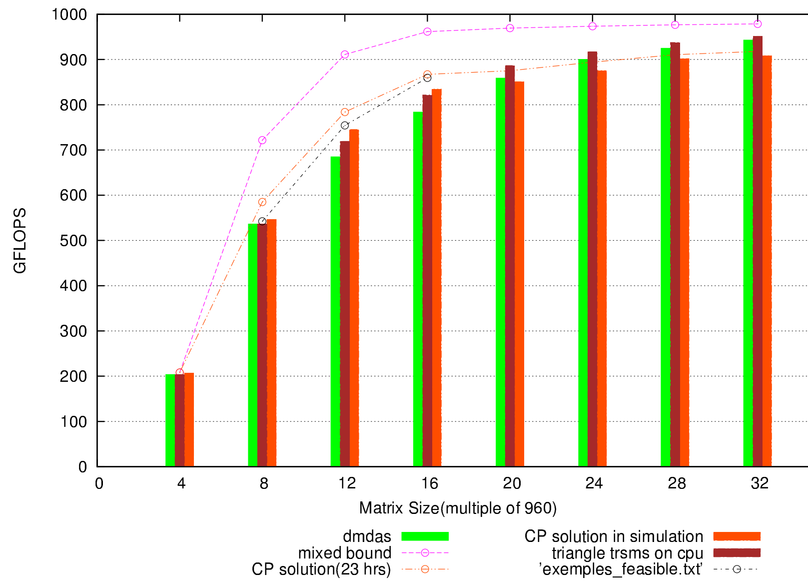


Figure 2.10: Heterogeneous unrelated simulated performance with static knowledge

We also tried to inject the CP schedule in actual execution for small matrices, however we did not get the good performance improvement compared to what we are getting in simulation. The CP formulation indeed does not account data transfers, since as described in Section 2.4.2, solving a CP with data transfers has shown intractable for the purpose at stake. Actual execution with CP schedule thus adds lots of idle time on resources during data transfer, and consequently does not reproduce the same performance in actual execution. The simulated execution has however allowed us to show, at least in the case without data transfers, that some heuristics get relatively close to an achievable CP solution.

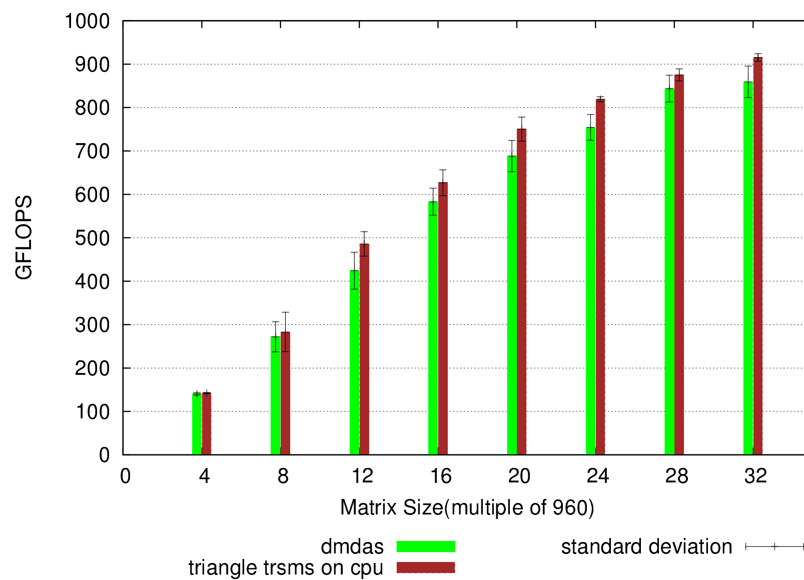


Figure 2.11: Heterogeneous actual performance with static knowledge

2.6 Discussion

2.6.1 dmda vs dmdas scheduler

We were expecting that *dmdas* would always perform better than *dmda* scheduler because it is also taking the HEFT priority into account while making scheduling decision. Nevertheless, we found a few cases where *dmda* outperforms *dmdas*. We investigated the generated trace files with *dmda* and *dmdas* schedulers in order to determine the reasons of this behavior and we found that *dmdas* puts emphasis on critical path rather than parallelism, since it selects some tasks in the beginning which are critical but are not generating enough level of parallelism. That introduces some idle time on the critical resource (GPUs) and degrades the overall performance of the system, which is a known defect of the HEFT scheduler in general. Figure 2.12 and Figure 2.13 show traces with *dmda* and *dmdas* schedulers.

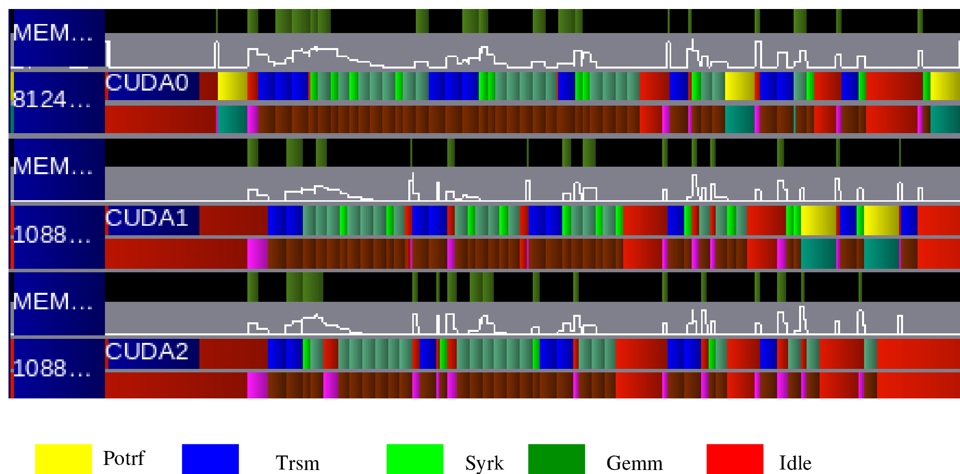


Figure 2.12: GPU trace for 8 $\times$ 8 tiles with dmda scheduler

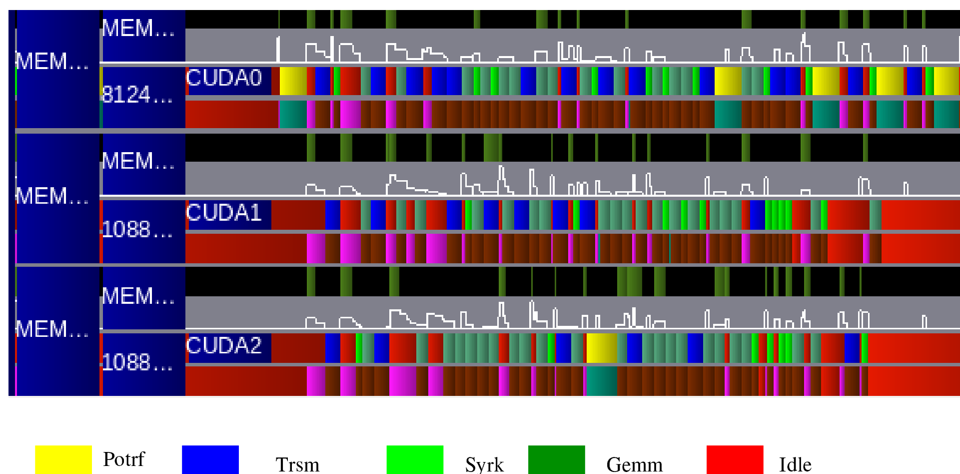


Figure 2.13: GPU trace for 8 $\times$ 8 tiles with dmdas scheduler

2.6.2 Mapping from Constraint Programming solution

We did some experiments by injecting only the mapping information (i.e. only the CPU/GPU information, not the exact task order) of the feasible solution statically obtained by constraint programming, and let the scheduler decide the precise ordering and worker dynamically. This extra information about resource allocation did not improve the performance of the system compared to the performance obtained by *dmda* and *dmdas* schedulers, which indicates that the feasible solution is highly dependent of the precise ordering chosen by constraint programming. This shows that heuristics required to achieve this performance are probably very complex, probably even beyond only backfilling.

2.6.3 Constraint Programming schedule in actual execution

We did some experiments by injecting the schedule obtained by constraint programming in actual execution for smaller matrices, but the performance improvement was not significant compared to the state-of-the art schedulers. After looking into traces we found that resources are idle for significant portion of time during data transfers. One of the prominent way to minimize the idle time on resources due to data transfer is to use prefetching in computational order, but using the prefetching very early also adds significant idle time on resources. Consider two data dependent tasks (second task is dependent on data of first task) schedule on two different workers with different memories, after execution of the first task, second task becomes ready and will initiate the data transfer request but it can be served only when all the already initiated data transfers (some of these data transfers may correspond to task which will be executed very late) by second worker are completed. One of the heuristic to minimize idle time on resource is to use limited prefetching but even this strategy won't solve the problem completely. Since performance in CP schedule is highly dependent on task order of whole schedule, therefore adding idle time on one of the resources may create idle time on other resources as well and degrades the overall performance dramatically.

2.7 Conclusion

In this work, we have bridged the gap between theoretical performance bounds and actually achieved performance on the dense Cholesky factorization. On the former side, we have proposed improved bounds which take into account both resource and task heterogeneity, as well as critical paths. On the latter side, we have introduced some static information into the dynamic task scheduler of StarPU, which brought the performance closer to the theoretical bounds, and very close to what a statically-optimized schedule can achieve. We have also shown that the performance achieved by such statically-optimized schedule depends on precise non-intuitive task ordering, which thus can not be reached by simple list-scheduling heuristics, even with backfilling.

We plan to try to verify the results on other hardware platforms, and apply the same methodology to other dense linear algebra algorithms, but we also plan to try other classes of applications, notably less irregular applications such as sparse linear algebra or FMM.

More generally, this work opens a bridge to closer interaction between applications and tasks schedulers. We have shown that while generic heuristics such as HEFT achieve very good performance, application-specific scheduling hints can noticeably improve performance. We aim at generalizing and formalizing this kind of information, so that scheduling experts can easily analyze achieved performance, optimize the schedule statically, and try to inject more or less application-specific scheduling hints into the scheduler, such as "this proportion of TRSM tasks should be run on CPUs", or "these TRSM tasks

should be run on CPUs", etc. The code produced for the purpose of the study will be reversed in the StarPU runtime system and Chameleon dense linear algebra library.

Part II

Scheduling Under Memory Constraints

Chapter 3

Memory-aware list scheduling for hybrid platforms

In the first part of this thesis, we took two courses of action to improve the performance of linear solvers. We first designed a new hybrid LU-QR factorization suited for distributed platforms. This algorithms was implemented in the PaRSEC framework, relying on its dynamic scheduler to take care of the resources management. We then focused on the dynamic scheduler itself by investigating how adding static information in the StarPU dynamic scheduler can improve its performance. We saw in Chapter 2 that the StarPU runtime system uses the HEFT strategy by default. Although the classical heuristic HEFT performs well on heterogeneous resources in term of completion time, it doesn't take into account the memory consumption. In this chapter, we dig further into the scheduling module of runtime systems by designing new memory-aware alternative to the classical heuristic HEFT. It is indeed important to minimize the maximum memory usage during a computation to avoid out-of-core memory access. Many works have been done on minimizing the memory consumption while traversing a tree-shaped or a general task graph workflow whose tasks require large I/O files. We can find many results in the literature when a single memory is available. But most nowadays machines are heterogeneous, as wee saw in the previous chapter. Multicores are associated with a dedicated accelerator, such as an FPGA or a GPU. These computational units work on their dedicated memory. In this chapter, when consider the problem of scheduling workflows on dual-memory clusters where two independent memories are available. The amount of used memory of each type at a given execution step strongly depends upon the ordering in which the tasks are executed, and upon when communications between both memories are scheduled. At first, we assess the complexity of this problem by focusing, on the simple case where the task graph is tree-shaped, and the mapping of the tasks onto the resources is already given. We establish the complexity of this two-memory scheduling problem, and provide inapproximability results. We then consider the general problem of scheduling an arbitrary workflow where each task can be mapped onto either resource and has a different processing time for each. The memory-aware heuristics provided in this chapter outperform the reference heuristics HEFT and MinMin used in runtime systems on a wide variety of problem instances.

3.1 Introduction

Modern computing platforms are heterogeneous: a typical node is composed of a multi-core processor equipped with a dedicated accelerator, such as an FPGA or a GPU. These two computational units (cores and accelerator) are strongly heterogeneous. To complicates 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. The nodes of the workflow correspond to tasks, and the edges correspond to the dependencies among the tasks. The dependencies are in the form of input and output files: each node accepts a (potentially large) file as input, and produces a set of files, each of them to be processed by a different successor. We consider in this chapter that we have two different processing units at our disposal, such as a CPU and a GPU. For sake of generality, we designate them by a color (namely blue and red). To execute a task on a given resource, the input file and all the output files of the task must fit within the corresponding memory. As the workflow is traversed, tasks are processed on different memories, and capacity constraints on both memory types must be met. In addition, when a task is executed on a different memory than its predecessor, say for example that a task executed on the blue memory has a successor executed on the red memory, a communication from the blue memory to the red memory must be scheduled before the successor can be processed (and again, the input file and all output files of this successor must fit within the red memory). 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 traversal.

In the first part of this chapter (Section 3.4), we consider the restricted problem where the workflow is tree-shaped and each task in the workflow is best suited to a given resource type (say a core or a GPU), and is colored accordingly. In this context, the objective is to determine an efficient traversal that minimizes the maximum amount of memory of each type needed to traverse the whole tree. This work mainly builds upon the pioneering work of Liu, who has studied tree traversals that minimize the peak amount of memory used on a homogeneous system, hence with a single memory type. Liu first restricted to depth-first traversals in [80], before dealing with an optimal algorithm for arbitrary traversals in [81]. In many situations, the optimal traversal is a depth-first traversal, but this is not always the case. An assessment of the relative performance of depth-first traversals versus optimal traversals is proposed by [74]. The main objective of this part is to extend these results to colored trees with two memory types, and tasks belonging to a given type. Clearly, the traversal, i.e., the order chosen to execute the tasks, and to perform the communications, plays a key role in determining which amount of each memory is needed for a successful execution of the whole tree. The interplay between both memories dramatically complicates the scheduling: it is no surprise that the complexity of the problem, that was polynomial with a unique memory, now becomes NP-complete.

In this first part, we concentrate on this difficult bi-criteria optimization problem, and derive several complexity results: NP-completeness of the problem, and inapproximability within a constant (α, β) factor pair of both absolute minimum memory amounts. Here the absolute minimum memory of a given type is computed when assuming an infinite amount of memory of the other type. We also provide a study of depth-first traversals and related variants. We show how to extend Liu's algorithm to compute the best depth-first traversal, which simultaneously minimizes both memory usages. However, while depth-first traversals were natural algorithms with a single memory, they severely constrain the activation of communication nodes with two memories. We show that the optimization problem is still NP-complete when relaxing the firing of communication nodes in depth-first traversal, which leads us to go beyond depth-first traversals and to introduce general heuristics. These heuristics extends Liu's optimal algorithm along various (greedy) decision criteria to trade-off the usage of both memory types. Finally, we assess the performance of all these heuristics using both randomly generated trees, and actual elimination trees that arise from the multifrontal factorization of sparse linear systems.

In the second part of this chapter (Section 3.5), we study a more realistic model with several complications: (i) tasks are not pre-assigned but can be dynamically assigned to either resource; (ii) task graphs are general DAGs rather then trees; and (iii) one aims at optimizing total execution time (or *makespan*) while minimizing memory usage. Here, the objective is makespan minimization, while enforcing that

memory capacities of each type are not exceeded. Given the negative results derived in the first part, there is little hope to design approximation algorithms. We lower our ambition and aim at designing efficient heuristics for this problem, which we validate through an extensive set of simulations for a variety of scientific benchmarks. However, one major theory-oriented contribution of this section is the derivation of an Integer Linear Program (ILP) formulation for the general problem. This linear program turns out very intricate, due to expressing all constraints related to memory usage, and it has a large number of variables and constraints. Still, it enables us to determine the optimal solution for small-size problems, up to 30 tasks, and thereby to assess the optimal performance of our heuristics for small instances.

HEFT [103] is widely used for scheduling scientific workflows on heterogeneous resources. It is an extension of critical-path list-scheduling that schedules the current ready task on the resource that will complete its execution as soon as possible (given already taken scheduling decisions). By considering task completion instead of task initiation, HEFT is able to take CPU speed heterogeneity into account. However, HEFT has no provision to optimize memory usage, even for a single-memory system, and a fortiori for a dual-memory one. Another main contribution of this chapter is to introduce a memory-aware variant of HEFT for dual-memory systems. Similarly, we design a memory-aware variant of MINMIN [25], another reference heuristic for DAGs where the next task to be executed is selected dynamically (rather than according to some static criteria as in HEFT): MINMIN picks the ready task which has the smallest completion time and executes it on the best available processor.

The rest of the chapter is organized as follows. We start with a brief overview of related work in Section 3.2. Then we detail the model and framework in Section 3.3. Section 3.4 is devoted to the complexity study of the restricted problem where the workflow is tree-shaped and the task are initially colored. We also study of depth-first traversals, a first class of (widely-used) heuristics, and introduce additional heuristics. In Section 3.5, we tackle the general problem, by expressing an optimal schedule in terms of the solution of a complex ILP. and introducing new heuristics. Finally, we provide concluding remarks in Section 3.6.

3.2 Related work

3.2.1 Task graph scheduling

Computations with dependencies are naturally modeled through task graphs, where nodes represent computational tasks and edges represent dependencies. Task graph scheduling has been the subject of a wide literature, ranging from theoretical studies to practical ones. On the theoretical side, the most used techniques are list scheduling [33], clustering [86], and task duplication [9]. On the practical side, task graphs have been widely used to model complex workflows in grid computing [48]. Scheduling task graphs on grids is the subject of a wide literature, and many tools exist to manage and schedule such workflows, such as MOTEUR [55]. These tools usually include scheduling heuristics to map workflow tasks onto available resources. These heuristics were often inherited from the task graph scheduling literature, and were more or less adapted to cope with the intrinsic heterogeneity of grid environments. The most famous task graph scheduling algorithm for grids and heterogeneous platforms is HEFT [103], which we use and adapt to our dual-memory context.

3.2.2 Pebble game and its variants

On the more theoretical side, this work builds upon the many papers that have addressed the pebble game and its variants. Scheduling a graph on one processor with the minimal amount of memory amounts to revisiting the I/O pebble game with pebbles of arbitrary sizes that must be loaded into main memory

before *firing* (executing) the task. The pioneering work of Sethi and Ullman [96] deals with a variant of the pebble game that translates into the simplest instance of the problem with a unique memory and where all files have weight 1. The concern in [96] was to minimize the number of registers that are needed to compute an arithmetic expression. The problem of determining whether a general DAG can be traversed with a given number of pebbles has been shown NP-hard by Sethi [95] if no vertex is pebbled more than once (the general problem allowing recomputation, that is, re-pebbling a vertex which have been pebbled before, has been proven PSPACE complete [52]). However, this problem has a polynomial complexity for tree-shaped graphs [96]. Recently, still in the contact of a single memory type, an extension of these results to parallel machines base been proposed in [83].

3.2.3 Scheduling with memory constraints

The problem of scheduling a task graph under memory constraints appears in the processing of scientific workflows whose tasks require large I/O files. Such workflows arise in many scientific fields, such as image processing, genomics or geophysical simulations. The problem of task graphs handling large data has been identified in [91] which proposes some simple heuristic solutions. Most existing theoretical studies are restricted to tree-shaped task graphs, that arise in some application domains such as the factorization of sparse matrices using the multifrontal method [81, 80]. We refer the interested reader to our recent paper [65] for an extended bibliography on adding memory constraints to the problem of scheduling tree-shaped task graphs.

3.2.4 Hybrid computing

Hybrid computing consists in the simultaneous use of CPUs and GPUs to optimize performance for high performance computing. Since CPUs and GPUs are powerful for specific and different tasks, its is natural to schedule tasks on their "favorite" resource, that is, the resource where their execution time is minimal. This has successfully been achieved to increase performance in linear algebra libraries [8, 70]. There also exist software tools that schedule an application composed of tasks with both CPU and GPU implementations on hybrid platforms: for instance, StarPU [15] optimizes the execution time of an application by scheduling its tasks on multiple kinds of resources, based on predictions of execution and data transfer times.

3.3 Model and framework

As stated above, we deal with general task graph traversals on a dual-memory system, where each task can be executed on either of the two processing units, that is, with its associated data in one of either memory. Dependencies are in the form of input and output files: each task accepts a set of files as input from each of its predecessors in the DAG, and produces a set of files to be consumed by each successor node. We start this section by formally writing all the constraints that need to be satisfied during a traversal. Finally, we state the target optimization problem in Section 3.3.3.

3.3.1 Flow and resources constraints

We consider, in this chapter, a dual-memory heterogeneous platform with P_1 identical processors which share the first memory and with P_2 identical processors which share the second memory. For clarity, in the rest of the chapter, the first memory will be referred to as the *blue* memory and the P_1 processors

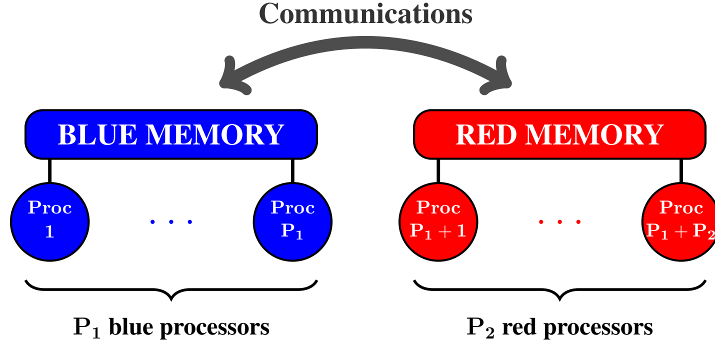


Figure 3.1: Description of the dual-memory platform.

sharing it will be called the *blue* processors. Similarly, the second memory and its processors will be associated to the color *red* as depicted in Figure 3.1.

The application is described by a Directed Acyclic Graph $\mathcal{D} = (V, E)$ composed of $|V| = n$ nodes, or tasks, numbered from 1 to n. We let $Succ(i) = \{j \in V \text{ s.t. } (i, j) \in E\}$ denote the set of successors of i and $Pred(i) = \{j \in V \text{ s.t. } (j, i) \in E\}$ denotes the set of predecessors of i. Dependencies imply a topological order, where a node has to be processed before its successors. Here are some definitions:

- Each task i in the DAG requires a processing time of $W_i^{(1)}$ on one of the *blue* processors and a processing time of $W_i^{(2)}$ on one of the *red* processors.
- Each communication $(i, j) \in E$ is instantaneous if nodes i and j are executed on processors that belong to the same memory. Otherwise, the file produced by node i and needed as input by node j has to be sent from one memory to the other. This transfer takes $C_{i,j}$ time units.

For example, consider the toy example DAG $\mathcal{D}_{ex}$ depicted in Figure 3.2. Task T_1 can be processed in $W_1^{(1)} = 3$ time units on a *blue* processor and in $W_1^{(2)} = 1$ time unit on a *red* processor. If tasks T_1 and T_2 are not executed on the same memory, the communication (T_1, T_2) will take $C_{1,2} = 1$ time unit to be processed. We point out that all communication times are set arbitrarily to 1 in this example (e.g., to account for a high start-up cost). Of course, an affine formula (such as $C_{i,j} = \alpha + \beta F_{i,j}$), or even arbitrary values, can be used in the model.

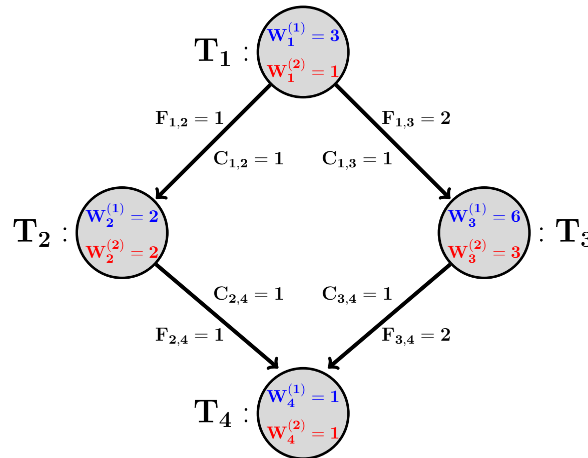


Figure 3.2: Description of $\mathcal{D}_{ex}$.

Given an application DAG, our goal is to determine where each task should be executed (the alloca-

tion) and at what time each task and communication may be started (the starting times). The allocation is described by function $proc : V \to [\![1, P_1 + P_2]\!]$ where $\forall i \in V$, $proc(i)$ represents the index of the processor that processes task i. $proc(i) \leq P_1$ represents a *blue* processor while $proc(i) > P_1$ represents a *red* processor. The starting times are expressed as two functions $\sigma : V \to \mathbb{R}^+$ and $\tau : E \to \mathbb{R}^+$ where $\forall i \in V$, $\sigma(i)$ represents the starting time of task i and $\forall (i, j) \in E$, $\tau(i)$ represents the starting time of communication (i, j).

Let W_i be the actual processing time of task i in the schedule $s = (\sigma, \tau, proc)$:

$$W_i = \begin{cases} W_i^{(1)} \text{ if } proc(i) \leq P_1 \\ W_i^{(2)} \text{ otherwise} \end{cases}$$

We note $COMM_{i,j}$ the actual time taken by communication (i, j) in the schedule $s = (\sigma, \tau, proc)$:

$$COMM_{i,j} = \begin{cases} 0 & \text{if } proc(i) \leq P_1 \text{ and } proc(j) \leq P_1 \\ 0 & \text{if } proc(i) > P_1 \text{ and } proc(j) > P_1 \\ C_{i,j} & \text{otherwise} \end{cases}$$

A schedule $s = (\sigma, \tau, proc)$ of $\mathcal{D}$ is a *valid schedule* if it respects:

- flow dependencies, $\forall (i, j) \in E$:

$$\begin{cases} \sigma(i) + W_i \leq \tau(i, j) \\ \tau(i, j) + COMM_{i,j} \leq \sigma(j) \end{cases}$$

- resource constraints, $\forall (i, j) \in V^2$:

$$proc(i) = proc(j) \implies \begin{cases} \sigma(i) \leq \sigma(j) + W_j \\ \text{or} \\ \sigma(j) \leq \sigma(i) + W_i \end{cases}$$

The makespan of the schedule is the finish time of the last task:

$$Makespan = \max_{i \in V} (\sigma(i) + W_i)$$

Back to the example $\mathcal{D}_{ex}$, on a dual-memory platform with one *blue* processor and one *red* processor ($P_1 = P_2 = 1$), consider the following schedule s_1 depicted in Figure 3.3 for :

$$\begin{cases} \sigma_1(T_1) = 0, \;\; \sigma_1(T_2) = 2, \;\; \sigma_1(T_3) = 1, \;\; \sigma_1(T_4) = 5 \\ \tau_1(T_1, T_2) = 1, \;\; \tau_1(T_2, T_4) = 4 \\ proc_1(T_1) = 2, \;\; proc_1(T_2) = 2, \;\; proc_1(T_3) = 1, \;\; proc_1(T_4) = 2 \end{cases}$$

Schedule $s_1 = (\sigma_1, \tau_1, proc_1)$ is a valid schedule for $\mathcal{D}_{ex}$, with $Makespan = 6$.

3.3.2 Memory constraints

As stated above, in our model, the dependencies are in the form of input and output files. Each node i in the DAG has an input file of size $F_{j,i}$ for each $j \in Pred(i)$. If i is not an input node, its input file is produced by its predecessors; if i is an input node, then it means that $Pred(i) = \emptyset$ and its input files may be of null size, or it may receive input from the outside world. Each non-terminal node i in the DAG, when executed, produces a file of size $F_{i,j}$ for each $j \in Succ(i)$. If i is a terminal node, then $Succ(i) = \emptyset$

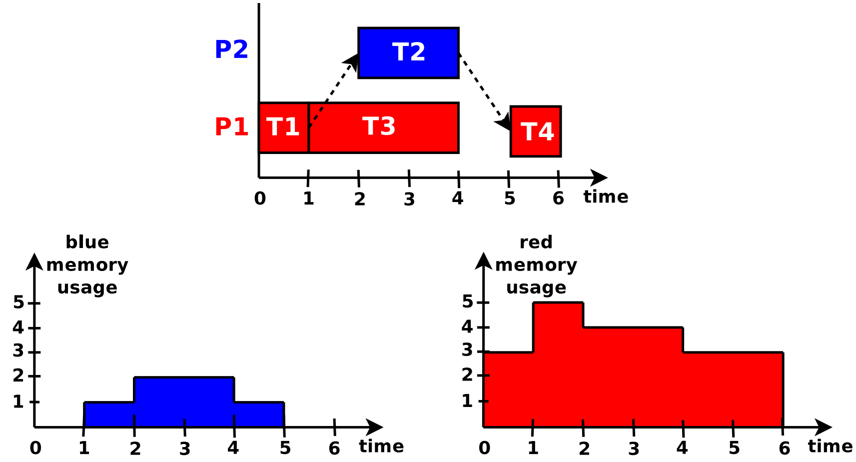


Figure 3.3: Representation of schedule s_1 for $\mathcal{D}_{ex}$.

and i produces a file of null size (we consider that terminal data produced by terminal nodes are directly sent to the outside world).

During the processing of a task i on one of the processors, the memory on which this processor operates must contain all the input and output files. The amount of memory $\mathit{MemReq}(i)$ that is needed for this processing is thus:

$$\mathit{MemReq}(i) = \left(\sum_{j \in \mathit{Pred}(i)} F_{j,i} \right) + \left(\sum_{j \in \mathit{Succ}(i)} F_{i,j} \right)$$

For instance, in $\mathcal{D}_{ex}$, $\mathit{MemReq}(T_3) = F_{1,3} + F_{3,4} = 4$. Note that the memory needed for the execution of the task itself can easily be accounted for, by adding a fictitious predecessor. After task i has been processed, its input files are discarded, while its output files are kept in memory until the processing of its successors. Thus, for a schedule $s = (\sigma, \tau, \mathit{proc})$ of $\mathcal{D}$, if a node i is processed by a *blue* processor, the actual amount of *blue* memory used to process the node i is:

$$\mathit{BlueMemUsed}(s,i) = \left(\sum_{j \in \mathit{Succ}(i)} F_{i,j} \right) + \sum_{e \in S_{\mathit{blue}}} F_e$$

where S_{blue} denotes the set of files (represented by the edges of $\mathcal{D}$) stored in the *blue* memory, when the scheduler decides to execute task i. Note that S_{blue} must contain the input files of task i. After the processing of node i, we have:

$$S_{\mathit{blue}} \leftarrow (S_{\mathit{blue}} \backslash \{(j,i), j \in \mathit{Pred}(i)\}) \cup \{(i,j), j \in \mathit{Succ}(i)\}$$

Of course, the same holds for *RedMemUsed* and S_{red} if i happens to be processed by a *red* processor. Initially, the input file of the root is arbitrarily located in S_{blue}.

Consider the schedule s_1 depicted in Figure 3.3. The execution of task T_1 uses $\mathit{RedMemUsed}(T_1) = F_{1,2}+F_{1,3} = 3$ units of *red* memory. The execution of task T_2 uses $\mathit{BlueMemUsed}(T_2) = F_{1,2}+F_{2,4} = 2$ units of *blue* memory. The execution of task T_3 uses $\mathit{RedMemUsed}(T_3) = F_{1,2} + F_{1,3} + F_{3,4} = 5$ units of *red* memory. And the execution of task T_4 uses $\mathit{RedMemUsed}(T_4) = F_{2,4} + F_{3,4} = 3$ units of *red* memory.

Each time there is a data dependence between two tasks assigned to different memories, the output file of the source task has to be loaded from one memory into the other. During the processing of the communication (i, j), both memories contain the file of size $F_{i,j}$ being copied. Thus, for instance, if i has been assigned on a *blue* processor and j has been assigned on a *red* processor, the amount of *blue* and *red* memory needed for this processing is $F_{i,j}$:

$$\mathit{BlueMemReq}(i, j) = F_{i,j}, \quad \mathit{RedMemReq}(i, j) = F_{i,j}$$

After the communication has been processed, the input file from the *blue* memory is discarded, while the output file is kept in the *red* memory until the processing of j. Thus, for a schedule $s = (\sigma, \tau, \mathit{proc})$ of $\mathcal{D}$, the actual amounts of memory used to process the communication (i, j) are:

$$\mathit{BlueMemUsed}(s, (i, j)) = F_{i,j} + \sum_{e \in S_{\mathit{blue}} \setminus \{(i,j)\}} F_e$$

$$\mathit{RedMemUsed}(s, (i, j)) = F_{i,j} + \sum_{e \in S_{\mathit{red}}} F_e$$

Note that S_{blue} must contain the input file of task i. After the processing of the communication (i, j) we have:

$$S_{\mathit{blue}} \leftarrow S_{\mathit{blue}} \setminus \{(i, j)\}, \qquad S_{\mathit{red}} \leftarrow S_{\mathit{red}} \cup \{(i, j)\}$$

It is important to state that communication (i, j) does not need to be fired right after the execution of task i. The only constraint is that the processing of communication (i, j) must follow the execution of i and precede the execution of j. This flexibility in the schedule severely complicates the search for efficient traversals.

3.3.3 Optimization problem

As stated above, we face an optimization problem under memory constraints. The *memory peak* is the maximum usage of each memory over the whole schedule $s = (\sigma, \tau, \mathit{proc})$ of the DAG $\mathcal{D}$, and is defined for the *blue* and the *red* memory by:

$$M^s_{\text{blue}}(\mathcal{D}) = \max_i \ \mathit{BlueMemUsed}(s, i)$$
$$M^s_{\text{red}}(\mathcal{D}) = \max_i \ \mathit{RedMemUsed}(s, i)$$

In practical settings, the amount of memory at disposal is limited. Let note $M^{(\mathit{blue})}$ and $M^{(\mathit{red})}$ the bounds on the *blue* and the *red* memories. We aim at finding the optimal schedule $s_{\mathit{opt}}(M^{(\mathit{blue})}, M^{(\mathit{red})})$ of the DAG $\mathcal{D}$, defined as the schedule with minimal makespan among all schedules s that does not require more memory than available, i.e., that enforce the bounds on memory peaks: $M^s_{\text{blue}}(\mathcal{T}) \leq M^{(\mathit{blue})}$ and $M^s_{\text{red}}(\mathcal{T}) \leq M^{(\mathit{red})}$.

Back to the schedule s_1 described in Figure 3.3, assume a dual-memory platform with one *blue* processor and one *red* processor. We compute that s_1 uses $M^{s_1}_{\text{blue}}(\mathcal{D}_{ex}) = 2$ units of *blue* memory and $M^{s_1}_{\text{red}}(\mathcal{D}_{ex}) = 5$ units of *red* memory. If we set the memory bounds $M^{(\mathit{blue})} = M^{(\mathit{red})} = 5$, it is clear that s_1 is the optimal schedule. But if we set $M^{(\mathit{blue})} = M^{(\mathit{red})} = 4$, s_1 is no longer an acceptable schedule. In this case, the optimal schedule for $\mathcal{D}_{ex}$ will be s_2, the schedule depicted in Figure 3.4. Schedule s_2 has a smaller memory peak than s_1 but has a larger $\mathit{Makespan} = 7$. This small example illustrates the necessary tradeoff between memory and makespan.

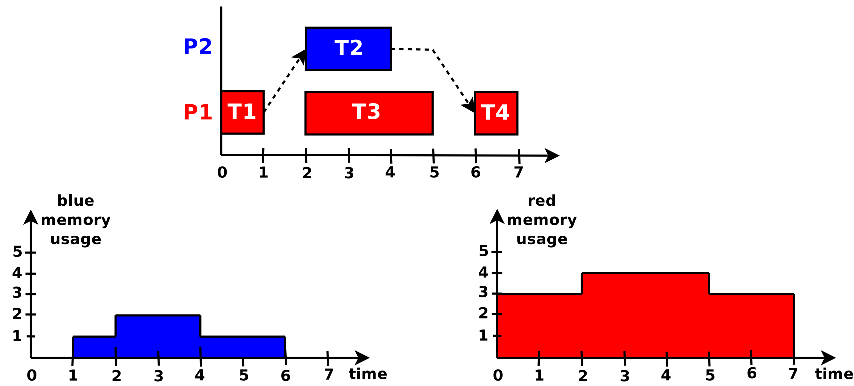


Figure 3.4: Representation of schedule s_2

3.4 Tree traversal with pre-assigned task

In this section, we focus on the restricted problem where the workflow is tree-shaped and each task is pre-assigned onto a resource. Determining a memory-efficient tree traversal is very important in sparse numerical linear algebra. The elimination tree is a graph theoretical model that represents the storage requirements, and computational dependencies and requirements, in the Cholesky and LU factorization of sparse matrices. Memory minimization is still a concern in modern multifrontal solvers when dealing with large matrices. In this section, we want to assess the complexity of minimizing the maximum memory usage when traversing a tree without makespan consideration. Thus, we consider one resource of each type and the mapping of the tasks onto the resources is already given. For commodity reasons, we will talk about colored trees where each task has the color of the resource it is assigned to. Thus, this coloring enforces the mapping $proc$ of the schedule $s = (\sigma, \tau, proc)$. The optimization problem becomes finding the starting time of every tasks and communications such that the *memory peaks* are minimized.

It is interesting to note that there is a complete equivalence between top-down traversals of out-trees (the problem addressed in this section) and bottom-up traversals of in-trees (as used in sparse matrices factorization). In a nutshell, one only needs to reverse the direction of the edges, and to execute the schedule backwards, to move from one variant to another[1]. In fact, the literature deals with both variants. The seminal paper of Liu [80] originally deals with post-order bottom-up traversals for in-trees, while we speak of depth-first top-down traversals for out-trees in this section, but there is no actual difference.

Complexity results for the bi-objective problem are derived in Section 3.4.1. Section 3.4.2 is devoted to the study of depth-first traversals, a first class of (widely-used) heuristics. Then we introduce additional heuristics in Section 3.4.3. The experimental evaluation of all the heuristics is conducted in Section 3.4.4.

3.4.1 Complexity results on trees

This section presents several important complexity results. We start with the NP-completeness of the two-memory minimization problem on colored trees in Section 3.4.1.1. Next we show in Section 3.4.1.2 that the problem reduces to traversing uncolored trees when one memory is unbounded. Finally, we prove in Section 3.4.1.3 that it is impossible to approximate both minimum memories within arbitrary

[1] This equivalence has been formally proven in [74] for single-memory platforms, and it is straightforward to extend the proof for two-memory systems.

constant factors.

3.4.1.1 Hardness of the problem

Our first result assesses the complexity of the problem, as formulated in the following definition.

Definition 3.1 (TWOMEMORYTRAVERSAL)**.** Given a tree $\mathcal{T}$ with n nodes, and two fixed memory amounts M_{red} and M_{blue}, does there exist a traversal σ of the tree such that $M^{\sigma}_{\text{blue}}(\mathcal{T}) \leq M_{blue}$ and $M^{\sigma}_{\text{red}}(\mathcal{T}) \leq M_{red}$?

Theorem 3.1. *The* TWOMEMORYTRAVERSAL *problem is NP-complete.*

Proof. The problem clearly belongs to NP, and the certificate is the ordered list of tasks (of both colors and including uncolored communication nodes) executed by the schedule; it is easy to maintain the amount of each memory required by the schedule, and to check that neither M_{red} nor M_{blue} is exceeded.

To establish the completeness, we use a reduction from the 2-Partition problem [49]. Consider an instance $Inst_1$ of the 2-Partition problem, with n integers $\{a_1, a_2, ..., a_n \parallel \sum_{i=1}^{n} a_i = S\}$. Consider an instance $Inst_2$ of the TWOMEMORYTRAVERSAL, consisting in the tree depicted on Figure 3.5. We set the bounds $M_{red} = 3S$ for the *red* memory and $M_{blue} = 2S$ for the *blue* memory. The construction of $Inst_2$ is polynomial in the size of $Inst_1$.

Assume first that $Inst_2$ has a solution. Any traversal must start with the root B_{root}. After it has been processed, $2S$ units of the *blue* memory are occupied, which means that it is full. Without loss of generality (by symmetry), assume that C is the next node to be executed. Then, we observe that if $C^{(2)}$ was the third executed node, we could never process R_{root} nor $R^{(2)}_{root}$ without violating the M_{red} bound for the *red* memory. Thus, the third executed node has to be R_{root}.

- We observe that the *red* tasks R_{big} and R_{free}, and each communication task C_i, all have to be processed before $R^{(2)}_{root}$, otherwise, since the execution of $R^{(2)}_{root}$ require $3S$ units of memory, it would violate the M_{red} bound for the *red* memory. Besides, since we can not execute $R^{(2)}_{root}$ before R_{big}, if $C^{(2)}$ were processed before R_{big}, there would be at least S units of memory in the *red* memory and the execution of R_{big} (which requires $\frac{5}{2}S$ units of the *red* memory) would violate the M_{red} bound. Thus, the node R_{big} has to be processed before $C^{(2)}$.

- Besides, let i_0 be the index of the first processed task B_i in the traversal. Its execution requires $a_{i_0} + \frac{3}{2}S$ units of the *blue* memory, which implies that it can not be processed before $C^{(2)}$ without violating the M_{blue} bound for the *blue* memory. Thus, the node R_{big} has to be processed before B_{i_0}.

According to the previous arguments, the only tasks that can be processed right after R_{root} and before R_{big} are the communication tasks C_i. Let I be the set of the indices of the tasks C_i executed after R_{root} and before R_{big}.

→ After the execution of R_{root}, there are $2S$ units occupied in the *red* memory and S units in the *blue* memory. Thus, to execute R_{big} without violating the M_{red} bound, the amount of *red* memory to free is at least $\frac{S}{2}$. This means that $\sum_{i \in I} a_i \geq \frac{S}{2}$.

→ Besides, if $\sum_{i \in I} a_i > \frac{S}{2}$, the execution of B_{i_0} (which requires at least $\sum_{i \in I} a_i + \frac{3}{2}S$ units of the *blue* memory) will violate the M_{blue} bound.

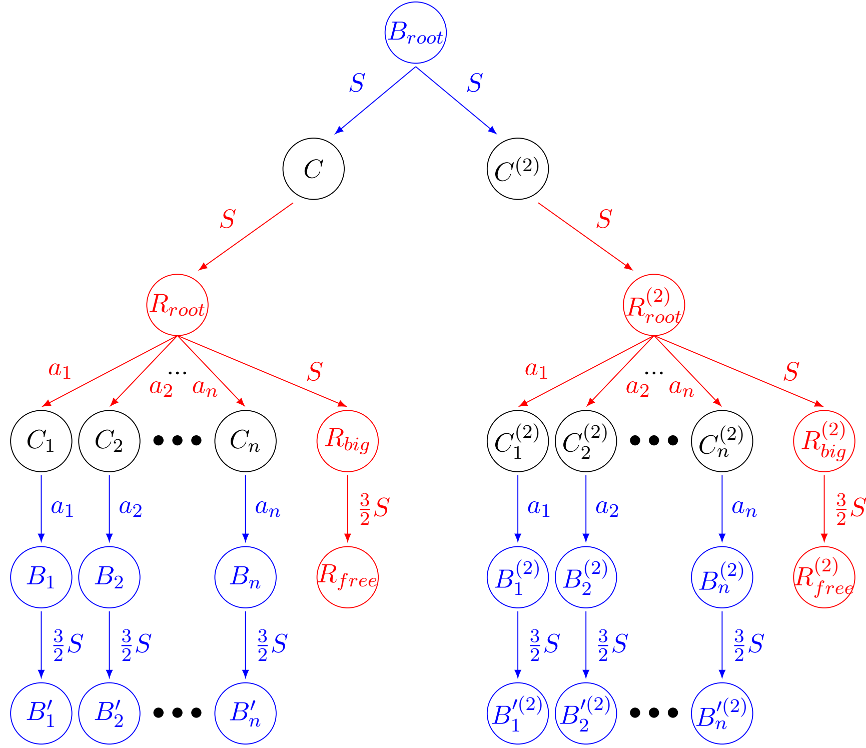


Figure 3.5: Tree used in the proof of Theorem 3.1

Thus, $\sum_{i \in I} a_i = \frac{S}{2}$, which implies that $Inst_1$ has a solution.

Suppose now that $Inst_1$ has a solution I. According to the previous reasoning, the sequence of nodes B_{root}; C; R_{root}; $\forall i \in I\ C_i$; R_{big} and R_{free} can be executed without violating the bounds on memories. After this sequence, there are $\frac{3}{2}S$ units occupied in the *blue* memory and the *red* one is empty. The node $C^{(2)}$ can be processed to load S units from the *blue* memory to the *red* one. Now, one of the *blue* node B_{i_0} with $i_0 \in I$ can be executed without violating the M_{blue} bound, followed by B'_{i_0}. Moreover, we can process every B_i and B'_i for all $i \in I$ to free the *blue* memory. Then, it is possible to execute every branch of C_i down to B'_i for all $i \notin I$. From this point on, we can process the sub-tree rooted at the node $R^{(2)}_{root}$ using the same pattern, which means that $Inst_2$ has a solution and concludes the proof. ■

3.4.1.2 When one memory is unbounded

In this section, we focus on the computation of $M^{\text{opt}}_{\text{red}}(\mathcal{T})$ (or $M^{\text{opt}}_{\text{blue}}(\mathcal{T})$) which represents the minimal peak memory reachable when there is no constraint on the other memory. We show that the computation of $M^{\text{opt}}_{\text{red}}(\mathcal{T})$ and $M^{\text{opt}}_{\text{blue}}(\mathcal{T})$ for a bi-colored tree $\mathcal{T}$ reduces to the computation of the minimal peak memory for an uncolored tree.

Definition 3.2. Given a bi-colored tree $\mathcal{T}$, we construct the corresponding uncolored (or for convenience, single-colored) tree $\mathcal{T}_{\text{blue}}$ by turning every communication node and *red* node into a *blue* node, and by turning every *red* edge of weight f_i into a *blue* edge of weight 0, as depicted in Figure 3.6. We construct the single-colored tree $\mathcal{T}_{\text{red}}$ in a similar way. We let M^{∞}_{blue} denote the minimal amount of memory needed to process $\mathcal{T}_{\text{blue}}$ (and similarly, M^{∞}_{red} for $\mathcal{T}_{\text{red}}$).

The following result is straightforward.

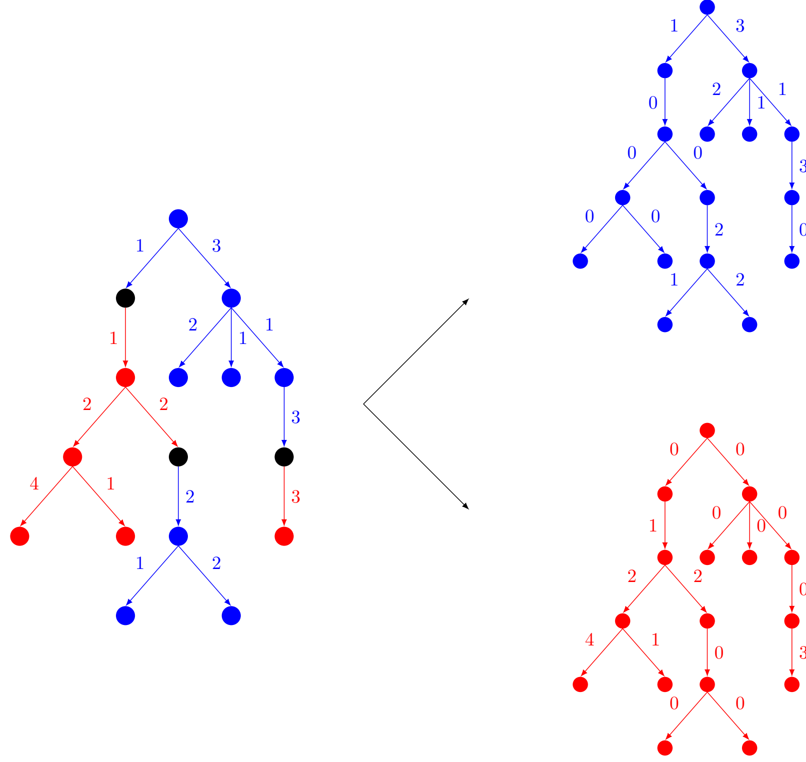


Figure 3.6: A bi-colored tree $\mathcal{T}$ and its corresponding single color trees $\mathcal{T}_{\text{blue}}$ and $\mathcal{T}_{\text{red}}$ in Definition 3.2.

Theorem 3.2. *For any bi-colored tree $\mathcal{T}$, we have $M^{\infty}_{\text{red}} = M^{\text{opt}}_{\text{red}}(\mathcal{T})$ and $M^{\infty}_{\text{blue}} = M^{\text{opt}}_{\text{blue}}(\mathcal{T})$.*

Proof. Given a bi-colored tree $\mathcal{T}$ with n nodes, consider $\mathcal{T}_{\text{blue}}$ and M^{∞}_{blue} as in Definition 3.2. We show here that $M^{\infty}_{blue} = M^{\text{opt}}_{\text{blue}}(\mathcal{T})$. The proof for $M^{\infty}_{red} = M^{\text{opt}}_{\text{red}}(\mathcal{T})$ is similar.

First, $\mathcal{T}$ and $\mathcal{T}_{\text{blue}}$ have the same shape. The only differences between $\mathcal{T}$ and $\mathcal{T}_{\text{blue}}$ are some edge values and the color of some vertices and edges. Thus, to any feasible traversal σ of $\mathcal{T}$, we can associate the corresponding feasible traversal σ_{blue} of $\mathcal{T}_{\text{blue}}$, and reciprocally. For any node $i \in [\![1, n]\!]$ of $\mathcal{T}$, its corresponding node in $\mathcal{T}_{\text{blue}}$ will be referred at as $i_{blue} \in [\![1, n]\!]$, thus $\sigma(i) = \sigma_{blue}(i_{blue})$. Moreover we show that $\mathit{BlueMemUsed}(\sigma, i) = \mathit{MemUsed}(\sigma_{blue}, i_{blue})$ for each node $i \in [\![1, n]\!]$,:

- If $\mathit{color}(i) = \mathit{blue}$, node i is not changed in $\mathcal{T}_{\text{blue}}$ as described in Definition 3.2. Thus, $\mathit{BlueMemReq}(i) = \mathit{MemReq}(i_{blue})$ and the size of the files stored in the memory after i_{blue} has been processed is the same that the files stored in the *blue* memory after i has been processed.

- If $\mathit{color}(i) = \mathit{red}$, then $\mathit{BlueMemReq}(i) = 0$ and no file is stored in the *blue* memory after i has been processed. Besides, for the corresponding node i_{blue} in $\mathcal{T}_{\text{blue}}$, we have $f_{i_{blue}} = 0$ and $f_j = 0$ for each $j \in \mathit{Succ}(i_{blue})$. Thus $\mathit{MemReq}(i_{blue}) = 0$ and no file is stored in the memory after i_{blue} has been processed.

- If i is uncolored (communication node), then $\mathit{BlueMemReq}(i) = f_i$. There are two sub-cases:

  - If i is a communication node from a *blue* node to a *red* node, its processing will store no file in the *blue* memory. According to the Definition 3.2, if j_{blue} denotes the successor of i_{blue}, we have $f_{i_{blue}} = f_i$ and $f_{j_{blue}} = 0$. Thus, $\mathit{MemReq}(i_{blue}) = f_i$ and no file is stored in the memory after i_{blue} has been processed.

- If i is a communication node from a *red* node to a *blue* node, its processing will store a file of size f_i in the *blue* memory. According to the Definition 3.2, if j_{blue} denotes the successor of i_{blue}, we have $f_{i_{blue}} = 0$ and $f_{j_{blue}} = f_i$. Thus, $MemReq(i_{blue}) = f_i$ and a file of size f_i is stored in the memory after i_{blue} has been processed.

During the whole process $BlueMemReq(\sigma, i) = MemReq(\sigma_{blue}, i_{blue})$. Besides, the size of the files stored in the *blue* memory after i has been processed and the size of the files stored in the memory after i_{blue} has been processed are equal. Thus $BlueMemUsed(\sigma, i) = MemUsed(\sigma_{blue}, i_{blue})$ and $M^{\text{opt}}_{\text{blue}}(\mathcal{T}) = M^{\infty}_{blue}$. ∎

3.4.1.3 Joint minimization of both objectives

Since the traversal problem is NP-complete, it is natural to wonder whether there it is possible to get a schedule with guaranteed blue and red peak memories, compared to the optimal ones. In this section, we show that a trade-off must be enforced between these two objectives: indeed, if one wants a strong guarantee on one memory (blue or red), then the produced schedule may be arbitrarily bad for the other memory. More specifically, we prove that there does not exist schedules that can simultaneously approximate both minimum memories $M^{\text{opt}}_{\text{blue}}(\mathcal{T})$ and $M^{\text{opt}}_{\text{red}}(\mathcal{T})$ within arbitrary constant factors, for any bi-colored tree $\mathcal{T}$. Since the (usually unfeasible) point of the Pareto diagram with coordinates $(M^{\text{opt}}_{\text{blue}}(\mathcal{T})$,$M^{\text{opt}}_{\text{red}}(\mathcal{T}))$ is sometimes called the *Zenith* in multi-objective optimization [46], this result amounts to proving that there exists no *Zenith*-approximation.

Definition 3.3. Given a bi-colored tree $\mathcal{T}$, we can construct the corresponding uncolored tree $\mathcal{T}_{\text{unco}}$ by turning every colored node of $\mathcal{T}$ into an uncolored node, as depicted in Figure 3.7. We let $M^{\text{opt}}_{\text{unco}}(\mathcal{T}_{\text{unco}})$ be the minimal amount of memory needed to process $\mathcal{T}_{\text{unco}}$.

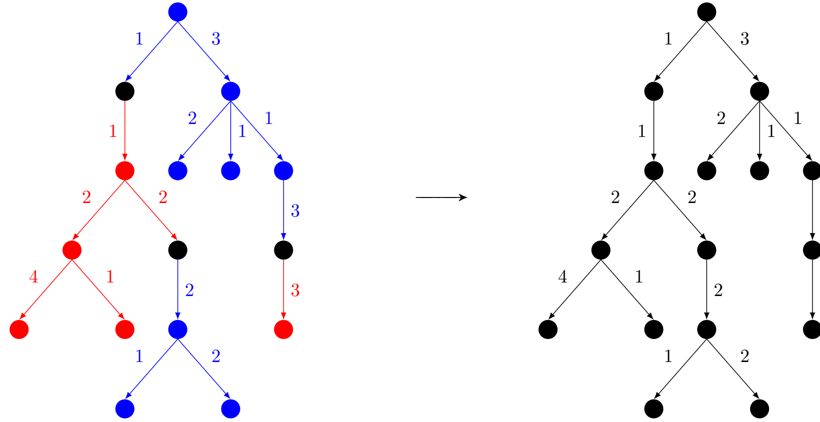


Figure 3.7: A bi-colored tree $\mathcal{T}$ and its corresponding uncolored tree $\mathcal{T}_{\text{unco}}$ in Definition 3.3.

The following lemma is helpful to prove the inapproximability theorem.

Lemma 3.1. *Given a bi-colored tree $\mathcal{T}$ with n nodes, consider an arbitrary traversal σ of $\mathcal{T}$ that requires an amount of* red *memory equal to $M^{\sigma}_{\text{red}}(\mathcal{T})$ and an amount of* blue *memory equal to $M^{\sigma}_{\text{blue}}(\mathcal{T})$. Then necessarily:*

$$M^{\sigma}_{\text{red}}(\mathcal{T}) + M^{\sigma}_{\text{blue}}(\mathcal{T}) \geq M^{\text{opt}}_{\text{unco}}(\mathcal{T}_{\text{unco}})$$

Proof. Let $\mathcal{T}_{\text{unco}}$ be the uncolored tree corresponding to $\mathcal{T}$ as described in Definition 3.3. We observe that $\mathcal{T}$ and $\mathcal{T}_{\text{unco}}$ have the same tasks, hence to any feasible traversal σ of $\mathcal{T}$, we can associate the corresponding feasible traversal σ_u of $\mathcal{T}_{\text{unco}}$, and reciprocally. For any node $i \in [\![1, n]\!]$ of $\mathcal{T}$, its corresponding node in $\mathcal{T}_{\text{unco}}$ will be referred to as $i_u \in [\![1, n]\!]$, thus $\sigma(i) = \sigma_u(i_u)$.

We will show that

$$\forall i \in [\![1, n]\!], \quad \textit{BlueMemUsed}(\sigma, i) + \textit{RedMemUsed}(\sigma, i) = \textit{MemUsed}(\sigma_u, i_u)$$

We proceed along the following case analysis:

- If $\textit{color}(i) = \textit{blue}$, then $\textit{BlueMemReq}(i) = \textit{MemReq}(i_u)$ and $\textit{RedMemReq}(i) = 0$. Besides, no file is stored in the *red* memory after i has been processed; also, the size of the files stored in the *blue* memory after i has been processed is the same as that of the files stored in the memory after i_u has been processed.

- If $\textit{color}(i) = \textit{red}$, then $\textit{RedMemReq}(i) = \textit{MemReq}(i_u)$ and $\textit{BlueMemReq}(i) = 0$. Besides, no file is stored in the *blue* memory after i has been processed; also, the size of the files stored in the *red* memory after i has been processed is the same as that the files stored in the memory after i_u has been processed.

- If i is uncolored (communication node), then $\textit{BlueMemReq}(i) + \textit{RedMemReq}(i) = 2 \times f_i = \textit{MemReq}(i_u)$. Besides, a file of size f_i will be stored in one of the two memories after i has been processed, and a file of size f_i will be stored in the memory after i_u has been processed.

During the whole traversal, we thus have $\textit{BlueMemReq}(\sigma, i) + \textit{RedMemReq}(\sigma, i) = \textit{MemReq}(\sigma_u, i_u)$. The sum of the size of the files stored in the *blue* memory and of the size of the files stored in the *red* memory after i has been processed is always equal to the size of the files stored in the memory after i_u has been processed. Thus $\textit{BlueMemUsed}(\sigma, i) + \textit{RedMemUsed}(\sigma, i) = \textit{MemUsed}(\sigma_u, i_u)$. This means that:

$$\begin{aligned}
M_{\text{unco}}^{\text{opt}}(\mathcal{T}_{\text{unco}}) &\leq M_{\text{unco}}^{\sigma_u}(\mathcal{T}_{\text{unco}}) \\
&= \max_i \textit{MemUsed}(\sigma_u, i) \\
&= \max_i \{\textit{BlueMemUsed}(\sigma, i) + \textit{RedMemUsed}(\sigma, i)\} \\
&\leq \max_i \{\textit{BlueMemUsed}(\sigma, i)\} + \max_i \{\textit{RedMemUsed}(\sigma, i)\} \\
&= M_{\text{red}}^{\sigma}(\mathcal{T}) + M_{\text{blue}}^{\sigma}(\mathcal{T})
\end{aligned}$$

which concludes the proof. ■

Theorem 3.3. *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.*

Proof. To establish this result, we proceed by contradiction. We therefore assume that there is an integer α, an integer β, and an algorithm $\mathcal{A}$ that processes any bi-colored tree $\mathcal{T}$ using a *blue* peak memory that is not greater than α times the optimal *blue* peak memory $M_{\text{blue}}^{\text{opt}}(\mathcal{T})$ and using a *red* peak memory that is not greater than β times the optimal *red* peak memory $M_{\text{red}}^{\text{opt}}(\mathcal{T})$. To derive the contradiction, we use the family of tree $(\mathcal{T}_n)_{n \in \mathbb{N}}$ depicted on Figure 3.8. $\mathcal{T}_n$ is defined recursively using $\mathcal{T}_{n-1}$. To help the reader to visualize $\mathcal{T}_n$, Figure 3.9 represents $\mathcal{T}_2$.

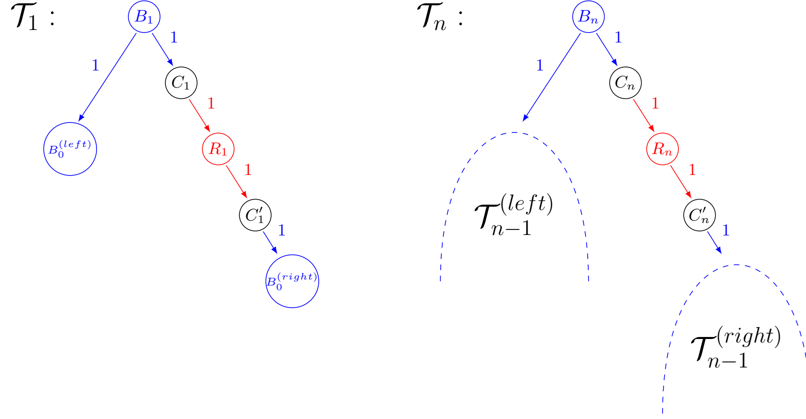


Figure 3.8: Recursive definition of $\mathcal{T}_n$ in the proof of Theorem 3.3

- $\forall \mathbf{n} \geq \mathbf{2}, \mathbf{M}^{\text{opt}}_{\text{blue}}(\mathcal{T}_\mathbf{n}) = \mathbf{3}$
  Consider the traversal σ_{blue} that processes $\mathcal{T}_n$ as follows:
  - If $n = 0$, σ_{blue} processes the node B_0
  - If $n > 0$, σ_{blue} processes the nodes B_n and C_n. Then $\mathcal{T}^{(left)}_{n-1}$ is processed recursively. Nodes R_n and C'_n follow. And finally $\mathcal{T}^{(right)}_{n-1}$ is processed recursively.

  At each step of this process, the traversal σ_{blue} does not use more than 3 units of *blue* memory. Since *BlueMemReq*$(B_{n-1}) = 3$, this proves that $M^{\text{opt}}_{\text{blue}}(\mathcal{T}_n) = 3$.

- $\forall \mathbf{n} \geq \mathbf{1}, \mathbf{M}^{\text{opt}}_{\text{red}}(\mathcal{T}_\mathbf{n}) = \mathbf{2}$
  Consider the traversal σ_{red} that processes $\mathcal{T}_n$ as follows. At step k:
  - If $k = 0$, σ_{red} processes the node B_0
  - If $k > 0$, σ_{red} processes the nodes B_k. Then $\mathcal{T}^{(left)}_{k-1}$ is processed recursively. Nodes C_k, R_k and C'_k follow. And finally $\mathcal{T}^{(right)}_{k-1}$ is processed recursively.

  At each step of this process, the traversal σ_{red} does not use more than 2 units of *red* memory. Since *RedMemReq*$(R_n) = 2$, this proves that $M^{\text{opt}}_{\text{red}}(\mathcal{T}_n) = 2$.

- Let $\mathcal{T}^{\text{unco}}_n$ be the uncolored tree corresponding to $\mathcal{T}_n$ as describe in Definition 3.3 and $M^{\text{opt}}_{\text{unco}}(\mathcal{T}^{\text{unco}}_n)$ the minimum amount of memory required to execute it. $\mathcal{T}^{\text{unco}}_2$ is depicted in Figure 3.9. We now prove by induction that $M^{\text{opt}}_{\text{unco}}(\mathcal{T}^{\text{unco}}_n) = n + 2$ for $n \geq 2$. As show in [81], post-order traversals are optimal for peak memory minimization of uncolored trees with unit costs. Besides, all post-order traversals of $\mathcal{T}^{\text{unco}}_n$ require the same amount of memory. Thus $M^{\text{opt}}_{\text{unco}}(\mathcal{T}^{\text{unco}}_n) = M^{\text{opt}}_{\text{unco}}(\mathcal{T}^{\text{unco}}_{n-1}) + 1$ for $n \geq 2$. Since $M^{\text{opt}}_{\text{unco}}(\mathcal{T}^{\text{unco}}_1) = 2$, we have the result.

By hypothesis, algorithm $\mathcal{A}$ can process any $\mathcal{T}_n$ with $M^{\mathcal{A}}_{\text{blue}}(\mathcal{T}_n) \leq \alpha.M^{\text{opt}}_{\text{blue}}(\mathcal{T}_n) = 3\alpha$ and $M^{\mathcal{A}}_{\text{red}}(\mathcal{T}_n) \leq \beta.M^{\text{opt}}_{\text{red}}(\mathcal{T}_n) = 2\beta$. Let $n_0 = \lceil 3\alpha + 2\beta \rceil$, we have:

$$\begin{aligned} M^{\mathcal{A}}_{\text{blue}}(\mathcal{T}_{n_0}) + M^{\mathcal{A}}_{\text{red}}(\mathcal{T}_{n_0}) &\leq 3\alpha + 2\beta \\ &< \lceil 3\alpha + 2\beta \rceil + 2 \\ &= M^{\text{opt}}_{\text{unco}}(\mathcal{T}^{\text{unco}}_{n_0}) \end{aligned}$$

This contradicts Lemma 3.1, which means that such an algorithm $\mathcal{A}$ cannot exist. ■

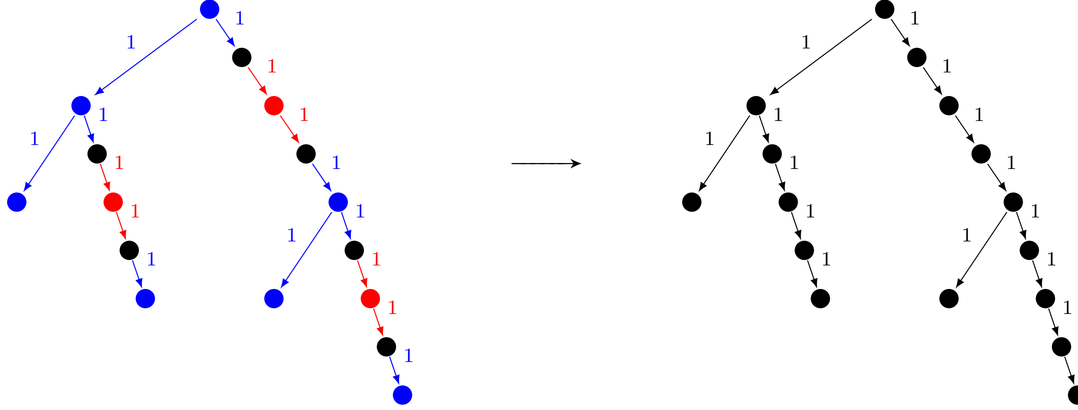


Figure 3.9: $\mathcal{T}_2$ and $\mathcal{T}_2^{\text{unco}}$ in the proof of Theorem 3.3.

3.4.2 Depth-first traversals

In this section, we study depth-first traversals, which are the equivalent of post-order traversals for in-trees. In the context of single-memory trees, depth-first traversals are known to be sub-optimal [81]: worse, their memory usage can be arbitrarily high as compared to that of the optimal solution [74]. Clearly, these negative results remain true in a two-memory framework (simply assume that one memory is infinite). Still, depth-first traversals are a natural heuristic for traversing tree graphs, and they enjoy a simple implementation and memory management. As such, they are the most commonly used traversals in actual sparse solvers like `MUMPS` [10, 11].

We show how to compute the optimal depth-first traversal in Section 3.4.2.1. It turns out that this traversal is optimal for both memory usages (among all depth-first traversals). However, depth-first traversals give no freedom on scheduling communication nodes. If we allow a communication node to be processed not immediately before its sub-tree, the ordering of the processing of the sub-trees and of the communication nodes will create a trade-off between both memory usages and will allow to decrease them. This leads us to define sloppy depth-first traversals, which we study in Section 3.4.2.2.

3.4.2.1 Strict depth-first traversals

Definition 3.4. A depth-first traversal is a feasible traversal that processes all nodes of a tree $\mathcal{T}$ by processing the root and, then, recursively processing all sub-trees. Hence, in a post-order traversal, after processing a node i, the whole sub-tree rooted at i is completely processed before any other node that does not belong to this sub-tree. Formally, a feasible traversal σ of the tree $\mathcal{T}$ with n nodes is a depth-first traversal if and only if for each node $r \in \mathcal{T}$, with two successors $i \in \mathit{Succ}(r)$ and $j \in \mathit{Succ}(r)$, we have:

$$\sigma(i) < \sigma(j) \Rightarrow (\forall u \in T_i, \sigma(u) < \sigma(j))$$

where T_i is the sub-tree rooted at the node i.

In the context of single-memory trees, depth-first traversals are known to be sub-optimal [81]: worse, their memory usage can be arbitrarily high as compared to that of the optimal solution [74]. Clearly, these negative results remain true in a two-memory framework (simply assume that one memory is infinite). Still, depth-first traversals are a natural heuristic for traversing tree graphs, and they enjoy a simple implementation and memory management. As such, they are the most commonly used traversals in actual sparse solvers. Algorithm 6 computes the optimal depth-first traversal: when it encounters a

blue node (respectively a red node), it applies the rule for minimizing the blue (resp. red) memory in depth-first traversals, which does not impact the amount of red (resp. blue) memory. It turns out that this traversal is optimal among all depth-first traversals for both memory usages.

Theorem 3.4. *Algorithm 6 returns the best depth-first traversal σ of $\mathcal{T}$ for both the* blue *and the* red *memories and the amount of memory M^{blue} and M^{red} used by σ.*

Algorithm 6: BestDepthFirstTraversal($\mathcal{T}$)

```
output: Schedule σ with peak blue memory M^blue and peak red memory M^red
root ← the root of T ;
CurrentMem ← 0;
(σ, M^blue, M^red) ← ([root] , 0, 0);
for i ∈ Succ(root) do
    (σ_i, M_i^blue, M_i^red) ← BestDepthFirstTraversal(T_i);
    CurrentMem ← CurrentMem + f_i
if color(root) = blue then
    for i ∈ Succ(root) in the increasing order of M_i^blue − f_i do
        σ ← [σ; σ_i];
        CurrentMem ← CurrentMem − f_i;
        M^blue ← max(M^blue, CurrentMem + M_i^blue);
    M^red ← max_{i∈Succ(root)} M_i^red;
if color(root) = red then
    for i ∈ Succ(root) in the increasing order of M_i^red − f_i do
        σ ← [σ; σ_i];
        CurrentMem ← CurrentMem − f_i;
        M^red ← max(M^red, CurrentMem + M_i^red);
    M^blue ← max_{i∈Succ(root)} M_i^blue;
if the root node is an uncolored communication node then
    i ← the unique successor of root; σ ← [σ; σ_i];
    if color(i) = blue then
        M^blue ← M_i^blue;
        M^red ← max(f_i, M_i^red);
    if color(i) = red then
        M^red ← M_i^red;
        M^blue ← max(f_i, M_i^blue);
return (σ, M^blue, M^red);
```

Proof. Finding the best depth-first traversal of $\mathcal{T}$ amounts to find the best ordering to process every sub-tree. We prove that the order of the recursive processes at each step in Algorithm 6 is the best for both memories.

- At a given step, if the root of the sub-tree is a communication node, we have no choice, and we recursively process the sub-tree rooted at its unique successor.

- At a given step, if the root r of the sub-tree is *blue*, then, the amount of *red* memory used to process this sub-tree will not depend on the order of the recursive processes to complete the sub-tree. Indeed, for each $i \in \mathit{Succ}(r)$, after the recursive process of T_i, $S_{\mathit{blue}} \leftarrow S_{\mathit{blue}} \setminus \{i\}$ and S_{red} is unchanged. Then, independently of the order of the recursive processes of every T_i, the amount of *red* memory required to process $\mathcal{T}$ with a depth-first traversal will be $\mathit{RedMemReq}(\mathcal{T}) = \max_{i \in \mathit{Succ}(root)} \mathit{RedMemReq}(T_i)$. Thus, at this step, we can only optimize the amount of *blue* memory. To do so, we use the optimal post-order traversal for uncolored trees provided by Liu [80]. This post-order traversal leads the best depth-first traversal for the *blue* memory at this step, and, thus, to the best depth-first traversal for both memories.

- At a given step, if the root of the sub-tree is *red*, the proof is similar.

■

3.4.2.2 Sloppy depth-first traversals

As explained in the previous section, the order of the sub-trees processed in a strict depth-first traversal does not influence the maximum usage of *red* memory for a tree rooted at a *blue* node, and vice versa. Thus, in a strict depth-first traversal, both memory usages are independent. This comes from the fact that strict depth-first traversals give no freedom on communications. If we allow a communication node to be processed not immediately before its sub-tree, the ordering of the processing of the sub-trees and of the communication nodes will create a trade-off between both memory usages. This leads us to define sloppy depth-first traversals.

Definition 3.5. A sloppy depth-first traversal is a feasible traversal similar to a depth-first traversal except that, after processing a communication node i, the whole sub-tree rooted at i is not necessarily processed immediately. We define *SloppyChildren*(i) as being the set of thc *red* and *blue* successors of i, together with the successors of the uncolored successors (these represent the set of the computational successors of i). Formally, a feasible traversal σ of the tree $\mathcal{T}$ with n nodes is a sloppy depth-first traversal if and only if for each node $r \in \mathcal{T}$, and for any two nodes $i \in \mathit{Succ}(r)$ and $j \in \mathit{SloppyChildren}(r) \cup \mathit{Succ}(r)$, we have:

$$\sigma(i) < \sigma(j) \Rightarrow (\forall u \in T_i, \sigma(u) < \sigma(j))$$

where T_i is the sub-tree rooted at the node i.

Definition 3.6 (TWOMEMORYSLOPPYDEPTHFIRSTTRAVERSAL)**.** Given a tree $\mathcal{T}$ with n nodes, and two fixed amount of memory M_{red} and M_{blue}, is there a sloppy depth-first traversal of the tree that need an amount of *red* memory inferior to M_{red} and an amount of *blue* memory inferior to M_{blue}?

Theorem 3.5. *The* TWOMEMORYSLOPPYDEPTHFIRSTTRAVERSAL *problem is NP-complete.*

Proof. The problem clearly belongs to NP, and the certificate is the ordered list of tasks (of both colors, and including communication nodes) executed by the schedule.

To establish the completeness, we use a reduction to the 2-Partition problem [49]. Consider an instance $Inst_1$ of the 2-Partition problem, with n integers $\{a_1, a_2, ..., a_n \parallel \sum_i a_i = S\}$. Consider an instance $Inst_2$ of the decision problem, consisting in the tree depicted on Figure 3.10. We set $M_{\mathit{red}} = 2S$ for the *red* tasks and $M_{\mathit{blue}} = 2S$ for the *blue* tasks. The construction of $Inst_2$ is polynomial in the size of $Inst_1$.

Assume first that $Inst_2$ has a solution. Any sloppy depth-first traversal must start with the root B_{root}. After it has been processed, $2S$ units of the *blue* memory are occupied, which means that this memory

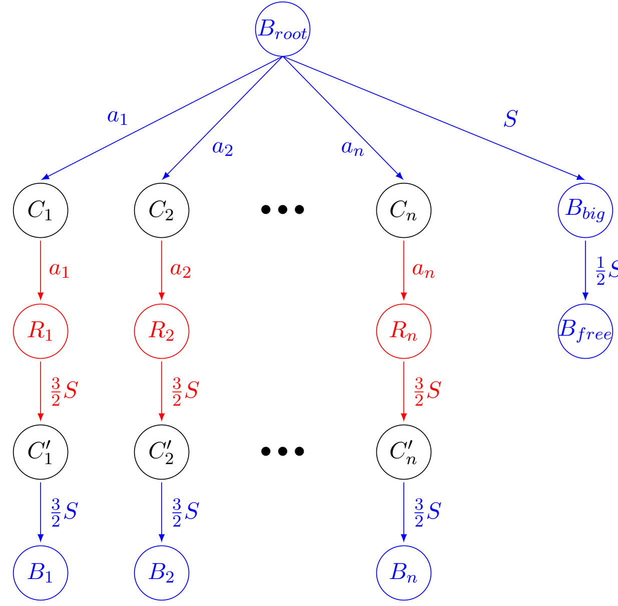


Figure 3.10: Tree corresponding to $Inst_2$ in the proof of Theorem 3.5

is full. Let i_0 be the index of the first *red* task R_i to be executed. We observe that B_{big} and B_{free} have to be processed before R_{i_0}, otherwise the process of C'_{i_0} (which occurs right after the process of R_{i_0} in a sloppy depth-first traversal) would violate the M_{blue} bound on the *blue* memory. Thus, the only tasks that can be processed right after B_{root} and before B_{big} are the communication tasks C_i. Let I be the set of the indices of the tasks C_i executed before B_{big}.

- If $\sum_{i\in I} a_i < \frac{S}{2}$, when the scheduler decides to execute B_{big}, the *blue* memory would be filled with $\sum_{i\notin I} a_i + S$ units. Thus the process of B_{big} will use *BlueMemUsed*$(B_{big}) = \sum_{i\notin I} a_i + S + \frac{3}{2}S > 2S$ units of *blue* memory, which violates the M_{blue} bound.

- If $\sum_{i\in I} a_i > \frac{S}{2}$, when the scheduler decides to execute R_{i_0}, the *red* memory would be filled with at least $\sum_{i\in I} a_i > \frac{S}{2}$ units. Thus the process of R_{i_0} will use at least *RedMemUsed*$(R_{i_0}) \geq \sum_{i\in I} a_i + \frac{3}{2}S > 2S$ units of *red* memory, which violates the M_{red} bound.

Thus, $\sum_{i\in I} a_i = \frac{S}{2}$, which implies that $Inst_1$ has a solution.

Suppose now that $Inst_1$ has a solution I. According to the previous reasoning, the sequence of nodes B_{root}; $\forall i \in I, C_i$; B_{big} and B_{free} can be executed without violating the bounds on memories. After this sequence, there are $\frac{S}{2}$ units occupied in the *blue* memory and in the *red* one. Now, one of the *red* node R_{i_0} with $i_0 \in I$ can be executed without violating the M_{red} bound, followed by C'_{i_0} and B_{i_0}. Moreover, we can process every R_i, C'_i and B_i $\forall i \in I$. Then, one is able to execute every branch of C_i down to B_i for all $i \notin I$, which means that $Inst_2$ has a sloppy depth-first solution and concludes our proof. ■

3.4.3 Heuristics

In addition to depth-first traversals, in this section we present three traversal heuristics which aim at minimizing both the blue and red memories. All three heuristics are based on the seminal work by Liu [81] who considers a single memory. We proposed different adaptation for two memories. We start with the simplest heuristic and then proceed to more elaborate ones.

Working with the uncolored tree: LiuUncolored We have shown that the problem TwoMemoryTraversal of finding a tree-traversal that minimizes both memory is NP-complete. However, when a single memory is considered, the problem becomes polynomial. It is thus natural to adapt the optimal algorithm for the single memory problem proposed by Liu [81], to bi-colored trees. The simplest adaptation amounts to considering the tree as uncolored, that is, as if all tasks were processed on the same computing unit with a single memory. On this uncolored tree, illustrated on Figure 3.6, we apply Liu's optimal algorithm. This heuristic computes an optimal traversal for the sum of the blue and the red memories. The intuition is that minimizing the sum of both memories will lead to a good memory usage for each of them. This heuristic is referred to as LiuUncolored in the following.

Refining the sum with weights: LiuWeightedSum One problem with the previous heuristic is that both memories may not be equivalent. For example, it may well be the case that (input and output) files used by red tasks are much larger that those used by blue tasks. In such a case, minimizing the sum may lead to a much larger amount of blue memory that would be needed, for example, in an optimal traversal for the blue memory. This behavior is not desirable, and we can slightly change the heuristic to (try to) avoid this. We first compute the optimal amount of blue (respectively red) memory that is needed to traverse the tree, as described in Section 3.4.1.2, and we denote this amount by M^{∞}_{blue} (resp. M^{∞}_{red}). Then, we normalize the memory weight of edges as follows: the memory weight f_i of the input edge of node i becomes f_i/M^{∞}_{blue} if this edge is blue, and f_i/M^{∞}_{red} if it is red. Then, the corresponding uncolored tree is considered and Liu's optimal algorithm is applied, as in the previous heuristic. This heuristic is called LiuWeightedSum in the following.

LiuWeightedMax In the previous heuristics, when applying Liu's algorithm to modified trees, we minimize the sum (or the weighted sum) of both memory amounts. However, to get closer to the Zenith point, we would like to minimize the maximum, or rather the weighted maximum of both memories. It is possible to modify Liu's algorithm for this new goal. Of course, the resulting algorithm is not optimal anymore (which is coherent with the NP-completeness of the TwoMemoryTraversal problem), but it can be used as a heuristic. Liu's algorithm is a recursive algorithm which, at each node r, combines optimal traversals for the subtrees rooted at the successors of r into an optimal traversal for the whole tree rooted in r. The combination relies on the definition of "hill-valley" segments: segments are defined by splitting a subtree schedule at different local minima (the "valley"). These segments are then sorted by non-increasing "hill" minus "valley" values (hill being the local peak memory of the segment). Liu [81] proves that such a combination of optimal subtree schedules leads to a global optimal schedule. In this heuristic, we replace the memory criterion used to define of the schedule by the maximum weighted memory: $\max(\frac{\mathit{BlueMemUsed}(\sigma,i)}{M^{\mathrm{opt}}_{\mathrm{blue}}(\mathcal{T})}, \frac{\mathit{RedMemUsed}(\sigma,i)}{M^{\mathrm{opt}}_{\mathrm{red}}(\mathcal{T})})$; we keep the same algorithm for combining subtree schedules. Of course, the proof of optimality does not hold for this new metric. This heuristic is called LiuWeightedMax in the following.

3.4.4 Experiments

In this section, we experimentally compare the memory usage of the heuristics proposed in the previous sections for TwoMemoryTraversal. For each heuristic among BestDepthFirst, LiuUncolored, LiuWeightedSum and LiuWeightedMax, we compute the amount of blue and red memory needed by the traversal. These values are compared to the minimum amount of blue (respectively red) memory needed when the red (resp. blue) memory is unbounded, as described in Section 3.4.1.2.

All heuristics have been implemented in C. The optimal value traversal for a single memory is computed using Liu's algorithm [81] written as a recursive code. Source code for all the algorithms, heuristics and experiments is publicly available at `http://perso.ens-lyon.fr/julien.herrmann/`.

3.4.4.1 Data Sets

We use four different sets of trees, ranging from actual trees arising in sparse matrix computations to random trees. We first describe the data set of uncolored trees which serves as a basis for our realistic colored trees.

Real uncolored trees for Cholesky factorization The UNCOLOREDREALTREES data set contains assembly trees for a set of sparse matrices obtained from the University of Florida Sparse Matrix Collection (http://www.cise.ufl.edu/research/sparse/matrices/). The chosen matrices satisfy the following assertions: not binary, not corresponding to a graph, square, having a symmetric pattern, a number of rows between 20,000 and 2,000,000, a number of non-zeros per row at least equal to 2.5, and a total number of non-zeros at most equal to 5,000,000; and each chosen matrix has the largest number of non-zeros among the matrices in its group satisfying the previous assertions. At the time of testing, there were 76 matrices satisfying these properties. We first order the matrices using MeTiS [77] (through the MeshPart toolbox [53]) and `amd` (available in Matlab), and then build the corresponding elimination trees using the `symbfact` routine of Matlab. We also perform a relaxed node amalgamation on these elimination trees to create assembly trees. We have created a large set of instances by allowing 1, 2, 4, and 16 (if more than 1.6×105 nodes) relaxed amalgamations per node. At the end we compute memory weights and processing times to accurately simulate the matrix factorization: we compute the memory weight n_i of a node as $\eta^2 + 2\eta(\mu - 1)$, where η is the number of nodes amalgamated, and μ is the number of non-zeros in the column of the Cholesky factor of the matrix which is associated with the highest node (in the starting elimination tree); the processing cost w_i of a node is defined as $2/3\eta^3 + \eta^2(\mu-1) + \eta(\mu-1)^2$ (these terms corresponds to one Gaussian elimination, two multiplications of a triangular $\eta \times \eta$ matrix with a $\eta \times (\mu - 1)$ matrix, and one multiplication of a $(\mu - 1) \times \eta$ matrix with a $\eta \times (\mu - 1)$ matrix). Edge weights f_i are computed as $(\mu - 1)^2$.

The resulting 644 trees contains from $2,000$ to $1,000,000$ uncolored nodes. Their depth ranges from 12 to $70,000$, and their maximum degree ranges from 2 to $175,000$.

Real colored trees for Cholesky factorization The REALTREES data set is obtained by coloring every tree in UNCOLOREDREALTREES in a meaningful way. Every tree node in UNCOLOREDREALTREES represents a step of a $(\eta + \mu - 1) \times (\eta + \mu - 1)$ matrix factorization, with a panel of size η. In practice, at each step of the factorization, we aim at processing the GEMM routine (which corresponds to the multiplication of the $(\mu - 1) \times \eta$ matrix with the $\eta \times (\mu - 1)$ matrix) on the GPU. Indeed, GEMMs can reach up to 99% of the GPU's theoretical peak performance. Thus, we split every node into two tasks: a *red* one corresponding to the GEMM routine, and a *blue* one corresponding to the rest of the factorization.

Real trees with random colors The RANDOMCOLOREDREALTREES data set is obtained by randomly coloring every node of every tree in UNCOLOREDREALTREES with an equiprobable choice in the set $\{red, blue\}$. Then, communication nodes are added between nodes of different colors.

Real trees with random weights and colors The RANDOMWEIGHTEDREALTREES data set is obtained by randomly coloring every node of every tree in UNCOLOREDREALTREES with an equiprobable choice in the set $\{red, blue\}$ and by randomly changing the nodes and edges weight. Every n_i is set to a random integer value in $[\![1, \frac{N}{500}]\!]$ where N is the size of the tree, and every f_i is set to a random integer value in $[\![1, N]\!]$. Then, communication nodes are added between nodes of different colors.

Random trees The RANDOMTREES data set is a set of 500 trees with random structure, random weights and random colors. Each tree has been generated as follows: the tree size N is randomly chosen in $[\![1, 32767]\!]$. Then, for each node $i \in [\![1, N]\!]$, its predecessor is randomly chosen in $[\![1, i-1]\!]$. The values of its n_i and f_i are uniformly chosen in $[\![1, 3276]\!]$, and its color is randomly chosen between *red* and *blue*. Then, communication nodes are added between nodes of different colors.

3.4.4.2 Results

In this section, we evaluate the performance of the four heuristics introduced above in terms of memory requirement. For every tree $\mathcal{T}$ in the data sets, and for every traversal σ returned by the heuristics, we compute the maximum relative overhead of each memory compared to the optimal value:
$MaxRelativeOverhead(\sigma, \mathcal{T}) = max(\frac{M^{\sigma}_{\text{blue}}(\mathcal{T}) - M^{\text{opt}}_{\text{blue}}(\mathcal{T})}{M^{\text{opt}}_{\text{blue}}(\mathcal{T})}, \frac{M^{\sigma}_{\text{red}}(\mathcal{T}) - M^{\text{opt}}_{\text{red}}(\mathcal{T})}{M^{\text{opt}}_{\text{red}}(\mathcal{T})})$.
As explained in Section 3.3, the optimal for both memories (also called *Zenith*) is a theoretical bound that may be not reachable. Thus, for a tree $\mathcal{T}$, there does not necessarily exist a traversal σ such that $MaxRelativeOverhead(\sigma, \mathcal{T}) = 0$. Detailed statistics for the four heuristics are given in Table 3.1. We make the following observations:

- For the REALTREES data set, BESTDEPTHFIRST statistically gives the best results, with an average relative overhead equal to 6.3%; it reaches the *Zenith* for 55.6% of the trees. This comes from the particular structure of the assembly trees. Indeed, most nodes in these assembly trees have an input file smaller than the sum of their output files: $f_i \leq \sum_{j \in Succ(i)} f_j$. This means that when we execute a node, it is more likely to be profitable to execute the whole subtree straightaway. This is why BESTDEPTHFIRST turns out to be the best heuristic for the REALTREES and the RANDOMCOLOREDREALTREES data sets. Besides, LIUUNCOLORED is very close to the BESTDEPTHFIRST performances on the REALTREES data set, with an average relative overhead equal to 6.6%. On the contrary, LIUWEIGHTEDMAX appears to be not well-designed for the structure of the assembly trees in REALTREES, with an average relative overhead equal to 8.4%; it can require up to 2.16 times the optimal memory for some trees.

- The structure of the trees in the RANDOMCOLOREDREALTREES data set is close to the trees in REALTREES, and the results are similar. BESTDEPTHFIRST statistically gives the best results with an average relative overhead equal to 3.8%, and LIUUNCOLORED provides the second best results with relative overhead equal to 5.2%.

- For the RANDOMWEIGHTEDREALTREES data set, file sizes are randomized, and BESTDEPTHFIRST is no longer adapted to such trees; it provides an average relative overhead equal to 20.9%. Much worse, LIUUNCOLORED can require up to 5.13 times the optimal memory for some trees in RANDOMWEIGHTEDREALTREES. On the contrary, LIUWEIGHTEDMAX appears to be well-designed for the trees in RANDOMWEIGHTEDREALTREES, with an average relative overhead two times lower than that of BESTDEPTHFIRST.

- The results for the RANDOMTREES data confirm that LIUWEIGHTEDMAX is the best of the four heuristics when dealing with trees with random structure. It gives the best results with an average

relative overhead equal to 3.4%, and exhibits a relative overhead inferior to 10% for 92% of the random trees.

Figures 3.12, 3.13 and 3.14 provide complete results of the simulations. In each figure, a point represents one scenario (one heuristic executed on one tree of the data set). To better visualize the distribution, we also plot a "cross" for each heuristic: the center of this cross is the average result, while the branches represent the scope of each objective between the $10th$ and $90th$ percentile of the distribution.

For the REALTREES data set, as explained above, we colored in *red* the nodes corresponding to the GEMM routine, and in *blue* the others nodes. Thus, every *red* nodes appears to have a communication node as predecessor, and an unique communication node as successor. With this structure, all of our heuristics gives the optimal memory usage for the *red* memory. This specification fits well with practice, where one aims at not overloading the GPU memory. Figure 3.11 provides the detailed distribution of the *blue* memory usage for the heuristics.

These figures exhibit the same trends for average values as observed in Table 3.1. For the RANDOMCOLOREDREALTREES data set in Figure 3.12, and for the RANDOMTREES data set in Figure 3.14, we see that many traversals returned by the heuristics are optimal for at least one of the two memories, whereas for the RANDOMWEIGHTEDREALTREES data set in Figure 3.13, many more of the returned traversals are non-optimal for either memory. We also observe that LIUUNCOLORED can require around 5 times the optimal red memory in two scenarios. These results show that the performance of the heuristics are strongly related to the structure of the trees. While BESTDEPTHFIRST achieves nice results for the realistic assembly trees, LIUWEIGHTEDMAX appears to be a better solution when dealing with more random structures.

Data set	Algorithm	Avg.	Max.	Std. Dev.	Frac. of Opt.	Frac. $\leq$ 10%
REALTREES	BESTDEPTHFIRST	6.3%	64.4%	8.0%	55.6%	73.7%
	LIUWEIGHTEDMAX	8.4%	116.5%	9.9%	49.8%	68.3%
	LIUWEIGHTEDSUM	7.5%	76.0%	9.1%	52.8%	70.6%
	LIUUNCOLORED	6.6%	60.0%	8.3%	55.0%	73.8%
RANDOMCOLOREDREALTREES	BESTDEPTHFIRST	3.8%	44.0%	5.4%	67.2%	83.9%
	LIUWEIGHTEDMAX	6.0%	52.3%	7.2%	51.4%	75.5%
	LIUWEIGHTEDSUM	5.9%	52.6%	7.3%	54.1%	75.8%
	LIUUNCOLORED	5.2%	52.6%	6.9%	59.7%	78.0%
RANDOMWEIGHTEDREALTREES	BESTDEPTHFIRST	20.9%	90.3%	18.6%	28.3%	44.6%
	LIUWEIGHTEDMAX	10.2%	88.2%	13.6%	39.8%	72.7%
	LIUWEIGHTEDSUM	13.4%	107.5%	16.3%	37.7%	65.2%
	LIUUNCOLORED	15.4%	413.1%	17.0%	26.5%	60.2%
RANDOMTREES	BESTDEPTHFIRST	4.5%	28.2%	4.3%	33.4%	83.4%
	LIUWEIGHTEDMAX	3.4%	23.5%	3.2%	26.0%	92.0%
	LIUWEIGHTEDSUM	4.4%	21.4%	3.7%	20.6%	86.0%
	LIUUNCOLORED	6.8%	32.9%	4.8%	14.6%	72.6%

Table 3.1: Statistics on the maximum relative overhead for each memory required by the four heuristics (comparison with the Zenith). Frac. of Opt. (respectively Frac $\leq$ 10%) counts the fractions of cases when the heuristics achieve the Zenith (resp. has a degradation not larger than 10%).

3.5 General problem

The simplified model used in previous section was useful to assert the intrinsic difficulty of the problem: it is NP-complete to decide whether there exists a tree traversal that satisfies bounds on each memory usage: worse, it is impossible to approximate within a constant factor pair both absolute minimum memory amounts. All theses results, although negative, laid the foundations of scheduling for dual-memory systems. In this section, we envision the fully general framework introduced in Section 3.2, where tasks

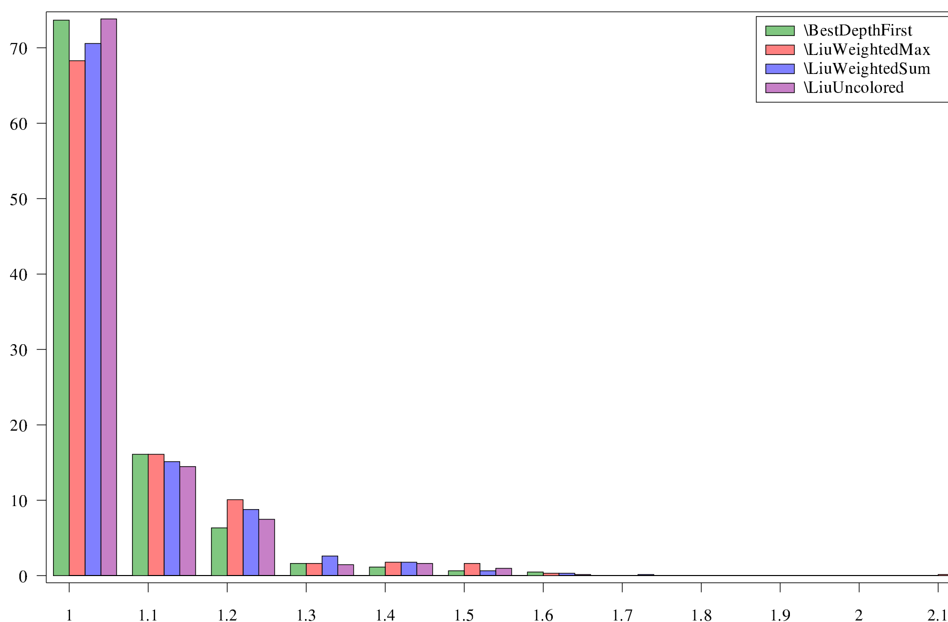


Figure 3.11: Percentage distribution of the *blue* memory usage for the REALTREES data set.

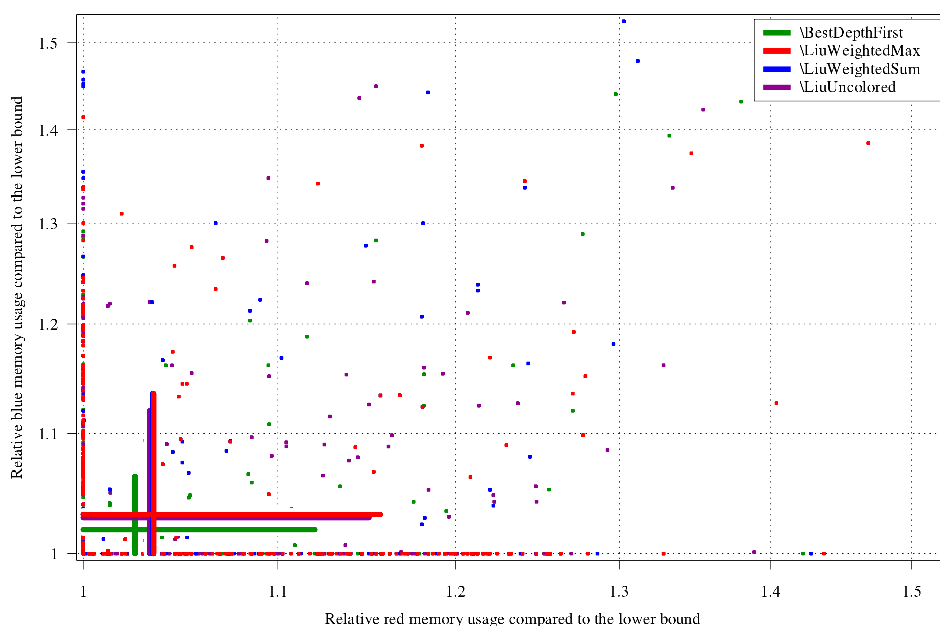


Figure 3.12: Distribution of each memory usage for the RANDOMCOLOREDREALTREES data set.

have different execution-times for each resource type (instead of being tied to a given resource as in previous section), and where concurrent execution of several tasks on each resource type is possible (instead of the fully sequential processing of the task graph that is assumed in previous section).

Section 3.5.1 is devoted to expressing an optimal schedule in terms of the solution of a complex ILP. We introduce the new heuristics in Section 3.5.2, and assess their performance through an extensive set of simulations in Section 3.5.3.

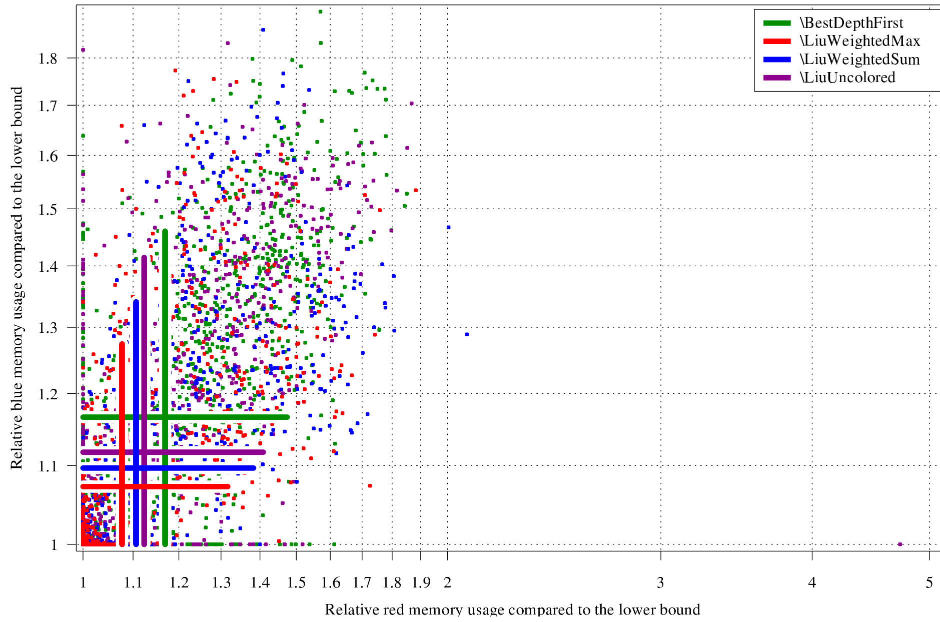


Figure 3.13: Distribution of each memory usage for the RANDOMWEIGHTEDREALTREES data set.

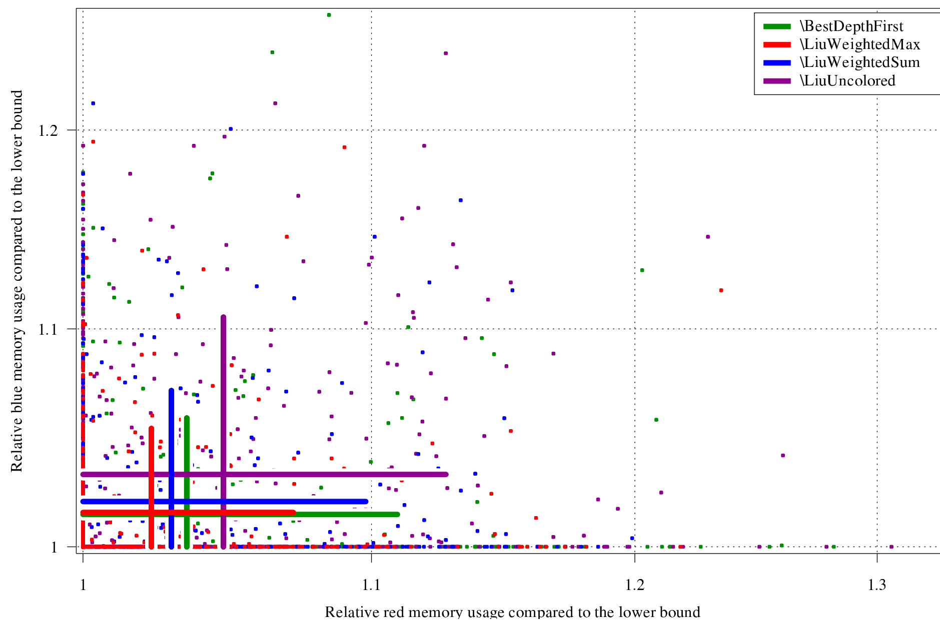


Figure 3.14: Distribution of each memory usage for the RANDOMTREES data set.

3.5.1 ILP formulation

In this section, we describe how to compute an optimal schedule σ_{opt} through a computationally expensive ILP (Integer Linear Program). The objective is twofold: (i) to provide an optimal solution for small instances and (ii) to compare the heuristics presented in the following section with the optimal schedule, to evaluate their absolute quality.

Our approach is motivated by the successful attempt to derive such an ILP formulation for several variants of the DAG scheduling problems, such as [105, 35]. However, to the best of our knowledge,

none of the existing ILP handles the memory usage of the schedule. A major contribution of this chapter is the introduction of additional constraints that enforce memory constraints, as those described in Section 3.3.2.

The variables used by our linear program are listed in Figure 3.15. The t_i's and τ_{ij}'s variables represent the starting time of the tasks and of the communications. M is the makespan value to minimize. The p_i's and b_i's variables describe the allocation of task i on the resources and are used to compute the value of the w_i's variables, which represent the actual computing time of task i. The ϵ_{ij}'s and δ_{ij}'s variables are used to enforce resources constraints. Finally, to compute the amount of memory used by the schedule at any time, we need to know the order in which all tasks and communications are processed. This is achieved through variables σ_{ij}, σ'_{kij}, m_{ij}, m'_{kij}, c_{ijk}, c'_{ijkp}, d_{ijk} and d'_{ijkp}. These numerous variables are needed to ensure that the schedule is properly defined, and that we precisely know which tasks are processed and which data are present in a given memory at any time, to ensure that the memory usage is kept below the prescribed bound.

M	makespan of the corresponding schedule
t_i	starting time of task i
τ_{ij}	starting time of communication (i, j)
p_i	index of the processor where the task i is to be executed
b_i	equal to 0 if task i is executed on the *red* memory and 1 if it is executed on the *blue* memory
w_i	actual computing time of task i in the corresponding schedule
ϵ_{ij}	equal to 1 if the processor index of task i is strictly less than that of task j and 0 otherwise
δ_{ij}	equal to 1 if task i and task j are executed on the same memory and 0 otherwise
σ_{ij}	1 if task i finishes before task j starts and 0 otherwise
σ'_{kij}	equal to 1 if task k finishes before communication (i,j) starts and 0 otherwise
m_{ij}	equal to 1 if task i starts before task j starts and 0 otherwise
m'_{kij}	equal to 1 if task k starts before communication (i,j) starts and 0 otherwise
c_{ijk}	equal to 1 if communication (i, j) starts before task k starts and 0 otherwise
c'_{ijkp}	equal to 1 if communication (i, j) starts before communication (k, p) starts and 0 otherwise
d_{ijk}	equal to 1 if communication (i, j) finishes before task k starts and 0 otherwise
d'_{ijkp}	equal to 1 if communication (i, j) finishes before communication (k, p) starts and 0 otherwise

Figure 3.15: Variables of the linear program

Due to the numerous variables that describe a schedule, the linear program counts a large number of constraints to ensure that these variables correspond to their definition given in Figure 3.15. For the sake of completeness, we give the whole linear program in Figure 3.16, and we detail the most significant constraints below.

Constraints (1) to (25) describes a schedule of the DAG onto the heterogeneous platform, and have nothing to do with memory constraints. They also ensure that communication times are respected when

$$
\begin{array}{rll}
\min_{t,p,\sigma,\epsilon} & M & \\
\forall i \in V, & t_i + w_i \leq M & (1) \\
\forall (i,j) \in E, & t_i + w_i \leq \tau_{ij} & (2) \\
\forall (i,j) \in E, & \tau_{ij} + (1-\delta_{ij})C_{ij} \leq t_j & (3) \\
\forall i \neq j \in V, & t_j - t_i - m_{ij}M_{max} \leq 0 & (4a) \\
\forall i \neq j \in V, & t_j - t_i + (1-m_{ij})M_{max} \geq 0 & (4b) \\
\forall k \in V, \forall (i,j) \in E, & \tau_{ij} - t_k - m'_{kij}M_{max} \leq 0 & (5a) \\
\forall k \in V, \forall (i,j) \in E, & \tau_{ij} - t_k + (1-m'_{kij})M_{max} \geq 0 & (5b) \\
\forall i \neq j \in V, & t_j - t_i - w_i - \sigma_{ij}M_{max} \leq 0 & (6a) \\
\forall i \neq j \in V, & t_j - t_i - w_i + (1-\sigma_{ij})M_{max} \geq 0 & (6b) \\
\forall k \in V, \forall (i,j) \in E, & \tau_{ij} - t_k - w_k - \sigma'_{kij}M_{max} \leq 0 & (7a) \\
\forall k \in V, \forall (i,j) \in E, & \tau_{ij} - t_k - w_k + (1-\sigma'_{kij})M_{max} \geq 0 & (7b) \\
\forall k \in V, \forall (i,j) \in E, & t_k - \tau_{ij} - c_{ijk}M_{max} \leq 0 & (8a) \\
\forall k \in V, \forall (i,j) \in E, & t_k - \tau_{ij} + (1-c_{ijk})M_{max} \geq 0 & (8b) \\
\forall (k,p) \neq (i,j) \in E, & \tau_{kp} - \tau_{ij} - c'_{ijkp}M_{max} \leq 0 & (9a) \\
\forall (k,p) \neq (i,j) \in E, & \tau_{kp} - \tau_{ij} + (1-c'_{ijkp})M_{max} \geq 0 & (9b) \\
\forall k \in V, \forall (i,j) \in E, & t_k - \tau_{ij} - (1-\delta_{ij})C_{ij} - d_{ijk}M_{max} \leq 0 & (10a) \\
\forall k \in V, \forall (i,j) \in E, & t_k - \tau_{ij} - (1-\delta_{ij})C_{ij} + (1-d_{ijk})M_{max} \geq 0 & (10b) \\
\forall (k,p) \neq (i,j) \in E, & \tau_{kp} - \tau_{ij} - (1-\delta_{ij})C_{ij} - d'_{ijkp}M_{max} \leq 0 & (11a) \\
\forall (k,p) \neq (i,j) \in E, & \tau_{kp} - \tau_{ij} - (1-\delta_{ij})C_{ij} + (1-d'_{ijkp})M_{max} \geq 0 & (11b) \\
\forall i,j \in V, & p_j - p_i - \epsilon_{ij}|P| \leq 0 & (12a) \\
\forall i \neq j \in V, & p_j - p_i - 1 + (1-\epsilon_{ij})|P| \geq 0 & (12b) \\
\forall i \in V, & p_i - |P_0| - |P|b_i \leq 0 & (13a) \\
\forall i \in V, & p_i - |P_0| - 1 + (1-b_i)(|P|+1) \geq 0 & (13b) \\
\forall i,j \in V, & m_{ij} + m_{ji} \geq 1 & (14) \\
\forall i,j \in V, & \sigma_{ij} + \sigma_{ji} \leq 1 & (15) \\
\forall (i,j) \in E, \forall k \in V, & m'_{kij} + c_{ijk} \geq 1 & (16) \\
\forall (i,j),(k,p) \in E, & c'_{ijkp} + c'_{kpij} \geq 1 & (17) \\
\forall (i,j),(k,p) \in E, & d'_{ijkp} + d'_{kpij} \leq 1 & (18) \\
\forall i \in V, \forall k \in V, & m_{ik} \geq \sigma_{ik} & (19) \\
\forall (i,j) \in E, \forall k \in V, & \sigma_{ik} \geq c_{ijk} & (20) \\
\forall (i,j) \in E, \forall k \in V, & c_{ijk} \geq d_{ijk} & (21) \\
\forall (i,j) \in E, \forall k \in V, & d_{ijk} \geq m_{jk} & (22) \\
\forall i,j \in V, & \delta_{ij} \leq 1 + b_i - b_j, \delta_{ij} \leq 1 + b_j - b_i, & \\
 & \delta_{ij} \geq b_i + b_j - 1 \text{ and } \delta_{ij} \geq 1 - b_i - b_j & (23) \\
\forall i \in V, & w_i \geq b_i W_i^{(2)} + (1-b_i)W_i^{(1)} & (24a) \\
\forall i \in V, & w_i \leq b_i W_i^{(2)} + (1-b_i)W_i^{(1)} & (24b) \\
\forall i \neq j \in V, & \sigma_{ij} + \sigma_{ji} + \epsilon_{ij} + \epsilon_{ji} \geq 1 & (25) \\
\forall i \in V, & \sum_{(k,p)\in E}(\delta_{ik}(m_{ki} - d_{kpi}) + \delta_{ip}(c_{kpi} - \sigma_{pi}))F_{kp} & \\
 & \leq b_i M_{blue} + (1-b_i)M_{red} & (26) \\
\forall (i,j) \in E, & \sum_{(k,p)\in E}(\delta_{kj}(m'_{kij} - d'_{kpij}) + \delta_{pj}(c'_{kpij} - \sigma'_{pij}))F_{kp} & \\
 & \leq b_j M^{(blue)} + (1-b_j)M^{(red)} + \delta_{ij}M_{max} & (27)
\end{array}
$$

Figure 3.16: Constraints of the ILP.

a data needs to be moved from one memory to another. Here is a short description of these constraints:

- Constraint (1) ensures that variable M representing the makespan will be larger than or equal to the completion time of the last task.
- Constraint (2) ensures that communication (i, j) starts after the completion of task i.
- Constraints (3) ensures that task j starts after the completion of every possible communication (i, j). We can note that, since $\delta_{ij} = 1$ if and only if task i and j are executed on the same memory, $(1 - \delta_{ij})C_{ij}$ is the actual processing time of communication (i, j).
- In Constraints (4a) and some of the following ones, we need an upper bound M_{max} on the possible value of M. This bound is set arbitrarily to $M_{max} = \sum_{i \in V} W_i^{(1)} + \sum_{i \in V} W_i^{(2)} + \sum_{(i,j) \in E} C_{i,j}$. Constraints (4a), (4b) and (14) ensure that $m_{i,j}$ and $m_{j,i}$ are correctly defined: $m_{i,j} = 1$ if $t_j > t_i$, $m_{i,j} = 0$ if $t_j > t_i$ and if $t_j = t_i$, at least one between m_{ij} and m_{ji} is equal to 1. This is important when computing the amount of memory in Constraint (26).
- Similarly Constraints (5a) to (18) ensure that m'_{kij}'s, σ_{ij}'s, σ'_{kij}'s, c_{ijk}'s, c'_{ijkp}'s, d_{ijk}'s, d'_{ijkp}'s, ϵ'_{ij}'s and b_i's variables are well defined.
- Constraint(19) ensures that task ordering is defined consistently, even for tasks with zero processing time (such tasks will appear when pipelining communications in Section 3.5.3).
- Constraint (20) ensures that if communication (i, j) starts before task k starts, task i must finish before task k starts. Constraints (21) and (22) ensure that the linear program defines a valid schedule for communications and tasks.
- Constraints (23) ensures that δ_{ij}'s variables are well defined, i.e., $\delta_{ij} = 1$ if and only if $b_i = b_j$.
- Constraints (24a) and (24b) ensure that w_i's variables are well defined, i.e., $w_i = W_i^{(1)}$ if and only if $b_i = 0$ and $w_i = W_i^{(2)}$ if and only if $b_i = 1$.
- Constraint (25) represents resource constraints as seen in Section 3.3.1: if two tasks are running at the same time, they are not on the same processor.

Finally, Constraint (26) deals with memory constraints, and ensures that the model defined in Section 3.3.2 is observed at the beginning of each task i. Specifically, $b_i M_{blue} + (1 - b_i) M_{red}$ is the memory bound on the memory on which task i is executed. When i is started, we ensure that the sum of the files stored in the corresponding memory when we start task i is smaller than this bound. We claim that $\forall (k, p) \in E$, the file of size F_{kp} will be in the corresponding memory when task i starts if and only if either "task i and task k are in the same memory and we started task k but communication (k, p) is not finished yet" or "task i and task p are in the same memory and we started communication (k, p) but task p is not finished yet". This explains Constraint (26). Similarly Constraint (27) ensures that the memory constraint is respected at the beginning of every communication (i, j).

Constraints (26) and (27) are not linear. However, they can be linearized using the technique presented in [105, 35]. To do so, we introduce the variables $\alpha_{kpi} = \delta_{ik}(m_{ki} - d_{kpi})$, $\beta_{kpi} = \delta_{ip}(c_{kpi} - \sigma_{pi})$, $\alpha'_{kpi} = \delta_{kj}(m'_{kij} - d'_{kpij})$ and $\beta'_{kpij} = \delta_{pj}(c'_{kpij} - \sigma'_{pij})$. Constraints (26) and (27) are then replaced by the constraints in Figure 3.17.

For an arbitrary DAG $\mathcal{D} = (V, E)$ with $|V| = n$ nodes and $|E| = m$ edges, the ILP has $O(m^2 + mn)$ variables and $O(m^2 + mn)$ constraints.

3.5.2 Heuristics

Given the complexity of optimizing 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.

$$
\begin{aligned}
\forall i \in V, \quad & \textstyle\sum_{(k,p)\in E}(\alpha_{kpi}+\beta_{kpi})F_{kp} \\
& \le b_i M^{(red)} + (1-b_i)M^{(blue)} && (26)\\
\forall i \in V, \forall (k,p) \in E, \quad & \alpha_{kpi} \ge \delta_{ik} + m_{ki} - d_{kpi} - 1 && (26a)\\
\forall i \in V, \forall (k,p) \in E, \quad & 2\alpha_{kpi} \le \delta_{ik} + m_{ki} - d_{kpi} && (26b)\\
\forall i \in V, \forall (k,p) \in E, \quad & \beta_{kpi} \ge \delta_{ip} + c_{kpi} - \sigma_{pi} - 1 && (26c)\\
\forall i \in V, \forall (k,p) \in E, \quad & 2\beta_{kpi} \le \delta_{ip} + c_{kpi} - \sigma_{pi} && (26d)\\
\forall (i,j) \in E, \quad & \textstyle\sum_{(k,p)\in E}(\alpha'_{kpij}+\beta'_{kpij})F_{kp} \\
& \le b_i M^{(red)} + (1-b_i)M^{(blue)} + \delta_{ij}M_{max} && (27)\\
\forall (i,j) \in E, \forall (k,p) \in E, \quad & \alpha'_{kpij} \ge \delta_{kj} + m'_{kij} - d'_{kpij} - 1 && (27a)\\
\forall (i,j) \in E, \forall (k,p) \in E, \quad & 2\alpha'_{kpij} \le \delta_{kj} + m'_{kij} - d'_{kpij} && (27b)\\
\forall (i,j) \in E, \forall (k,p) \in E, \quad & \beta'_{kpij} \ge \delta_{pj} + c'_{kpij} - \sigma'_{pij} - 1 && (27c)\\
\forall (i,j) \in E, \forall (k,p) \in E, \quad & 2\beta'_{kpij} \le \delta_{pj} + c'_{kpij} - \sigma'_{pij} && (27d)
\end{aligned}
$$

Figure 3.17: Linearization of the last two constraints of the ILP.

3.5.2.1 The MEMHEFT algorithm

MEMHEFT is based on HEFT (Heterogeneous Earliest Finish Time) [103]. The HEFT algorithm is highly competitive and widely used to schedule static DAGs 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. In our model, there are only two processor types, hence each selected task will be mapped on one of two candidates, namely the processors with earliest available time in each type. In other words, the *processor selection phase* can be renamed as the *memory selection phase*. In addition, MEMHEFT checks memory usage, as explained below.

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

Memory selection phase. For each selected task i and for each memory $\mu \in \{red, blue\}$, we have to compute $EST^{(\mu)}(i)$ the earliest execution start time of task i on memory μ (derived from a given partial schedule). This earliest execution start time has to take into account (i) resource, (ii) precedence, and (iii) memory constraints.

From a resource perspective, task i can not be executed on memory μ before one of the processors operating on memory μ is available. Thus $resource_EST^{(\mu)}(i)$, the earliest start time of task i on memory μ from a resource point of view, can be expressed as:

$$resource_EST^{(\mu)}(i) = \min_{proc \text{ in } \mu \text{ mem}} \{avail[proc]\}$$

where $avail[proc]$ is the finish time of the last task assigned to $proc$ in the partial schedule.

From a precedence perspective, all immediate predecessors $j \in Pred(i)$ of task i must have been scheduled. Thus $precedence_EST^{(\mu)}(i)$, the earliest start time of task i on memory μ from a prece-

dence point of view, is expressed as:

$$precedence_EST^{(\mu)}(i) = \max_{j \in Pred(i)} \{AFT(j) + \delta_j^{(\mu)} C_{j,i}\}$$

where $\delta_j^{(\mu)} = 0$ is task j is executed on memory μ, 1 otherwise, and $AFT(j)$ is the actual finish time of task j in the partial schedule.

From a memory perspective, we have to keep trace of the memory consumption of our schedule to ensure that it does not violate the memory constraints. Thus, the MEMHEFT algorithm maintains for each memory μ the function $free_mem^{(\mu)}(t)$ that represents the amount of the μ memory available at time t in the partial schedule. Here $free_mem^{(\mu)}$ is a staircase function (the definition space $\mathbb{R}$ can be partitioned in a finite number of intervals where $free_mem^{(\mu)}$ is constant) that can be stored as a list of couples $[(x_1, val_1), .., (x_\ell, val_\ell)]$ such that:

$$\forall i \in [1, \ell - 1],\ \forall t \in [x_i, x_{i+1}[,\ free_mem^{(\mu)}(t) = val_i$$

and $\forall t \geq x_\ell,\ free_mem^{(\mu)}(t) = val_\ell$. Note that val_ℓ can be non-zero since the partial schedule may keep some files $F_{i,j}$ stored in the memories if task i has been scheduled but task j has not. Thus, to process task i on memory μ at time t without violating memory constraints, there must be enough available memory to store all the input files of task i that were not stored on memory μ yet, and all its output files. Thus, the earliest start time of task i on memory μ from the memory point of view can be expressed as:

$$\begin{aligned} task_mem_EST^{(\mu)}(i) = min\ \{t,\ \text{such that}\ \forall t' \geq t,& \\ free_mem^{(\mu)}(t') \geq \sum_{j \in Pred(i)} (1 - \delta_j^{(\mu)}) F_{j,i} + \sum_{j \in Succ(i)} F_{i,j}\}& \end{aligned}$$

If $free_mem^{(\mu)}$ is stored as a list of size ℓ, $task_mem_EST^{(\mu)}(i)$ can be computed in time $O(\ell)$.

The MEMHEFT algorithm enforces that when a task i is assigned to the memory μ, every communication $(j, i) \in E$ such that $\delta_j^{(\mu)} = 0$ will start as late as possible, and they will all have a processing time $C_i^{(\mu)} = \max_{(j,i) \in E}\{(1 - \delta_j^{(\mu)}) C_{i,j}\}$. Thus, to process task i on memory μ, the earliest start time of every communication $(j, i) \in E$ from the memory point of view can be expressed as:

$$comm_mem_EST^{(\mu)}(i) = min\ \{t,\ \text{such that}\ \forall t' \geq t, free_mem^{(\mu)}(t') \geq \sum_{j \in Pred(i)} (1 - \delta_j^{(\mu)}) F_{j,i}\}$$

If $free_mem^{(\mu)}$ is stored as a list of size ℓ, $comm_mem_EST^{(\mu)}(i)$ can be computed in time $O(\ell)$.

Finally, the earliest execution start time of task i on memory μ will be expressed as:

$$\begin{aligned} EST^{(\mu)}(i) = max\ \{&resource_EST^{(\mu)}(i), \\ &precedence_EST^{(\mu)}(i), \\ &task_mem_EST^{(\mu)}(i), \\ &comm_mem_EST^{(\mu)}(i) + C_i^{(\mu)}\} \end{aligned}$$

The selected task i is assigned to the memory μ_{min} that minimizes its earliest finish time $EFT^{(\mu)}(i) = EST^{(\mu)}(i) + W_i^{(\mu)}$ and then, to the $proc$ that minimizes the idle time $EST(i, \mu_{min}) - avail_proc(proc)$.

3.5.2.2 The MEMMINMIN algorithm

The MEMMINMIN algorithm does not include a task prioritizing phase but dynamically decides the order in which tasks are mapped onto resources. It is the memory-aware counterpart of the MINMIN heuristic [25]. 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 memory $\mu_{\min} \in \{red, blue\}$ that minimizes $EFT^{(\mu)}(i)$ as defined in Section 3.5.2.1 (computed from a partial schedule).

For a DAG $\mathcal{D}$ with $|V| = n$ nodes and $|E| = m$ edges, both heuristics have a worst-case complexity of $O(n^2(n + m))$. Their pseudo-code is available in [66].

3.5.3 Simulation results

In this section, we conduct several simulations to compare the two heuristics MEMHEFT and MEMMINMIN proposed in Section 3.5.2, and to assess their absolute performance w.r.t. to the (optimal) ILP solution (Section 3.5.1). For each heuristic, we compute its makespan for various amounts of the available *blue* and *red* memories. The heuristics have been implemented in Python 2.7. Source code for all the algorithms, heuristics and simulations is publicly available at http://perso.ens-lyon.fr/julien.herrmann/. The optimal makespan for small graphs has been computed by solving the ILP using the IBM® ILOG® CPLEX® Interactive Optimizer 12.5.0.0.

3.5.3.1 Experimental setup

We use four different sets of DAGs: (i) two synthetic sets (randomly generated) of different sizes ,SMALLRANDSET, and LARGERANDSET; and (ii) two applicative sets (from linear algebra benchmarks), LUSET and CHOLESKYSET.

Random task graphs The first and second sets are random DAGs, generated using the Directed Acyclic Graph GENerator (DAGGEN)[2]. DAGGEN uses four popular parameters to define the shape of the DAG: *size*, *width*, *density* and *jumps*.

- The *size* determines the number of node in the DAG. Nodes are organized in levels.
- The *width* determines the maximum parallelism in the DAG, that is the number of tasks in the largest level. A small value leads to "chain" graphs and a large value to "fork-join" graphs.
- The *density* denotes the number of edges between two levels of the DAG, with a low value leading to few edges and a large value to many edges.
- Finally random edges are added that go from level l to levels $l + 1 \ldots l + jumps$.

The first two parameters take values between 0 and 1. This DAG generation procedure is similar to the one used in [85].

SMALLRANDSET is a set of 50 randomly generated DAGs using values $size = 30$, $width = 0.3$, $density = 0.5$ and $jumps = 5$. Then, for each node, the values $W_i^{(1)}$ and $W_i^{(2)}$ are randomly chosen between 1 and 20 and, for each edge, the values $C_{i,j}$ and $F_{i,j}$ are randomly chosen between 1 and 10. One graph of SMALLRANDSET is depicted in Figure 3.18.

LARGERANDSET is a set of 100 randomly generated DAGs using values $size = 1000$, $width = 0.3$, $density = 0.5$ and $jumps = 5$. Then, for each node and each edge, the values $W_i^{(1)}$, $W_i^{(2)}$, $C_{i,j}$ and $F_{i,j}$ are randomly chosen between 1 and 100. One graph of LARGERANDSET is depicted in Figure 3.19.

[2] The code for the generator is publicly available at https://github.com/frs69wq/daggen.

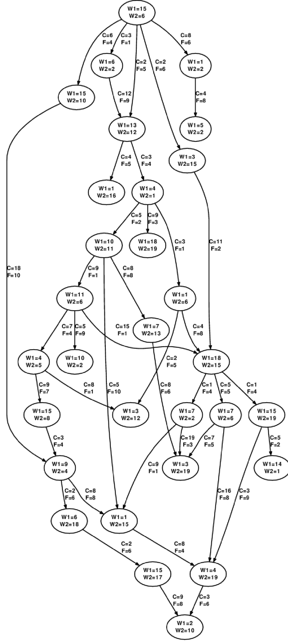


Figure 3.18: One DAG in SMALLRANDSET.

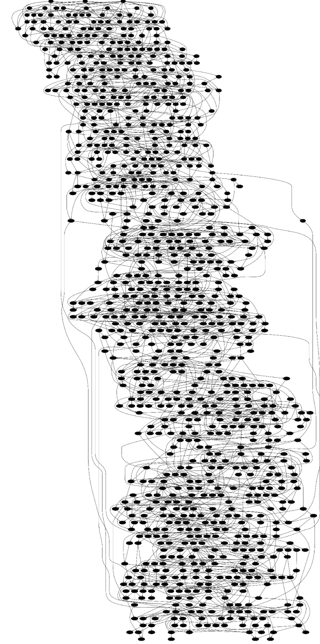


Figure 3.19: One DAG in LARGERANDSET.

Linear algebra task graphs The third and four sets contain representative DAGs from dense linear algebra kernels.

LUSET contains DAGs representing the task graph of the LU factorization of a tiled square matrix. At each step of this factorization, the diagonal tile is factored with a GETRF kernel, the first row and the first column of tiles are eliminated with a TRSM kernel, and the remaining tiles are updated with a GEMM kernel. Another step of the LU factorization is then applied on the trailing matrix involving a workflow dependencies among the kernels working on the same tiles.

CHOLESKYSET contains DAGs representing the task graph of the Cholesky factorization of tiled symmetrical matrix. At each step of this factorization, the diagonal tile is factored with a POTRF kernel, the first column and the diagonal of tiles are processed with respectively a TRSM and a SYRK kernel, and the remaining tiles are updated with a GEMM kernel. Another step of the Cholesky factorization is then applied on the trailing matrix involving a workflow dependencies among the kernels working on the same tiles.

More details on the tiled LU and Cholesky factorizations can be found in [28]. The classic DAGs of both factorizations do not exactly fit our model, as the output of a node (typically the kernel used for factoring a diagonal tile) may be used as an input for several other tasks. Hence we add a linear pipeline of fictitious null-size tasks that models the broadcast of the output to the target tasks. The DAG for the LU factorization of a $n \times n$ tiled matrix has $\frac{4}{3}n^3$ nodes, whereas the DAG for the Cholesky factorization has $\frac{2}{3}n^3$ nodes (and there are $O(n^2)$ fictitious tasks).

The running times of the linear kernels have been measured on the *mirage* platform, an heterogeneous system composed of two Intel hexacore processors X5650 at 2.67 GHz having 12 MB of L3 cache for a total of 12 cores and 36 GB of main memory, equipped with three NVIDIA Tesla M2070 GPUs having 6 GB of memory each. We associate the *blue* processors to the CPUs and the *red* processors to the GPUs. The running times were estimating by performing measurement with the MAGMA library [8] (using tiles of size 192×192 in double precision) and are given in Table 3.2.

For communication costs, the average observed time to send one tile from a CPU to a GPU was

Kernels	getrf	gemm	trsm_l	trsml_u	potrf	syrk
On CPUs	450	1450	990	830	450	990
On GPUs	10490	140	150	400	10490	150

Table 3.2: Average running time in *ms* of the linear algebra kernels on a 192×192 tile

approximatively 50 ms, thus all $C_{i,j}$ have been set to this value. The files sent and received by the tasks contain the value of the tiles. Since all tiles have the same size, we consider that for each edge (i, j) in the DAG, $F_{i,j} = 1$, one unit of memory corresponding to one tile.

3.5.3.2 Results

SMALLRANDSET To assess the absolute performance of the heuristics, we compare them to the optimal schedule found by the ILP described in Section 3.5.1. Note that SMALLRANDSET is the only set for which the ILP is able to compute a solution in a reasonable time. We aim at finding a schedule for each DAG in SMALLRANDSET with the smallest makespan as possible and under the same memory bound for each memory $M^{(blue)} = M^{(red)} = M^{(bound)}$.

First, we compute for each DAG $\mathcal{D}$ the makespan $Makespan_{HEFT}$ returned by the classical memory-oblivious HEFT algorithm and its maximum usage of each memory $M_{\text{blue}}^{HEFT}(\mathcal{D})$ and $M_{\text{red}}^{HEFT}(\mathcal{D})$. The idea is that the classical HEFT algorithm will not be able to schedule $\mathcal{D}$ on a platform with less than these amounts of *blue* and *red* memory. It is also clear that if the memory bounds respect $M^{(blue)} \geq M_{\text{blue}}^{HEFT}(\mathcal{D})$ and $M^{(red)} \geq M_{\text{red}}^{HEFT}(\mathcal{D})$, MEMHEFT will take exactly the same decisions as HEFT. Thus if $M^{(bound)} \geq max(M_{\text{blue}}^{HEFT}(\mathcal{D}), M_{\text{red}}^{HEFT}(\mathcal{D}))$, the performance of MEMHEFT will be the same as that of HEFT. Figure 3.20 reports the performances of MEMHEFT and MEMMINMIN if $M^{(bound)} = \alpha \times max(M_{\text{blue}}^{HEFT}(\mathcal{D}), M_{\text{red}}^{HEFT}(\mathcal{D}))$ with $\alpha \in [0, 1]$ being the relative memory compared to the amount needed by HEFT. 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 DAGs successfully scheduled with the given memory bounds (to be read on the left scale). Dotted lines show the fraction of DAGs in SMALLRANDSET that our heuristics manage to schedule with the given memory bounds (to be read on the right scale).

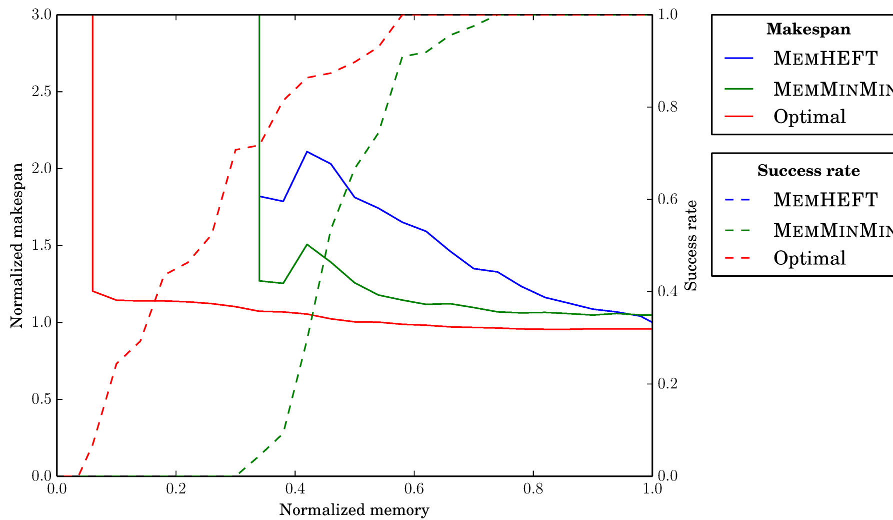


Figure 3.20: Results for SMALLRANDSET.

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 3.20 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 DAGs in SMALLRANDSET with this memory bound. Our heuristics can provide a feasible schedule for every DAG in SMALLRANDSET when the memory bound is greater than 75% of the amount required by HEFT, whereas, in theory, every DAG can be scheduled down to 60% of this amount. In addition to the global view for SMALLRANDSET, detailed results for the DAG of Figure 3.18 are provided in Figure 3.21.

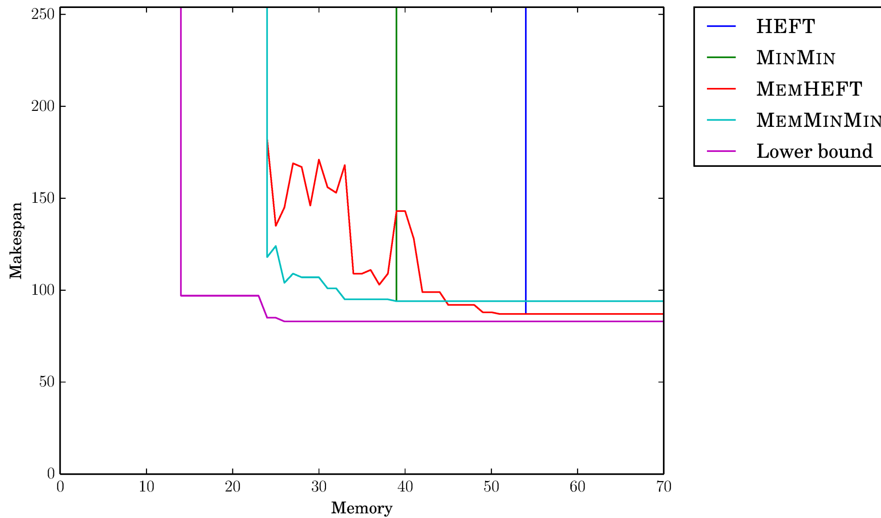


Figure 3.21: Makespan for the DAG in SMALLRANDSET depicted in Figure 3.18.

LARGERANDSET The same experimental procedure has been applied to LARGERANDSET, except that the optimal schedule cannot be computed in reasonable time anymore. The average relative makespan of our heuristics are depicted in Figure 3.22. We see that both MEMHEFT and MEMMINMIN succeed to schedule all the DAGs in LARGERANDSET 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. Finally, specific results for the one DAG depicted in Figure 3.19 are provided in Figure 3.23.

LUSET and CHOLESKYSET We provide results for numerical algebra sets corresponding to a 13 $\times$ 13 tiled matrix. Figure 3.24 depicts the results for LU factorization, whereas Figure 3.25 deals with Cholesky factorization. Contrarily to the previous section, 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. Indeed, Figure 3.24 shows that MEMMINMIN fails

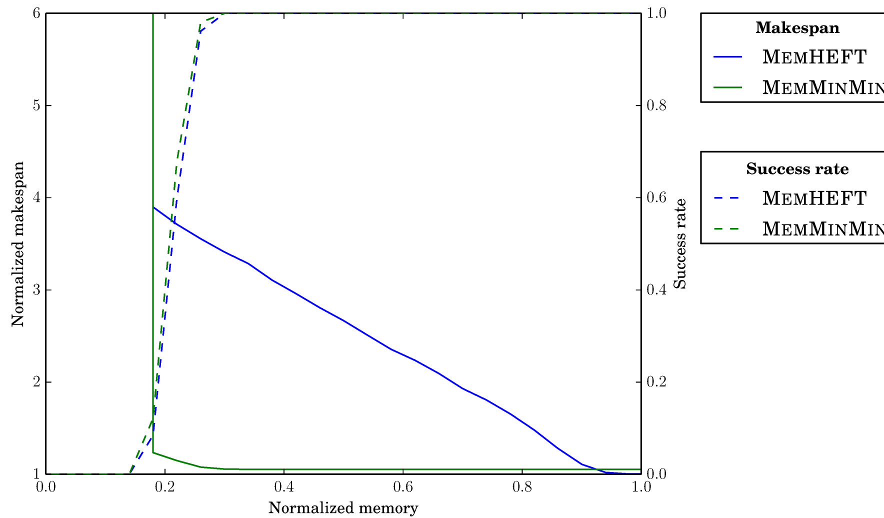


Figure 3.22: Results for LARGERANDSET.

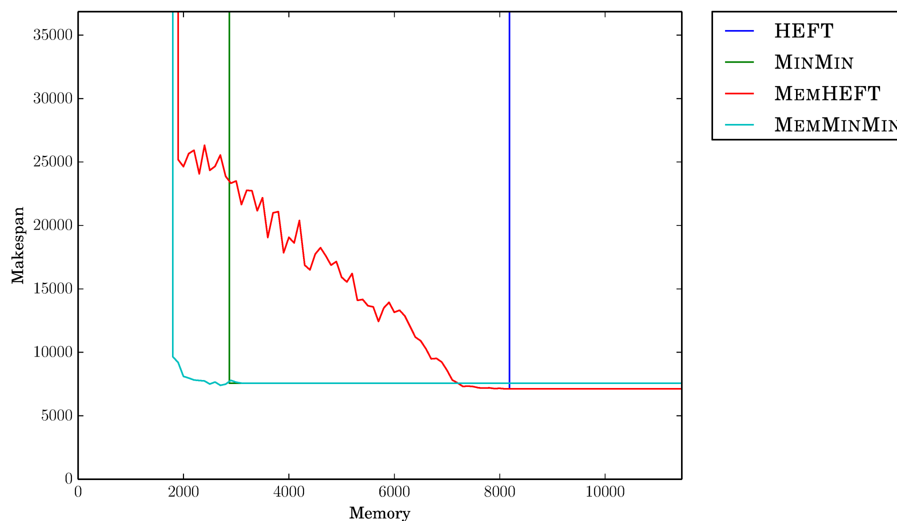


Figure 3.23: Makespan for the DAG in LARGERANDSET depicted in Figure 3.19.

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 DAGs, 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 DAG. 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 trade-offs. 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.

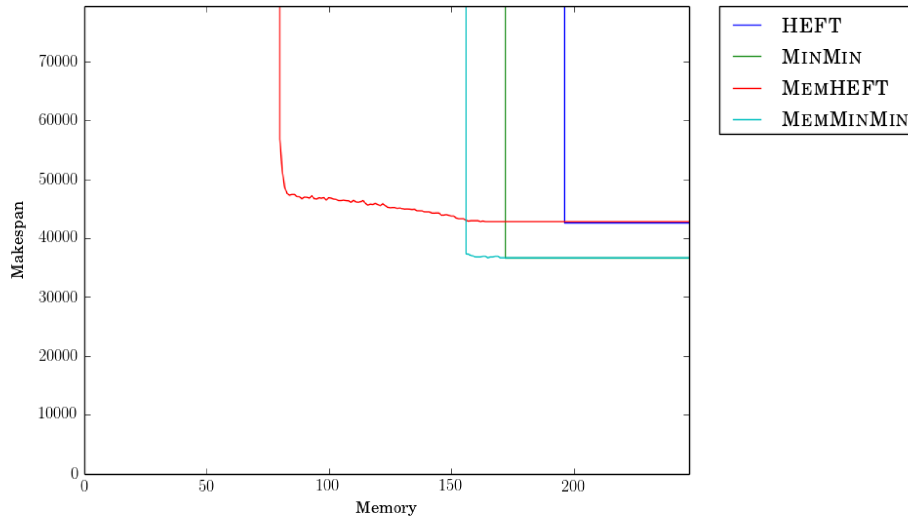


Figure 3.24: MEMHEFT and MEMMINMIN results on the DAG representing an LU factorization of a 13×13 tiled matrix.

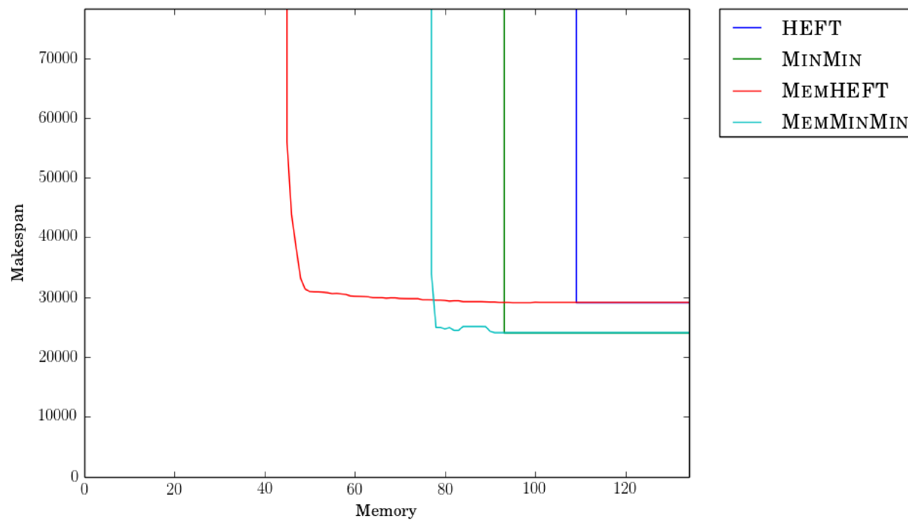


Figure 3.25: MEMHEFT and MEMMINMIN results on the DAG representing an Choleski factorization of a 13×13 tiled matrix.

3.6 Conclusion

In this chapter, we have investigated the problem of scheduling a task graph on a dual-memory system, i.e. an heterogeneous platform made of two types of memories, with several processors attached to each memory. Dual-memory systems include emerging hybrid computing platforms, which usually includes one or several accelerators (such as GPU) in addition to multicore CPUs.

Our first contribution was to propose a simple model that captures the complexity of the problem. We have studied the bi-criteria memory minimization problem that arises when traversing a colored task tree for a system composed of two different computing units with their own memory. We have proved that the search for an optimal solution is NP-complete, and that it was impossible to approximate

both memories by any pair of constant factors. In addition, we have determined the optimal depth-first traversal, which turns out to minimize both memories simultaneously. This depth-first traversal achieves nice results for realistic assembly trees. We have also proposed several heuristics, based upon extensions of Liu's optimal algorithm for the one-memory problem. These heuristics provide very good solutions when dealing with arbitrary tree graphs.

Given the NP-hardness of the restricted problem, we have proposed several approaches for the general case. We have first provided an exact resolution through the design of an intricate ILP which is able to compute an optimal schedule for medium-size instances (up to 30 tasks). Then, we have proposed two memory-aware heuristics for larger instances, which are the counterparts of the classical HEFT and MINMIN algorithms. We have studied the performance of these new heuristics through extensive simulations on different task graphs, and compared them to the optimal solution for small instances.

An interesting future work would be to include some of the proposed heuristics in an actual runtime toolkit for hybrid platform such as StarPU [15]. It would also be of interest to adapt the heuristics to more complex platforms, such as hybrid platforms with several types of accelerators, and/or including more than two memories.

Chapter 4

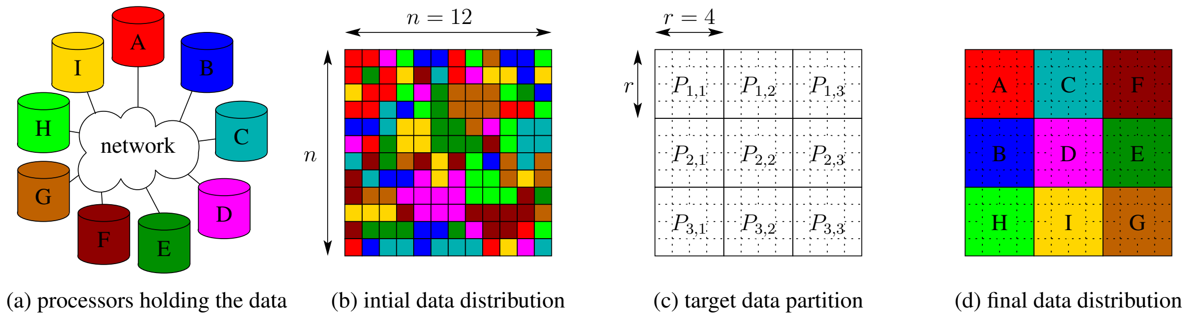
Assessing the cost of redistribution followed by a computational kernel

When dealing with a distributed computational platform, the data layout has a strong impact on the overall performance. As said in the introduction of this thesis, modern linear algebra libraries use level-3 BLAS kernels that partition the matrices into blocks. We saw in Chapter 1 that, in a distributed memory context, the blocks are mapped onto the resources using a 2D block-cyclic distribution, which nicely balances the workload across processors. When factorizing a matrix, we always considered that the blocks were initially mapped onto the distributed platform with a 2D block-cyclic pattern, allowing a efficient owner-compute strategy during the factorization. In this chapter, we consider the case where the matrix may have an arbitrary initial distribution, requiring a redistribution. We do not restrict ourselves to targeting a 2D block-cyclic distribution, but we rather tackle the general redistribution problem. The classical redistribution problem aims at optimally scheduling communications when reshuffling from an initial data distribution to a target data distribution. This target data distribution is usually chosen to optimize some objective for the algorithmic kernel under study (good computational balance or low communication volume or cost), and therefore to provide high efficiency for that kernel. However, the choice of a distribution minimizing the target objective is not unique. This leads to generalizing the redistribution problem as follows: find a re-mapping of data items onto processors such that the data redistribution cost is minimal, and the operation remains as efficient. This chapter studies the complexity of this generalized problem. We compute optimal solutions and evaluate, through simulations, their gain over classical redistribution. We also show the NP-hardness of the problem to find the optimal data partition and processor permutation (defined by new subsets) that minimize the cost of redistribution followed by a simple computational kernel. Finally, experimental validation of the new redistribution algorithms are conducted on a multicore cluster, for both a 1D-stencil kernel and a more compute-intensive dense linear algebra routine.

4.1 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. Data redistribution algorithms are critical to many applications, and therefore have received considerable attention. The data redistribution problem can be stated informally as follows: given N data items that are currently distributed across P processors, redistribute them according to a different target layout. Consider for instance a dense square matrix $A = (a_{ij})_{0 \leq i,j < n}$ of size n, whose initial distribution is

random, and that must be redistributed into square blocks across a $p \times p$ 2D-grid layout. A scenario for this problem is that the matrix has been generated by a Monte-Carlo method and is now needed for some matrix product $C \leftarrow C + AB$. Assume for simplicity that p divides n, and let $r = n/p$. In this example, $N = n^2$, $P = p^2$, and the redistribution will gather a block of $r \times r$ data elements of A on each processor, as illustrated on Figure 4.1. More precisely, all the elements of block $A_{i,j} = (a_{k,\ell})$, where $ri \leq k < (r+1)i$ and $rj \leq \ell < (r+1)j$, must be sent to processor $P_{i,j}$. This example illustrates the classical redistribution problem. Depending upon the cost model for communications, various optimization objectives have been considered, such as the total volume of data that is moved from one processor to another, or the total time for the redistribution, if several communications can take place simultaneously. We detail classical cost models in Section 4.2, which is devoted to related work.



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

Figure 4.1: Example of matrix redistribution with $N = 12^2$ data blocks and $P = 3^2$ processors. Each color in the data distributions corresponds to a processor, e.g., all red data items reside on processor A.

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. Continuing with the example, the $p \times p$ 2D-grid will be virtual, i.e., an overlay topology mapped into the physical topology, forcing the interconnection switch to emulate a 2D-grid. Notwithstanding, the layout of the processors in the grid remains completely flexible. For instance, the processors labeled $P_{1,1}$, $P_{1,2}$ and $P_{2,1}$ can be *any* processors in the platform, and we have the freedom to choose which three processors will indeed be labeled as the top-left corner processors of the virtual grid. Now, to describe the matrix product on the 2D-grid, we say that data will be sent *horizontally* between $P_{1,1}$ and $P_{1,2}$, and vertically between $P_{1,1}$ and $P_{2,1}$, but this actually means that these messages will be routed by the actual network, regardless of the physical position of the three processors in the platform.

This leads us to revisit the redistribution problem, adding the flexibility to select the *best* assignment of data on the processors (according to the cost model). The problem can be formulated as mapping a *partition* of the initial data onto the resources: there are P data subsets (the blocks in the example) to be assembled onto P processors, with a huge (exponential) number, namely $P!$, of possible mappings. An intuitive view of the problem is to assign the same color to all data items that initially reside on the same processor, and to look for a coloring of the virtual grid that will minimize the redistribution cost. For instance, in Figure 4.1, most data items of the block allocated to the virtual processor $P_{1,1}$ are initially colored red (they reside on the red processor A), so we decided to map $P_{1,1}$ on processor A to avoid moving these items.

One major goal of this chapter 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. There

are $P!$ possible redistributions, and we aim at finding the one minimizing a predefined cost-function. In this chapter, we use the two most widely-used criteria in the literature to compute the cost of a redistribution:

- **Total volume.** In this model, the platform is not dedicated, and the objective is to minimize the total communication volume, i.e., the total amount of data sent from one processor to another. 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.** In this model, the platform is dedicated to the application, and several communications can take place in parallel, provided that they involve different processor pairs. This is the one-port bi-directional model used in [68, 71]. The quantity to minimize is the number of parallel steps, where a step is a collection of unit-size messages that involve different processor pairs.

One major contribution of this chapter is the design of an algorithm solving this optimization problem for either criterion. We also provide various experiments to quantify the gain that results from choosing the optimal mapping rather than a *canonical* mapping where processors are labeled arbitrarily, and independently of the initial data distribution.

As mentioned earlier, a redistribution is usually motivated by the need to efficiently execute in parallel a subsequent computational kernel. In most cases, there may well be several data partitions that are suitable for the efficient execution of this kernel. The optimal partition also depends upon the initial data distribution. Coming back to the introductory example, where the redistribution is followed by a matrix product, we may ask whether a full block partition is absolutely needed? If the original data is distributed along a suitable, well-balanced distribution, a simple solution is to compute the product in place, using the *owner computes* rule, that is, we let the processor holding $C_{i,j}$ compute all $A_{i,k}B_{k,j}$ products. This means that elements of A and B will be communicated during the computation, when needed. On the contrary, if the original distribution has a severe imbalance, with some processors holding significantly more data than others, a redistribution is very likely needed. But in this latter case, do we really need a perfect full block partition? In fact, the optimization problem is the following: given an initial data distribution, what is the best data partition, and the best mapping of this partition onto the processors, to minimize total execution time, defined as the sum of the redistribution time and of the execution of the kernel. Another major contribution of this chapter is to assess the complexity of this intricate problem. Finding the optimal partition mapping becomes NP-complete when coupling the redistribution with a simple computational kernel such as an iterative 1D-stencil kernel. Here the optimization objective is the sum of the redistribution time (computed using either of the two criteria above, with all communications serialized or with communications organized in parallel steps), and of the parallel execution time of a few steps of the stencil. Intuitively this confirms that determining the optimal data partition and its mapping is a difficult task. Stencil computations naturally favor block distributions, in order to communicate only block frontiers at each iteration. But this has to be traded-off with the cost of moving the data from the initial distribution, with the number of iterations, and with the possible imbalance of the final distribution that is chosen (whose own impact depends upon the communication-to-computation ratio of the machine). Altogether, it is no surprise that all these possibilities lead to a hard combinatorial problem.

Finally, this chapter provides an experimental validation of the new redistribution algorithms conducted on a multicore cluster. We first experiment with the 1D-stencil algorithm and obtain performance improvements in total execution time that strongly depend on initial distributions. Different data configurations have been tested to assess this gain. For a more compute-intensive dense linear algebra routine,

such as QR factorization, redistributing the data items can also be necessary. The 2D block-cyclic partition is known to offer a good trade-off between the amount of communications during the QR factorization and the load balancing among processors. Using the algorithms to determine the best distribution compatible with the 2D block-cyclic partition provides significant improvement in the completion time.

The rest of the chapter is organized as follows. We survey related work in Section 4.2. We detail the model and formally state the optimization problems in Section 4.3. We deal with the problem of finding the best redistribution for a given data partition in Section 4.4. Sections 4.4.1 and 4.4.2 provide algorithms computing the optimal solution, while Section 4.4.3 reports simulation results showing the gain over redistributing to an arbitrary compatible distribution. In Section 4.5, we couple the redistribution with a stencil kernel, and show that finding the optimal data partition, together with the corresponding redistribution, is NP-complete. Experiments conducted on a multicore cluster are reported in Section 4.6. Section 4.6.1 is devoted to the experimental setup. Section 4.6.2 provides results when redistribution is followed by a stencil kernel, while Section 4.6.3 deals with QR factorization. We provide final remarks and directions for future work in Section 4.7.

4.2 Related work

4.2.1 Communication model

The *macro-dataflow model* has been widely used in the scheduling literature (see the survey papers [84, 97, 33, 47] and the references therein). In this model, the cost to communicate L bytes is $\alpha + L\beta$, where α is a start-up cost and β is the inverse of the bandwidth. In this chapter, we consider large, same-sized data items, so we can safely restrict to *unit* communications that involve a single data item; we integrate the start-up cost into the cost of a unit communication.

In the macro-dataflow model, communication delays from one task to its successor are taken into account, but communication resources are not limited. First, a processor can send (or receive) any number of messages in parallel, hence an unlimited number of communication ports is assumed (this explains the name *macro-dataflow* for the model). Second, the number of messages that can simultaneously circulate between processors is not bounded, hence an unlimited number of communications can simultaneously occur on a given link. In other words, the communication network is assumed to be contention-free, which of course is not realistic as soon as the processor number exceeds a few units.

A much more realistic communication model is the *one-port bidirectional model* where at a given time-step, any processor can communicate with at most one other processor in both directions: sending to and receiving from. Thus, communications can occur in parallel, provided that they involve disjoint pairs of sending/receiving processors. The one-port model was introduced by Hollermann et al. [68], and Hsu et al. [71]. It has been widely used since, both for homogeneous and heterogeneous platforms [17, 18].

4.2.2 General data redistribution

The complexity of scheduling data redistribution in distributed architectures strongly depends on the network model. When the network has a general graph topology, achieving the minimal completion time for a set of communications is NP-complete, even when the time required to move a data along any link is constant [92].

In this context, several variants of the *one-port bidirectional model* have been considered. The first variant is an unidirectional one-port model, where a processor can participate in only one communication

at a time (either as a sender or as a receiver); with this variant, the redistribution problem becomes NP-complete [78]. A second variant consists of assuming that each processor p has a number of ports $v(p)$ that represents the maximum number of simultaneous transfers that this processor can be involved in [34]. Finally, in a third variant [13], processors have memory constraints that must be enforced during the redistribution process.

4.2.3 Array redistribution

A specific class of redistribution problems has received considerable attention, namely the redistribution of arrays that are already distributed in a block-cyclic fashion over a multidimensional processor grid. This interest was originally motivated by the HPF [79] programming style, in which scientific applications are decomposed into phases. At each phase, there is an optimal distribution of the data arrays onto the processor grid. Typically, arrays are distributed according to a `CYCLIC(r)` pattern[1] along one or several dimensions of the grid. The best value of the distribution parameter r depends on the characteristics of the algorithmic kernel as well as on the communication-to-computation ratio of the target machine [41]. Because the optimal value of r changes from phase to phase and from one machine to another (think of a heterogeneous environment), run-time redistribution turns out to be a critical operation, as stated in [76, 106, 108, 102] (among others). Communications are scheduled into parallel steps, which involve different processor pairs. The model comes in two variants, synchronous and asynchronous. In the synchronous variant, the cost of a parallel step is the maximal size of a message and the objective is to minimize the sum of the costs of the steps [106, 38]. In the asynchronous model, some overlap is allowed between communication steps [62]. Finally, the ScaLAPACK library provides a set of routines to perform array redistribution [88]. A total exchange is organized between processors, which are arranged as a (virtual) caterpillar. The total exchange is implemented as a succession of synchronous steps.

We point out that all the works referenced in Section 4.2.2 and in this one deal with a fixed target distribution. To the best of our knowledge, this work is the first to consider target data partitions rather than target data distributions, thereby allowing to choose the best data redistribution among $P!$ candidates, where P is the number of enrolled processors. Also, this work is the first to study the cost of coupling a redistribution with a computational kernel, which is a very important problem in practice.

4.3 Model and framework

This section details the framework and formally states the optimization problems. We start with a few definitions.

4.3.1 Definitions

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

Definition 4.1 (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.

[1]The definition is the following: let an array $X[0...M-1]$ be distributed according to a block-cyclic distribution `CYCLIC(r)` onto a linear grid of P processors. Then element $X[i]$ is mapped onto processor $p = \lfloor i/r \rfloor \bmod P$, $0 \le p \le P-1$.

Definition 4.2 (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 jth *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 4.3 (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, $\mathcal{D}(x) = \sigma(\mathcal{P}(x))$.

4.3.2 Cost of a redistribution

In this section, we formally state the two metrics for the cost of a redistribution, namely the total volume and the number of parallel steps. Both metrics assume that the communication of one data item from one processor to another takes the same amount of time, regardless of the item and of the location of the source and target processors. Indeed, data items can be anything from single elements to matrix tiles, columns or rows, so that our approach is agnostic of the granularity of the redistribution. As already mentioned, many modern interconnection networks are fully-connected switches, and they can implement any (same-length) communication in the same amount of time. Note that with asymmetric networks, it is always possible to use the worst-case communication time between any processor pair as the unit time for a communication.

4.3.2.1 Total volume

For this metric, we simply count the number of data items that are sent from one processor to another. This metric may be pessimistic if some parallelism is possible, but it provides an interesting measure of the overhead of the redistribution, especially if the platform is not dedicated.

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 $\mathcal{D}_{tar}(x) = j$. For a given processor i, let s_i (respectively r_i) be the total number of data items that processor i must send (respectively receive) during the redistribution. We have $s_i = \sum_{j \neq i} q_{i,j}$ and $r_i = \sum_{j \neq i} q_{j,i}$. The total communication volume of the redistribution is defined as

$$RedistVol(\mathcal{D}_{ini} \rightarrow \mathcal{D}_{tar}) = \sum_i s_i = \sum_i r_i.$$

4.3.2.2 Number of parallel steps

With this metric, some communications can take place in parallel, provided that each of them involves a different processor pair (sender and receiver). This communication model is the bidirectional one-port model introduced in [68, 71] and accounts for contention when communications take place simultaneously.

We define a parallel step 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 $\mathit{RedistSteps}(\mathcal{D}_{ini} \to \mathcal{D}_{tar})$ as the minimal number of parallel steps that are needed to perform the redistribution.

4.3.3 Optimization problems

Here, we formally introduce the optimization problems that we study in Sections 4.4 and 4.5 below.

4.3.3.1 Best redistribution compatible with a given partition

In the optimization problems of Section 4.4, the data partition is given, and we aim at finding the best compatible target distribution (among $P!$ ones). More precisely, given an initial data distribution $\mathcal{D}_{ini}$ and a target data partition $\mathcal{P}_{tar}$, we aim at finding a data distribution $\mathcal{D}_{tar}$ that is compatible with $\mathcal{P}_{tar}$ and such that the redistribution cost from $\mathcal{D}_{ini}$ to $\mathcal{D}_{tar}$ is minimal. Since we have two cost metrics, we define two problems:

Definition 4.4 (VOLUMEREDISTRIB)**.** Given $\mathcal{D}_{ini}$ and $\mathcal{P}_{tar}$, find $\mathcal{D}_{tar}$ compatible with $\mathcal{P}_{tar}$ such that $\mathit{RedistVol}(\mathcal{D}_{ini} \to \mathcal{D}_{tar})$ is minimized.

Definition 4.5 (STEPREDISTRIB)**.** Given $\mathcal{D}_{ini}$ and $\mathcal{P}_{tar}$, find $\mathcal{D}_{tar}$ compatible with $\mathcal{P}_{tar}$ such that $\mathit{RedistSteps}(\mathcal{D}_{ini} \to \mathcal{D}_{tar})$ is minimized.

We show in Section 4.4 that both problems have polynomial complexity.

4.3.3.2 Best partition and best compatible redistribution

In the optimization problems of Section 4.5, the data partition is no longer fixed. 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 will have 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 $\mathcal{D}_{tar}$ will itself depend upon $\mathcal{D}_{tar}$. We model the total cost as the sum of the time of the redistribution and of the computation. Letting τ_{comm} denote the time to perform a communication, the time to execute the redistribution is either $\mathit{RedistVol}(\mathcal{D}_{ini} \to \mathcal{D}_{tar}) \times \tau_{comm}$ or $\mathit{RedistSteps}(\mathcal{D}_{ini} \to \mathcal{D}_{tar}) \times \tau_{comm}$, depending upon the communication model. This leads to the following two problems:

Definition 4.6 (VOLPART&REDISTRIB)**.** Given $\mathcal{D}_{ini}$, find $\mathcal{P}_{tar}$, and $\mathcal{D}_{tar}$ compatible with $\mathcal{P}_{tar}$, such that $T_{total} = \mathit{RedistVol}(\mathcal{D}_{ini} \to \mathcal{D}_{tar}) \times \tau_{comm} + T_{comp}(\mathcal{P}_{tar})$ is minimized.

Definition 4.7 (STEPPART&REDISTRIB)**.** Given $\mathcal{D}_{ini}$, find $\mathcal{P}_{tar}$, and $\mathcal{D}_{tar}$ compatible with $\mathcal{P}_{tar}$, such that $T_{total} = \mathit{RedistSteps}(\mathcal{D}_{ini} \to \mathcal{D}_{tar}) \times \tau_{comm} + T_{comp}(\mathcal{P}_{tar})$ is minimized.

Note that both problems require that we are able to compute $T_{comp}(\mathcal{P}_{tar})$ for any target data partition $\mathcal{P}_{tar}$. This is realistic only for very simple computational kernels. In Section 4.5, we consider such a kernel, namely the 1D-stencil. We show the NP-completeness of both VOLPART&REDISTRIB and STEPPART&REDISTRIB for this kernel, thereby assessing the difficulty to couple redistribution and computations.

4.4 Redistribution

This section deals with the VOLUMEREDISTRIB and STEPREDISTRIB problems: 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. We show that both problems have polynomial complexity. As a side note, we point out that these results directly extend to the case where we have different numbers of processors for the source and target distributions.

4.4.1 Total communication volume

Theorem 4.1. *Given an initial data distribution $\mathcal{D}_{ini}$ and target data partition $\mathcal{P}_{tar}$, Algorithm 7 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. Using the definition of s_i and r_i from Section 4.3.2.1, the total volume of communications during the redistribution phase from the initial distribution to the target distribution is

$$RedistVol(\mathcal{D}_{ini} \rightarrow \mathcal{D}_{tar}) = \sum_{0 \leq i \leq P-1} s_i = \sum_{0 \leq i \leq P-1} r_i.$$

Solving VOLUMEREDISTRIB 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 7 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 [94]). Here n=P and m=P^2, hence the overall complexity of Algorithm 7 is $O(NP^2 + P^3)$. ■

Note that, in Algorithm 7, the complexity of computing egde weights may easily be reduced to $O(NP + P^2)$: (i) we first initialize all weights to 0 (in $O(P^2)$), (ii) then, for each data item x and each $i \neq \mathcal{D}_{ini}(x)$, the weight of edge $(i, \mathcal{P}_{tar}(x))$ is incremented (in $O(NP)$). With this optimization, the complexity of Algorithm 7 can be reduced to $O(NP + P^3)$.

4.4.2 Number of parallel communication 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 4.2. *Given an initial data distribution $\mathcal{D}_{ini}$ and target data partition $\mathcal{P}_{tar}$, Algorithm 8 computes a data distribution $\mathcal{D}_{tar}$ compatible with $\mathcal{P}_{tar}$ such that $RedistSteps(\mathcal{D}_{ini} \rightarrow \mathcal{D}_{tar})$ is minimized, and its complexity is $O(NP^2 + P^{\frac{9}{2}})$.*

Algorithm 7: 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 $\mathcal{D}_{tar}$

Proof. First, given an initial data distribution $\mathcal{D}_{ini}$ and a target distribution $\mathcal{D}_{tar}$, we can compute $RedistSteps(\mathcal{D}_{ini} \to \mathcal{D}_{tar})$ as

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

This well-known result [38] is a direct consequence of König's theorem (see Theorem 20.1 in [94]) stating that the edge-coloring number of a bipartite multigraph is equal to its maximum degree.

Algorithm 8 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 optimal solution to STEPREDISTRIB. 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 8 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 8 prunes edges starting from the heaviest ones, these edges are still in G when Algorithm 8 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 8. Thus $m_{ret} = m_{opt}$ and the matching returned by Algorithm 8 is a solution to STEPREDISTRIB.

Again, computing edge weights can be done with complexity $O(NP^2 + P^2)$. Algorithm 8 uses the Hopcroft–Karp Algorithm [69] to find the maximum cardinality matching of 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 8 has a worst-case complexity of $O(NP^2 + P^{\frac{9}{2}})$. ■

Note that, like in previous section, the complexity of Algorithm 8 can easily be reduced to $O(NP + P^{\frac{9}{2}})$ with an optimized weight computation.

Algorithm 8: 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 $RedistSteps(\mathcal{D}_{ini} \rightarrow \mathcal{D}_{tar})$ is minimized
$A \leftarrow \{0, \dots, P-1\}$ (set of processors)
$B \leftarrow \{0, \dots, P-1\}$ (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 $\mathcal{M}_{save}$

4.4.3 Evaluation of optimal vs. arbitrary redistributions

In this section, we conduct several simulations to illustrate the interest of the two algorithms introduced above. In particular, we show that in many cases, it is important to optimize the mapping rather than resorting to an arbitrary mapping which could induce many more communications. Source code for the algorithms and simulations is publicly available at http://perso.ens-lyon.fr/julien.herrmann/.

4.4.3.1 Random balanced initial data distribution

First we consider a random balanced initial data distribution $\mathcal{D}_{ini}$, where each processor initially hosts D data items, and each data item has the same probability to reside on any processor. Most parallel applications require perfect load balancing to achieve good performance, and thus a balanced data partition. Therefore, we consider here a balanced target data partition $\mathcal{P}_{tar}$ (each of its P components includes D data items). We denote by $\mathcal{D}_{can}$ the canonical data distribution (compatible with partition $\mathcal{P}_{tar}$) which maps its jth component onto processor j.

As seen in Section 4.3, the volume of communication involved during the redistribution from $\mathcal{D}_{ini}$ to $\mathcal{D}_{can}$ is $RedistVol(\mathcal{D}_{ini} \rightarrow \mathcal{D}_{can}) = \sum_{0 \leq j \leq P-1} |\{x \text{ s.t. } \mathcal{P}_{tar}(x) = j \text{ and } \mathcal{D}_{ini}(x) \neq j\}|$. Since each component of $\mathcal{P}_{tar}$ has cardinal D and $\mathcal{D}_{ini}(x)$ is equal to j with a probability $\frac{1}{P}$ for any processor j and any data item x, we can compute the expected volume of communication: $E(RedistVol(\mathcal{D}_{ini} \rightarrow \mathcal{D}_{can})) = D(P-1)$. Thus, picking an arbitrary target distribution leads to an average volume of communications linear in P.

Each processor hosts D data items at the beginning and at the end of the redistribution phase. Thus, according to Section 4.4.2, the number of steps required to schedule the redistribution phase is equal to D if and only if one of the P processors has to send its complete initial data set during the redistribution

phase. This happens with probability

$$p = 1 - \left(1 - \left(\frac{P-1}{P}\right)^D\right)^P.$$

This probability is equal to 0.986 for $P = 10$ and $D = 10$, and is non-decreasing with P, which means that the worst number of steps is reached in almost all cases for average values of D. This shows that picking an arbitrary data distribution $\mathcal{D}_{can}$ is suboptimal most of the time. Instead, we can use Algorithm 7 to find the data distribution $\mathcal{D}_{vol}$ that minimizes the volume of communications involved in the redistribution phase, and Algorithm 8 to find the data distribution $\mathcal{D}_{steps}$ that minimizes the number of steps of the redistribution phase. Figure 4.2 depicts the relative volume of communications and the relative number of redistribution steps when using target data distributions $\mathcal{D}_{vol}$ and $\mathcal{D}_{steps}$. The results are normalized with the performance of the arbitrary target distribution $\mathcal{D}_{can}$. The simulations have been conducted with $P = 32$ processors and up to $D = 20$ data items residing on each of them. These values correspond to an application dealing with $32 \times 20 = 640$ data items and running on a distributed cluster of 32 processors, which is a realistic problem size when considering linear algebra problems, since each data item is a matrix tile. For these values, the arbitrary target distribution $\mathcal{D}_{can}$ requires on average 620 communications and involves 20 parallel steps with a probability larger than $1 - 3.3 \times 10^{-11}$. Each point in Figure 4.2 represents the average results and the standard deviation on a set of 50 random initial distributions. The best data distributions for the communication volume and for the communication steps represent a 10% improvement compared to an arbitrary target distribution when $D \geq 10$, and a larger improvement for smaller values of D. The results for these two data distributions are really close and present a small standard deviation.

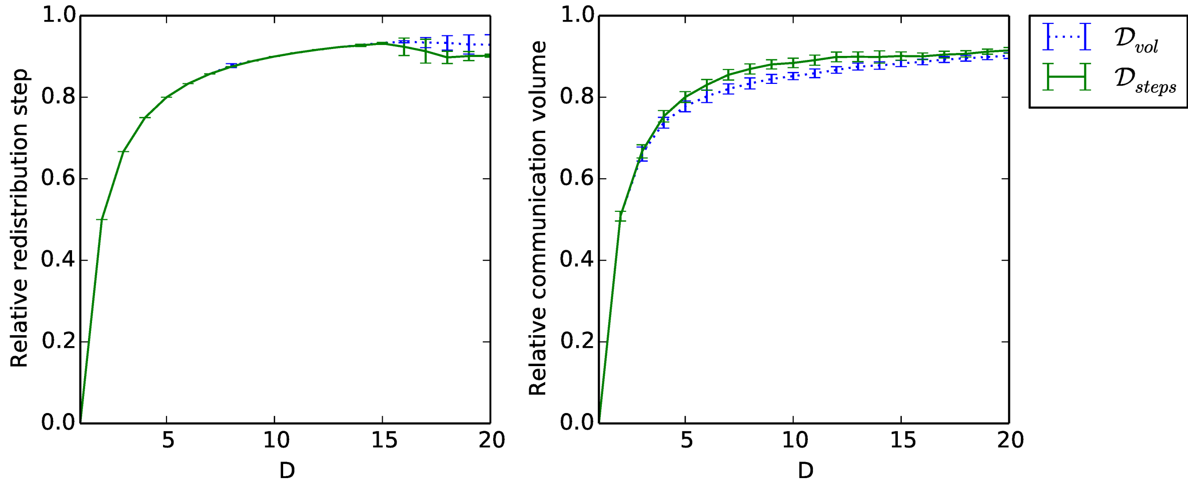


Figure 4.2: Performance of $\mathcal{D}_{vol}$ (computed by Algorithm 7) and $\mathcal{D}_{steps}$ (computed by Algorithm 8) relatively to the canonical distribution, for a random initial distribution and for both metrics.

4.4.3.2 Skewed balanced initial data distribution

Real world data distributions are usually not random. Some data are more likely to be initially hosted by some particular processor. In this section, we show the possible gain of using the proposed algorithms for skewed initial distributions. We consider a balanced target data partition $\mathcal{P}_{tar}$ where each of its P

components includes D elements of data. For $0 \leq \alpha \leq 1$, we note $\mathcal{D}_{ini}^{\alpha}$ the initial data distribution which maps $\lfloor \alpha D \rfloor$ data items of the jth component of $\mathcal{P}_{tar}$ on processor $(j+1) \mod P$, and which randomly maps the other $D - \lfloor \alpha D \rfloor$ data items of this component to all P processors. Note that $\mathcal{D}_{ini}^{0}$ represents a random balanced data distribution as studied in previous section.

We still use $\mathcal{D}_{can}$, the arbitrary target distribution which maps the jth component of $\mathcal{P}_{tar}$ onto processor j, as a comparison basis. During the redistribution phase from $\mathcal{D}_{ini}^{\alpha}$ to $\mathcal{D}_{can}$, each processor sends at least $\lfloor \alpha D \rfloor$ of its elements. With the skewed distribution, we can compute the expected volume of communications of $\mathcal{D}_{can}$: $E(RedistVol(\mathcal{D}_{ini}^{\alpha} \rightarrow \mathcal{D}_{can})) = D(P-1) + \lfloor \alpha D \rfloor$. The number of steps required to schedule the redistribution phase from $\mathcal{D}_{ini}^{\alpha}$ to $\mathcal{D}_{can}$ is equal to D with probability $1 - \left(1 - (\frac{P-1}{P})^{D - \lfloor \alpha D \rfloor}\right)^{P}$.

Figure 4.3 depicts the relative volume of communication and the relative number of redistribution steps for the target distributions $\mathcal{D}_{vol}$ (obtained with Algorithm 7) and $\mathcal{D}_{steps}$ (obtained with Algorithm 8), normalized with the performance of the arbitrary target distribution $\mathcal{D}_{can}$. The simulations have been conducted with $P = 32$ processors, $D = 20$ elements of data on each of them and α varying from 0 to 1. When α is close to 0, $\mathcal{D}_{ini}^{\alpha}$ is close to a random balanced data distribution and we retrieve the results of the previous section. When α is larger than 0.2, for any j, the proportion of data in the jth component of $\mathcal{P}_{tar}$ that are initially hosted by processor $(j+1) \mod P$ is significant. Thus, mapping this component onto processor $(j+1) \mod P$ becomes the best solution to reduce both the volume of communication and the number of communication steps. In this case, Algorithm 7 and Algorithm 8 provide the same target data distribution. Both objectives decrease linearly with α since the proportion of data that are initially mapped onto the correct processor increases linearly with α.

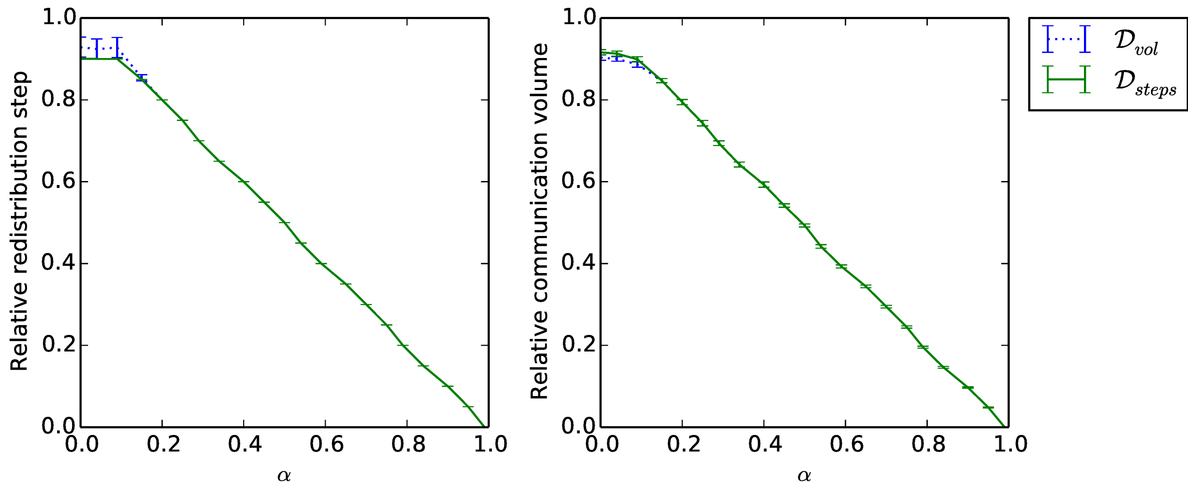


Figure 4.3: Performance of $\mathcal{D}_{vol}$ (computed by Algorithm 7) and $\mathcal{D}_{steps}$ (computed by Algorithm 8) relatively to the canonical distribution, for a skewed initial distribution and for both metrics.

4.5 Coupling redistribution and stencil computations

In this section, we focus on a simple, yet realistic, application to assess the complexity of redistribution when coupled to a computational kernel. We consider a 1D-stencil iterative algorithm, which updates the elements of an array in parallel, according to the value of their direct neighbors. Stencil computations are widely used to numerically solve partial differential equations [98]. We first detail the application

model before establishing the NP-completeness of minimizing the cost of a redistribution followed by the execution of the kernel.

4.5.1 Application model

We consider here a three-point stencil with circular arrangement of the data. More precisely, to compute the value $x(i,t)$ of the data at position i at step t, we need its value and those of its left and right neighbors at the previous step, namely $x(i,t-1)$, $x(i-1 \mod N, t-1)$, and $x(i+1 \mod N, t-1)$. If the neighbors are not stored on the same processor, their value has to be received from the processors hosting them. Thus, each iteration of the stencil algorithm consists in two phases, the *communication phase* when the value of each data item is sent to the processors hosting its neighbors, and the *computation phase*, when each data item is updated according to a given kernel using these values (see Algorithm 9). The update kernel depends on the application.

Algorithm 9: One iteration of the unidimensional stencil algorithm

Result: N data items numbered from 0 to $N-1$ and their distribution $\mathcal{D}$ on P processors
for $0 \le x \le N-1$ *in parallel* **do**
    $\ell_x \leftarrow (x-1) \mod N$;
    $r_x \leftarrow (x+1) \mod N$;
    if $\mathcal{D}(\ell_x) \neq \mathcal{D}(x)$ **then**
        Processor $\mathcal{D}(x)$ receives data item ℓ_x from processor $\mathcal{D}(\ell_x)$;
    if $\mathcal{D}(r_x) \neq \mathcal{D}(x)$ **then**
        Processor $\mathcal{D}(x)$ receives data item r_x from processor $\mathcal{D}(r_x)$;
for $0 \le x \le N-1$ *in parallel* **do**
    Processor $\mathcal{D}(x)$ updates data item x using ℓ_x and r_x;

Given a data partition $\mathcal{P}_{tar}$, let N_{ij} be the number of data items sent by the processor hosting the ith component of $\mathcal{P}_{tar}$ to the processor hosting the jth component during one communication phase of the stencil algorithm: N_{ij} is the number of left or right neighbors in the ith component of data items in the jth component. Formally:

$$N_{ij} = |\{0 \le x \le N-1 \text{ s.t. } \mathcal{P}_{tar}(x) = i \text{ and } (\mathcal{P}_{tar}(x-1 \mod N) = j \text{ or } \mathcal{P}_{tar}(x+1 \mod N) = j)\}|.$$

The workload ℓ_i of the processor i hosting the ith component of $\mathcal{P}_{tar}$ is:

$$\ell_i = |\{0 \le x \le N-1 \text{ s.t. } \mathcal{P}_{tar}(x) = i\}|.$$

Given a data partition $\mathcal{P}_{tar}$, the running time of the stencil algorithm depends on the communication model, but not on the actual data distribution, provided that it is compatible with $\mathcal{P}_{tar}$. Let τ_{comm} be the time needed to perform one communication (see Section 4.3.3), and let τ_{calc} be the time needed to perform one data update for the considered stencil application. The processing time for K iterations of the stencil with the two communication models is the following (using the notations of Section 4.3.3):

- **Total volume:** For problem VOLPART&REDISTRIB, $T_{comp}(\mathcal{P}_{tar}) = K \times T_{vol}^{iter}(\mathcal{P}_{tar})$, where

$$\begin{aligned} T_{vol}^{iter}(\mathcal{P}_{tar}) = \ & \tau_{comm} \times \left(\sum_{0 \le i \le P-1} \sum_{j \neq i} N_{ij} \right) \\ & + \tau_{calc} \times \max_{0 \le i \le P-1} \ell_i \end{aligned}.$$

The first term corresponds to the serialization of all communications, and the second one to the parallel processing of the updates.

- **Number of parallel steps:** For problem STEPPART&REDISTRIB, $T_{comp}(\mathcal{P}_{tar}) = K \times T^{iter}_{steps}(\mathcal{P}_{tar})$, where

$$T^{iter}_{steps}(\mathcal{P}_{tar}) = \begin{array}{l} \tau_{comm} \times \max\limits_{0 \le i \le P-1} \left(\sum\limits_{j \ne i} N_{ij}, \sum\limits_{j \ne i} N_{ji} \right) \\ + \tau_{calc} \times \max\limits_{0 \le i \le P-1} \ell_i \end{array}.$$

Here the first term corresponds to the time needed to perform the required number of communication steps, and the second term is unchanged.

4.5.2 Complexity

Assume without loss of generality that N is a multiple of P. There is a well-known optimal data partition for the 1D-stencil kernel, namely the full block partition (data item i is assigned to component $\lfloor iP/N \rfloor$). This *canonical* partition $\mathcal{P}_{can}$ minimizes the duration of the communication phase (only two items are sent/received per component of the partition) and the computation phase is perfectly balanced.

Starting from an initial data distribution $\mathcal{D}_{ini}$, we can use either Algorithm 7 or 8 to find a target distribution $\mathcal{D}_{tar}$ which is compatible with the full-block partition $\mathcal{P}_{can}$ and whose redistribution cost is minimal. However, redistributing from $\mathcal{D}_{ini}$ to $\mathcal{D}_{tar}$ may induce a large overhead on the total execution time, which is fully justified only when the number of iterations K is large enough. It may be useful to avoid a costly redistribution for small values of K, and to find a target redistribution which is a trade-off between minimizing redistribution time and processing time. Actually, finding such a trade-off distribution is an NP-complete problem for both communication models. We define the two decision problems associated to VOLPART&REDISTRIB and STEPPART&REDISTRIB:

Definition 4.8 (DECISIONVOLPART&REDISTRIB)**.** 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 $\mathcal{D}_{ini}$ and a bound T_{MAX}, are there a partition $\mathcal{P}_{tar}$, and a distribution $\mathcal{D}_{tar}$ compatible with $\mathcal{P}_{tar}$, such that:
$T_{total}(\mathcal{D}_{ini}, \mathcal{D}_{tar}) = \mathit{RedistVol}(\mathcal{D}_{ini} \to \mathcal{D}_{tar}) \times \tau_{comm} + T_{comp}(\mathcal{P}_{tar}) \le T_{MAX}$?

Definition 4.9 (DECISIONSTEPPART&REDISTRIB)**.** 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 $\mathcal{D}_{ini}$ and a bound T_{MAX}, are there a partition $\mathcal{P}_{tar}$, and a distribution $\mathcal{D}_{tar}$ compatible with $\mathcal{P}_{tar}$, such that:
$T_{total}(\mathcal{D}_{ini}, \mathcal{D}_{tar}) = \mathit{RedistSteps}(\mathcal{D}_{ini} \to \mathcal{D}_{tar}) \times \tau_{comm} + T_{comp}(\mathcal{P}_{tar}) \le T_{MAX}$?

Theorem 4.3. *The* DECISIONVOLPART&REDISTRIB *problem with the 1D-stencil kernel is strongly NP-complete.*

Proof. We first prove that DECISIONVOLPART&REDISTRIB belongs to NP. Given $\mathcal{D}_{tar}$, it is possible to compute in polynomial time the redistribution time $\mathit{RedistVol}(\mathcal{D}_{ini} \to \mathcal{D}_{tar}) \times \tau_{comm}$ and the cost of the K iterations of the stencil algorithm $T_{comp}(\mathcal{P}_{tar})$, and thus to check whether T_{total} is smaller than T_{MAX} or not. Thus, DECISIONVOLPART&REDISTRIB is in NP.

To establish the completeness, we use a reduction from the 3-Partition problem, which is known to be NP-complete in the strong sense [49]. We consider the following instance $Inst_0$ of the 3-Partition problem: let a_i^0, $1 \le i \le 3m$, be $3m$ integers and B^0 an integer such that $\sum a_i^0 = mB^0$. We enforce the additional (usual) constraint that $\forall i, B^0/4 < a_i^0 < B^0/2$ [49]. To solve $Inst_0$, we need to solve the following question: is there a partition of the a_i^0's in m subsets $S_1, ..., S_m$

such that, $\forall S_k, \sum_{i \in S_k} a_i^0 = B^0$. Note that if there is a solution, then each subset will contain exactly 3 elements, due to the additional constraint.

We then transform (in polynomial time) $Inst_0$ into another instance $Inst_1$ of the 3-Partition problem as follows: let $a_i = 385m \times a_i^0$ for $1 \leq i \leq 3m$ and $B = 385m \times B^0$. Obviously, $Inst_1$ has a solution if and only if $Inst_0$ has a solution. This new instance of the 3-Partition problem has the following properties:

- $\forall \mathbf{i}, \mathbf{B/4} < \mathbf{a_i} < \mathbf{B/2}$, since $\forall i, B^0/4 < a_i^0 < B^0/2$
- $\mathbf{B^2} > \mathbf{5m} \times \mathbf{B} + \mathbf{96m}$ since $\frac{1}{m}(B^2 - 5m \times B) = 385mB^0(385B^0 - 5) > 96$ because $m \geq 1$ and $B^0 \geq 1$.
- $\mathbf{B} > \mathbf{384m}$ since $B^0 \geq 1$.

Given $Inst_1$, we build the following instance $Inst_2$ of the DECISIONVOLPART&REDISTRIB problem, illustrated in Figure 4.4. In $Inst_2$, we set the number of processors to $P = 12m$, the number of 1D-stencil steps to $K = 1$, elementary communication and computing times $\tau_{comm} = 1$ and $\tau_{calc} = B^2$. We also set the time bound to $T_{\text{MAX}} = 96m + 5mB + 8B^3$. Finally, Figure 4.4 represents the initial data distribution $\mathcal{D}_{ini}$ of $96mB$ elements on the $12m$ different processors. To clarify the proof, we split the $12m$ processors into 4 different groups. There are $3m$ processors in group 1, m processors in group 2, $4m$ processors in group 3 and $4m$ processors in group 4. Processors in group k are denoted by $P_i^{(k)}$. Figure 4.4 depicts the initial data distribution $\mathcal{D}_{ini}$. For example, the $2B$ first consecutive elements are stored on $P_1^{(1)}$, the first processor in group 1. The next $2B$ elements are stored on $P_1^{(4)}$, the first processor in group 4. Note that in the fifth set of $3m$ block values (a_1, $2B$, a_2, $2B, \ldots, a_{3m}$, $2B$), the $3m$ blocks of size $2B$ are distributed on the m group-4 processors $P_{3m+1}, \ldots, P_{4m}$ in a round robin way (the first block goes to P_{3m+1}, the second one to $P_{3m+2}, \ldots$, the m-th block goes to P_{4m}, the $m+1$-th goes ot P_{3m+1}, etc.

The construction of $Inst_2$ is polynomial in the size of $Inst_1$, and thus, in the size of $Inst_0$. We show that $Inst_2$ has a solution if and only if $Inst_1$ has a solution.

We first assume that $Inst_2$ has a solution and we let $\mathcal{D}_{tar}$ denote be the final distribution of data. A *connected component* of processor p is defined as a set of consecutive items hosted on processor p. A *maximal connected component* of processor p is a connected component of processor p which is not strictly included in another connected component of processor p. For instance, in $\mathcal{D}_{ini}$ depicted in Figure 4.4, each group-1 processor has 5 maximal connected components in $\mathcal{D}_{ini}$. Let C_p be the number of maximal connected components on processor p for the distribution $\mathcal{D}_{tar}$. At each stencil step, each processor has to send only the two items at each border of each of its maximal connected components. Thus, with $l_p = |\{x \text{ s.t. } \mathcal{D}_{tar}(x) = p\}|$ being the workload of processor p as introduced in Section 4.5.1, $T_{vol}^{iter}(\mathcal{P}_{tar}) = 2 \times \sum_p C_p + B^2 \times \max_p l_p$, and $T_{total}(\mathcal{D}_{ini}, \mathcal{D}_{tar}) = \textit{RedistVol}(\mathcal{D}_{ini} \rightarrow \mathcal{D}_{tar}) + 2 \times \sum_p C_p + B^2 \times \max_p l_p$. Since, $\mathcal{D}_{tar}$ is a solution to $Inst_2$, we know that:

$$\textit{RedistVol}(\mathcal{D}_{ini} \rightarrow \mathcal{D}_{tar}) + 2 \times \sum_p C_p + B^2 \times \max_p l_p \leq 96m + 5mB + 8B^3. \tag{4.1}$$

Let us first show that $\forall \mathbf{p}, \ \mathbf{l_p} = \mathbf{8B}$:

- $\max_p l_p \leq 8B$ because otherwise, we would have:

$$T_{vol}^{stencil}(\mathcal{D}_{ini}, \mathcal{D}_{tar}) \geq B^2 \times \max_p l_p \geq B^2 \times (8B+1) > T_{\text{MAX}},$$

  since $B^2 > 5m \times B + 96m$.

- There are a total of $96mB$ elements of data and $12m$ processors, thus $\forall p, \ l_p = 8B$.

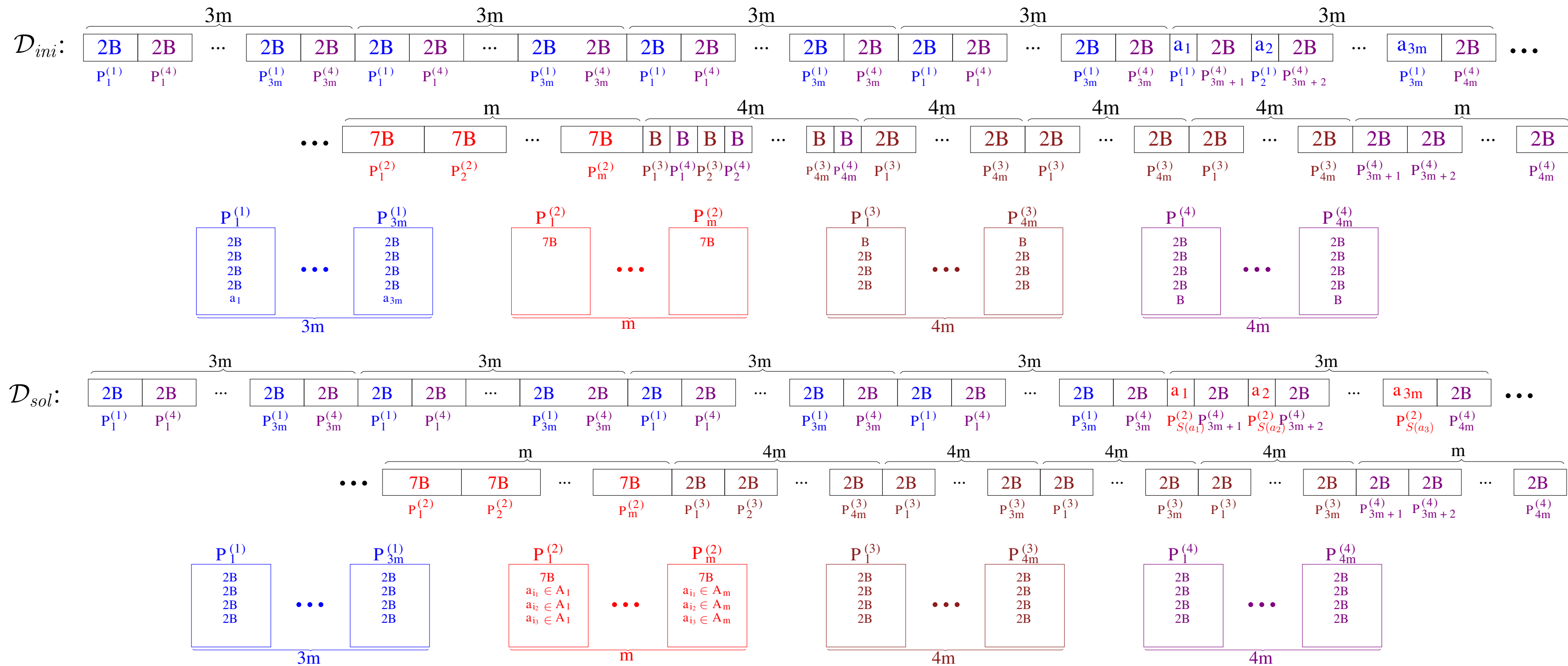


Figure 4.4: $\mathcal{D}_{ini}$ and $\mathcal{D}_{sol}$ in the proof of Theorems 4.3 and 4.4.

Thus $\max_p l_p = 8B$ and Equation 4.1 becomes:

$$RedistVol(\mathcal{D}_{ini} \rightarrow \mathcal{D}_{tar}) + 2 \times \sum_p C_p \leq 96m + 5mB. \tag{4.2}$$

For each processor $P_i^{(k)}$, let $S_i^{(k)}$ (respectively $R_i^{(k)}$) be the number of elements sent (respectively received) by processor $P_i^{(k)}$ during the redistribution phase. We naturally have

$$\sum_{k,i} S_i^{(k)} = \sum_{k,i} R_i^{(k)} = RedistVol(\mathcal{D}_{ini} \rightarrow \mathcal{D}_{tar}).$$

Let us show that $\sum_p C_p \geq 48m$. Initially, in $\mathcal{D}_{ini}$, there are $52m$ maximal connected components among all the processors. There are only two different ways to decrease the global number of maximal connected components: merging two existing connected components by receiving all the data between them, or sending one entire maximal connected component to one of the processors that host a maximal connected component next to it. We first consider the first option. In $\mathcal{D}_{ini}$, two maximal connected components hosted by the same processor are separated by more than $6mB$ elements. Thus, to merge two existing maximal connected components in $\mathcal{D}_{ini}$, a processor would have to receive more than $6mB$ elements during the redistribution phase, which is not possible according to Equation 4.2, since $B > 384m$. We now consider the second option (sending one entire maximal connected component).

- Let assume that a processor $P_i^{(1)}$ sends one of its entire maximal connected components to one of its neighbors. The only neighbors of $P_i^{(1)}$ are processors of group-4. This means that a processor $P_j^{(4)}$ will receive at least $B/4$ elements from $P_i^{(1)}$ during the redistribution phase, since $\forall i, B/4 < a_i$. However, at the end of the redistribution phase, $P_j^{(4)}$ can only host $8B$ elements, thus it will have to send at least $\frac{5}{4}B$ elements during the redistribution phase:

$$RedistVol(\mathcal{D}_{ini} \rightarrow \mathcal{D}_{tar}) = \sum_{(k,p) s.t. (k,p) \neq (4,j)} S_p^{(k)} + S_j^{(4)} \geq 5mB + B/4,$$

which is not possible according to Equation 4.2 and since $B > 384m$.

- Let assume that a processor $P_i^{(2)}$ sends one of its entire maximal connected components to one of its neighbors. This means that processor $P_i^{(2)}$ will send at least $7B$ elements during the redistribution phase and thus, we will have

$$RedistVol(\mathcal{D}_{ini} \rightarrow \mathcal{D}_{tar}) = \sum_{(k,p) s.t. (k,p) \neq (2,i)} S_p^{(k)} + S_i^{(2)} \geq 5mB + 7B,$$

which again, is not possible according to Equation 4.2 and since $B > 384m$.

- Let assume that a processor $P_i^{(3)}$ sends one of its entire maximal connected components to one of its neighbors. This means that $P_i^{(3)}$ will send at least B elements during the redistributing phase and thus, we will have

$$RedistVol(\mathcal{D}_{ini} \rightarrow \mathcal{D}_{tar}) = \sum_{(k,p) s.t. (k,p) \neq (3,i)} S_p^{(k)} + S_i^{(3)} \geq 5mB + B,$$

which, again, is not possible.

Thus, the only remaining option to decrease the number of maximal connected component is that some processor $P_i^{(4)}$ sends at least one entire connected component to one of its neighbors. Assume that it sends at least two of its entire maximal connected components to one of its neighbors. This means that $P_i^{(4)}$ will send at least $3B$ elements during the redistributing phase and thus, we will have

$$RedistVol(\mathcal{D}_{ini} \rightarrow \mathcal{D}_{tar}) = \sum_{(k,p) s.t. (k,p) \neq (3,i)} S_p^{(k)} + S_i^{(4)} \geq 5mB + 2B,$$

which, again, is not possible according to Equation 4.2 and since $B > 384m$.

Thus each processor $P_i^{(4)}$ can send only one of its entire maximal connected components to one of its neighbors. There are $4m$ processors in group-4, so we can reduce the number of maximal connected components by only $4m$. Since there are $52m$ maximal connected components in $\mathcal{D}_{ini}$, we have in $\mathcal{D}_{tar}$:

$$\sum_p C_p \geq 48m. \tag{4.3}$$

Then, we show that $RedistVol(\mathcal{D}_{ini} \rightarrow \mathcal{D}_{tar}) = 5mB$.

- Equation 4.2 and Equation 4.3 lead to: $RedistVol(\mathcal{D}_{ini} \rightarrow \mathcal{D}_{tar}) = \sum_{k,i} R_i^{(k)} \leq 5mB$
- Initially, in $\mathcal{D}_{ini}$, each processor in group-2 or group-3 hosts $7B$ elements of data and since $\forall p,\ l_p = 8B$ in $\mathcal{D}_{tar}$, the group-2 and group-3 processors each have to receive at least B elements of data during the redistribution phase ($\forall i, R_i^{(2)} \geq B$ and $R_i^{(3)} \geq B$). Thus at least $5mB$ elements of data have to be communicated during the redistribution phase:

$$RedistVol(\mathcal{D}_{ini} \rightarrow \mathcal{D}_{tar}) = \sum_{k,i} R_i^{(k)} \geq 5mB.$$

Thus $RedistVol(\mathcal{D}_{ini} \rightarrow \mathcal{D}_{tar}) = 5mB$ and Equation 4.2 becomes:

$$\sum_p C_p = 48m. \tag{4.4}$$

We now bound the number of elements sent and received by processors in group 2, 3 and 4.

- Each group-2 processor $P_i^{(2)}$ and each group-3 processor $P_i^{(3)}$ hosts $7B$ elements in the initial distribution $\mathcal{D}_{ini}$, and $8B$ elements in the final distribution $\mathcal{D}_{tar}$. Thus, they each have to receive at least B elements of data. There are $5m$ of them, so they can receive only B elements of data each, and no other processor can receive any data. More formally, we have $\forall i$: $R_i^{(1)} = 0$, $R_i^{(2)} = B$, $R_i^{(3)} = B$ and $R_i^{(4)} = 0$.
- Each group-1 processor $P_i^{(1)}$ hosts $8B + a_i$ elements in $\mathcal{D}_{ini}$, and $8B$ elements in $\mathcal{D}_{tar}$. Each group-4 processor $P_i^{(4)}$ hosts $9B$ elements in $\mathcal{D}_{ini}$, and $8B$ elements in $\mathcal{D}_{tar}$. Again, this means that each group-1 processor $P_i^{(1)}$ can send only a_i elements of data, each group-4 processor $P_i^{(4)}$ can send only B elements of data and no other processors can send any data. That is, $\forall i$: $S_i^{(1)} = a_i$, $S_i^{(2)} = 0$, $S_i^{(3)} = 0$ and $S_i^{(4)} = B$.

Since $\sum_p C_p = 48m$, each group-4 processor has to send one and only one of its entire maximal connected components to one of its neighbor (as we have seen earlier, there is no other way to decrease the global number of maximal connected components). Since $\forall i$: $S_i^{(4)} = B$, group-4 processors can only send their maximal connected component of size B. The only neighbors of these maximal connected components are some group-3 processors. Thus, each group-4 processor will send its entire maximal connected component of size B to a group-3 processor, and group-3 processors can not receive anything else from any other processor.

Gathering all the results shown above, we can state that group-1 processors can only send their a_i elements to group-2 processors during the redistribution phase. If a processor $P_i^{(1)}$ splits its a_i consecutive elements and send them to two different group-2 processors, this would create an extra maximal connected component on the group-2 processors. Thus, **each group-1 processor has to send its $\mathbf{a_i}$ elements to the same group-2 processor**.

Let A_k be the set of the sizes of the maximal connected components received by $P_k^{(2)}$ during the redistribution phase. The A_k sets represent a partition of the a_i's and $\forall k, \sum_{a_i \in A_k} a_i = R_k^{(2)} = B$. Hence the A_k sets are a solution of $Inst_1$.

Suppose now that $Inst_1$ has a solution. Let A_k be the 3-Partition of the integers a_i and consider the distribution $\mathcal{D}_{sol}$ described in Figure 4.4. To perform the redistribution from $\mathcal{D}_{ini}$ to $\mathcal{D}_{sol}$, each group-2 and group-3 processors has to send or receive B elements of data, which means that $RedistVol(\mathcal{D}_{ini} \rightarrow \mathcal{D}_{sol}) = 5mB$. In addition, in $\mathcal{D}_{sol}$, there are $48m$ maximal connected components. Thus, $T_{total}(\mathcal{D}_{ini}, \mathcal{D}_{sol}) = 96m + 5mB + 8B^3 \leq T_{MAX}$, which means that $Inst_2$ has a solution. This concludes the proof. ■

Theorem 4.4. STEPPART&REDISTRIB *problem with the 1D-stencil kernel is strongly NP-complete.*

Proof. The proof of Theorem 4.4 is similar to that of Theorem 4.3. We consider the same instances $Inst_0$ and $Inst_1$ of the 3-Partition problem [49]. We build the instance $Inst_2$ depicted in Figure 4.4 for the DECISIONSTEPPART&REDISTRIB problem, as in the previous proof, except that we now set $T_{\text{MAX}} = 8 + B + 8B^3$. We want to show that $Inst_2$ has a solution if and only if $Inst_1$ has a solution.

Assume first that $Inst_2$ has a solution and use the same notations as above. We have the inequality:

$$\begin{aligned} T_{total}(\mathcal{D}_{ini}, \mathcal{D}_{tar}) &= RedistSteps(\mathcal{D}_{ini} \rightarrow \mathcal{D}_{tar}) + 2 \times \max_p C_p + B^2 \times \max_p l_p \\ &\leq 8 + B + 8B^3. \end{aligned} \tag{4.5}$$

As in the previous proof, we can easily show that $\forall p,\ l_p = 8B$. Thus, Equation 4.5 becomes:

$$RedistSteps(\mathcal{D}_{ini} \rightarrow \mathcal{D}_{tar}) + 2 \times \max_p C_p \leq 8 + B. \tag{4.6}$$

Then we show that $RedistSteps(\mathcal{D}_{ini} \rightarrow \mathcal{D}_{tar}) = \max_{(p,k)} max(S_p^{(k)}, R_p^{(k)}) = B$:

- Initially, in $\mathcal{D}_{ini}$, the processor $P_1^{(4)}$ hosts $9B$ elements of data; since $\max_p l_p = 8B$ in $\mathcal{D}_{tar}$, the processor $P_1^{(4)}$ has to send at least B elements of data during the redistribution phase. Thus, $S_1^{(4)} \geq B$ and $RedistSteps(\mathcal{D}_{ini} \rightarrow \mathcal{D}_{tar}) \geq B$.

- If one processor sends $B + 1$ elements of data during the redistribution phase, we would have $RedistSteps(\mathcal{D}_{ini} \rightarrow \mathcal{D}_{tar}) \geq B + 1$ and $2 \times \max_p C_p \leq 7$, so $\max_p C_p \leq 3$. Initially, in $\mathcal{D}_{ini}$, processor $P_1^{(1)}$ hosts 5 maximal connected components and could have at most 3 maximal

connected components in $\mathcal{D}_{tar}$. There are only two different ways to decrease the number of maximal connected components in a processor: sending one entire maximal connected component to another processor or merging two existing connected components by receiving all the data between them. Both options are impossible in this case, because processor $P_1^{(1)}$ would have to send or receive more than $2B$ elements during the redistribution phase, which is impossible according to Equation 4.5.

Thus $RedistSteps(\mathcal{D}_{ini} \rightarrow \mathcal{D}_{tar}) = B$ and Equation 4.6 becomes:

$$\max_p C_p \leq 4. \tag{4.7}$$

We now bound the number of elements sent and received by processors in group 2, 3 and 4. We naturally have

$$\forall P_i^{(k)},\ max(S_i^{(k)}, R_i^{(k)}) \leq RedistSteps(\mathcal{D}_{ini} \rightarrow \mathcal{D}_{tar}) = B.$$

- Each group-2 processor $P_i^{(2)}$ hosts $7B$ elements in the initial distribution $\mathcal{D}_{ini}$, and $8B$ elements in the final distribution $\mathcal{D}_{tar}$. This means that $R_i^{(2)} - S_i^{(2)} = B$. Since $max(S_i^{(2)}, R_i^{(2)}) \leq B$, we have: $\mathbf{R_i^{(2)} = B}$ and $\mathbf{S_i^{(2)} = 0}$.
- Each group-3 processor $P_i^{(3)}$ hosts $7B$ elements in $\mathcal{D}_{ini}$, and $8B$ elements in $\mathcal{D}_{tar}$. Again, this means that $R_i^{(3)} - S_i^{(3)} = B$ and since $max(S_i^{(3)}, R_i^{(3)}) \leq B$, we necessarily have: $\mathbf{R_i^{(3)} = B}$ and $\mathbf{S_i^{(3)} = 0}$.
- Each group-4 processor $P_i^{(4)}$ hosts $9B$ elements in $\mathcal{D}_{ini}$, and $8B$ elements in $\mathcal{D}_{tar}$. Again, $S_i^{(4)} - R_i^{(4)} = B$ and we have $\mathbf{R_i^{(4)} = 0}$ and $\mathbf{S_i^{(4)} = B}$.

From these results, using the same reasoning as in the previous proof, we can show that:

- group-1 and group-2 processors do not send or receive any data to or from a group-3 or group-4 processor.
- Each group-1 processor does not keep any data received during the redistribution phase.
- Each group-1 processor has to send its $\mathbf{a_i}$ elements to the same group-2 processor.

Let A_k be the set of the sizes of the maximal connected components received by $P_k^{(2)}$ during the redistribution phase: we have shown that the A_k sets are a solution of $Inst_1$.

Suppose now that $Inst_1$ has a solution. As in the previous proof, we can show that the distribution $\mathcal{D}_{sol}$ described in Figure 4.4 is a solution for $Inst_2$, which concludes the proof.

■

4.6 Experiments

The algorithms designed in Section 4.4 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. Section 4.5 proved 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. In this section, we show that the redistribution algorithms introduced in Section 4.4 are good enough to provide performance improvements in real-life applications. The experiments are conducted on a multicore cluster for the 1D-Stencil kernel and, then, for a more compute-intensive dense linear algebra routine, namely the QR factorization.

4.6.1 Setup

We have implemented the 1D-stencil kernel of Section 4.5 on top of the PARSEC runtime [23, 22]. In addition, we have also implemented a QR factorization algorithm on top of PARSEC, in order to experiment with a widely used computation-intensive numerical linear algebra routine.

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 (ICL) in Knoxville, TN. This cluster has 16 multi-core nodes, each equipped with 8 cores, and an InfiniBand 10G interconnection network. Each node features two Intel Westmere-EP E5606 CPUs at 2.13GHz. The system is running the Linux 64bit operating system, version 3.7.2-x86_64. The software was compiled with the Intel Compiler Suite 2013.3.163. BLAS kernels were provided by the MKL library, and OpenMPI 1.4.3 was used for MPI communications by the PARSEC runtime version 1.1. Each computational thread is bound to a single core using the HWLOC 1.7.1 library. We use all 16 nodes, whose aggregated theoretical peak performance is $1,091$ GFLOP/sec.

4.6.2 Stencil

The stencil algorithm described in Algorithm 9 can use diverse patterns to update the data, depending upon the target application. In our experiments, data items operated upon are blocks of 1.6×10^6 double-precision floats. Experimentally, we observe that the average communication time of such blocks between two nodes of *Dancer* is 100 milliseconds. The data items are initially distributed on the 16 processors according to a random balanced distribution as described in Section 4.4.3. We used a set of 30 randomly generated initial data distributions.

Figure 4.5 depicts the performance of the 1D-stencil algorithm when the update kernel takes on average 100 milliseconds to compute the new value of one data item, so that the communication-to-computation ratio is $\tau_{comm}/\tau_{calc} = 1$. Each sub-figure represents a different number of stencil iterations ($K = 0$ to 9). In each sub-figure, we have executed K stencil iterations with 4 different strategies. In the *owner computes* strategy, the data items are not moved and the stencil algorithm 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 $\mathcal{P}_{can}$ described in Section 4.5.2: (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 $\mathcal{D}_{vol}$; and (iii) the distribution that minimizes the number

of redistribution steps $\mathcal{D}_{steps}$. We compute the processing time of the redistribution followed by the K stencil iterations. The time needed to compute the target distributions depends on the number of processors and data items but does not depend on the size of the data items. Usually the size of the data items is large enough for the computation time of the algorithm presented in Section 4.4 to be negligible, therefore it is not included in the figures. Each cross shows the performance for one of the 30 initial data distributions, and the plain lines shows the average performance on the 30 initial data distributions. The first observation is that the standard deviation of the processing time for all the initial data distribution is very small. Moreover, in all sub-figures, we observe that the performances for target distributions $\mathcal{D}_{vol}$ and $\mathcal{D}_{steps}$ are indistinguishable. This is in accordance with the results in Section 4.4.3 showing that, on random balanced initial distributions, Algorithm 7 and Algorithm 8 provide similar performances for both metrics. We observe that the processing time of the three redistribution strategies slightly increases with the number of stencil iterations, i.e., one stencil iteration is very fast (roughly 400 milliseconds for 16 data items per processor) when processed on the optimal data partition described in Section 4.5.2. However, the *owner computes* strategy is less efficient as soon as we have to process more than one stencil iteration. In the top-left sub-figure, $K = 0$ so that no iteration is executed. Data items are moved from their initial processor to their target processor and then moved back onto their initial position. It thus depicts the performance of two consecutive redistributions. The *owner computes* strategy has a processing time close to zero which corresponds to the overhead of the PaRSEC runtime. Both redistribution strategies computed by Algorithm 7 and Algorithm 8 provide a 20% improvement over $\mathcal{D}_{can}$. This improvement decreases when the number of iterations increases. Indeed, the only difference between the performances of $\mathcal{D}_{can}$, $\mathcal{D}_{vol}$ and $\mathcal{D}_{steps}$ comes from the redistribution phase: the heavier the computation, the less significant the redistribution phase.

Figure 4.6 depicts the performance of the 1D-stencil algorithm when the update kernel is less expensive, so that $\tau_{comm}/\tau_{calc} = 10$. Hence, in this experiment, the cost of communicating a data element is greater than the computation time and we have to take special care to the redistribution. With a faster computing kernel, the overall computation time is inferior to the one in Figure 4.5 but the *owner computes* strategy is still less efficient than the redistributing strategies as soon as we have to do more than one iteration. The difference between the performances of $\mathcal{D}_{can}$, $\mathcal{D}_{vol}$ and $\mathcal{D}_{steps}$ is smaller in percentage than in Figure 4.5.

Altogether, the experiments show that (i) redistributing towards a better data distribution is more suitable than performing the algorithm in place with the random initial distribution, as soon as the computational cost in non-negligible; and (ii) redistributing towards a data distribution that minimizes the cost of the redistribution phase rather than towards an arbitrary one does improve the performance, especially when the time to communicate a data item is significant.

4.6.3 QR factorization

In this section, we deal with a more compute-intensive kernel, namely the QR factorization, which consists of decomposing a square matrix A into the product of two matrices $Q \times R$ such that Q is an orthogonal matrix and R is an upper triangular matrix. QR factorization is a widely used linear algebra algorithm for solving linear systems and linear least squares problems.

4.6.3.1 Framework

To optimize performance, the matrix is usually stored in tiled form: A has n tiles per row or column and each tile is a block of $n_b \times n_b$ floating point numbers. The matrix is then factored with a tiled QR factorization algorithm using orthogonal Householder matrices, as in [28, 90]. The $N = n^2$ matrix

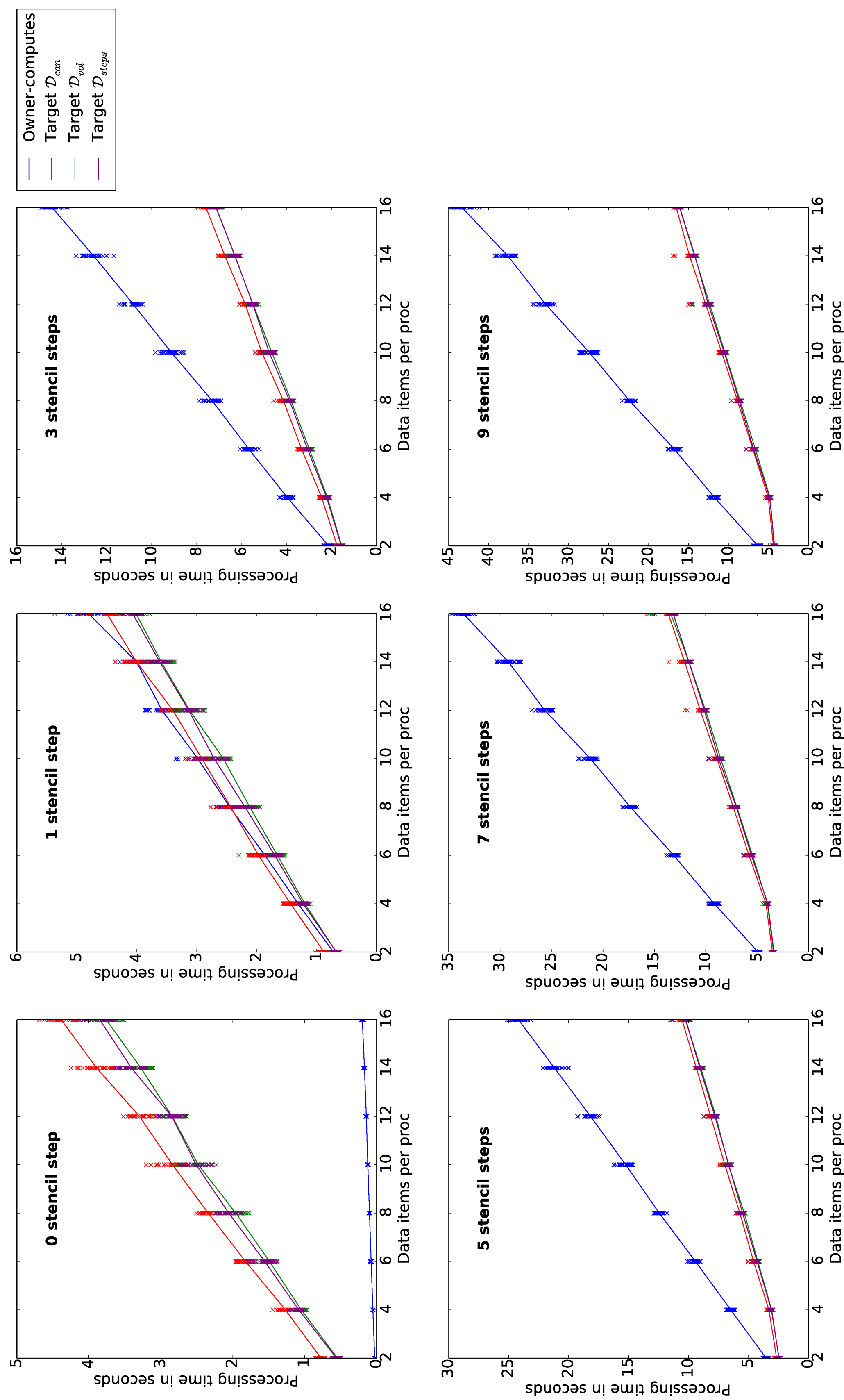


Figure 4.5: Performance of the stencil algorithm for $\tau_{comm}/\tau_{calc} = 1$.

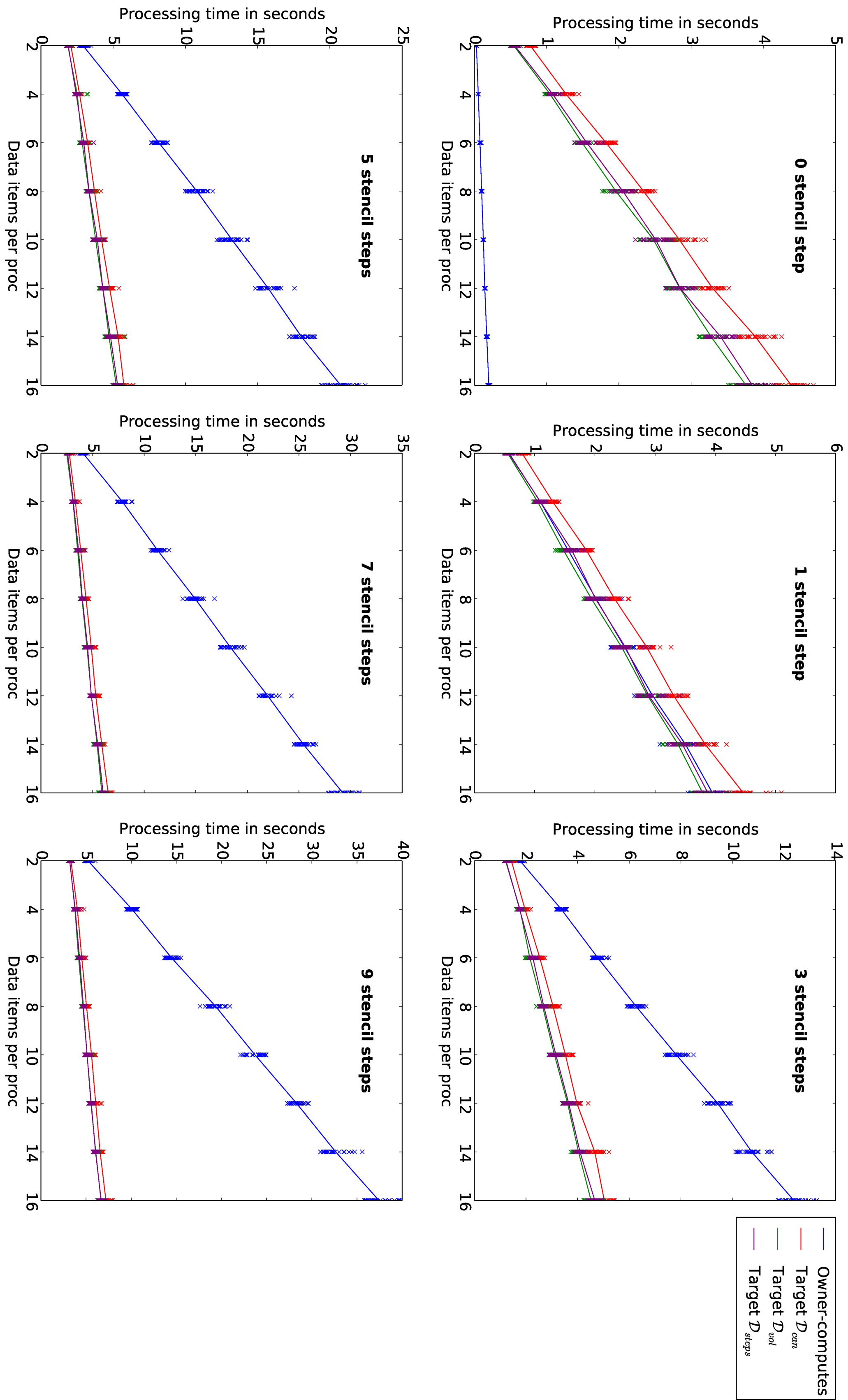


Figure 4.6: Performance of the stencil algorithm for $\tau_{comm}/\tau_{calc} = 10$.

tiles are the data items of the application. Initially, the tiles are arbitrarily distributed, and this initial distribution may not be suitable for the QR factorization. We aim to redistribute the N data items towards a better data partition. However, as opposed to the 1D-stencil algorithm, the QR factorization algorithm has a complex workflow, and it is impossible to predict its processing time accurately: given a data partition $\mathcal{P}$, we cannot easily compute $T_{comp}(\mathcal{P})$.

However, even though there is no explicit model for the cost of a QR factorization performed on a specific data partition, 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 [32]. 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 $(i \mod p) \times p + (j \mod q)$. In the following, this data partition will be referred to as $\mathcal{P}_{tar}$, and the objective is to redistribute the N data items towards a distribution compatible with $\mathcal{P}_{tar}$.

Similarly to Section 4.6.2, we compare 4 redistribution strategies. In the *owner computes* strategy, data items are not moved and the QR factorization is performed in place. In the other strategies, we redistribute data items towards three target distributions compatible with $\mathcal{P}_{tar}$: (i) the distribution $\mathcal{D}_{can} = \mathcal{P}_{tar}$ with the original (arbitrary) labeling of the processors; (ii) the distribution that minimizes the volume of communications $\mathcal{D}_{vol}$; and (iii) the distribution that minimizes the number of redistribution steps $\mathcal{D}_{steps}$.

4.6.3.2 Setup

A highly optimized version of the QR factorization implemented on top of the PaRSEC runtime is available in the DPLASMA library [20]. We have modified this implementation to deal with different data distributions. We use a wide range of matrix sizes, with tiles of size $n_b = 200 \times 200$ double-precision floating point numbers. Our objective is to highlight the impact of the target data distribution on the performance of the QR algorithm, but a tile size of 200×200 is reasonable as it ensures near peak performance on the execution platform.

As already mentioned, real-life distributions are not random. We conduct experiments on 2 different sets of initial distributions for the matrix tiles, one artificially generated and one modeling an Earth Science application [100]:

- $SkewedSet$: 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. The processor with index $i \in [\![0, P-1]\!]$ has a probability $\frac{2i}{P(P-1)}$ to receive each tile. Thus, the workload among processors is likely to be imbalanced. The redistribution strategies toward $\mathcal{D}_{vol}$ and $\mathcal{D}_{steps}$ should 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.

- $ChunkSet$: This distribution set comes from an Earth Science application [100]. 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 4.7 depicts the data distribution of a matrix in $ChunkSet$ where matrix tiles of the same color are initially stored on the same processor.

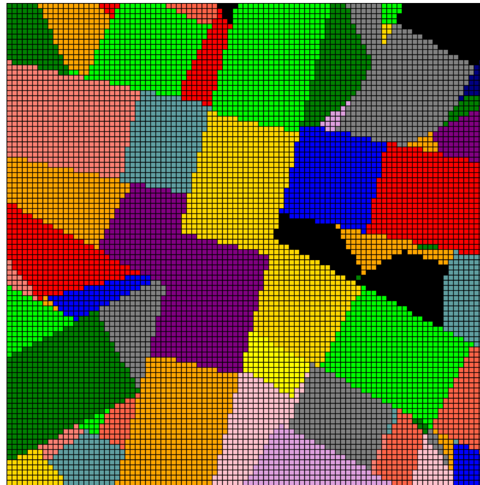


Figure 4.7: The initial distribution of a tiled matrix in $\mathit{ChunkSet}$.

4.6.3.3 Results

Table 4.1 presents the results of the experiments for initial distributions in $\mathit{SkewedSet}$. Each line corresponds to the average results on 50 matrices with $n \times n$ tiles. Columns 1 to 4 give the volume of tiles communicated during the redistribution phase for the four strategies. As expected, redistributing towards the arbitrary distribution $\mathcal{D}_{can}$ requires moving almost every tile while redistributing towards $\mathcal{D}_{vol}$ or $\mathcal{D}_{steps}$ involves almost half as many communications. Columns 5 to 8 present the number of redistribution steps required to schedule the redistribution for the four strategies. We observe that $\mathcal{D}_{vol}$ or $\mathcal{D}_{steps}$ are identical, since Algorithm 7 and Algorithm 8 manage to find the 2D block-cyclic distribution used as reference when building $\mathit{SkewedSet}$. Columns 9 and 10 present the total volume of tiles communicated during the QR factorization. It appears that redistributing towards a 2D block-cyclic partition divides by more than 3 the amount of communications involved in the QR factorization. The gain obtained by redistributing the data according to a suitable partition is significant, and can be seen in the total completion times shown in columns 11 to 14. Columns 15 to 17 present the percentage of improvement provided by the redistribution strategies over the *owner computes* strategy.

Table 4.2 presents the results of the experiments for initial distributions in $\mathit{ChunkSet}$. Each line corresponds to the average results on 50 matrices, as before. The three redistribution strategies perform similarly, with around 90% of the tiles moved during the redistribution phase. Contrary to the previous case, it appears that redistributing towards a 2D block-cyclic partitioning does not lead to a reduction of the volume of communication involved during the QR factorization. Indeed, the *owner computes* strategy requires fewer communications than the other strategies for larger matrices in $\mathit{ChunkSet}$, due to the chunk distribution of the tiles. However, it does not lead to better performance results. Indeed, the three redistribution strategies require more communications to ensure a better load balancing, which leads to a 10-15% improvement on the total completion times compared to the *owner computes* strategy on large matrices.

In summary, we conclude that redistributing towards a suitable data partition for the QR factorization leads to significant improvement, compared to not redistributing the data as with the *owner computes* strategy. Initial distributions in $\mathit{SkewedSet}$ are a good example where redistributing data is essential. Sometimes, like in $\mathit{ChunkSet}$, redistribution strategies involve a bigger amount of communications during the QR factorization but lead to a better load balancing across processors, which is enough to be profitable in the end.

n	Vol. of comm. in the redist. phase				Nb. of steps in the redist. phase				Vol. of comm. in QR fact.		Total completion time			
	owner	can	vol	step	owner	can	vol	step	owner	can-vol-step	owner	can	vol	step
16	0	249	119	119	0	37	32	32	1,973	831	3.34	1.94	2.02	1.89
34	0	1,119	538	538	0	157	149	149	15,118	5,720	25.22	9.06	8.14	8.48
52	0	2,616	1,257	1,257	0	378	353	353	49,707	16,669	78.11	26.75	23.07	22.51
70	0	4,739	2,286	2,286	0	680	638	638	114,127	37,942	182.96	53.44	50.11	52.32
88	0	7,492	3,615	3,615	0	1,092	1,019	1,019	219,757	69,951	344.92	100.87	94.61	95.14

Table 4.1: Results for the different initial distributions in $SkewedSet$.

n	Vol. of comm. in the redist. phase				Nb. of steps in the redist. phase				Vol. of comm. in QR fact.		Completion time			
	owner	can	vol	step	owner	can	vol	step	owner	can-vol-step	owner	can	vol	step
16	0	240	205	233	0	34	31	31	1140	831	2.63	1.92	1.89	1.89
34	0	1,087	1,004	1,072	0	153	142	142	6,380	5,720	11.05	8.77	8.23	8.61
52	0	2,526	2,425	2,518	0	349	338	338	19,295	16,669	30.22	26.12	22.41	22.31
70	0	4,598	4,459	4,575	0	681	660	659	29,606	37,942	61.16	53.41	52.29	58.21
88	0	7,271	7,129	7,242	0	963	951	951	45,311	69,951	114.69	100.88	96.70	99.36

Table 4.2: Results for the different initial distributions in $ChunkSet$.

4.7 Conclusion

In this chapter, we have studied the problem of finding the best data redistribution, given a target data partition. We have used two cost metrics, the total volume of communications and the number of parallel redistribution steps. We have provided algorithms computing the optimal solution for both metrics, and shown through simulations that they achieve significant gain over redistributing to an arbitrary fixed distribution. We have also proved that finding the optimal data partition that minimizes the completion time of the redistribution followed by a 1D-stencil kernel 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 chapter 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 new redistribution strategies presented in this chapter 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.

Future work will be devoted to further investigating the Earth Science application. We have restricted to redistributing data towards the canonical 2D block-cyclic partition, but more experiments are needed to determine the best partition, given the initial distributions that typically arise for this application.

Part III

Auto-adjoint computations

Chapter 5

Optimal multistage algorithms for adjoint computation

When scheduling task graph workflows with large I/O files, we saw in Chapter 3 that the ordering in which the tasks are executed strongly impacts the memory consumption. We designed new memory-aware scheduling techniques for such workflows. But other techniques exist to deal with memory limitations. In some context, when the memory constraints are really tight, it may be necessary to delete some output files during the processing and recompute them later when needed. In this chapter, we target a scientific application, namely the adjoint computation, that provides no parallelism across tasks and where the recomputation technique is necessary to handle the memory usage. The goal is to decide which output files to store in the memory to process the workflows in a minimal time, while respecting the memory constraints. Stumm and Walther provided an optimal algorithm for adjoint computation when the output files can be stored on a single limited memory. Here, we reexamine their work and provide an optimal algorithm for this problem when there are two levels of checkpoints, in memory and on disk. Previously, optimal algorithms for adjoint computations were known only for a single level of checkpoints with no writing and reading costs; a well-known example is the binomial checkpointing algorithm of Griewank and Walther [58]. Stumm and Walther [101] extended that binomial checkpointing algorithm to the case of two levels of checkpoints, but they did not provide any optimality results. We bridge the gap by designing the first optimal algorithm in this context. We experimentally compare our optimal algorithm with that of Stumm and Walther to assess the difference in performance.

5.1 Introduction

The need to efficiently compute the derivatives of a function arises frequently in many areas of scientific computing, including mathematical optimization, uncertainty quantification, and nonlinear systems of equations. When the first derivatives of a scalar-valued function are desired, so-called adjoint computations can compute the gradient at a cost equal to a small constant times the cost of the function itself, without regard to the number of independent variables. This adjoint computation can arise from discretizing the continuous adjoint of a partial differential equation [75, 54] or from applying the so-called reverse or adjoint mode of algorithmic (also called automatic) differentiation to a program for computing the function [59]. In either case, the derivative computation applies the chain rule of differential calculus starting with the dependent variables and propagating back to the independent variables. Thus, it reverses the flow of the original function evaluation. In general, intermediate function values are not available at the time they are needed for partial derivative computation and must be stored or recomputed [51].

A popular storage or recomputation strategy for functions that have some sort of natural “time step” is to save (checkpoint) the state at each time step during the function computation (forward sweep) and use this saved state in the derivative computation (reverse sweep). If the storage is inadequate for all states, one can checkpoint only some states and recompute the unsaved states as needed. Griewank and Walther prove that given a fixed number of checkpoints, the schedule that minimizes the amount of recomputation is a binomial checkpointing strategy [57, 58]. The problem formulation they used implicitly assumes that reading and writing checkpoints are essentially free, but the number of available checkpoints is limited (see Problem 1 below). In [101], Stumm and Walter consider the case where checkpoints can be written to either memory or disk. The number of checkpoints to disk is effectively unlimited but the time to read or write a checkpoint can no longer be ignored (see Problem 2). We consider the same situation. In contrast to Stumm and Walther, however, we do not restrict ourselves to a single binomial schedule but instead prove that there exists a time-optimal schedule possessing certain key properties (including no checkpoints written to disk after the first checkpoint to memory has been written), and we provide a polynomial time algorithm for determining an optimal schedule.

The rest of this chapter is organized as follows. Section 5.2 introduces terminology and the general problem framework. Section 5.3 establishes several properties that must hold true for some optimal schedule and provides an algorithm for identifying this schedule. Section 5.4 compares our checkpointing schedule with that of Stumm and Walther. We conclude with some thoughts on future research directions.

5.2 Framework

5.2.1 The AC problem

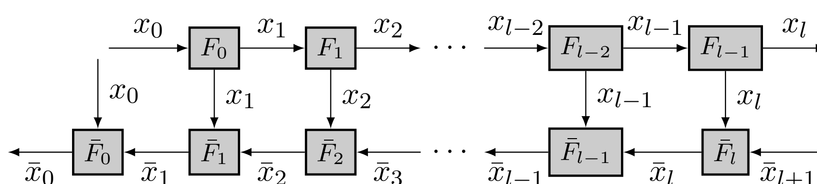


Figure 5.1: The AC dependence graph.

Definition 5.1 (Adjoint Computation (AC) [58, 101])**.** An adjoint computation (AC) with l time steps can be described by the following set of equations:

$$F_i(x_i) = x_{i+1} \text{ for } 1 \leq i < l \tag{5.1}$$

$$\bar{F}_i(x_i, \bar{x}_{i+1}) = \bar{x}_i \text{ for } 1 \leq i \leq l \tag{5.2}$$

The dependencies between these operations[1] are represented by the graph $\mathcal{G} = (V, E)$ depicted in Figure 5.1.

The F computations are called *forward* steps. The $\bar{F}$ computations are called *backward* steps. If $\bar{x}_l$ is initialized appropriately, then at the conclusion of the adjoint computation, $\bar{x}_1$ will contain the gradient with respect to the initial state (x_1).

[1] In the original approach by Griewank [57], an extra F_l operation was included. It is not difficult to take this extra operation into account.

Definition 5.2 (Platform)**.** We consider a platform with three storage locations:

- *Buffers:* there are two buffers, the top buffer and the bottom buffer. The top buffer is used to store a value x_i for some i, while the bottom buffer is used to store a value $\bar{x}_j$ for some j. For a computation (F or $\bar{F}$) to be executed, its input values have to be stored in the buffers. Let $\mathcal{B}^\top$ and $\mathcal{B}^\perp$ denote the content of the top and bottom buffers. In order to start the execution of the graph, x_0 must be stored in the top buffer and $\bar{x}_{l+1}$ in the bottom buffer. Hence, without loss of generality, we assume that at the beginning of the execution, $\mathcal{B}^\top = \{x_0\}$ and $\mathcal{B}^\perp = \{\bar{x}_{l+1}\}$.

- *Memory:* there are c_m slots of memory where the content of a buffer can be stored. The time to write from buffer to memory is w_m. The time to read from memory to buffer is r_m. Let $\mathcal{M}$ be the set of x_i and $\bar{x}_i$ values stored in the memory. The memory is empty at the beginning of the execution ($\mathcal{M} = \emptyset$).

- *Disks:* there are c_d slots of disks where the content of a buffer can be stored.. The time to write from buffer to disk is w_d. The time to read from disk to buffer is r_d. Let $\mathcal{D}$ be the set of x_i and $\bar{x}_i$ values stored in the disk. The disk is empty at the beginning of the execution ($\mathcal{D} = \emptyset$).

Memory and disk are generic terms for a two-level storage system, modeling any platform with a dual memory system, including (i) a cheap-to-access first-level memory, of limited size; and (ii) and a costly-to-access second-level memory, whose size is very large in comparison with the first-level memory. The pair *(memory, disk)* can be replaced by *(cache, memory)* or *(disk, tape)* or any relevant hardware combination.

Intuitively, the core of the AC problem is the following. After the execution of a forward step, its output is kept in the top buffer only. If it is not saved in memory or disk before the next forward step, it is lost and will have to be recomputed when needed for the corresponding backward step. When no disk storage is available, the problem is to minimize the number of recomputations in the presence of limited (but cheap-to-access) memory slots. When disk storage is added, the problem becomes even more challenging: saving data on disk can save some recompilation, and a trade-off must be found between the cost of disk accesses and that of recomputations.

The problem with only memory and no disk (Problem 1: PROB(l, c_m) below) has been solved by Griewank and Walther [58], using a binomial checkpointing algorithm called REVOLVE. In accordance to the scheduling literature, we use the term *makespan* for the total execution time.

Problem 1 (PROB(l, c_m))**.** *We want to minimize the makespan of the AC problem with the following parameters:*

		Initial state:
AC graph:	*size l*	
Steps:	u_f, u_b	
Memory:	c_m, $w_m = r_m = 0$,	$\mathcal{M}_{ini} = \emptyset$
Disks:	$c_d = 0$	
Buffers:	$\mathcal{B}^\top$, $\mathcal{B}^\perp$	$\mathcal{B}^\top_{ini} = \{x_0\}$, $\mathcal{B}^\perp_{ini} = \{\bar{x}_{l+1}\}$

In this chapter we consider the problem with limited memory and infinite disk (Problem PROB$_\infty(l, c_m, w_d, r_d)$ below). The main goal of this chapter is to provide the first optimal algorithm for PROB$_\infty(l, c_m, w_d, r_d)$.

Problem 2 (PROB$_\infty(l, c_m, w_d, r_d)$)**.** *We want to minimize the makespan of the AC problem with the following parameters:*

		Initial state:
AC graph:	*size* l	
Steps:	u_f, u_b	
Memory:	c_m, $w_m = r_m = 0$,	$\mathcal{M}_{ini} = \emptyset$
Disks:	$c_d = +\infty, w_d, r_d$,	$\mathcal{D}_{ini} = \emptyset$
Buffers:	$\mathcal{B}^\top$, $\mathcal{B}^\perp$	$\mathcal{B}^\top_{ini} = \{x_0\}$, $\mathcal{B}^\perp_{ini} = \{\bar{x}_{l+1}\}$

5.2.2 Algorithm model

We next detail the elementary operations that an algorithm can perform.

F_i Execute one forward computation F_i (for $i \in \{0, \dots, l-1\}$). Note that by definition, for F_i to occur, x_i should be in the top buffer before (i.e., $\mathcal{B}^\top = \{x_i\}$) and x_{i+1} will be in the top buffer after (i.e., $\mathcal{B}^\top \leftarrow \{x_{i+1}\}$). This operation takes a time $\mathsf{time}(F_i) = u_f$.

$\bar{F}_i$ Execute the backward computation $\bar{F}_i$ ($i \in \{0, \dots, l\}$). Note that by definition, for $\bar{F}_i$ to occur, x_i should be in the top buffer and $\bar{x}_{i+1}$ in the bottom buffer (i.e., $\mathcal{B}^\top = \{x_i\}$ and $\mathcal{B}^\perp = \{\bar{x}_{i+1}\}$) and $\bar{x}_i$ will be in the bottom buffer after (i.e., $\mathcal{B}^\perp \leftarrow \{\bar{x}_i\}$). This operation takes a time $\mathsf{time}(\bar{F}_i) = u_b$.

W_i^m Write the value x_i of the top buffer into the memory. Note that by definition, for W_i^m to occur, x_i should be in the top buffer (i.e., $\mathcal{B}^\top = \{x_i\}$) and there should be enough space for x_i in the memory (i.e., $|\mathcal{M}| < c_m$); x_i will be in the memory after (i.e., $\mathcal{M} \leftarrow \mathcal{M} \cup \{x_i\}$). This operation takes time $\mathsf{time}(W_i^m) = w_m$.

D_i^m Discard the value x_i of the memory (i.e., $\mathcal{M} \leftarrow \mathcal{M} \backslash \{x_i\}$). This operation takes a time $\mathsf{time}(D_i^m) = 0$. This operation is introduced only to clarify the proofs, since a D_i^m operation is always immediately followed by a W_i^m operation. In other words, all write operations overwrite the content of some memory slot, and we simply decompose an overwrite operation into a discard operation followed by a write operation.

R_i^m Read the value x_i in the memory, and put it into the top buffer. Note that by definition, for R_i^m to occur, x_i should be in the memory (i.e., $x_i \in \mathcal{M}$) and x_i will be in the top buffer after (i.e., $\mathcal{B}^\top = \{x_i\}$). This operation takes a time $\mathsf{time}(R_i^m) = r_m$.

W_i^d Write the value x_i of the top buffer into the disk. Note that by definition, for W_i^d to occur, x_i should be in the top buffer (i.e., $\mathcal{B}^\top = \{x_i\}$) and x_i will be in the disk after (i.e., $\mathcal{D} \leftarrow \mathcal{D} \cup \{x_i\}$). This operation takes a time $\mathsf{time}(W_i^d) = w_d$.

R_i^d Read the value x_i in the disk and puts it into the top buffer. Note that by definition, for R_i^d to occur, then x_i should be in the disk (i.e., $x_i \in \mathcal{D}$) and x_i will be in the top buffer after (i.e., $\mathcal{B}^\top = \{x_i\}$). This operation takes a time $\mathsf{time}(R_i^d) = r_d$.

D_i^d Discard the value x_i of the disk (i.e., $\mathcal{D} \leftarrow \mathcal{D} \backslash \{x_i\}$). This operation takes a time $\mathsf{time}(D_i^d) = 0$. The same comment as for D_i^m operations holds: all disk writes, just as memory writes, are overwrite operations, which we decompose as indicated above. Both discard operations are introduced for the clarity of the proofs.

For conciseness, we let $F_{i \to i'}$ denote the sequence $F_i \cdot F_{i+1} \cdot \ldots \cdot F_{i'}$.

Because of the shape of the AC dependence graph (see Figure 5.1), any algorithm solving Problem 2 will have the following unique structure,

$$\mathfrak{S}_l \cdot \bar{F}_l \cdot \mathfrak{S}_{l-1} \cdot \bar{F}_{l-1} \cdot \ldots \cdot \mathfrak{S}_0 \cdot \bar{F}_0, \tag{5.3}$$

where for all i, $\mathfrak{S}_i$ is a sequence of operations that does not contain $\bar{F}_i$ (hence the $\bar{F}_i$ following $\mathfrak{S}_i$ is the first occurrence of $\bar{F}_i$).

Definition 5.3 (iteration i)**.** Given an algorithm that solves the AC problem, we let *iteration* i (for $i = l \dots 0$) be the sequence of operations $\mathfrak{S}_i \cdot \bar{F}_i$. Let l_i be the execution time of iteration i.

With this definition, the makespan of an algorithm is $\sum_{i=1}^{l} l_i$.

5.3 Solution of PROB$_\infty(l, c_m, w_d, r_d)$

In this section we show that we can compute in polynomial time the solution to PROB$_\infty(l, c_m, w_d, r_d)$. To do so, we start by showing some properties on an optimal algorithm.

- We show that iteration l starts by writing some values x_i on disk checkpoints before doing any memory checkpoints (Lemma 5.6).
- Once this is done, we show that we can partition the initial AC graph into connected subgraphs by considering the different subgraphs between consecutive disk checkpoints (Proposition 5.1). Each of these subgraphs can be looked at (and solved) independently.
- We give some details on how to solve the problem on all subgraphs. In particular we show that (i) we do not write any additional values to disks in order to solve them (Lemma 5.2); and (ii) we show that similarly to the first iteration, the algorithm writes some values to memory checkpoints and we can partition these subgraphs by considering the different subgraphs between memory checkpoints (Lemma 5.3).
- To solve these subgraphs, we introduce new problems (Problems 3 and 4) that inherit the properties of the general problem.
- We show how to compute the size of the different connected subgraphs through a dynamic programming algorithm (§ 5.3.2).

5.3.1 Properties of an optimal algorithm

We show here some dominance properties. That is, there exist optimal algorithms that obey these properties, even though not all optimal algorithms do.

Lemma 5.1. *There exists an optimal solution that has the following structure,* $\mathfrak{S}_l \cdot \bar{F}_l \cdot \mathfrak{S}_{l-1} \cdot \bar{F}_{l-1} \cdot \ldots \cdot \mathfrak{S}_0 \cdot \bar{F}_0$*, and that satisfies*
(P0):

(i) There are no D^d-type operations (we do not discard from disks).

(ii) Each D^m-type operation is immediately followed by a W^m-type operation (we discard a value from memory only to overwrite it).

(iii) Each R-type operation is not immediately followed by another R-type operation.

(iv) Each W^m-type operation (resp. W^d-type operation) is not immediately followed by another W^m-type operation (resp. W^d-type operation).

(v) Each W^m-type operation is not immediately followed by another D^m-type operation.

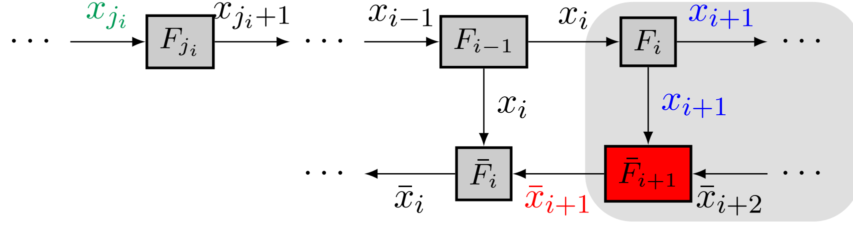


Figure 5.2: After executing $\bar{F}_{i+1}$, x_{i+1} (blue) is in the top buffer, and $\bar{x}_{i+1}$ (red) is in the bottom buffer. For the remainder of the execution, the algorithm will not need the grey area anymore; hence it will need to fetch x_{j_i} (green) from a checkpoint slot.

(vi) There are no $\bar{F}$-type operations in any $\mathfrak{S}_i$ sequence (backward steps are not recomputed)*;*

(vii) During $\mathfrak{S}_i$ ($i < l$), there are no F_i to F_{l-1} operations (nor actions involving x_{i+1} to x_l);

(viii) In particular, for all $i < l$, the first operation of sequence $\mathfrak{S}_i$ is a R-type operation; in other words, there exist j_i and $s \in \{m, d\}$ such that $\mathfrak{S}_i = R^s_{j_i}\tilde{\mathfrak{S}}_i$;

(ix) $\forall i$, there is at least one R^m_i operation between a W^m_i and a D^m_i operations.

(x) If $l > 0$, the first operation of the algorithm is a W-type operation.

Proof. Some of the intuitions of this lemma can be grasped from Figure 5.2. Note that removing an operation from the optimal solution cannot increase the makespan.

(i) We have an infinite number of disk slots available: there is no need to discard any value from it.

(ii) Discard a value from the memory is useless if the memory is not full and if we do not need to write a new value in it.

(iii) If we had two consecutive reads in the sequence, then the only action of the first read would be to put some value x_i in the top buffer. However, the second read would immediately overwrite this value, making the first read unnecessary. Thus, the first read can be removed.

(iv) It is useless to write the same value in the same storage twice in a row.

(v) Similar to the previous point, from (ii) a D^m operation is immediately followed by a W^m operation; hence this would be writing twice in the same storage in a row.

(vi) The reason there are no $\bar{F}$-type operations in all $\mathfrak{S}_i$ is that we have a dedicated buffer for the $\bar{x}_i$ values. The only operations that use the $\bar{x}_i$ values are $\bar{F}$-type operations. Also, to execute $\bar{F}_i$, we need only the value of $\bar{x}_{i+1}$ that is already stored in the bottom buffer at the beginning of $\mathfrak{S}_i$. Hence, removing the additional $\bar{F}$-type operations from $\mathfrak{S}_i$ can only improve the execution time.

(vii) Operations involving F_i to F_{l-1} (or their output values) during $\mathfrak{S}_i$ would be useless; see Figure 5.2.

(viii) After the execution of $\bar{F}_i$, the content of the top buffer is x_i. The value x_i is useless for $\bar{F}_{i-1}$ (see the previous point). Hence, at the begining of $\mathfrak{S}_{i-1}$, we need to read the content of a storage slot before executing any F-type, $\bar{F}$-type, or W-type operation. Furthermore, because of property (ii), doing a D-type operation will not permit an R-type operation before the next W-type operation. Hence, the first operation of sequence $\mathfrak{S}_i$ is necessarily an R-type operation.

(ix) Assume that there exists i such that there are no R_i^m operations between a W_i^m and a D_i^m operations. It is useless to write the value x_i in the memory and discard it without reading it in between. Thus the W_i^m and D_i^m operations can be removed at no additional time delay.

(x) The first operation of the solution cannot be an R-type or a D-type operation since at the beginning of the execution the memory and the disk are empty. If $l > 0$, the forward step F_0 has to be executed before the backward step $\bar{F}_l$. Thus the first operation cannot be an $\bar{F}$-type operation. Now assume that the first operation is an F-type operation. It then has to be F_0. After the execution of F_0, the value x_1 is in the top buffer, and the value x_0 is not stored anywhere. There is no way to recompute the value x_0, thus to execute $\bar{F}_0$, which would then prevent computing $\bar{x}_0$ (absurd). Thus, at the beginning of the execution, we have to store the value x_0 either in the memory or in the disk, and the first operation of the algorithm is a W-type operation.

■

Lemma 5.2. *There exists an optimal solution to Problem 2 that satisfies (P0) and*
(P1):

(i) All disk checkpoints are executed during the first iteration $\mathfrak{S}_l$.

(ii) For all i, $1 \le i \le l-1$, W_i^d operations are executed before W_i^m operations.

Proof. Suppose $\mathcal{S}$ is an optimal solution that satisfies (P0). We will show that we can transform it into a solution that satisfies (P0) and (P1):

(i) Iteration l passes through all forward computations. If a value x_i is saved on disk later on during the algorithm, we could as well save it after the first execution of F_{i-1} (in $\mathfrak{S}_l$) with no additional time delay, since we have an infinite number of slots on disk.

(ii) Let assume that there exists i, $0 \le i \le l-1$, such that W_i^m is executed before W_i^d in $\mathcal{S}$. By definition, when W_i^m is executed, the value x_i is stored in the top buffer. Thus we can execute W_i^d right before W_i^m, instead of later, at no additional time delay, since the amount of available disk slots is infinite. This new solution still satisfies (P0).

■

The following lemma and its proof are inspired by Lemma 3.1 by Walther [107].

Lemma 5.3 (Memory Checkpoint Persistence)**.** *There exists an optimal solution to Problem 2 that satisfies (P0), (P1), and*
(P2):

(i) Let $i < l$; if W_i^m is executed, then there are no D_i^m operations until after the execution of $\bar{F}_i$ (that is, until the beginning of iteration $i-1$).

(ii) Moreover, until that time, no operation involving F_0 to F_{i-1} or values x_0 to x_{i-1} is taken.

The intuition behind this result is that if we were to discard the value x_i before executing $\bar{F}_{i+1}$, then a better solution would have stored x_{i+1} in the first place. Furthermore, because $r_m = 0$, we can show that until $\bar{F}_i$, all actions involving computations F_0 to F_{i-1} or values x_0 to x_{i-1} do not impact the actions that lead to an $\bar{F}_j$ operation, $j \ge i$, and thus can be moved to a later time at no additional time delay (and potentially reducing the makespan of the algorithm).

Proof. Let $\mathcal{S}$ be an optimal solution that satisfies (P0) and (P1) but not (P2). We can transform it into a solution that satisfies (P0), (P1), and (P2). We iteratively transform $\mathcal{S}$ to increase the number of i values respecting (P2), without increasing the makespan of the schedule. This transformation can be applied as many times as necessary to reach a schedule that satisfies (P2).

Assume, first, that $\mathcal{S}$ does not satisfy (P2(i)). Let x_i be the first value such that at some point during $\mathcal{S}$, x_i is stored in the memory and discarded before executing $\bar{F}_i$. Thus we can write

$$\mathcal{S} = \mathcal{S}_0 \cdot W_i^m \cdot \mathcal{S}_1 \cdot D_i^m \cdot \mathcal{S}_2 \cdot \bar{F}_i \cdot \mathcal{S}_3,$$

where $\mathcal{S}_0$, $\mathcal{S}_1$, $\mathcal{S}_2$, and $\mathcal{S}_3$ are sequences of operations that do not include $\bar{F}_i$. Since $\mathcal{S}$ satisfies (P0(ix)), there is at least one R_i^m operation in $\mathcal{S}_1$. Let us prove that all these R_i^m operations are immediately followed by an F_i operation.

- The R_i^m operations cannot be immediately followed by an $\bar{F}$-type operation, because the only $\bar{F}$-type operation allowed after an R_i^m operation is $\bar{F}_i$ (since the value x_i would be in the top buffer) and there are no $\bar{F}_i$ in $\mathcal{S}_1$.
- According to (P0(ix)), the R_i^m operations cannot be immediately followed by another R-type operation.
- The R_i^m operations cannot be immediately followed by a D^m-type operation because, according to (P0(ii)), the next operation would be a W^m-type operation. However, since the value x_i would be in the top buffer, the only W^m-type operation allowed would be W_i^m, which is useless since the value x_i is already in the memory.
- The R_i^m operations cannot be immediately followed by a D^d-type operation because, according to (P0(i)), there are no D^d-type operations in $\mathcal{S}$.
- The R_i^m operations cannot be immediately followed by a W^m-type operation, because the only W^m-type operation allowed after an R_i^m operation is W_i^m (since the value x_i would be in the top buffer), which is useless since the value x_i is already in the memory.
- The R_i^m operations cannot be immediately followed by a W^d-type operation, because the only W^d-type operation allowed after an R_i^m operation is W_i^d (since the value x_i would be in the top buffer), which is impossible according to (P1(ii)) since a W_i^m has already been executed before $\mathcal{S}_1$.

So, all these R_i^m operations are immediately followed by an F-type operation; since the value x_i is in the top buffer, this operation is F_i. Thus, any R_i^m in $\mathcal{S}_1$ is followed by F_i.

Let us now focus on the first operation in $\mathcal{S}_1$. It cannot be a $\bar{F}$-type (the only possible $\bar{F}$-type is $\bar{F}_i$), W^m-type ((P0(iv))), W^d-type ((P1(ii))), or D-type ((P0(v))). Hence, the first operation in $\mathcal{S}_1$ is either a R-type operation or a F-type operation (in which case, it is F_i since $\mathcal{B}^\top = \{x_i\}$ at the beginning of $\mathcal{S}_1$).

- Assume that $\mathcal{S}_1 = F_i \cdot \mathcal{S}_1'$. Let $\mathcal{S}_1''$ be the sequence $\mathcal{S}_1'$ where every occurrence of $R_i^m \cdot F_i$ has been replaced by R_{i+1}^m. We know that the schedule $\mathcal{S}' = \mathcal{S}_0 \cdot F_i \cdot R_{i+1}^m \cdot \mathcal{S}_1'' \cdot D_{i+1}^m \cdot \mathcal{S}_2 \cdot \bar{F}_i \cdot \mathcal{S}_3$ is correct and has a makespan at least as good as $\mathcal{S}$ (since there is at least one occurrence of $R_i^m \cdot F_i$ in $\mathcal{S}_1'$).
- Assume that there exist j and s (either equal to m or d) such that $\mathcal{S}_1 = R_j^s \cdot \mathcal{S}_1'$. Let $\mathcal{S}_1''$ be the sequence $\mathcal{S}_1'$ where every occurrence of $R_i^m \cdot F_i$ has been replaced by R_{i+1}^m. We know that the schedule $\mathcal{S}' = \mathcal{S}_0 \cdot F_i \cdot R_{i+1}^m \cdot R_j^s \cdot \mathcal{S}_1'' \cdot D_{i+1}^m \cdot \mathcal{S}_2 \cdot \bar{F}_i \cdot \mathcal{S}_3$ is correct and has a makespan at least as good as $\mathcal{S}$ (since there is at least one occurrence of $R_i^m \cdot F_i$ in $\mathcal{S}_1'$).

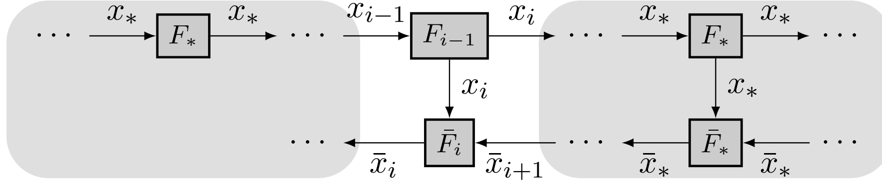


Figure 5.3: Consider a subsequence of $\mathcal{S}_1$ comprised between two consecutive R-type operations: R_j^s and $R_{j'}^{s'}$. If $j \geq i$, then by definition the subsequence will activate only parts of the right area (and not overwrite any checkpoint from the left area). If $j < i$, then we have shown that the subsequence will activate only parts of the left area (no forward sweep through F_{i-1}).

Hence we were able to transform $\mathcal{S}$ into $\mathcal{S}$' without increasing the makespan, so that the number of values i, $0 \leq i < l$, that does not respect (P2(i)) decreases. We repeat this transformation until the new schedule satisfies (P2(i)).

Let us now consider (P2(ii)). Let $\mathcal{S}$ be an optimal solution that satisfies (P0), (P1), and (P2(i)). Let i, $0 \leq i < l$, such that there exists W_i^m in $\mathcal{S}$. We can write

$$\mathcal{S} = \mathcal{S}_0 \cdot W_i^m \cdot \mathcal{S}_1 \cdot \bar{F}_i \cdot \mathcal{S}_2 \cdot D_i^m \cdot \mathcal{S}_3,$$

where $\mathcal{S}_1$ is a sequence of operations that do not include D_i^m. There are no F_{i-1} in $\mathcal{S}_1$ because their only impact on the memory is to put the value x_i in the top buffer which could be done with R_i^m for no time delay.

Consider now two consecutive R-type operations of $\mathcal{S}_1$. Because there are no F_{i-1} operations in $\mathcal{S}_1$, we know that between these two R-type operations, and with the definitions of $\mathcal{A}$ and $\mathcal{B}$ in Figure 5.3, either only elements of $\mathcal{A}$ are activated (and no element of $\mathcal{B}$) or only elements of $\mathcal{B}$ are activated (and no element of $\mathcal{A}$).

Consider now the last R-type operation R_j^s of $\mathcal{S}_1$ such that $j < i$ (s being equal to m or d). All W^m-type operations written after this operation and before the next R-type operation of $\mathcal{S}_1$ involve some values in $\mathcal{A}$. Hence by (P2(i)), they are not discarded until after $\bar{F}_i$. Furthermore, because R_j^s is the last such operation, we know that they are not used in $\mathcal{S}_1$ either. Hence we can move this sequence of operations (the sequence between R_j^s and the next R-type operation) right after $\bar{F}_i$ at no additional time delay. This operation can be repeated until there are no more such operations in $\mathcal{S}_1$. We then proceed with these operations recursively in the appearance order of the W_i^m. This shows that we can construct an optimal schedule that satisfies (P2(ii)). ■

Lemma 5.4. *There exists an optimal algorithm for Problem 2 that satisfies (P0), (P1), (P2), and* (P3): *There is only one R-type operation (the first one) in every iteration $\mathfrak{S}_i$, where $i < l$.*

Proof. Let $\mathcal{S}$ be an optimal schedule that satisfies (P0), (P1), and (P2). We show that we can transform it into an optimal schedule that satisfies (P0–3).

To show this result, for any $i \geq 0$, we inductively show that if we have a solution such that the property is true in iterations l to $i+1$, then we can transform it into a solution such that this property in true in iterations l to i.

Assume that $\mathcal{S}$ does not satisfies (P3). Let $\mathfrak{S}_i$ be the first iteration of $\mathcal{S}$ that includes more than one R-type operation.

If $i = 0$, then all R-type operations are $R_{\bar{0}}$ by (P2). Hence we can remove any of them until there is only one.

$$
\begin{array}{rl}
\tilde{\mathfrak{S}}^{(2)} = & R_0^s F_1 W_1^m F_2 F_3 F_4 F_5 W_5^m F_6 W_6^m \\
\tilde{\mathfrak{S}}^{(3)} = & \qquad\qquad R_3^{s'} F_4 F_5 \qquad F_6 W_6^m F_7 F_8 F_9 W_9^m \\
\hline
\tilde{\mathfrak{S}}^{(4)} = & R_0^s F_1 W_1^m F_2 F_3 F_4 F_5 W_5^m F_6 W_6^m F_7 F_8 F_9 W_9^m
\end{array}
$$

Figure 5.4: Example of the merging operation.

Otherwise assume that $i \geq 1$. Let $R_{j_1}^{s_1}$ and $R_{j_2}^{s_2}$ (where $s_1, s_2 \in \{m, d\}$) be the last two R-type operations in $\mathfrak{S}_i$. According to (P0(iv)), we know that step i does not involve x_{i+1} to x_l, so $j_1 \leq i$ and $j_2 \leq i$. We can write

$$\mathfrak{S}_i = \mathfrak{S}^{(1)} \cdot R_{j_1}^{s_1} \cdot \mathfrak{S}^{(2)} \cdot R_{j_2}^{s_2} \cdot \mathfrak{S}^{(3)},$$

where $\mathfrak{S}^{(2)}$ and $\mathfrak{S}^{(3)}$ are sequences of operations that do not include any R-type operation. According to (P0(vi)), there are no $\bar{F}$-type operations in $\mathfrak{S}^{(2)}$ and $\mathfrak{S}^{(3)}$ either. Since the value x_{j_2} is in the top buffer at the beginning of $\mathfrak{S}^{(3)}$, we know that the first F-type operation of $\mathfrak{S}^{(3)}$ has to be F_{j_2}. We know that the first operation after $\mathfrak{S}_i$ is $\bar{F}_i$. Thus the last F-type operation of $\mathfrak{S}^{(3)}$ is F_{i-1}. Since there are no R-type operations in $\mathfrak{S}^{(3)}$, we know that the sequence $\mathfrak{S}^{(3)}$ includes all F-type operations from F_{j_2} to F_{i-1}.

Similarly, the sequence $\mathfrak{S}^{(2)}$ includes all F-type operations from F_{j_1} to $F_{j_{\max}}$ operations with $j_{\max} \leq i-1$.

- If $j_{\max} \geq j_2$, we note $j_{\min} = \min(j_1, j_2)$ and $s_{\min}$ the corresponding value of s_1 or s_2. Iteration $\mathfrak{S}_i$ includes each F-type operations from $F_{j_{\min}}$ to F_i (possibly twice). Let us build the sequence of operations $\mathfrak{S}^{(4)}$ from the sequence $F_{j_{\min} \to i}$ where each operation F_k if immediately followed by W_k^m if W_k^m is present in either $\mathfrak{S}^{(2)}$ or $\mathfrak{S}^{(3)}$ (see Figure 5.4 for this transformation). Thus we know that the sequence $\mathfrak{S}_i' = \mathfrak{S}^{(1)} \cdot R_{j_{\min}}^{s_{\min}} \cdot \mathfrak{S}^{(4)}$ will have the exact same impact on the memory as $\mathfrak{S}_i$ without increasing the makespan. Transforming iteration $\mathfrak{S}_i$ into sequence $\mathfrak{S}_i'$ reduces by one the number of readings in iteration i.

- If $j_{\max} < j_2$, sequences of operations $\mathfrak{S}^{(2)}$ and $\mathfrak{S}^{(3)}$ are disjoint. Thus $\mathfrak{S}^{(2)}$ has no impact on iteration i and can be moved to the beginning of the next operation. Thus transforming iteration $\mathfrak{S}_i$ into $\mathfrak{S}_i' = \mathfrak{S}^{(1)} \cdot R_{j_2}^{s_2} \cdot \mathfrak{S}^{(3)}$ and moving $R_{j_1}^{s_1} \cdot \mathfrak{S}^{(2)}$ to the beginning of iteration $\mathfrak{S}_{i+1}$ will not increase the makespan of $\mathcal{S}$ and will reduce by one the number of readings in iteration i.

Hence we have shown that until there is only one R-type operation in iteration i, we can reduce by one the number of R-type operations in iteration i (and leave as they were iterations l to $i+1$). Thus, if the property is true in iteration l to $i+1$, then we can transform it into a solution such that this property in true in iteration l to i. This concludes the proof. ■

Corollary 5.1 (Description of iteration $i < l$)**.** *Given an optimal algorithm for Problem 2 that satisfies (P0–3), each iteration $i < l$ can be written as*

$$\mathfrak{S}_i = R_{j_i}^{s_i} \tilde{\mathfrak{S}}_i \tag{5.4}$$

for some j_i and for some $s_i \in \{m, d\}$, where $\tilde{\mathfrak{S}}_i$ is composed only of F-type, W^m-type, and D^m-type operations (possibly empty). Furthermore, it goes through all operations F_j for j from j_i to $i-1$.

Lemma 5.5 (Description of iteration l)**.** *There exists an optimal algorithm for Problem 2 that satisfies (P0–3) and*

(P4): *There are no R-type operations in iteration $\mathfrak{S}_l$. Hence, every F-type operation is executed once and only once in iteration $\mathfrak{S}_l$.*

Proof. Let $\mathcal{S}$ be an optimal schedule that satisfies (P0–3). We can write

$$\mathcal{S} = \mathfrak{S}_l \cdot \bar{F}_l \cdot \mathfrak{S}_{l-1} \cdot \bar{F}_{l-1} \cdot \ldots \cdot \mathfrak{S}_0 \cdot \bar{F}_0.$$

We know that at the end of the execution of $\mathfrak{S}_l$, the top buffer contains the value x_l and the bottom buffer contains the value $\bar{x}_{l+1}$. Let $\mathcal{M}_l$ and $\mathcal{D}_l$ be the state of the memory and the disk at the end of the execution of $\mathfrak{S}_l$.

Let $\mathfrak{S}'_l$ be the sequence of operations $F_{0\to l-1}$ where every operation F_i is immediately followed by (i)$W_i^d W_i^m$ if $x_i \in \mathcal{M}_l \cap \mathcal{D}_l$; (ii) else, W_i^m if $x_i \in \mathcal{M}_l$; (iii) else W_i^d if $x_i \in \mathcal{D}_l$. At the end of the execution of $\mathfrak{S}'_l$, the memory and the disk will be in the states $\mathcal{M}_l$ and $\mathcal{D}_l$. Furthermore, if $x_i \in \mathcal{M}_l$ (resp. if $x_i \in \mathcal{D}_l$), the sequence $\mathfrak{S}_l$ contains the operation W_i^m (resp. W_i^d). Thus all W-type operations of $\mathfrak{S}'_l$ are in $\mathfrak{S}_l$. Moreover, $\mathfrak{S}_l$ has to contain all the forward steps from F_0 to F_{l-1}. Thus, every operation in $\mathfrak{S}'_l$ is included in $\mathfrak{S}_l$. Hence, the sequence $\mathcal{S}' = \mathfrak{S}'_l \cdot \bar{F}_l \cdot \mathfrak{S}_{l-1} \cdot \bar{F}_{l-1} \cdot \ldots \cdot \mathfrak{S}_0 \cdot \bar{F}_0$ is valid and has a makespan not larger than $\mathcal{S}$. $\mathcal{S}'$ is then optimal and satisfies (P0–4). ∎

Lemma 5.6. *There exists an optimal algorithm for Problem 2 that satisfies (P0–4) and*
(P5): *Given i, j, $0 \le i, j \le l-1$, all W_i^d operations are executed before any W_j^m operation.*

Hence, during the first iteration $\mathfrak{S}_l$, we first assign the disk checkpoints before assigning the memory checkpoints. The idea is that the farther away in the graph a checkpoint is set, the more times it is going to be read during the execution.

Proof. Let $\mathcal{S}$ be an optimal algorithm that satisfies (P0–4), but not (P5). We show that we can transform it into an optimal algorithm that also satisfies (P5) without increasing the makespan.

By contradiction, assume that there exist i and j such that W_i^m is executed before W_j^d. According to (P1(i)), every write on the disk occurs during the first iteration $\mathfrak{S}_l$. Thus W_i^m also occurs in iteration $\mathfrak{S}_l$. According to (P4), the F-type operations are not re-executed in iteration $\mathfrak{S}_l$. Thus, necessarily, $i < j$.

According to (P0(ix)), since the algorithm wrote the value x_i in the memory and the value x_j in the disk, they are read later in the schedule. Let $\mathfrak{S}_{i_m}$ be a step when R_i^m occurs and $\mathfrak{S}_{i_d}$ a step when R_j^d occurs. Then we have: $i \le i_m$ by (P0(iv)) and $j \le i_d$ by (P0(iv)). Finally, there is only one R-type operation per step (Corollary 5.1); thus $i_m \neq i_d$.

Assume first that $i_m > i_d$. From Corollary 5.1, the first operation of $\mathfrak{S}_{i_m}$ is R_i^m. Thus the value x_i is in the top buffer at the beginning of $\mathfrak{S}_{i_m}$. Furthermore, by definition of the steps, the value x_{i_m} has to be in the top buffer at the end of $\mathfrak{S}_{i_m}$. Since there are no other R-type operations in $\mathfrak{S}_{i_m}$, all forward steps from F_i to F_{i_m} are executed in $\mathfrak{S}_{i_m}$. In particular F_{j-1} is executed in $\mathfrak{S}_{i_m}$. Let n be the number of consecutive F-type operations right before F_{j-1}. Thus $\mathfrak{S}_{i_m}$ has the shape: $\mathfrak{S}_{i_m} = R_i^m \cdot \mathfrak{S}^1 \cdot F_{(j-n)\to(j-1)} \cdot \mathfrak{S}^2$ where the last operation of $\mathfrak{S}^1$ is not a F-type operation. Recall that the time for a disk read is r_d and for a forward step is u_f.

- Assume that $(n+1)u_f > r_d$. Then the sequence $R_i^m \cdot \mathfrak{S}^1 \cdot R_j^d \cdot \mathfrak{S}^2$ has a smaller execution time than $\mathfrak{S}_{i_m}$ contradicting the optimality.
- Assume that $(n+1)u_f \le r_d$. According to Corollary 5.1, if $\mathfrak{S}^1$ is not empty, then its last operation is W_{j-n-1}^m (if it is empty, then $i = j-n-1$). (P2) ensures that x_{j-n-1} will not be discarded until after x_j has become useless (after $\bar{F}_j$). Since $(n+1)u_f \le r_d$, we could replace all future instances of R_j^d by $R_{j-n-1}^m \cdot F_{(j-n)\to j}$. Hence W_j^d would be useless, which would contradict the optimality of the schedule.

Hence, $i_m < i_d$. In particular, this is true for any i_m and i_d. Then we can show with similar arguments that this is true until $\bar{F}_i$. Hence, there are not any R_j^d until after $\bar{F}_i$. Since $j > i$, this means that there will not be anymore R_j^d operations at all. Finally, this shows that the execution of W_j^d is useless and can be removed. ■

Proposition 5.1 (Disk Checkpoint Persistence)**.** *Given an optimal algorithm for Problem 2 that satisfies (P0–5), then after any operation W_i^d, there are no F_j operations for $0 \leq j \leq i-1$ (nor actions involving the values x_j for $0 \leq j \leq i-1$) until after the execution of $\bar{F}_i$.*

Proof. Let $\mathcal{S}$ be an optimal algorithm that satisfies (P0–5). Assume by contradiction that $\mathcal{S}$ includes W_i^d and there exists $j \geq i$ and $i' < i$ such that iteration $\mathfrak{S}_j$ involves $F_{i'}$. In particular, according to Corollary 5.1, it involves F_{i-1}. According to Corollary 5.1, there exist k, $s \in \{m, d\}$, and n maximum such that

$$\mathfrak{S}_j = R_k^s \mathfrak{S}^1 \cdot F_{(i-1-n)\to(i-1)} \cdot \mathfrak{S}^2.$$

We can first show that $r_d > n$. Indeed, otherwise, $\mathfrak{S}^1 \cdot R_{i'}^d \cdot \mathfrak{S}^2$ has a smaller execution time than does $\mathfrak{S}_j$ for the same result, contradicting the optimality. Let us now show that we can remove all appearances of R_i^d in the schedule, hence decreasing the execution time, which would contradict the optimality of the algorithm.

- Consider the occurrence of R_i^d after $\mathfrak{S}_j$. By maximality of n, if $\mathfrak{S}^1$ is not empty, then the last operation of $\mathfrak{S}^1$ is W_{i-1-n}^m (Corollary 5.1). If $\mathfrak{S}^1$ is empty, then $k = i-1-n$ and $s = m$ (otherwise we could replace $R_k^d F_{(i-1-n)\to(i-1)}$ by R_i^d which would contradict the optimality of the algorithm). Thus, in both cases the value x_{i-1-n} is stored in the memory during $\mathfrak{S}_j$, and (P2) ensures that it will not be discarded until after x_{i-1-n} has become useless (after $\bar{F}_i$). Because $r_d > n$, we can replace all later appearances of R_i^d by $R_{i-1-n}^m F_{(i-1-n)\to(i-1)}$ at no additional time delay.
- Let us now consider the eventual occurrences of R_i^d anterior to iteration j. Necessarily, all R_i^d anterior to $\mathfrak{S}_j$ are followed by F_i (otherwise one of them is followed by W_i^m and it is not permitted by the memory checkpoint persistence property (P2)). Hence, we can store the value x_{i+1} in the disk instead of the value x_i during $\mathfrak{S}_l$ ($\mathfrak{S}_l$ goes through all forward operation according to (P4)). Then, it is possible to replace all $R_i^d F_i$ operations by R_{i+1}^d reducing the execution time.

Hence, we can decrease the execution time by removing all appearances of R_i^d in the schedule, which shows the contradiction. ■

5.3.2 Optimal algorithm

We construct here a dynamic program that solves Problem 2 optimally. To do so we introduce two auxiliary dynamic programs during the construction. The time complexity of our optimal algorithm is $O(l^2)$.

Definition 5.4 ($\mathrm{Opt}_0(l, c_m)$)**.** Let $l \in \mathbb{N}$ and $c_m \in \mathbb{N}$, $\mathrm{Opt}_0(l, c_m)$ is the execution time of an optimal solution to $\textsc{Prob}(l, c_m)$.

Note that $\mathrm{Opt}_0(l, c_m)$ is the execution time of the routine $\textsc{Revolve}(l, c_m)$, from Griewank and Walther [58].

Definition 5.5 ($\mathrm{Opt}_\infty(l, c_m, w_d, r_d)$)**.** Let $l \in \mathbb{N}$, $c_m \in \mathbb{N}$, $w_d \in \mathbb{R}$ and $r_d \in \mathbb{R}$, $\mathrm{Opt}_\infty(l, c_m, w_d, r_d)$ is the execution time of an optimal solution to $\textsc{Prob}_\infty(l, c_m, w_d, r_d)$.

To compute $\text{Opt}_\infty(l, c_m, w_d, r_d)$, we first focus on the variant of Problem 2 where the input value x_0 is initially in both the top buffer and the disk:

Problem 3 ($\text{PROB}_\infty^{(d)}(l, c_m, w_d, r_d)$). *We want to minimize the makespan of the AC problem with the following parameters:*

		Initial state:
AC graph:	*size* l	
Steps:	u_f, u_b	
Memory:	c_m, $w_m = r_m = 0$,	$\mathcal{M}_{ini} = \emptyset$
Disks:	$c_d = +\infty, w_d, r_d$,	$\mathcal{D}_{ini} = \{x_0\}$
Buffers:	$\mathcal{B}^\top$, $\mathcal{B}^\bot$	$\mathcal{B}^\top_{ini} = \{x_0\}$, $\mathcal{B}^\bot_{ini} = \{\bar{x}_{l+1}\}$

Definition 5.6 ($\overline{\text{PROB}}_\infty^{(d)}(l, c_m, w_d, r_d)$). Let $l \in \mathbb{N}$, $c_m \in \mathbb{N}$, $w_d \in \mathbb{R}$ and $r_d \in \mathbb{R}$, $\overline{\text{PROB}}_\infty^{(d)}(l, c_m, w_d, r_d)$ is the subproblem of $\text{PROB}_\infty^{(d)}(l, c_m, w_d, r_d)$ where the space of solution is restricted to the schedules that satisfy the following properties:
(P6):

(i) (P0(i)), (P0(ii)), (P0(iii)), (P0(iv)), and (P0(v)) are matched.

(ii) Given i, there are no operations F_j for $i \leq j \leq l-1$ after the execution of $\bar{F}_i$.

(iii) (Memory and disk checkpoint persistence) Let $s \in \{m, d\}$ and $i < l$, if W_i^s is executed, then there are no D_i^s until after the execution of $\bar{F}_i$. Moreover no operations involving F_0 to F_{i-1} or values x_0 to x_{i-1} are taken until after the execution of $\bar{F}_i$.

(iv) Let $0 \leq i, j \leq l-1$. All W_i^d operations are executed before any W_j^m operation. Moreover, all W_i^d operations are executed during iteration l.

Note that (P6) is a subset of (P0–5). Let $\text{Opt}_\infty^{(d)}(l, c_m, w_d, r_d)$ be the execution time of an optimal solution to $\overline{\text{PROB}}_\infty^{(d)}(l, c_m, w_d, r_d)$.

Theorem 5.1 (Optimal solution to $\text{PROB}_\infty(l, c_m, w_d, r_d)$). *Let $l \in \mathbb{N}$, $c_m \in \mathbb{N}$, $w_d \in \mathbb{R}$, and $r_d \in \mathbb{R}$.*

$$Opt_\infty(l, c_m, w_d, r_d) = \min \begin{cases} Opt_0(l, c_m) \\ w_d + Opt_\infty^{(d)}(l, c_m, w_d, r_d) \end{cases}$$

Proof. Let

$$A = \text{Opt}_\infty(l, c_m, w_d, r_d)$$
$$B = \min \begin{cases} \text{Opt}_0(l, c_m) \\ w_d + \text{Opt}_\infty^{(d)}(l, c_m, w_d, r_d) \end{cases}$$

Let us show that $A \leq B$. Every solution to $\text{PROB}(l, c_m)$ is also a solution to $\text{PROB}_\infty(l, c_m, w_d, r_d)$. Hence,

$$\text{Opt}_\infty(l, c_m, w_d, r_d) \leq \text{Opt}_0(l, c_m).$$

Let $\bar{\mathcal{S}}_\infty^{(d)}$ be a solution to $\overline{\text{PROB}}_\infty^{(d)}(l, c_m, w_d, r_d)$. Then the sequence $W_0^d \cdot \bar{\mathcal{S}}_\infty^{(d)}$ is a solution to $\text{PROB}_\infty(l, c_m, w_d, r_d)$. Indeed, the only difference between $\overline{\text{PROB}}_\infty^{(d)}(l, c_m, w_d, r_d)$ and $\text{PROB}_\infty(l, c_m, w_d, r_d)$ is that in $\overline{\text{PROB}}_\infty^{(d)}(l, c_m, w_d, r_d)$, the value x_0 is stored in the disk initially. Thus

$$\text{Opt}_\infty(l, c_m, w_d, r_d) \leq w_d + \text{Opt}_\infty^{(d)}(l, c_m, w_d, r_d)$$

and $\text{Opt}_\infty(l, c_m, w_d, r_d) \leq \min\{\text{Opt}_0(l, c_m);\; w_d + \text{Opt}^{(d)}_\infty(l, c_m, w_d, r_d)\}$.

Let us show that $A \geq B$. According to § 5.3.1, there exists at least an optimal algorithm $\mathcal{S}_\infty$ to solve $\text{PROB}_\infty(l, c_m, w_d, r_d)$ that satisfies (P0–5). According to (P0(x)), the first operation of $\mathcal{S}_\infty$ is a W-type operation.

- If it is a W^m-type operation, according to (P5), there are no W^d-type operations is $\mathcal{S}_\infty$. Hence the disk is not used at all in $\mathcal{S}_\infty$, and $\mathcal{S}_\infty$ is also a solution to $\text{PROB}(l, c_m)$. Thus

$$\text{Opt}_\infty(l, c_m, w_d, r_d) \geq \text{Opt}_0(l, c_m).$$

- If it is a W^d-type operation, $\mathcal{S}_\infty$ has the shape $\mathcal{S}_\infty = W^d_0 \cdot \mathcal{S}'_\infty$, where $\mathcal{S}'_\infty$ is a solution to $\text{PROB}^{(d)}_\infty(l, c_m, w_d, r_d)$. Since $\mathcal{S}_\infty$ satisfies (P0), (P1), (P2), (P3), (P4), and (P5), $\mathcal{S}'_\infty$ satisfies (P6). Hence $\mathcal{S}'_\infty$ is a solution to $\overline{\text{PROB}}^{(d)}_\infty(l, c_m, w_d, r_d)$. Thus

$$\text{Opt}_\infty(l, c_m, w_d, r_d) \geq w_d + \text{Opt}^{(d)}_\infty(l, c_m, w_d, r_d).$$

We get that $\text{Opt}_\infty(l, c_m, w_d, r_d) \geq \min\{\text{Opt}_0(l, c_m);\; w_d + \text{Opt}^{(d)}_\infty(l, c_m, w_d, r_d)\}$, which concludes the proof. ■

To compute $\text{Opt}^{(d)}_\infty(l, c_m, w_d, r_d)$, we need to consider the problem with only one disk slot containing x_0 at the beginning of the execution:

Problem 4 ($\text{PROB}^{(d)}_1(l, c_m, w_d, r_d)$)**.** *We want to minimize the makespan of the AC problem with the following parameters.*

		Initial state:
AC graph:	*size* l	
Steps:	u_f, u_b	
Memory:	c_m, $w_m = r_m = 0$,	$\mathcal{M}_{ini} = \emptyset$
Disks:	$c_d = 1, w_d, r_d,$	$\mathcal{D}_{ini} = \{x_0\}$
Buffers:	$\mathcal{B}^\top$, $\mathcal{B}^\perp$	$\mathcal{B}^\top_{ini} = \{x_0\}$, $\mathcal{B}^\perp_{ini} = \{\bar{x}_{l+1}\}$

Definition 5.7 ($\overline{\text{PROB}}^{(d)}_1(l, c_m, w_d, r_d)$)**.** Let $l \in \mathbb{N}$, $c_m \in \mathbb{N}$, $w_d \in \mathbb{R}$ and $r_d \in \mathbb{R}$, $\overline{\text{PROB}}^{(d)}_1(l, c_m, w_d, r_d)$ is the subproblem of $\text{PROB}^{(d)}_1(l, c_m, w_d, r_d)$, where the space of solution is restricted to the schedules that satisfy (P6) and that do not contain any W^d-type operation (and, therefore, the value x_0 is never discarded from the disk).

Let $\text{Opt}^{(d)}_1(l, c_m, w_d, r_d)$ be the execution time of an optimal solution to $\overline{\text{PROB}}^{(d)}_1(l, c_m, w_d, r_d)$.

Theorem 5.2 (Optimal solution to $\overline{\text{PROB}}^{(d)}_\infty(l, c_m, w_d, r_d)$)**.** *Given $l \in \mathbb{N}$, $c_m \in \mathbb{N}$, $w_d \in \mathbb{R}$ and $r_d \in \mathbb{R}$: If $l = 0$, then*

$$Opt^{(d)}_\infty(0, c_m, w_d, r_d) = u_b$$

else

$$Opt^{(d)}_\infty(l, c_m, w_d, r_d) = \min_{1 \leq j \leq l-1} \begin{cases} Opt^{(d)}_1(l, c_m, w_d, r_d) \\ j u_f + Opt_\infty(l-j, c_m, w_d, r_d) + r_d + Opt^{(d)}_1(j-1, c_m, w_d, r_d) \end{cases}$$

Proof. For $l = 0$, the result is immediate. Let us prove the result for $l \geq 1$. Let

$$A = \text{Opt}_\infty^{(d)}(l, c_m, w_d, r_d)$$
$$B = \min_{1 \leq j \leq l-1} \begin{cases} \text{Opt}_1^{(d)}(l, c_m, w_d, r_d) \\ ju_f + \text{Opt}_\infty(l-j, c_m, w_d, r_d) + r_d + \text{Opt}_1^{(d)}(j-1, c_m, w_d, r_d) \end{cases}$$

Let us show that $A \leq B$. Every solution to $\overline{\textsc{Prob}}_1^{(d)}(l, c_m, w_d, r_d)$ is also a solution to $\overline{\textsc{Prob}}_\infty^{(d)}(l, c_m, w_d, r_d)$. Hence,

$$\text{Opt}_\infty^{(d)}(l, c_m, w_d, r_d) \leq \text{Opt}_1^{(d)}(l, c_m, w_d, r_d).$$

Given j, $1 \leq j \leq l-1$, let $\mathcal{S}_1^{(d)}$ be an optimal solution to $\overline{\textsc{Prob}}_1^{(d)}(j-1, c_m, w_d, r_d)$. Let $\mathcal{S}_\infty$ be an optimal solution to $\textsc{Prob}_\infty(l-j, c_m, w_d, r_d)$ that satisfies (P0–5). Let $\mathcal{S}'_\infty$ be the sequence $\mathcal{S}_\infty$ where every index of the operations are increased by j (F_i becomes F_{i+j}, $W_i^{(m)}$ becomes $W_{i+j}^{(m)} \dots$). $\mathcal{S}'_\infty$ is still valid and has the same makespan as $\mathcal{S}_\infty$. Then, the sequence $F_{0\to j} \cdot \mathcal{S}'_\infty \cdot R_0^d \cdot \mathcal{S}_1^{(d)}$ is a solution to $\textsc{Prob}_\infty^{(d)}(l, c_m, w_d, r_d)$. By construction this sequence also satisfies (P6). Its execution time is $ju_f + \text{Opt}_\infty(l-j, c_m, w_d, r_d) + r_d + \text{Opt}_1^{(d)}(j-1, c_m, w_d, r_d)$. Thus, for all $1 \leq j \leq l-1$:

$$\text{Opt}_\infty^{(d)}(l, c_m, w_d, r_d) \leq ju_f + \text{Opt}_\infty(l-j, c_m, w_d, r_d) + r_d + \text{Opt}_1^{(d)}(j-1, c_m, w_d, r_d).$$

In particular it is smaller than the minimum over all j, hence the result.

Let us show that $A \geq B$. Let $\mathcal{S}_\infty^{(d)}$ be an optimal solution to $\overline{\textsc{Prob}}_\infty^{(d)}(l, c_m, w_d, r_d)$. $\mathcal{S}_\infty^{(d)}$ satisfies (P6) and its makespan is $\text{Opt}_\infty^{(d)}(l, c_m, w_d, r_d)$.

Assume first that there is at least one W-type operation in $\mathcal{S}_\infty^{(d)}$. Consider the first one. We can prove that it occurs before the first $\bar{F}$-type operation.

- If it is a W^d-type, then it occurs before the first $\bar{F}$-type according to (P6(iv))
- If it is a W^m-type, then obviously $c_m > 0$. If no W^m-operation occured during iteration l, then at the beginning of iteration $l-1$, R_0^d is executed. Hence a better solution would be better to start with W_0^m and D_0^m at the beginning of iteration $l-1$, which contradicts the optimality of $\mathcal{S}_\infty^{(d)}$.

Hence, the first W-type operation in $\mathcal{S}_\infty^{(d)}$ occurs before the first $\bar{F}$-type operation. Then two possibilities exist.

- The first operation in $\mathcal{S}_\infty^{(d)}$ is W_0^m. Since $\mathcal{S}_\infty^{(d)}$ satisfies (P6(iv)), there are no W^d-type operations in $\mathcal{S}_\infty^{(d)}$. Thus no other value than x_0 will be stored in the disk during the execution. Furthermore, because x_0 is stored in memory, it will not be read from the disk (otherwise $\mathcal{S}_\infty^{(d)}$ will not be optimal). Thus $\mathcal{S}_\infty^{(d)}$ is also a solution to $\overline{\textsc{Prob}}_1^{(d)}(l, c_m, w_d, r_d)$ and

$$\text{Opt}_\infty^{(d)}(l, c_m, w_d, r_d) \geq \text{Opt}_1^{(d)}(l, c_m, w_d, r_d).$$

- Otherwise, the first operation in $\mathcal{S}_\infty^{(d)}$ is not W_0^m. Consider the first W-type operation. Because it occurs before the first $\bar{F}$-type operation, it cannot be W_0^m (otherwise it would be the first operation in $\mathcal{S}_\infty^{(d)}$), nor W_0^d (x_0 is already stored on disk). Let W_j^s ($s \in \{m, d\}$) be the first W-type operation in $\mathcal{S}_\infty^{(d)}$, then $j > 0$.

⋆ We proved that there are no $\bar{F}$-type operations before W_j^s in $\mathcal{S}_\infty^{(d)}$. By definition, there are no W-type operations before W_j^s in $\mathcal{S}_\infty^{(d)}$. Since before W_j^s the memory is empty, there are no D-type operations in $\mathcal{S}_\infty^{(d)}$. Since the only value in one of the storage is x_0 (in the disk), the only possible R-type operation before W_j^s would be R_0^d. The only reason to execute R_0^d would be to perform F_0. However, F_0 can be executed at the beginning of $\mathcal{S}_\infty^{(d)}$ at no cost, since the value x_0 is already in the top buffer. Thus, there are only F-type operations before W_j^s in $\mathcal{S}_\infty^{(d)}$ (P6(i)).
⋆ According to (P6(iii)), after W_j^s, there are no operations involving values x_0 to x_{j-1} until after the operation $\bar{F}_j$.
⋆ According to (P6(ii)), there are no operations involving values x_j to x_l after the operation $\bar{F}_j$.
⋆ Since after the operation $\bar{F}_j$ the content of the top buffer is useless, the first operation after $\bar{F}_j$ has to be an R-type operation.
⋆ Moreover, since the only value from x_0 to x_{j-1} in one of the storage after $\bar{F}_j$ is x_0 (in the disk), it has to be R_0^d. Thus, based on all these considerations, $\mathcal{S}_\infty^{(d)}$ has the following shape:

$$\mathcal{S}_\infty^{(d)} = F_{0\to j} \cdot W_j^s \cdot \mathcal{S}_1 \cdot R_0^d \cdot \mathcal{S}_2$$

where (i) no operations involve values x_0 to x_{j-1} in $\mathcal{S}_1$ and (ii) no operations involve values x_j to x_l in $\mathcal{S}_2$.

Let $\mathcal{S}'_1$ be the sequence $W_j^s \cdot \mathcal{S}_1$, where every index of the operations is decreased by j (F_i becomes F_{i-j}, W_i^m becomes W_{i-j}^m,...). Then $\mathcal{S}'_1$ is a solution to $\text{PROB}_\infty(l-j, c_m, w_d, r_d)$, whose makespan is necessarily not smaller than $\text{Opt}_\infty(l-j, c_m, w_d, r_d)$.

On the other hand, the sequence $\mathcal{S}_2$ executes all the $\bar{F}$-type operations from $\bar{F}_{j-1}$ down to $\bar{F}_0$ with no operations involving values x_j to x_l. Furthermore, $\mathcal{S}_2$ does not use disk slots, except the one already used by value x_0. Since $\mathcal{S}_\infty^{(d)}$ satisfies (P0(5)), $\mathcal{S}_2$ satisfies (P6). Hence $\mathcal{S}_2$ is a solution to $\overline{\text{PROB}}_\infty^{(d)}(j-1, c_m, w_d, r_d)$, and its makespan is greater than $\text{Opt}_\infty^{(d)}(j-1, c_m, w_d, r_d)$. Finally, we have

$$\text{Opt}_\infty^{(d)}(l, c_m, w_d, r_d) \geq j u_f + \text{Opt}_\infty(l-j, c_m, w_d, r_d) + r_d + \text{Opt}_1^{(d)}(j-1, c_m, w_d, r_d);$$

in particular it is smaller than B.

We note that if there is no W-type operation, because we do not use any additional disk slot, $\mathcal{S}_\infty^{(d)}$ is also a solution to $\overline{\text{PROB}}_1^{(d)}(l, c_m, w_d, r_d)$ and $\text{Opt}_\infty^{(d)}(l, c_m, w_d, r_d) \geq \text{Opt}_1^{(d)}(l, c_m, w_d, r_d)$. This shows that $A \geq B$ and concludes the proof. ■

Theorem 5.3 (Optimal solution to $\overline{\text{PROB}}_1^{(d)}(l, c_m, w_d, r_d)$). *Let $l \in \mathbb{N}$, $c_m \in \mathbb{N}$, $w_d \in \mathbb{R}$ and $r_d \in \mathbb{R}$: If $l = 0$, then*

$$Opt_1^{(d)}(0, c_m, w_d, r_d) = u_b$$

else

$$Opt_1^{(d)}(l, c_m, w_d, r_d) = \min_{1\leq j\leq l-1} \begin{cases} Opt_0(l, c_m) \\ j u_f + Opt_0(l-j, c_m) + r_d + Opt_1^{(d)}(j-1, c_m, w_d, r_d) \end{cases}$$

Proof. For $l = 0$, the result is immediate. Let us prove the result for $l \geq 1$.

Let

$$A = \text{Opt}_1^{(d)}(l, c_m, w_d, r_d)$$
$$B = \min_{1 \le j \le l-1} \begin{cases} \text{Opt}_0(l, c_m) \\ ju_f + \text{Opt}_0(l-j, c_m) + r_d + \text{Opt}_1^{(d)}(j-1, c_m, w_d, r_d) \end{cases}$$

Let us show that $A \le B$. Every solution to PROB(l, c_m) is also a solution to $\overline{\text{PROB}}_1^{(d)}(l, c_m, w_d, r_d)$. Hence,

$$\text{Opt}_1^{(d)}(l, c_m, w_d, r_d) \le \text{Opt}_0(l, c_m).$$

Given j, $1 \le j \le l-1$, let $\mathcal{S}_2$ be an optimal solution to $\overline{\text{PROB}}_1^{(d)}(j-1, c_m, w_d, r_d)$. Let $\mathcal{S}_1$ be an optimal solution to PROB$(l-j, c_m)$. Let $\mathcal{S}'_1$ be the sequence $\mathcal{S}_1$ where every index of the operations are increased by j (F_i becomes F_{i+j}, $W_i^{(m)}$ becomes $W_{i+j}^{(m)} \ldots$). $\mathcal{S}'_1$ is still valid and has the same makespan as $\mathcal{S}_1$. Then, the sequence $F_{0\to j} \cdot \mathcal{S}'_1 \cdot R_0^d \cdot \mathcal{S}_2$ is a solution to PROB$_1^{(d)}(l, c_m, w_d, r_d)$. By construction this sequence does not contain any W^d-type operation since neither $\mathcal{S}'_1$ nor $\mathcal{S}_2$ do. Its execution time is $ju_f + \text{Opt}_0(l-j, c_m) + r_d + \text{Opt}_1^{(d)}(j-1, c_m, w_d, r_d)$. Thus, for all $1 \le j \le l-1$:

$$\text{Opt}_1^{(d)}(l, c_m, w_d, r_d) \le ju_f + \text{Opt}_0(l-j, c_m) + r_d + \text{Opt}_1^{(d)}(j-1, c_m, w_d, r_d)\}.$$

In particular it is smaller than the minimum over all j, hence the result.

Let us show that $A \ge B$. Let $\mathcal{S}_1^{(d)}$ be an optimal solution to $\overline{\text{PROB}}_1^{(d)}(l, c_m, w_d, r_d)$. $\mathcal{S}_1^{(d)}$ satisfies (P6) and does not contain any W^d-type operations. Its makespan is $\text{Opt}_1^{(d)}(l, c_m, w_d, r_d)$.

First, note that if $c_m > 0$, then there is a W^m-type operation in iteration l of $\mathcal{S}_1^{(d)}$. Otherwise, if no W^m-operation occured during iteration l, then at the beginning of iteration $l-1$, R_0^d is executed. Hence a better solution would be better to start with W_0^m and D_0^m at the beginning of iteration $l-1$, which contradicts the optimality of $\mathcal{S}_1^{(d)}$.

Hence, the first W-type operation in $\mathcal{S}_1^{(d)}$ occurs before the first $\bar{F}$-type operation. Then there are two possibilities.

- The first operation in $\mathcal{S}_1^{(d)}$ is W_0^m. Because x_0 is stored in memory, it will not be read from the disk (otherwise $\mathcal{S}_\infty^{(d)}$ will not be optimal). Thus $\mathcal{S}_\infty^{(d)}$ is also a solution to $\overline{\text{PROB}}_1^{(d)}(l, c_m, w_d, r_d)$ and
$$\text{Opt}_1^{(d)}(l, c_m, w_d, r_d) \ge \text{Opt}_0(l, c_m).$$

- Otherwise, the first operation in $\mathcal{S}_1^{(d)}$ is not W_0^m. Consider the first W-type operation. It occurs before the first $\bar{F}$-type operation and hence cannot be W_0^m (otherwise it would be the first operation in $\mathcal{S}_1^{(d)}$). Let W_j^m (there are no W^d-type operations) be the first W-type operation in $\mathcal{S}_1^{(d)}$, $j > 0$.

  ⋆ We proved that there are no $\bar{F}$-type operations before W_j^m in $\mathcal{S}_\infty^{(d)}$. By definition, there are no W-type operations before W_j^m in $\mathcal{S}_1^{(d)}$. Since the only value in one of the storage slots is x_0 (in the disk), the only possible R-type operation before W_j^s would be R_0^d. The only reason to execute R_0^d would be to perform F_0. However, F_0 can be executed at the beginning of $\mathcal{S}_1^{(d)}$ at no cost, since the value x_0 is already in the top buffer. Thus, there are only F-type operations before W_j^m in $\mathcal{S}_\infty^{(d)}$ (P6(i)).

  ⋆ According to (P6(iii)), after W_j^m, there are no operations involving values x_0 to x_{j-1} until after

the operation $\bar{F}_j$.
⋆ According to (P6(ii)), there are no operations involving values x_j to x_l after the operation $\bar{F}_j$.
⋆ Since after the operation $\bar{F}_j$, the content of the top buffer is useless, the first operation after $\bar{F}_j$ has to be an R-type operation.
⋆ Moreover, since the only value from x_0 to x_{j-1} in one of the storage slots after $\bar{F}_j$ is x_0 (in the disk), it has to be R^d_0. Thus, based on all this considerations, $\mathcal{S}^{(d)}_1$ has the following shape:

$$\mathcal{S}^{(d)}_1 = F_{0\to j} \cdot W^m_j \cdot \mathcal{S}_1 \cdot R^d_0 \cdot \mathcal{S}_2$$

where (i) no operations involve values x_0 to x_{j-1} in $\mathcal{S}_1$ and (ii) no operations involve values x_j to x_l in $\mathcal{S}_2$.

Let $\mathcal{S}'_1$ br the sequence $W^m_j \cdot \mathcal{S}_1$, where every index of the operations is decreased by j (F_i becomes F_{i-j}, W^m_i becomes $W^m_{i-j}, \ldots$). Then $\mathcal{S}'_1$ is a solution to $\text{PROB}(l-j, c_m)$, whose makespan is necessarily not smaller than $\text{Opt}_0(l-j, c_m, w_d, r_d)$.

On the other hand, the sequence $\mathcal{S}_2$ executes all the $\bar{F}$-type operations from $\bar{F}_{j-1}$ down to $\bar{F}_0$ with no operations involving values x_j to x_l. Furthermore, $\mathcal{S}_2$ does not use disk slots, except the one already used by value x_0. Thus $\mathcal{S}_2$ is a solution to $\overline{\text{PROB}}^{(d)}_1(j-1, c_m, w_d, r_d)$, and its makespan is greater than $\text{Opt}^{(d)}_\infty(j-1, c_m, w_d, r_d)$. Finally, we have

$$\text{Opt}^{(d)}_\infty(l, c_m, w_d, r_d) \geq j u_f + \text{Opt}_\infty(l-j, c_m, w_d, r_d) + r_d + \text{Opt}^{(d)}_1(j-1, c_m, w_d, r_d);$$

in particular it is smaller than B.

Note that if there is no W-type operation, because we do not use any additional disk slot, $\mathcal{S}^{(d)}_\infty$ is also a solution to $\overline{\text{PROB}}^{(d)}_1(l, c_m, w_d, r_d)$ and $\text{Opt}^{(d)}_\infty(l, c_m, w_d, r_d) \geq \text{Opt}^{(d)}_1(l, c_m, w_d, r_d)$. This shows that $A \geq B$ and concludes the proof. ■

5.4 Simulations

In this section we compare our optimal algorithm with the only (to the best of our knowledge) algorithm for multilevel checkpointing, introduced by Stumm and Walther [101].

5.4.1 Stumm and Walther's algorithm (SWA$^\star(l, c_m, w_d, r_d)$)

Stumm and Walther [101] solve Problem 2 using a variant of REVOLVE [107]. REVOLVE takes l the size of the AC graph and s the number of storage slots as argument and returns an optimal solution for Problem 1. Stumm and Walther show that in REVOLVE(l, s), some storage slots are less used than others. They design the SWA algorithm that takes l the size of the AC graph, c_m the number of memory slots, and c_d the number of disk slots as argument and returns the solution REVOLVE$(l, c_d + c_m)$ where the c_d storage slots that are the least used are considered as disk slots and all the others are considered as memory slots.

To solve Problem 2, SWA* returns the best solution among the solutions returned by SWA(l, c_m, c_d, w_d, r_d), that is,

$$\text{SWA}^\star(l, c_m, w_d, r_d) = \min_{c_d=0\ldots l-c_m} \text{SWA}(l, c_m, c_d, w_d, r_d)$$

(having more than l storage slots is useless).

5.4.2 Simulation setup

For the simulations we have tested our algorithm and Stumm and Walther's algorithm on AC graphs of size up to 20,000 with different numbers of memory checkpoints. In global ocean circulation modeling [64], a graph of size 8,640 represents one year of results with an hourly timestep.

In the experiments, we normalize all time values by setting u_f to 1. We take $u_b = 2.5$ as a representative value [64]. Here we present results for $c_m \in \{2, 5, 10, 25\}$ and $w_d = r_d \in \{1, 2, 5, 10\}$.

In Figure 5.5 we reproduce Stumm and Walther's results in order to study the behavior of SWA. We plot the execution time of SWA as a function of c_d for a fixed graph size (note that we used a logarithmic scale for the horizontal axis for better readability). We compare it with the optimal solution $\text{Opt}_\infty(l, c_m, w_d, r_d)$.

In Figures 5.6 and 5.7 we plot the ratio between SWA* and $\text{Opt}_\infty(l, c_m, w_d, r_d)$ as a function of the size of the AC graph with different values of c_m, w_d and r_d.

5.4.3 Simulation results

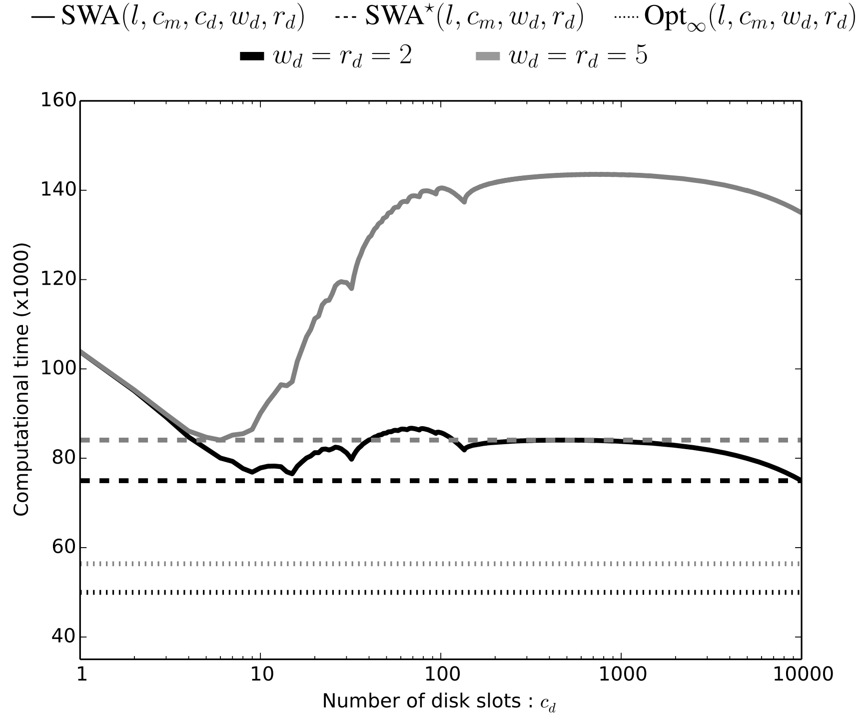


Figure 5.5: Makespan of SWA on an AC graph of size 10,000 as a function of c_d for $c_m = 5$. SWA* and $\text{Opt}_\infty(l, c_m, w_d, r_d)$ are also plotted for comparison.

First we observe the behavior of SWA given the amount of available disk slots for different disk access costs. The two plots in Figure 5.5 are representative of the two behaviors we observed for SWA during our experiments. We can see that the makespan of SWA(l, c_m, c_d, w_d, r_d) is always strictly higher than the optimal one for an infinite number of disk slots $\text{Opt}_\infty(l, c_m, w_d, r_d)$ (see the dotted line). For all values studied, the evolution of the execution time of SWA when the amount of storage increases follows a specific pattern that can be divided into three phases.

1. A very fast decrease with the first additional disk slots.

2. A succession of small increases and decreases.

3. A slow but steady decrease until all steps are stored (remember that the horizontal axis has a logarithmic scale for better readability).

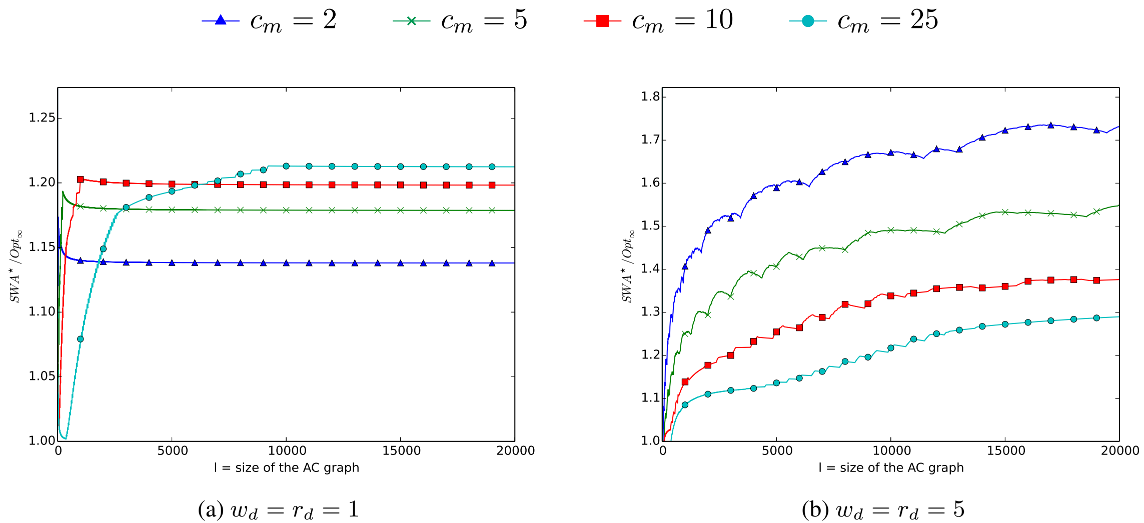
In all our experiments, the minimum for SWA is reached either at the end of step 1 with a very low number of disk checkpoints ($c_d = r_d = 5$ in Figure 5.5) or at the end of step 3 with a very high number of disk checkpoints ($c_d = r_d = 2$ in Figure 5.5), depending on the disk access costs and the number of memory slots. Eventually, given the general shape of the SWA performances, we assume that when the size of the graph increases enough, the minimum value is always reached at the end of step 3, when every output of the AC graph is stored in one of the storage slots.

Note that Stumm and Walther observed a fourth phase [101] where the computational time increases again when the number of disk checkpoints gets closer to the total number of steps. They explained it by saying that when the volume of data stored on the disk reaches a threshold, the cost of a disk access increases, which in turn increases the computational time. We do not observe such a fourth step because we plot the computational time obtained when giving the model parameters as input to SWA (and the cost of disk access remains constant).

In the following, the time complexity of SWA* does not allow us to run large instances of l. To be able to plot SWA* for large AC graphs, we plot a faster version of SWA* that takes the previous remarks into account, namely, that assume that the minimum is either reached at the end of phase 1 (for small values of c_d) or at the end of phase 3. More precisely, we consider that the end of phase 1 is reached before a number of disk size equal to 200 (for the problem sizes considered in this chapter), and we plot a faster version of SWA$^\star(l, c_m, w_d, r_d)$:

$$\text{SWA}^\star(l, c_m, w_d, r_d) = \min\left(\min_{c_d=0\ldots200} \text{SWA}(l, c_m, c_d, w_d, r_d), \text{SWA}(l, c_m, l-c_m, w_d, r_d)\right).$$

This assumption allows us to compute SWA* for large values of l and to compare it with the optimal computational time for an infinite number of disk slots.



(a) $w_d = r_d = 1$

(b) $w_d = r_d = 5$

Figure 5.6: Ratio SWA$^\star(l, c_m, w_d, r_d)/\text{Opt}_\infty(l, c_m, w_d, r_d)$ as a function of l.

Figures 5.6 and 5.7 depict the overhead of using SWA* compared with the optimal solution $\text{Opt}_\infty(l, c_m, w_d, r_d)$ that we designed in § 5.3.2. For unlimited disk slots, SWA* returns the best solution

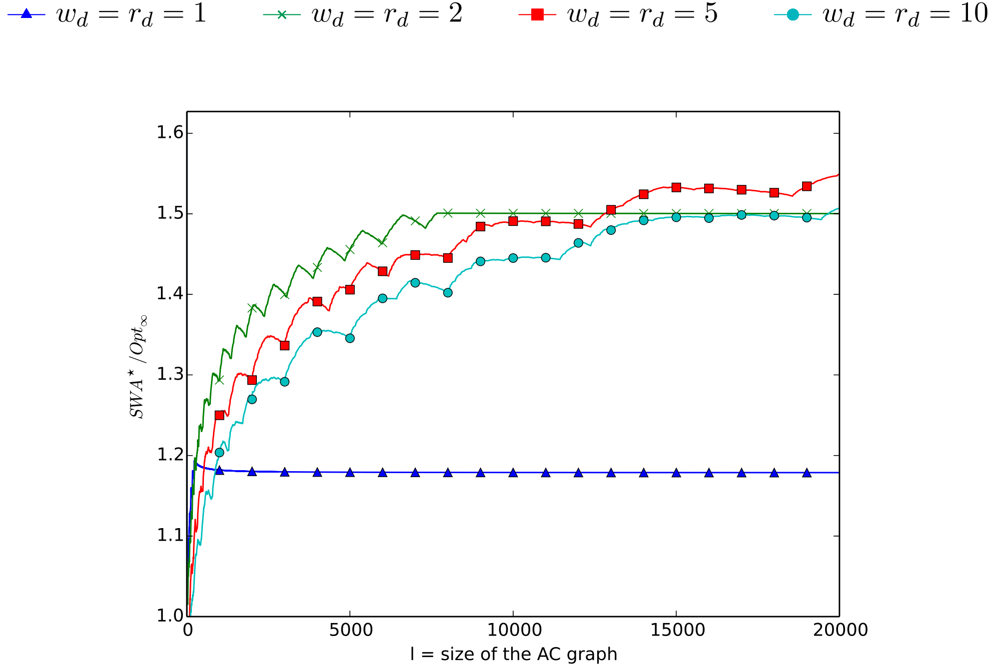


Figure 5.7: Ratio $\text{SWA}^\star(l, c_m, w_d, r_d)/\text{Opt}_\infty(l, c_m, w_d, r_d)$ as a function of l, for $c_m = 5$.

among the solutions returned by SWA, and this solution is always greater than $\text{Opt}_\infty(l, c_m, w_d, r_d)$. We observe that the ratio increases until the size of the graph reaches a threshold where the ratio becomes constant. This is particularly visible in Figure 5.6a where we can see that the value of this threshold increases with the number of memory slots c_m. In practice, the threshold delimits the moment when the number of disk slots used by SWA* goes from a relatively small number (when the minimum for $\text{SWA}(l, c_m, c_d, w_d, r_d)$ is reached at the end of phase 1) to $c_d = l - c_m$ (when the minimum for $\text{SWA}(l, c_m, c_d, w_d, r_d)$ is reached when all forward steps are checkpointed, at the end of phase 3).

We are interested in the limit ratio reached after the threshold because we are considering the problem for very large graphs. In Figures 5.6a and 5.7 we can see that this ratio increases when c_m or w_d and r_d increase. When $r_d = w_d = 1$, the ratio limit for $c_m = 2$ is approximately 1.14, which means that SWA* is 14% slower than the optimal algorithm we designed in §5.3.2. For a memory of size $c_m = 10$, this overhead increases to 20% for large AC graphs. When $r_d = w_d = 5$, the ratio limit is not reached for AC graphs of size inferior to 20,000. But since the ratio for $c_m = 2$ will be higher than 1.6, we can state that SWA* will perform at least 60% slower than the optimal algorithm on large AC graphs for any memory sizes.

5.5 Conclusion and future work

In this chapter we have provided optimal algorithms for the adjoint checkpointing problem with two storage locations: a bounded number of memory slots with zero access cost and an infinite number of disk slots with a given write and read costs. We have compared our optimal solution with existing work, showing that our solution gives significantly better execution time.

We have identified applications in computational fluid dynamics and earth systems modeling that could benefit from our approach. We will examine whether the theoretical benefits of the optimal mul-

tistage schedule can be realized in practice. Future theoretical directions include the solution to the online AC problem (where the size l of the AC graph is not known before execution), within the same framework. Another possible extension could be to solve the same problem as in this chapter but with a limited number of disk checkpoints. Large-scale platforms are failure-prone, and checkpointing for resilience in addition to checkpointing for performance will lead to challenging algorithmic problems. As an intermediate step, we will examine the problem of maximizing progress during a fixed time period. This situation arises in practice when the job scheduler limits the maximum duration of jobs (limits such as 12 hours are common).

Conclusion

Throughout this thesis, we have designed memory-aware algorithms and scheduling techniques suited for modern memory architectures. We have shown special interest in improving the performance of matrix computations on multiple levels. At a high level, we have introduced new numerical algorithms for solving linear systems on large distributed platforms. Most of the time, these linear solvers rely on runtime systems to handle resources allocation and data management. We also focused on improving the dynamic schedulers embedded in these runtime systems by adding static information to their decision process. We proposed new memory-aware dynamic heuristics to schedule workflows, that could be implemented in such runtime systems.

Altogether, we have dealt with multiple state-of-the-art factorization algorithms used to solve linear systems, like the LU, QR and Cholesky factorizations. We targeted different platforms ranging from multicore processors to distributed memory clusters, and worked with several reference runtime systems tailored for these architectures, such as PaRSEC and StarPU. On a theoretical side, we took special care of modelling convoluted hierarchical memory architectures. We have classified the problems that are arising when dealing with these storage platforms. We have designed many efficient polynomial-time heuristics on general problems that had been shown NP-complete beforehand. Our main contributions are stated in the following paragraphs.

Chapter 1: Mixing LU and QR factorization algorithms to design high-performance dense linear algebra solvers

The first contribution of this thesis is the introduction of a new linear solver that offers an alternative to state-of-the-art solvers on distributed platforms. We pointed out the fact that LU steps and QR steps can be mixed during a factorization. The resulting hybrid LU-QR factorization accelerates the classical QR algorithm by introducing some LU steps whenever these do not compromise stability. We decided to implement the hybrid LU-QR algorithm using the HQR algorithm for the QR steps and the LUPP algorithm for the LU steps. The HQR algorithm offers a higher degree of parallelism than the classical QR factorization by introducing several eliminator tiles inside a panel. All tiles hosted on the same processor are eliminated by the same tile. These elimination tiles are then zeroed out by the diagonal tile using a reduction tree across processors. The LUPP algorithm is the classical LU factorization with partial pivoting, that finds the largest element of each column of the panel and uses it as pivot. Theoretically, the hybrid factorization concept can be expanded, and any variant of the LU and QR factorizations could be used. We could even alternate two different variants of the LU algorithm such as the LU factorization without pivoting that offers a high level of parallelism and the LU factorization with partial pivoting that provides better stability results in practice. This chapter opens the way to new factorization methods with high potential.

The choice to perform an LU step or a QR step is handled by a robustness criterion. In this chapter, we introduced three robustness criteria, all of them being motivated by a numerical study. The criteria rely on a threshold α that allows one to tighten or loosen the stability requirement. Moreover, we

have compared the effective performance of all criteria on actual distributed clusters, by conducting an extensive set of experiments. We have discussed how the size of these platforms impacts performance and stability. We saw that the three criteria provide a wide range of stability/performance trade-offs on random matrices, thanks to the tunable threshold α. They also manage to detect stability deterioration on most tested ill-conditioned matrices on which LUPP fails because of large growth factors. The Max criterion appeared to be the best choice since it did not fail on any pathological matrix of the Higham's Matrix Computation Toolbox [67].

Chapter 2: Bridging the gap between experimental performance and theoretical bounds for the Cholesky factorization on heterogeneous platforms

In this chapter, we have investigated to what extent the performance of the dynamic schedulers for the Cholesky factorization on heterogeneous platforms can be improved. Runtime systems usually use generic dynamic heuristics to allocate tasks onto available resources. The HEFT heuristic has the quality to be simple to implement, and to provide good performance in practice. We investigated how its local vision of the task graph workflow to schedule, may result in bad local decisions that can have consequences on the overall performance. We studied how adding static information to dynamic schedulers can prevent this phenomenon. We analytically computed a set of TRSM tasks that could be forced to execute on CPUs, even if their processing time would have been shorter on the GPUs. We selected these tasks based on their distance to the critical path of the application. Forcing the dynamic scheduler of the StarPU runtime system to make these non-intuitive decisions led to performance improvements for both simulations and actual experiments.

To quantify the room for improvement between these results and the best feasible solution, we derived upper bounds for the best achievable performance. We proposed to consider three bounds: one involving the critical path of the Cholesky factorization, one based of the maximum workload that resources can process, and a third one computed by a linear program using both notions. Experiments showed that we managed to raise the StarPU scheduler performance close to its best achievable result.

Chapter 3: Memory-aware list scheduling for hybrid platforms

This chapter has addressed the problem of scheduling general workflows on a heterogeneous architecture with two memories. We provided an extensive complexity study for the restricted problem where the workflow is tree-shaped and the tasks are already mapped on the resources. We showed the NP-hardness of determining if there exists a traversal of the tree that does not violate the memory constraints when both memories are limited. Worse, given two constants α and β, it is impossible in the general case to find a traversal that is both an α-approximation for one memory peak minimization and a β-approximation for the other memory peak minimization. However, we provided the optimal depth-first traversal for an arbitrary tree, which turns out to minimize both memories simultaneously.

In addition, we considered the general problem of scheduling an arbitrary workflow where each task can be mapped onto either resource, and has a different processing time for each type. We designed two dynamic memory-aware algorithms, MEMHEFT and MEMMINMIN, based on the reference heuristics HEFT and MINMIN. MEMHEFT (respectively MEMMINMIN) behaves like HEFT (respectively MEMMINMIN) when the memory constraints are not critical, and provide valid schedules with a reasonable makespan overhead when reference heuristics fail to respect memory constraints. These memory-aware heuristics could be implemented in runtime systems like PARSEC or StarPU when one wants to enforce a memory usage bound to avoid out-of-core memory access, for instance.

Chapter 4: Assessing the cost of redistribution followed by a computational kernel

In this chapter, we took into consideration the fact that sometimes, in the context of distributed computing, the initial data distribution is not suited for the computational kernel that we want to process. In such situations, the data elements have to be redistributed across the resources. We assumed that an optimal (or close to optimal) data partition for the computational kernel is known. We provided polynomial algorithms to compute the best data distribution, given the target data partition, for two metrics: minimizing the total volume of communications during the redistribution, and minimizing the number of parallel redistribution steps. However we showed that, when the redistribution is followed by a computation kernel, redistributing the data elements toward this targed data distribution is not always the best option in term of overall completion time. Actually, finding the optimal data partition that minimizes the completion time of the redistribution followed by a computational kernel is NP-complete, even for a simple computational kernel like the 1D-stencil algorithm.

We considered four strategies to tackle this problem and tried them in the PaRSEC framework, for both the 1D-stencil kernel and the QR factorization algorithm. Our experiments showed that the new redistribution strategies presented in this chapter lead to better performance in all cases, for arbitrary initial data distributions, as well as for data distributions arising from Earth Science applications.

Chapter 5: Optimal multistage algorithm for adjoint computation

The last contribution of this thesis concerns a different scientific application, namely adjoint computation. The workflow arising in adjoint computation provides no parallelism across tasks and, in order to meet the memory usage constraints, we need to use a different strategy than in the rest of this thesis. Indeed, since there is not enough space in the memory to store every output file of the workflow, we need to decide which files will be saved, and which files will be deleted and recomputed later when needed. The case where only one limited memory is available has already been solved in the literature. In this chapter we considered the case where two different storages are available: the memory and the disk. Reading and writing checkpoints in memory are supposed to be free in terms of time, but the number of available slots is limited. On the contrary, the number of slots in the disk is unlimited but the time to read or write a file can no longer be ignored. We provided the first polynomial time algorithm that computes the best schedule for adjoint computation in this dual-storage context. Our algorithm relies on a convoluted dynamic program, and we proved the optimality of the computed solution. We also assess the performance of existing heuristics for this problem, compared to our optimal solution, through a full set of simulations.

Perspectives

Throughout this thesis, we mentioned at the end of each chapter some future work and research directions that could be investigated. Here, we outline multiple possible extensions to our work, along with more general, long-term oriented, research directions.

Linear algebra

In this thesis we have laid the foundation for hybrid linear solvers that alternate steps of different factorization algorithms. The main quality of our hybrid LU-QR algorithm is that it restricts the search of pivots inside the diagonal domain when doing so is numerically safe, thereby avoiding unnecessary communications across processors. But many variants of the hybrid LU-QR algorithm introduced in

this thesis could be implemented. An interesting application could be to derive LU algorithms with several eliminators per panel. Such "multi-killer" algorithms are well suited to modern distributed architectures. Indeed, at each step of the factorization, every tile of the panel except one per domains is zeroed out using a local tile hosted on the same processor. These operations do not involve communications across processors. Then, the remaining tiles are zeroed out using a reduction tree across domains. This approach has been proven effective for the QR factorization [42]. However, multi-killer LU factorizations have always turned out to be unstable so far. A hybrid algorithm coupling a multi-killer LU factorization with another more stable LU algorithm, like the LU factorization with partial pivoting, could be a pioneering method to develop a stable and effective LU algorithm with several eliminators per panel. Another course of study could be to focus on designing new factorization algorithms that are intrinsically communication-avoiding. This approach has shown to be effective and the communication-avoiding LU (CALU [61]) is known to be stable and efficient on both parallel distributed platforms and single multicore nodes.

Scheduling under memory constraints

All the memory-aware scheduling techniques introduced in this thesis could not be implemented in actual runtime systems, due to a lack of time. We designed memory-aware alternatives to the classical dynamic schedulers HEFT and MINMIN that would deserve to be tried in real-world conditions. These workflow schedulers ensure that the memory peak usage is smaller that a given bound and can prove useful in runtime systems like StarPU, that operate on shared-memory parallel platforms. They can help avoiding out-of-core memory access and better handling convoluted hierarchical memory layouts. Many other phenomena can be taken into account to model data communications. In this thesis, we only considered platforms with the bidirectional one-port model (where every communication can be processed in parallel as long as all senders and all receivers are different), or the bus model (where every communication has to processed sequentially). These assumptions correspond to a model with no contention between communications, and to a model with infinite contention, respectively. We could consider alternative cases where communications can be processed in parallel but their processing time depends on the number of communication taking place at the same time. This communication model can be considered more realistic and would lead to new interesting algorithmic problems.

Long-term perspectives

In this thesis, we devoted our efforts to improve resource and memory management on modern architectures, one necessary step to reach the Exascale performance mark. But many other challenges need to be faced to this end. Among them, with the increasing number of components, the reliability of the entire computational system is becoming an issue. Even if each component is quite reliable individually, the mean time between failures in next generation supercomputers is expected to drop drastically. Large scientific applications on these platforms will have to deal with a higher frequency of failures and errors, in the future. There are many possible approaches to remedy this problem. In matrix computations, we can mention algorithm-based fault tolerance techniques, using checksums that allow one to detect at each operation if a soft error occurred during the computation. It can be interesting to develop other low-overhead methods to detect soft errors. It could also be interesting to investigate how to efficiently schedule a workflow knowing that resources are more likely to fail at some point. New models of memory subject to failures can be introduced. It can be interesting, in auto-adjoint computations for instance, to consider the fact that one memory may be likely to lose every stored output file at some point, and to see how the probability of failures impacts the optimal algorithm introduced in this thesis.

Bibliography

[1] "Top500 list of June 2015," http://www.top500.org/lists/2015/06/.

[2] "LAPACK: Linear algebra package," 1992. [Online]. Available: http://www.netlib.org/lapack/

[3] "HPL: A portable implementation of the high-performance linpack benchmark for distributed-memory computers," 2008. [Online]. Available: http://www.netlib.org/benchmark/hpl/

[4] "Chameleon, a dense linear algebra software for heterogeneous architectures," 2014. [Online]. Available: https://project.inria.fr/chameleon

[5] "The white house, office of the press secretary, executive order: Creating a national strategic computing initiative," 2015, https://www.whitehouse.gov/the-press-office/2015/07/29/executive-order-creating-national-strategic-computing-initiative.

[6] Advanced Scientific Computing Advisory Committee (ASCAC), "Ten technical approaches to address the challenges of Exascale computing," http://science.energy.gov/~/media/ascr/ascac/pdf/meetings/20140210/Top10reportFEB14.pdf.

[7] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov, "Faster, cheaper, better - a hybridization methodology to develop linear algebra software for gpus," LAPACK Working Note 230, 2010.

[8] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The plasma and magma projects," *Journal of Physics: Conference Series*, vol. 180, no. 1, p. 012037, 2009.

[9] I. Ahmad and Y.-K. Kwok, "On exploiting task duplication in parallel program scheduling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 9, pp. 872–892, 1998.

[10] 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*, vol. 23, no. 1, pp. 15–41, 2001.

[11] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet, "Hybrid scheduling for the parallel solution of linear systems," *Parallel Computing*, vol. 32, no. 2, pp. 136–156, 2006.

[12] P. Amestoy, J.-Y. L'Excellent, and S. Pralet, "Personal communication," Jan. 2013.

[13] E. Anderson, J. Hall, J. Hartline, M. Hobbes, A. Karlin, J. Saia, R. Swaminathan, and J. Wilkes, "Algorithms for data migration," *Algorithmica*, vol. 57, no. 2, pp. 349–380, 2010.

[14] C. Augonnet, S. Thibault, and R. Namyst, "Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures," in *Proceedings of the International Euro-Par Workshops 2009, HPPC'09*, ser. Lecture Notes in Computer Science, vol. 6043. Delft, The Netherlands: Springer, Aug. 2009, pp. 56–65.

[15] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[16] M. Baboulin, J. J. Dongarra, J. Herrmann, and S. Tomov, "Accelerating linear system solutions using randomization techniques," *ACM Trans. Mathematical Software*, vol. 39, no. 2, pp. 8:1–8:13, 2013.

[17] O. Beaumont, V. Boudet, and Y. Robert, "A realistic model and an efficient heuristic for scheduling with heterogeneous processors," in *HCW'2002, the 11th Heterogeneous Computing Workshop*. IEEE Computer Society Press, 2002.

[18] P. Bhat, C. Raghavendra, and V. Prasanna, "Efficient collective communication in distributed heterogeneous systems," *Journal of Parallel and Distributed Computing*, vol. 63, pp. 251–263, 2003.

[19] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszcezek, A. YarKhan, and J. Dongarra, "Distributed dense numerical linear algebra algorithms on massively parallel architectures: DPLASMA," ICL, UTK, Tech. Rep., 2010.

[20] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. 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.

[21] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, "PaRSEC: Exploiting heterogeneity to enhance scalability," *IEEE Computing in Science and Engineering*, vol. 15, no. 6, pp. 36–45, 2013.

[22] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. 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.

[23] ——, "DAGuE: A generic distributed DAG engine for high performance computing," *Parallel Computing*, vol. 38, no. 1, pp. 37–51, 2012.

[24] H. Bouwmeester, M. Jacquelin, J. Langou, and Y. Robert, "Tiled QR factorization algorithms," in *Proc. ACM/IEEE SC11 Conference*. ACM Press, 2011.

[25] 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," *J. Parallel Distrib. Comput.*, vol. 61, no. 6, pp. 810–837, 2001.

[26] J. Bruck, C.-T. Ho, E. Upfal, S. Kipnis, and D. Weathersby, "Efficient algorithms for all-to-all communications in multiport message-passing systems," *IEEE Trans. Parallel Distributed Systems*, vol. 8, no. 11, pp. 1143–1156, 1997.

[27] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "Parallel tiled QR factorization for multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 13, pp. 1573–1590, 2008.

[28] ——, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Computing*, vol. 35, no. 1, pp. 38–53, 2009.

[29] H. Casanova, A. Legrand, and M. Quinson, "SimGrid: a Generic Framework for Large-Scale Distributed Experiments," in *proceedings of the 10th IEEE International Conference on Computer Modeling and Simulation (UKSim)*, Apr. 2008.

[30] E. Chan, F. G. Van Zee, P. Bientinesi, E. S. Quintana-Orti, G. Quintana-Orti, and R. Van de Geijn, "SuperMatrix: a multithreaded runtime scheduling system for algorithms-by-blocks," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, 2008, p. 123–132.

[31] H. Chetto, M. Silly, and T. Bouchentouf, "Dynamic scheduling of real-time tasks under precedence constraints," *Real-Time Systems*, vol. 2, no. 3, pp. 181–194, 1990.

[32] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. Whaley, "ScaLAPACK: a portable linear algebra library for distributed memory computers — design issues and performance," *Computer Physics Communications*, vol. 97, no. 1–2, p. 1–15, 1996.

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

[34] E. G. Coffman, M. R. Garey, D. S. Johnson, and A. S. LaPaugh, "Scheduling file transfers," *SIAM J. Comput.*, vol. 14, pp. 744–780, 1985.

[35] T. Davidović, L. Liberti, N. Maculan, and N. Mladenović, "Towards the optimal solution of the multiprocessor scheduling problem with communication delays," in *In MISTA Proceedings*, 2007.

[36] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, "Communication-optimal parallel and sequential QR and LU factorizations," *SIAM J. Scientific Computing*, vol. 34, no. 1, pp. A206–A239, 2012.

[37] J. W. Demmel, N. J. Higham, and R. S. Schreiber, "Block lu factorization," *Numerical Linear Algebra with Applications*, vol. 2, no. 2, pp. 173–190, 1995.

[38] F. Desprez, J. Dongarra, A. Petitet, C. Randriamaro, and Y. Robert, "Scheduling block-cyclic array redistribution," *IEEE Trans. Parallel Distributed Systems*, vol. 9, no. 2, pp. 192–205, 1998.

[39] C. D. Dhillon, J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, "Lapack working note 95 scalapack: A portable linear algebra library for distributed memory computers - design issues and performance," 1995.

[40] S. Donfack, J. Dongarra, M. Faverge, M. Gates, J. Kurzak, P. Luszczek, and I. Yamazaki, "On algorithmic variants of parallel Gaussian elimination: Comparison of implementations in terms of performance and numerical properties," LAPACK Working Note 280, Jul. 2013.

[41] J. J. Dongarra and D. W. Walker, "Software libraries for linear algebra computations on high performance computers," *SIAM Review*, vol. 37, no. 2, pp. 151–180, 1995.

[42] J. Dongarra, M. Faverge, T. Hérault, M. Jacquelin, J. Langou, and Y. Robert, "Hierarchical QR factorization algorithms for multi-core cluster systems," *Parallel Computing*, vol. 39, no. 4-5, pp. 212–232, 2013.

[43] J. Dongarra, M. Faverge, H. Ltaief, and P. Luszczek, "Achieving numerical accuracy and high performance using recursive tile LU factorization with partial pivoting," *Concurrency and Computation: Practice and Experience*, 2013, available online.

[44] J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK benchmark: Past, present, and future," *Concurrency and Computation: Practice and Experience*, vol. 15, pp. 803–820, 2003.

[45] I. S. Duff and S. Pralet, "Strategies for scaling and pivoting for sparse symmetric indefinite problems," *SIAM J. Matrix Anal. Appl.*, vol. 27, no. 2, pp. 313–340, 2005.

[46] P.-F. Dutot, K. Rzadca, E. Saule, D. Trystram *et al.*, "Multiobjective scheduling," in *Introduction to Scheduling*. CRC Press, 2010.

[47] H. El-Rewini, H. H. Ali, and T. G. Lewis, "Task scheduling in multiprocessing systems," *Computer*, vol. 28, no. 12, pp. 27–37, 1995.

[48] I. Foster, *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2004.

[49] M. R. Garey and D. S. Johnson, *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[50] T. Gautier, X. Besseron, and L. 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*, ser. PASCO '07. New York, NY, USA: ACM, 2007, pp. 15–23.

[51] R. Giering and T. Kaminski, "Recomputations in reverse mode AD," in *Automatic Differentiation: From Simulation to Optimization*, ser. Computer and Information Science, G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, Eds. New York: Springer, 2002, ch. 33, pp. 283–291.

[52] J. R. Gilbert, T. Lengauer, and R. E. Tarjan, "The pebbling problem is complete in polynomial space," *SIAM J. Comput.*, vol. 9, no. 3, pp. 513–524, 1980.

[53] J. R. Gilbert, G. L. Miller, and S.-H. Teng, "Geometric mesh partitioning: Implementation and experiments," *SIAM Journal on Scientific Computing*, vol. 19, no. 6, pp. 2091–2110, 1998.

[54] M. B. Giles and N. A. Pierce, "An introduction to the adjoint approach to design," *Flow, Turbulence and Combustion*, vol. 65, pp. 393–415, 2000.

[55] T. Glatard, J. Montagnat, D. Lingrand, and X. Pennec, "Flexible and efficient workflow deployement of data-intensive applications on grids with MOTEUR," *Int. Journal of High Performance Computing and Applications*, 2008.

[56] R. L. Graham, "Bounds for certain multiprocessing anomalies," *Bell System Technical Journal*, vol. 45, no. 9, pp. 1563–1581, 1966.

[57] A. Griewank, "Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation," *Optimization Methods and software*, vol. 1, no. 1, pp. 35–54, 1992.

[58] A. Griewank and A. Walther, "Algorithm 799: Revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation," *ACM Transactions on Mathematical Software (TOMS)*, vol. 26, no. 1, pp. 19–45, 2000.

[59] ——, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, 2nd ed., ser. Other Titles in Applied Mathematics. Philadelphia, PA: SIAM, 2008, no. 105.

[60] L. Grigori, J. W. Demmel, and H. Xiang, "Communication avoiding Gaussian elimination," in *Proc. ACM/IEEE SC08 Conference*, 2008.

[61] ——, "CALU: a communication optimal LU factorization algorithm," *SIAM Journal on Matrix Analysis and Applications*, vol. 32, no. 4, pp. 1317–1350, 2011.

[62] M. Guo and Y. Pan, "Improving communication scheduling for array redistribution," *J. Parallel Distrib. Comput.*, vol. 65, no. 5, 2005.

[63] F. Gustavson, "High-performance linear algebra algorithms using new generalized data structures for matrices," *IBM Journal of Research and Development*, vol. 47, no. 1, pp. 31–55, Jan 2003.

[64] P. Heimbach, C. Hill, and R. Giering, "An efficient exact adjoint of the parallel MIT general circulation model, generated via automatic differentiation," *Future Generation Computer Systems*, vol. 21, no. 8, pp. 1356–1371, 2005.

[65] J. Herrmann, L. Marchal, and Y. Robert, "Model and complexity results for tree traversals on hybrid platforms," in *Euro-Par 2013 - Parallel Processing*. Springer Verlag, 2013.

[66] ——, "Memory-aware list scheduling for hybrid platforms," INRIA, Research Report 8641, Jan. 2014.

[67] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*. SIAM Press, 2002.

[68] L. Hollermann, T. S. Hsu, D. R. Lopez, and K. Vertanen, "Scheduling problems in a practical allocation model," *J. Combinatorial Optimization*, vol. 1, no. 2, pp. 129–149, 1997.

[69] J. E. Hopcroft and R. M. Karp, "An n^5/2 algorithm for maximum matchings in bipartite graphs," *SIAM Journal on computing*, vol. 2, no. 4, pp. 225–231, 1973.

[70] M. Horton, S. Tomov, and J. Dongarra, "A class of hybrid lapack algorithms for multicore and gpu architectures," in *Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on*, july 2011, pp. 150 –158.

[71] T. S. Hsu, J. C. Lee, D. R. Lopez, and W. A. Royce, "Task allocation on a network of processors," *IEEE Trans. Computers*, vol. 49, no. 12, pp. 1339–1353, 2000.

[72] F. D. Igual, E. Chan, E. S. Quintana-Ortí, G. Quintana-Ortí, R. A. van de Geijn, and F. G. V. Zee, “The FLAME approach: From dense linear algebra algorithms to high-performance multi-accelerator implementations,” *Journal of Parallel and Distributed Computing*, 2012.

[73] D. Irony and S. Toledo, “The snap-back pivoting method for symmetric banded indefinite matrices,” *SIAM journal on matrix analysis and applications*, vol. 28, no. 2, pp. 398–424, 2006.

[74] M. Jacquelin, L. Marchal, Y. Robert, and B. Ucar, “On optimal tree traversals for sparse matrix factorization,” *IPDPS'11*, 2011.

[75] A. Jameson, “Aerodynamic shape optimization using the adjoint method,” *Lectures at the Von Karman Institute, Brussels*, 2003.

[76] E. T. Kalns and L. M. Ni, “Processor mapping techniques towards efficient data redistribution,” *IEEE Trans. Parallel Distributed Systems*, vol. 6, no. 12, pp. 1234–1247, 1995.

[77] G. Karypis and V. Kumar, *MeTiS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version 4.0*, U. of Minnesota, Dpt. of Comp. Sci. and Eng., Army HPC Research Center, Minneapolis, 1998.

[78] Y.-A. Kim, “Data migration to minimize the total completion time,” *J. Algorithms*, vol. 55, no. 1, pp. 42–57, 2005.

[79] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. S. Jr., and M. E. Zosel, *The High Performance Fortran Handbook*. The MIT Press, 1994.

[80] J. W. H. Liu, “On the storage requirement in the out-of-core multifrontal method for sparse factorization,” *ACM Trans. Math. Software*, vol. 12, no. 3, pp. 249–264, 1986.

[81] ——, “An application of generalized tree pebbling to sparse matrix factorization,” *SIAM J. Algebraic Discrete Methods*, vol. 8, no. 3, 1987.

[82] G. Manimaran and C. S. R. Murthy, “An efficient dynamic scheduling algorithm for multiprocessor real-time systems,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 9, no. 3, pp. 312–319, 1998.

[83] L. Marchal, O. Sinnen, and F. Vivien, “Scheduling tree-shaped task graphs to minimize memory and makespan,” INRIA, Research report 8082, 2012, accepted for publication in IPDPS'13, 27th International Parallel and Distributed Processing Symposium.

[84] M. G. Norman and P. Thanisch, “Models of machines and computation for mapping in multicomputers,” *ACM Computing Surveys*, vol. 25, no. 3, pp. 103–117, 1993.

[85] T. N'Takpé, F. Suter, and H. Casanova, “A comparison of scheduling approaches for mixed-parallel applications on heterogeneous platforms,” in *ISPDC*, 2007, pp. 250–257.

[86] M. A. Palis, J.-C. Liou, and D. S. L. Wei, “Task clustering and scheduling for distributed memory parallel architectures,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 1, pp. 46–55, 1996.

[87] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta, “Hierarchical task-based programming with StarSs,” *International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 284–299, 2009.

[88] L. Prylli and B. Tourancheau, "Efficient block-cyclic data redistribution," in *EuroPar'96*, ser. Lectures Notes in Computer Science, vol. 1123. Springer Verlag, 1996, pp. 155–164.

[89] G. Quintana-Ortí, E. S. Quintana-Ortí, R. A. Geijn, F. G. V. Zee, and E. Chan, "Programming matrix algorithms-by-blocks for thread-level parallelism," *ACM Transactions on Mathematical Software (TOMS)*, vol. 36, no. 3, p. 14, 2009.

[90] G. Quintana-Ortí, E. S. Quintana-Ortí, R. A. van de Geijn, F. G. Van Zee, and E. Chan, "Programming matrix algorithms-by-blocks for thread-level parallelism," *ACM Trans. Math. Softw.*, vol. 36, no. 3, pp. 1–26, 2009.

[91] A. Ramakrishnan, G. Singh, H. Zhao, E. Deelman, R. Sakellariou, K. Vahi, K. Blackburn, D. Meyers, and M. Samidi, "Scheduling data-intensiveworkflows onto storage-constrained distributed resources," in *CCGRID'07*. IEEE, 2007.

[92] P. Rivera-Vega, R. Varadarajan, and S. Navathe, "Scheduling data redistribution in distributed databases," in *Proc. Sixth Int. Conf. Data Engineering*, 1990, pp. 166–173.

[93] V. Sarkar, *Partitioning and scheduling parallel programs for multiprocessors*. MIT press, 1989.

[94] A. Schrijver, *Combinatorial Optimization: Polyhedra and Efficiency*, ser. Algorithms and Combinatorics. Springer-Verlag, 2003, vol. 24.

[95] R. Sethi, "Complete register allocation problems," in *STOC'73: Proceedings of the fifth annual ACM symposium on Theory of computing*. ACM Press, 1973, pp. 182–195.

[96] R. Sethi and J. Ullman, "The generation of optimal code for arithmetic expressions," *J. ACM*, vol. 17, no. 4, pp. 715–728, 1970.

[97] B. A. Shirazi, A. R. Hurson, and K. M. Kavi, *Scheduling and load balancing in parallel and distributed systems*. IEEE CS Press, 1995.

[98] G. Smith, *Numerical Solutions of Partial Differential Equations: Finite Difference Methods*. Clarendon Press, Oxford, 1985.

[99] L. Stanisic, S. Thibault, A. Legrand, B. Videau, and J.-F. Méhaut, "Modeling and Simulation of a Dynamic Task-Based Runtime System for Heterogeneous Multi-Core Architectures," in *Euro-par - 20th International Conference on Parallel Processing*. Porto, Portugal: Springer-Verlag, Aug. 2014.

[100] M. Stonebraker, J. Duggan, L. Battle, and O. Papaemmanouil, "SciDB DBMS research at M.I.T," *IEEE Data Eng. Bull.*, vol. 36, no. 4, pp. 21–30, 2013.

[101] P. Stumm and A. Walther, "Multistage approaches for optimal offline checkpointing," *SIAM Journal on Scientific Computing*, vol. 31, no. 3, pp. 1946–1967, 2009.

[102] R. Thakur, A. Choudhary, and G. Fox, "Runtime array redistribution in HPF programs," in *Scalable High-Performance Computing Conference, 1994., Proceedings of the*, May 1994, pp. 309–316.

[103] H. Topcuoglu, S. Hariri, and M. Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel Distributed Systems*, vol. 13, no. 3, pp. 260–274, 2002.

[104] H. Topcuoglu, S. Hariri, and M.-y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, 2002.

[105] S. Venugopalan and O. Sinnen, "Optimal linear programming solutions for multiprocessor scheduling with communication delays," in *ICA3PP (1)*, 2012, pp. 129–138.

[106] D. W. Walker and S. W. Otto, "Redistribution of block-cyclic data distributions using MPI," *Concurrency: Practice and Experience*, vol. 8, no. 9, pp. 707–728, 1996.

[107] A. Walther, "Program reversal schedules for single-and multi-processor machines," Ph.D. dissertation, PhD thesis, Institute of Scientific Computing, Technical University Dresden, Germany, 1999.

[108] L. Wang, J. M. Stichnoth, and S. Chatterjee, "Runtime performance of parallel array assignment: an empirical study," in *1996 ACM/IEEE Supercomputing Conference*, 1996.

[109] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995.

[110] A. YarKhan, J. Kurzak, and J. Dongarra, *QUARK Users' Guide: QUeueing And Runtime for Kernels*, UTK ICL, 2011.

Publications

Articles in international refereed journals

[J1] Marc Baboulin, Jack Dongarra, Julien Herrmann, and Stanimire Tomov. Accelerating linear system solutions using randomization techniques. *ACM Transactions on Mathematical Software (TOMS)*, 39(2):8, 2013.

[J2] Julien Herrmann, Loris Marchal, and Yves Robert. Memory-aware tree traversals with pre-assigned tasks. *Journal of Parallel and Distributed Computing (JPDC)*, 75:53–66, 2015.

[J3] 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*, 2015.

[J4] Mathieu Faverge, Julien Herrmann, Julien Langou, Bradley Lowery, Yves Robert, and Jack Dongarra. Mixing LU and QR factorization algorithms to design high-performance dense linear algebra solvers. *Journal of Parallel and Distributed Computing (JPDC)*, 85C:32–46, 2015.

Articles in international refereed conferences

[C1] Julien Herrmann, Loris Marchal, and Yves Robert. Model and complexity results for tree traversals on hybrid platforms. In *Proc. of Euro-Par Parallel Processing*, pages 647–658. Springer, 2013.

[C2] Julien Herrmann, Loris Marchal, Yves Robert, et al. Memory-aware list scheduling for hybrid platforms. In *Workshop on Advances in Parallel and Distributed Computational Models (APDCM)*, 2014.

[C3] Mathieu Faverge, Julien Herrmann, Julien Langou, Bradley Lowery, Yves Robert, and Jack Dongarra. Designing LU-QR hybrid solvers for performance and stability. In *Proc. of IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2014.

[C4] Thomas Herault, Julien Herrmann, Loris Marchal, and Yves Robert. Determining the optimal redistribution for a given data partition. In *Proc. of IEEE International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 95–102. IEEE, 2014.

[C5] 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 *Proc. of Heterogeneity in Computing Workshop (HCW)*, 2015.

Research reports

[RR1] Julien Herrmann, Loris Marchal, and Yves Robert. Memory-aware list scheduling for hybrid platforms. Research Report RR-8461, INRIA, February 2014.

[RR2] Julien Herrmann, Loris Marchal, and Yves Robert. Tree traversals with task-memory affinities. Research Report RR-8226, INRIA, February 2013.

[RR3] Thomas Hérault, Julien Herrmann, Loris Marchal, and Yves Robert. Determining the optimal redistribution. Research Report RR-8499, INRIA, March 2014.

[RR4] Guillaume Aupy, Julien Herrmann, Paul Hovland, and Yves Robert. Optimal Multistage Algorithm for Adjoint Computation. Research Report RR-8721, INRIA, April 2014.