Scheduling task graphs on modern computing platforms

Bertrand Simon

Numéro National de Thèse : 2018LYSEN022

THÈSE de DOCTORAT DE L'UNIVERSITÉ DE LYON

opérée par

l'École Normale Supérieure de Lyon

*École Doctorale N°*512
Informatique et Mathématiques de Lyon

Spécialité : Informatique

présentée et soutenue publiquement le 04/07/2018, par :

Bertrand SIMON

Ordonnancement de graphes de tâches sur des plates-formes de calcul modernes

Scheduling task graphs on modern computing platforms

Devant le jury composé de :

Claire	HANEN	Professeure, Université Paris Nanterre	*Rapporteure*
Safia	KEDAD-SIDHOUM	Professeure, CNAM, Paris	*Rapporteure*
Sascha	HUNOLD	Assistant-professor, TU Wien (Autriche)	*Examinateur*
Uwe	SCHWIEGELSHOHN	Professor, TU Dortmund (Allemagne)	*Examinateur*
Loris	MARCHAL	Chargé de recherche, CNRS, ENS de Lyon	*Co-encadrant*
Frédéric	VIVIEN	Directeur de recherche, Inria, ENS de Lyon	*Directeur de thèse*

Acknowledgments

Je souhaite remercier en premier lieu Claire Hanen et Safia Kedad-Sidhoum pour avoir accepté d'être rapporteures de cette thèse, et donc d'avoir consacré une partie de leur temps à lire en détail ce manuscript, ainsi que pour leurs remarques judicieuses qui ont permis de l'améliorer. Je suis également reconnaissant à Uwe Schwiegelshohn et Sascha Hunold d'avoir accepté d'évaluer mes travaux en tant que président et membre du jury.

Merci à Loris et Frédéric qui m'ont encadré durant ces trois années. Merci à la fois pour la direction scientifique au quotidien, les conseils professionnels, mais aussi les moments plus détendus. Tous les doctorants n'ont pas cette chance.

J'ai beaucoup appris pendant ma thèse notamment en travaillant avec des personnes qui m'ont apporté différentes visions de la recherche et méthodes de travail. Merci à Abdou Guermouche, Michael Bender, Rob Johnson, Shikha Singh, Samuel McCauley, Oliver Sinnen et Louis-Claude Canon.

Je remercie tous les membres (actuels, anciens, et de passage) de l'équipe ROMA, dans laquelle règne une très bonne ambiance de travail, entrecoupée de pauses ludiques, gourmandes ou/et anisées. En particulier les futurs docteurs avec qui j'ai partagé un bureau durant ces trois années: Guillaume, Julien, Aurélien, Loïc et Gilles. Je remercie également tous les doctorants du LIP que j'ai cotôyés, Evelyne, Lætitia et Marie, qui m'ont considérablement facilité la vie au laboratoire ainsi que tous les membres du LIP avec qui j'ai interagi pendant ma thèse.

Impossible de ne pas mentionner ici les PhDisc, avec qui j'ai passé de nombreux week-ends d'ultimate durant ces trois ans. Déjà quatre thèses soutenues et plusieurs qui vont suivre dans les prochains mois, je compte sur les nouvelles recrues pour poursuivre la série!

Merci à Uderzo et Goscinny, sans qui les introductions de chapitres auraient manqué d'originalité.

Merci enfin à ma famille qui m'a soutenu tout au long de ma thèse et bien avant.

Introduction

Modern computing platforms are designed to solve increasingly complex scientific problems, that come from areas as various as astronomy, genomics, geophysics, or image processing. These applications can benefit from an improved performance in several ways. In the context of numerical simulations, this improvement may be used to rely on a more complex mathematical model, to study a larger system, to increase the length of the simulation, or to improve the accuracy by reducing the time steps or the mesh size. The computational power of modern platforms is increasing due to the combination of many computing units, which may be specialized in different sorts of computations: a typical computer is now composed of many classical CPUs (Central Processing Units) associated with some specialized units such as GPUs (Graphics Processing Units) or Xeon Phi. Because of the large number of computing units in a modern platform, these units are divided into *nodes* which are themselves organized into a specific hierarchy. The available memory is in turn distributed over the nodes. This distribution and the use of several layers of memories of different speeds lead to complex memory accesses, which is known as a Non-Uniform Memory Access (NUMA) architecture. Furthermore, transferring data between two nodes is highly time-consuming, and this cost depends on the connection between involved nodes. Two computers therefore differ in multiple ways: the number of processing units, the type of each processing unit, the amount of memory available, the memory access times, the way these resources are organized into a hierarchy, ... Due to the complexity and specific features of modern platforms, implementing an application in order to fully harness the capacities of a given platform is far from an easy task, and the lack of portability is furthermore highly problematic.

Several approaches have been used to abstract the specificities of the targeted platform in order to allow a programmer to write efficient and portable parallel code. One of the most natural way to exploit potential parallelism in a sequential code is to flag certain loops which can be executed concurrently, as implemented in the OpenMP API [116] released in 1997. A downside of this approach is the restriction of the parallelism possibilities, as it may introduce unnecessary synchronization points. During the same period, Cilk [35] proposed a different approach, which has then been adopted in OpenMP 3.0 in 2008. The programmer has here the possibility to call a function via a *task*, which may be executed on a different thread, and may wait for a given result before being executed. This method allows to easily implement parallel recursive algorithms, following for instance the *divide-and-conquer* mechanism. It suffers however from limitations on more complex programs. A paradigm which allows more general parallelism has been introduced quite recently and is now widely adopted [57], for instance in OpenMP 4.0 since 2013: *task graphs*, also named *workflows*. The user creates tasks, as previously, but specifies their data dependences. Any (acyclic) graph of tasks can now therefore be constructed. Typically, an application is composed of a few main operations which are called many times (e.g., a matrix-matrix product). The idea here to cope with hybrid platforms is to provide several implementations of each operation in order to allow its execution on different types of processors. A dedicated software, named a *runtime scheduler*, builds the task graph corresponding to the program and schedules the tasks by allocating each one on some resources of the targeted platform. This software decides the allocation of

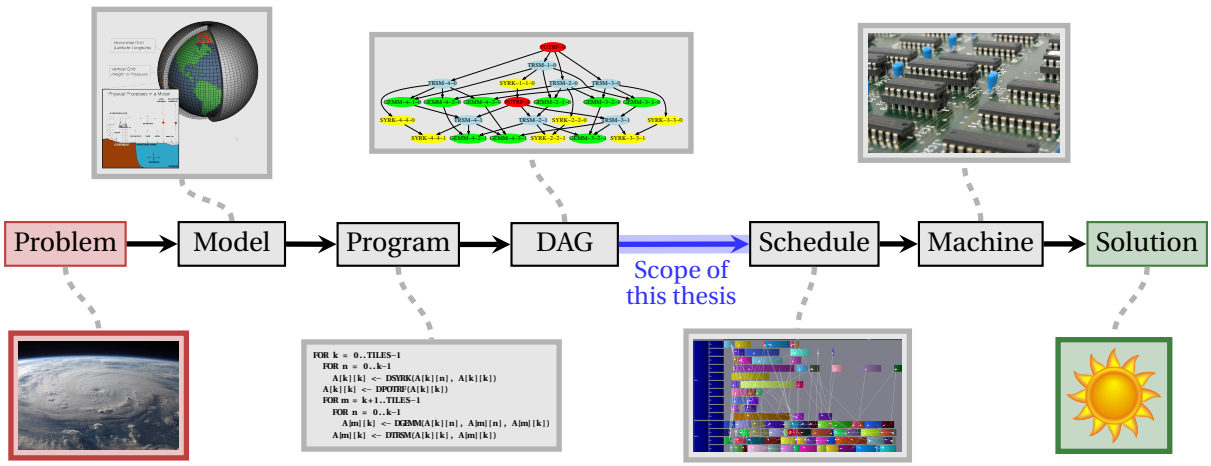


Figure 1: Several steps from a scientific problem to the computation of a solution.[1]

each task: how many computing units and which ones will be dedicated to them. The developer of a scientific application does not need to take into account the specificities of the targeted platform in his code, or even to adapt it to different platforms: everything is in the hands of the scheduler. This concept has been applied in many contexts, although often primarily motivated by linear algebra, as evidenced by the number of softwares currently developed and dedicated to task graph scheduling on parallel platforms. We can cite among them: StarPU [17] from Inria Bordeaux, France; XKAAPI [68] from Inria Grenoble, France; StarSS [119] from Barcelona Supercomputing Center, Spain; QUARK [147] and PaRSEC [36] from ICL, University of Tenessee, Knoxville, USA. The dense linear algebra library MAGMA [137] also relies on DAGs.

If we take a step back, the complete process from a given problem to the computation of a solution is summarized in Figure 1. Following the example of weather forecasting, mathematical models have been designed to describe the relevant physical interactions. The earth and the atmosphere are divided into a 3D grid. The interactions between gridboxes, following the mathematical model, are implemented into a computer program. This program is then interpreted as a DAG, or a workflow. A schedule of this DAG on the targeted machine is computed by a dedicated scheduler. Finally, the execution of this schedule leads to the result, here, a weather prediction. The objective of this thesis is to provide theoretical algorithms and results in order to help schedulers compute a satisfactory schedule of such a DAG on a given platform.

Representing a consequent workload as a graph of smaller tasks is actually not a new idea. Before its widespread usage in runtime schedulers, this concept has been used in many areas of the scheduling domain. Even before computer science scheduling studies, scientific management already included such a paradigm. In this context, the workload does not represent programs to be executed on a computer, but a project which has to be completed by human labor. In the 1950s, the Program Evaluation and Review Technique (PERT), developed for the U.S. Navy, along with the Critical Path Method, already used this technique for large project management, such as the development of the Polaris submarine. The main objective was to identify the critical tasks, i.e., tasks for which any delay will impact the final completion date, in order to prioritize them. This was a major advance in scientific project management, initiated at the end of the 19th century by the Taylorism and later the Fordism, as it allowed to represent more complex projects than traditional Gantt charts, which were introduced at the beginning of the 20th century. In the 1960s and 1970s, many task scheduling problems have been studied in the computing

[1] All images are either self-produced, under the CC0 license, or in the public domain.

context, considering for instance several machines of potentially different types, independent tasks or precedence constraints, and different objectives such as minimizing the makespan or the sum of the completion times, classified under the now widely adopted $\alpha|\beta|\gamma$ notation introduced in the survey of Graham et al. [74]. Graham's list scheduling algorithm [72] designed in late 1960s to schedule DAGs on parallel platforms is still a reference. In parallel, task trees have been used to represent arithmetic expressions, where each node is a single operation. This model has lead to algorithms ensuring for instance an efficient usage of registers, see for example [130]. Similar techniques have then been used to reduce the memory consumption of linear algebra workflows [106]. With the increased importance of heterogeneous architectures, the task graph model has been thoroughly studied, as discussed above. One of the most well-known algorithm is HEFT [139], a low-complexity heuristic designed in 2002 for minimizing the makespan on a heterogeneous platform. The improvement of the computing power of modern machines and the generalization of the task graph model bring new challenges to the scheduling community. We will explore some of them in this manuscript.

In this thesis, we consider the problem of scheduling a graph of tasks on a modern, therefore complex, platform. As explained in the beginning of this introduction, such platforms present many specificities, which cannot be all addressed in this manuscript. We therefore focus on shared-memory platforms, which may represent one node of a more complex platform. Specifically, we consider the three following challenges.

- Some applications are described as a graph of tasks where each task can be parallelized, which is known as *task parallellism*. Allocating more processors to a task generally decreases its computing time. In addition, several tasks of the graph may also be executed simultaneously (i.e., *graph parallelism*). There are therefore two conflicting types of parallelism to cope with: several tasks can be computed simultaneously, but a single task can be terminated faster using more resources. The scheduler needs to decide the number of resources allocated to each task, in addition to the traditional scheduling problem. We study this problem on a homogeneous platform composed of identical processors.

- Hybrid platforms are composed of several types of processors, for instance CPUs and GPUs. The execution time of each task is typically much shorter on one type of resource. Because of their cost, there are usually far fewer GPUs, which makes them a rare resource: many tasks can be accelerated on GPUs, but an efficient schedule may allocate many of them on CPUs. One of the main scheduling choices in this context is then to decide on which type of resource each task will be executed. We study this problem in an online setting, where the graph is gradually discovered, which increases the difficulty of the decision.

- The processing power of modern platforms increases at a faster rate than the available memory. Therefore, for more and more applications running on a shared-memory platform, the priority becomes to reduce the memory consumption in order to avoid expensive memory transfers. We study two scenarios under this perspective. First, we aim at preventing dynamic schedulers from running out of memory when possible. These schedulers generally start multiple tasks in parallel, which may lead to a memory shortage. We study how to restrict this parallelism while maintaining the performance. Then, if the data files manipulated are too large, the memory will not be sufficient. In this case, the objective is not to prevent these transfers, but to optimize their efficiency, in order to minimize their impact.

The studies conducted in this thesis focus on the theoretical side of the problems. Therefore, we don't claim that our contributions may be directly implemented in an actual runtime scheduler. The

objective is instead to influence future implementations of schedulers, by exhibiting efficient algorithms under well-defined and nevertheless realistic models. In each studied problem, we aim at grasping the main features leading to complex scheduling problems in order to propose generic solutions, which are independent of the nature of the task graph or of the specificities of the platform, provided that they fit into the adopted model.

The main contributions of each chapter are summarized below.

Chapter 1: The speedup model of Prasanna and Musicus for parallel tasks [C2]

We focus in the first two chapters of this thesis on scheduling graphs of parallel tasks, which means that several processors can be allocated to each task, and more precisely on malleable tasks, for which the allocation can change during the execution of a task. In this framework, the objective of minimizing the makespan on identical processors has been addressed in several studies, but most of them remain quite general and therefore lead to complex algorithms. We thus focus on specific applications for which we can estimate the speedup of the tasks, i.e., their acceleration factor as a function of the processing power allocated to them. We then aim at designing simple algorithms with performance guarantees that can be implemented in a runtime scheduler.

In this chapter, we target workflows corresponding to the elimination trees arising in the multifrontal factorization of sparse matrices. The studied DAGs therefore have a tree structure. Simulations show that a speedup function equal to p^α, where p is the number of allocated processors and α is a number smaller than 1, suits quite well the behavior obtained when using some factorization kernels for a reasonable number of processors. Such a speedup model has already been studied by Prasanna and Musicus, and the optimal strategy has been defined through the use of complex optimization techniques. We propose a new and simpler proof of this optimal strategy which gives more insights on the underlying concepts. We then study the same scheduling problem on two multicores, where no task can be split between the two nodes. We prove that this problem is NP-complete, we provide an approximation algorithm when the nodes are identical, and an FPTAS for independent tasks when the number of processors per node may differ.

Chapter 2: The two-threshold roofline speedup model for parallel tasks [J2]

The speedup model adopted in the previous chapter presented several limitations and appeared to be too specific. In order to overcome these weaknesses, we design and study a more accurate and general speedup model for the same problem. Based on benchmarks, we assume that the parallelization of a task is composed of three phases. When few processors are allocated to a task, the speedup is perfect. When many processors are allocated, the speedup is constant. When an intermediate number of processors is allocated, the speedup is linear but not perfect. The thresholds defining these phases, as well as the maximum speedup, depend on the task. Benchmarks on linear algebra kernels show that this model is able to fit actual tasks with a high accuracy. Minimizing the makespan of a graph of tasks under this model is unfortunately NP-complete. Two algorithms coming from the literature and not designed specifically for this model have an approximation ratio depending on tasks' speedup function. We propose a new scheduling policy that takes advantage of the knowledge of this model and has the same approximation ratio. This heuristic turns out to be more efficient than the others on synthetic graphs.

Chapter 3: Exploiting hybrid platforms in an online setting [C6]

The platform studied in the previous chapters is assumed to be composed of identical processors. As motivated earlier, modern computing platforms rely more and more on several dedicated computing units, such as GPUs or Xeon Phis, combined with many traditional CPUs. This heterogeneity in the computing resources leads to difficult scheduling choices: on which type of processor each task should be executed? Typically, a task can be accelerated if scheduled on one of the few available GPUs, but this rare resource may be used more efficiently if allocated to another task. This concern is preponderant when designing dynamic runtime schedulers. Indeed, for some applications, the workflow to schedule, represented by a DAG of tasks, is not known in advance, or is too large to allow complex computations. Therefore, when a task becomes available, the scheduler has to allocate this task with little information on the remainder of the graph. For some applications, it is nevertheless possible to precompute some information on the descendants of each task. We focus on a platform composed of two types of processors, for instance m CPUs and k GPUs, with $m \geq k$. Each task is sequential (i.e., can be executed on one processor) and has a known running time depending solely on the type of processor on which it is executed. Our contribution is twofold. On the theoretical side, we show several lower bounds on online algorithms competitivity. An algorithm aware of every available task cannot be better than $\sqrt{m/k}$ - competitive. We study the influence on this bound of the addition of flexibility on the way tasks are processed and of pre-computed information on the graph. On the algorithmic side, we improve an existing online algorithm to obtain a $(2\sqrt{m/k}+1)$ - competitive algorithm. We also design an algorithm which has a better behavior on non-pathological instances, while maintaining a competitive ratio in $O(\sqrt{m/k})$. We show on simulations that this heuristic presents performance close to an offline scheduling algorithm (which has full information on the task graph). All the results have finally been extended to multiple types of processors.

Chapter 4: Coping with a limited available memory [C7]

In the previous chapters, we did not focus on the memory consumption of the computed schedule, and therefore implicitly assumed that few data transfers occurred for the targeted applications. Indeed, when the objective is solely to minimize the makespan, typical schedules execute many tasks concurrently. If the available memory is not sufficient, this may lead to a memory shortage, and the computed schedule needs to rely on swap mechanisms or out-of-core execution. These operations have a dramatic impact on the final execution time, and need then be avoided or optimized.

We first focus on the case where it is possible to avoid out-of-core execution. The obtained schedule is then usually much more efficient. To address this problem, we consider a DAG whose execution may or may not fit into the available memory depending on the order in which the tasks are executed. An interesting result is that deciding whether there exists a schedule that does not fit into memory is polynomial. Dynamic runtime schedulers usually have efficient strategies to minimize the makespan provided that there is no memory shortage. These strategies are typically adapted from theoretical results such as the ones presented in the previous chapters. The problem is then to guide these schedulers in order to prevent them from making choices that lead to a memory shortage. This question is difficult on arbitrary DAGs because deciding whether there exists a schedule fitting in memory is NP-complete. However, the graphs considered are generally easy to schedule sequentially without exceeding the memory limit. Our solution consists in adding dependences to a DAG in order to obtain a graph for which every schedule will fit in main memory. We use the critical path of the resulting graph in order to assess the quality of the solution, i.e., the makespan that will be obtained by a scheduler. Unfortunately, even given the knowledge of a memory-efficient sequential schedule, finding such a graph with additional fictitious dependences that minimizes the critical path is NP-complete; hence, we rely on heuristics. Specifically,

we detect a cut of the graph corresponding to a set of files that may be stored simultaneously in a schedule, but does not fit into memory. Then, we design several heuristics which add a fictitious dependence in order to prevent the scheduler to reach this cut. This procedure is repeated until no such cut exists. Simulations on realistic graphs show that such a strategy allows to efficiently prevent memory shortage while preserving enough parallelism in the graph.

Chapter 5: Minimizing I/Os when processing a tree [W1]

Contrarily to the case studied in the previous chapter, some applications may require too much memory, so that it is impossible to execute them using exclusively the main memory. We target in this chapter the elimination tree workflows that arise in multifrontal factorizations. The considered DAG is therefore a tree, for which we assume that no schedule fits into the main memory. The objective is then to minimize the transfers (or I/Os) between the core memory and an infinite disk. Several related problems have already been studied. Deciding whether a graph can be computed without I/Os is polynomial for trees but NP-complete for DAGs. If the files produced by the tasks have to be either totally in core memory or totally in secondary storage, minimizing the I/Os is again NP-hard. We focus here on files which can be divisible: fractions of the files can be moved to secondary storage. This model is relevant when dealing with large files which can be split in several pages. We first study a subclass of schedules, named postorder traversals, which completely process a subtree before starting another one. This class of schedules is often used in actual solvers for data locality reasons. The optimal schedule of this class can be computed in polynomial time. We prove that its performance can be arbitrarily far from that of general schedules, but is optimal when all files have a unit size. In order to address the general problem, we propose an integer linear program and a polynomial heuristic, which appears to be close to the optimal solution in simulations. The complexity of the general problem remains however open.

Chapter 6: Data structures for external memory [C3, C5]

Nota Bene: This chapter briefly exposes the results obtained during a research visit at the Stony Brook University, NY USA, in the team of Michael Bender. The subject is therefore not related to the title of this manuscript.

In this chapter, we design several data structures aimed at performing well under the external memory model. As in the previous chapter, this model accounts for two levels of memory, which can be named as RAM and disk. Operations have to be performed on RAM. When this memory is full, a memory transfer, or I/O, is necessary in order to move contiguous elements from RAM to disk. These transfers may dictate the final running time and need then to be minimized.

In a first project, we study the complexity of computing prime number tables. Since the sieve of Eratosthenes, most studies have focused only on the number of operations and the space usage. We design data structures which dramatically reduce the required I/Os, while performing few operations. The second project focuses on history-independent data structure. This property ensures that the current state of the structure reveals no information on past operations. We first design a skip list (a history-independent and simple data structure performing the same operations as a self-balancing binary search tree) matching the optimal external memory bounds with high probability. The second data structure yields a near-optimal history-independent cache-oblivious B-tree, i.e., one of the standard search structure in external memory. This data structure maintains a dynamic set of elements in sorted order in a linear-size array. We design a history-independent version with the same complexity guarantees.

Contents

Introduction v

French summary xv

Preliminaries xxv

1 The speedup model of Prasanna and Musicus for parallel tasks 1
- 1.1 Related work . . . 3
  - 1.1.1 Models of parallel tasks . . . 3
  - 1.1.2 Results for moldable tasks . . . 4
  - 1.1.3 Results for malleable tasks . . . 4
  - 1.1.4 Series-parallel graphs . . . 4
- 1.2 Experimental evaluation of the model . . . 4
- 1.3 Application model . . . 8
- 1.4 Optimal solution for shared-memory platforms . . . 9
- 1.5 Simulations . . . 14
- 1.6 Extensions to distributed memory . . . 16
  - 1.6.1 Two homogeneous multicore nodes . . . 16
  - 1.6.2 Two heterogeneous multicore nodes . . . 25
- 1.7 Conclusion . . . 28

2 The two-threshold roofline speedup model for parallel tasks 29
- 2.1 Application model . . . 30
- 2.2 Experimental validation of the model . . . 32
- 2.3 Problem complexity . . . 34
- 2.4 Heuristics description and approximation analysis . . . 38
  - 2.4.1 Performance analysis of Proportional Mapping . . . 39
  - 2.4.2 Optimizations of Proportional Mapping . . . 40
  - 2.4.3 A novel algorithm: Greedy-Filling . . . 41
  - 2.4.4 The FlowFlex algorithm . . . 43
- 2.5 Experimental comparison . . . 44
  - 2.5.1 Datasets . . . 44
  - 2.5.2 Results . . . 45
- 2.6 Conclusion . . . 48

3 Exploiting hybrid platforms in an online setting **49**
3.1 Related work . . . 50
3.2 Lower bound on online algorithms competitiveness . . . 51
3.3 Competitive algorithms . . . 58
3.3.1 The Quick Allocation (QA) algorithm . . . 58
3.3.2 A tunable competitive algorithm which performs well in practice . . . 63
3.4 The allocation is more difficult than the schedule . . . 64
3.5 Extension to multiple types of processors . . . 66
3.6 Simulations . . . 68
3.6.1 Baseline heuristics . . . 68
3.6.2 Experimental setup . . . 69
3.6.3 Results . . . 69
3.7 Towards an offline approximation algorithm . . . 72
3.8 Conclusion . . . 75

4 Coping with a limited available memory **77**
4.1 Related work . . . 78
4.2 Problem modeling . . . 79
4.2.1 Formal description . . . 79
4.2.2 Emulation of other memory models . . . 81
4.2.3 Peak memory minimization in the proposed model . . . 84
4.3 Computing the maximal peak memory . . . 84
4.3.1 Complexity of the problem . . . 85
4.3.2 Explicit algorithm . . . 86
4.4 Lowering the maximal peak memory of a graph . . . 88
4.4.1 Complexity analysis . . . 89
4.4.2 Finding an optimal partial serialization through ILP . . . 91
4.4.3 Heuristic strategies to compute a partial serialization . . . 93
4.4.4 Computing a sequential schedule for MINLEVELS . . . 94
4.5 Simulation results . . . 95
4.6 Conclusion . . . 103

5 Minimizing I/Os when processing a tree **105**
5.1 Related work . . . 106
5.2 Problem modeling and basic results . . . 107
5.2.1 Model and notation . . . 107
5.2.2 Towards a compact solution . . . 108
5.2.3 Related algorithms . . . 109
5.3 Existing solutions are not satisfactory . . . 110
5.3.1 Computing the best postorder traversal . . . 110
5.3.2 POSTORDERMINIO is optimal on homogeneous trees . . . 111
5.3.3 Postorder traversals are not competitive . . . 117
5.3.4 OPTMINMEM is not competitive . . . 118
5.3.5 Unknown complexity . . . 119
5.4 ILP formulation of the problem . . . 122
5.5 Heuristic . . . 124
5.6 Numerical results . . . 126

5.6.1 Datasets . . . 126
5.6.2 Results . . . 127
5.7 Conclusion . . . 130

6 Data structures for external memory **131**
6.1 Introduction to the computational model . . . 132
6.2 The I/O complexity of computing prime tables . . . 132
6.3 History-independent sparse tables and dictionaries . . . 134
6.3.1 External memory skip list . . . 134
6.3.2 History-independent packed-memory array . . . 135
6.4 Conclusion . . . 135

Conclusion **137**

Appendices **141**

A The I/O complexity of computing prime tables [LATIN 2016 conference] **141**

B Anti-persistence on persistent storage: history-independent sparse tables and dictionaries [PODS 2016 conference] **177**

Bibliography **193**

List of publications **203**

French summary

Les plates-formes de calcul modernes sont conçues pour résoudre des problèmes scientifiques de plus en plus complexes, qui viennent de domaines variés tels que l'astronomie, la génomique, la géophysique ou le traitement d'images. Ces applications peuvent bénéficier d'une performance accrue de plusieurs manières. Dans le contexte des simulations numériques, cette amélioration peut être utilisée pour utiliser un modèle mathématique plus complexe, étudier un système plus grand, augmenter la longueur de la simulation ou améliorer la précision en diminuant le pas de temps ou la taille de la grille. La puissance de calcul des plates-formes modernes augmente grâce à l'utilisation en parallèle de plusieurs unités de calcul, qui peuvent être spécialisées en différents types de calcul: un ordinateur typique est maintenant composé d'un grand nombre de composants CPU (Central Processing Unit) classiques associés à plusieurs unités spécialisées tels que des GPU (Graphics Processing Unit) ou des Xeon Phi. Puisque les plates-formes modernes sont composées d'un grand nombre d'unités de calcul, ces unités sont divisées en *nœuds* qui sont eux-mêmes organisés selon une hierachie spécifique. La mémoire disponible est à son tour distribuée parmi ces nœuds. Cette distribution et l'utilisation de plusieurs niveaux de mémoire de différentes vitesses conduisent à des accès mémoires complexes, phénomène connu sous le nom d'une architecture NUMA (Non-Uniform Memory Access). En outre, transférer des données entre deux nœuds demande beaucoup de temps, et ce coût dépend de la connection entre les nœuds impliqués. Deux ordinateurs diffèrent donc dans de multiples aspects: le nombre d'unités de calcul, le type de chaque unité de calcul, la quantité de mémoire disponible, les temps d'accès mémoire, la manière dont ces ressources sont organisées en une hiérarchie, ... À cause de la complexité et des particularités des plates-formes modernes, implémenter une application de manière à exploiter au mieux les capacités d'une plate-forme donnée est loin d'être une tâche facile, et le manque de portabilité est de plus fortement problématique.

Plusieurs approches ont été menées pour abstraire les particularités de la plate-forme visée afin de permettre au programmeur d'écrire un code parallèle efficace et portable. L'un des moyens les plus naturels pour exploiter le parallélisme potentiel d'un code séquentiel consiste à identifier certaines boucles qui peuvent être exécutées en parallèle, comme implémenté dans l'interface OpenMP sortie en 1997. Un défaut de cette approche est la restriction des possibilités de parallélisation, car des points de synchronisation superflus peuvent être introduits. Durant la même période, l'interface Cilk propose une approche différente, qui a ensuite été adoptée dans OpenMP 3.0 en 2008. Le programmeur a ici la possibilité d'appeler une fonction à travers une *tâche*, qui peut être éxécutée sur un thread différent, et peut attendre le calcul de certains résultats avant d'être démarrée. Cette méthode permet d'implémenter facilement des algorithmes parallèles récursifs, qui suivent par exemple le méchanisme *diviser pour régner*. Elle souffre néanmoins de limitations pour des programmes plus complexes. Un paradigme permettant un parallélisme plus général a été développé assez récemment et est maintenant généralement adopté, par exemple dans OpenMP 4.0 depuis 2013: les *graphes de tâches*. L'utilisateur créé des tâches, comme précédemment, mais spécifie directement leurs dépendances. Tout graphe de tâche (acyclique) peut donc être construit. Typiquement, une application est composée de quelques opérations principales utilisées très souvent (e.g., un produit de matrices). L'idée pour exploiter des plates-formes hybrides

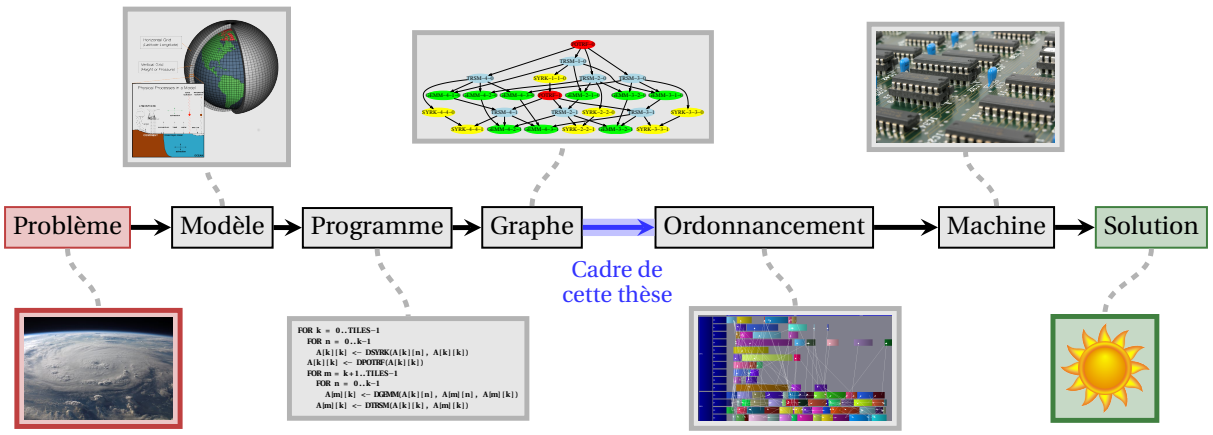


Figure 2: Plusieurs étapes d'un problème scientifique au calcul d'une solution.

consiste à fournir plusieurs implémentations de chaque opération afin de permettre leur exécution sur différents types de processeurs. Un logiciel dédié, appelé un ordonnanceur *runtime*, construit le graphe de tâches correspondant au programme et ordonnance les tâches en allouant chacune sur des ressources de la plate-forme visée. Ce logiciel décide de l'allocation de chaque tâche: combien d'unités de calcul et lesquelles sont dédiées à une tâche. Le développeur de l'application scientifique n'a donc pas besoin de prendre en compte les particularités de la plate-forme visée dans son code: tout est dans les mains de l'ordonnanceur. Ce concept a été appliqué dans de nombreux contextes, souvent motivés par l'algèbre linéaire, comme en atteste le nombre de logiciels actuellement développés et dédiés à l'ordonnancement de graphes de tâches sur des plates-formes parallèles. Parmi eux, nous pouvons citer: StarPU de Inria Bordeaux, France; XKAAPI de Inria Grenoble, France; StarSS du Barcelona Supercomputing Center, Espagne; QUARK et PaRSEC de l'ICL, University of Tenessee, Knoxville, États-Unis. La bibliothèque d'algèbre linéaire dense MAGMA repose aussi sur des graphes de tâches.

Pour prendre du recul, le processus complet d'un problème jusqu'au calcul d'une solution peut être schématiquement résumé par la figure 2. Suivant l'example de prédictions météorologiques, des modèles mathématiques ont été conçus pour décrire les interactions physiques pertinentes. La Terre et son atmosphère sont divisées en une grille tridimensionnelle. Les interactions entre les cases de la grille, suivant le modèle mathématique, sont implémentées dans un programme informatique. Ce programme est ensuite interprété comme un graphe de tâches. Un ordonnancement de ce graphe sur la machine visée est calculé par un ordonnanceur dédié. Finalement, l'exécution de cet ordonnancement mène au résultat, ici, une prédiction météorologique. L'objectif de cette thèse est de fournir des algorithmes et résultats théoriques afin d'aider les ordonnanceurs à calculer un ordonnancement satisfaisant pour un graphe de tâche sur une plate-forme donnée.

Représenter une charge de travail conséquente comme un graphe de tâches plus petites n'est en fait pas une idée nouvelle. Avant son usage généralisé dans les ordonnanceurs, ce concept a été utilisé dans plusieurs domaines de l'ordonnancement. Même avant l'étude d'ordonnancements informatiques, la gestion scientifique du travail se servait déjà d'un paradigme similaire. Dans ce contexte, la charge de travail ne représente pas des programmes à exécuter sur un ordinateur, mais un projet qui doit être traité par une main d'œuvre humaine. Dans les années cinquante, le programme PERT (Programme Evaluation and Review Technique) développé pour la marine américaine, ainsi que la méthode du chemin critique (Critical Path Method), utilisaient déjà cette technique pour identifier les tâches critiques, i.e., les tâches pour lesquelles tout délai retarde la date de finition du projet, afin de les traiter en priorité. Ce fût une avancée majeure dans la gestion scientifique de projets, initiée à la fin du XIX^{e}

siècle par le Taylorisme puis le Fordisme, car des projets plus complexes pouvaient être étudiés, contrairement aux traditionnels diagrammes de Gantt, introduits au début du XX^{e} siècle. Dans les années soixante et soixante-dix, beaucoup de problèmes d'ordonnancements ont été étudiés en informatique, considérant par exemple plusieurs machines de différents types, des tâches indépendantes ou des contraintes de précédence, et différents objectifs comme la minimization du temps d'exécution ou la somme des temps de terminaison, classifiés selon la notation de Graham $\alpha|\beta|\gamma$ désormais largement adoptée. L'algorithme d'ordonnancement de liste de Graham, conçu à la fin des années soixante pour ordonnancer des graphes sur des plates-formes parallèles est encore une référence. En parallèle, les arbres de tâches ont été utilisés pour représenter des expressions arithmétiques, où chaque nœud représente une unique opération. Ce modèle a mené à des algorithmes assurant par exemple une utilisation efficace des registres. Des techniques similaires ont ensuite été utilisées pour réduire la consommation mémoire d'applications d'algèbre linéaire. Avec l'importance accrue des architectures hétérogènes, le modèle de graphe de tâches a été profondément étudié, comme exposé plus haut. L'un des algorithmes les plus connus est HEFT (Heterogeneous Earliest Finish Time), une heuristique de faible complexité conçue en 2002 pour minimiser le temps d'exécution d'un graphe sur une plate-forme hétérogène. L'amélioration de la puissance de calcul des plates-formes modernes et la généralisation du modèle de graphes de tâches apporte de nouveaux défis en ordonnancement. Nous allons explorer quelques-uns d'entre eux dans ce manuscrit.

Dans cette thèse, nous considèrons le problème d'ordonnancement d'un graphe de tâches sur une plate-forme moderne, donc complexe. Comme expliqué au début de cette introduction, de telles plates-formes présentent de nombreuses particularités, qui ne peuvent être toutes prises en compte dans ce manuscrit. On se concentre donc sur des plates-formes à mémoire partagée, qui peuvent représenter un nœud d'une plate-forme complexe. Plus précisément, nous considèront les trois défis suivants.

- Certaines applications sont décrites par des graphes de tâches où chaque tâche peut être parallélisée, ce qui est appelé *parallélisme de tâche*. Allouer plus de processeurs sur une tâche permet généralement de diminuer son temps d'exécution. En outre, plusieurs tâches du graphe peuvent être exécutées simultanément (*parallélisme de graphe*). Il y a donc deux types de parallélisme contradictoires à exploiter: plusieurs tâches peuvent être exécutées simultanément, mais chaque tâche peut être terminée plus rapidement en utilisant plus de ressources. L'ordonnanceur doit donc décider du nombre de ressources allouées à chaque tâche, en plus du problème traditionnel d'ordonnancement. Nous étudions ce problème sur une plate-forme homogène composée de processeurs identiques.

- Les plates-formes hybrides comportent plusieurs types de processeurs, par exemple des CPU et des GPU. Le temps d'exécution de chaque tâche est typiquement plus court sur une des ressources. À cause de leur côut, les GPU sont généralement moins nombreux, et constituent donc une ressource rare: beaucoup de tâches pourraient être accélérées sur GPU, mais un ordonnancement efficace doit généralement en allouer la plupart sur les CPU. Une des principales décisions dans ce contexte consiste à choisir le type de ressource sur lequel chaque tâche sera exécutée. Nous étudions ce problème dans un contexte *online* (ou *en ligne*), le graphe étant progressivement découvert, ce qui augmente la difficulté de la décision.

- La puissance de calcul des plates-formes modernes augmente plus rapidement que la mémoire disponible. En conséquence, pour de plus en plus d'applications exécutées sur une plate-forme à mémoire partagée, la priorité devient de réduire la consommation mémoire afin d'éviter de coûteux transferts de mémoire. Nous étudions deux scenarios sous cette perspective. Tout d'abord,

nous prenons pour objectif d'empêcher un ordonnanceur dynamique d'être à court de mémoire. Ces ordonnanceurs démarrent généralement beaucoup de tâches en parallèle, ce qui peut entraîner une pénurie de mémoire. Nous étudions comment restreindre le parallélisme tout en maintenant un bon niveau de performance. Ensuite, si les fichiers de données manipulés sont trop volumineux, la mémoire disponible ne sera pas suffisante. Dans ce cas, l'objectif n'est plus d'éviter ces transferts, mais d'optimiser leur efficacité, afin de minimiser leur effet.

Les travaux conduits dans cette thèse se concentrent sur l'aspect théorique des problèmes. Par conséquent, nous n'affirmons pas que nos contributions peuvent être implémentées directement dans les ordonanceurs actuellement développés. L'objectif est plutôt d'influencer les futures implémentations de ces ordonnanceurs, en fournissant des algorithmes efficaces sous un modèle bien défini mais néanmoins réaliste. Dans chaque étude, nous visons à saisir les principaux aspects qui mènent à des problèmes complexes d'ordonnancement afin de proposer des solutions génériques, qui sont indépendantes de la nature du graphe de tâche ou des particularités de la plate-forme visée, à condition que le modèle adopté soit adequat.

Les contributions principales de chaque chapitre sont résumées ci-dessous.

Chapitre 1: Le modèle d'accélération de Prasanna et Musicus pour les tâches parallèles

Nous nous intéressons dans les deux premiers chapitres de cette thèse à l'ordonnancement de tâches parallèles, ce qui veut dire que plusieurs processeurs peuvent être alloués à chaque tâche, et plus précisément aux tâches malléables, pour lesquelles l'allocation peut varier au cours de l'exécution de la tâche. Dans ce cadre, l'objectif de minimiser le temps d'exécution sur une plate-forme composée de processeurs identiques a été étudié dans de nombreux travaux, mais la plupart s'intéresse à un cadre général et mène à des algorithmes complexes. Nous nous concentrons sur des applications spécifiques pour lesquelles il est possible d'estimer l'accélération des tâches, i.e., le rapport entre le temps d'exécution séquentiel et le temps d'exécution sur plusieurs processeurs en fonction du nombre de processeurs alloués à la tâche. L'objectif est de concevoir des algorithmes à la fois simples à mettre en œuvre dans un ordonnanceur et avec des garanties de performances.

Dans ce chapitre, nous nous intéressons aux applications correspondant aux arbres d'élimination qui interviennent lors de la factorisation multifrontale de matrices creuses. Les graphes étudiés ont donc une structure d'arbre. Des simulations montrent qu'une accélération égale à p^α, où p est le nombre de processeurs alloués et α est un nombre entre 0 et 1, correspond plutôt bien au comportement obtenu en utilisant certains noyaux de factorisation pour un nombre de processeurs raisonnable. Un tel modèle d'accélération a déjà été étudié par Prasanna et Musicus, et la stratégie optimale a été définie à travers l'utilisation de techniques complexes d'optimisation. Nous proposons une nouvelle preuve plus simple de cette stratégie optimale qui donne plus d'intuition sur les concepts sous-jacents. Nous généralisons ensuite le problème à deux nœuds multi-cœurs, où une tâche ne peut être divisée sur les deux nœuds. Nous montrons que ce problème est NP-complet, fournissons une approximation lorsque les nœuds sont identiques et une FPTAS pour des tâches indépendantes quand le nombre de cœurs par nœud diffère.

Chapitre 2: Le modèle d'accélération à deux seuils et un plateau pour les tâches parallèles

Le modèle d'accélération adopté dans le chapitre précédent s'est avéré comporter plusieurs limitations et être trop spécifique. Afin de pallier ces faiblesses, nous concevons dans ce chapitre un modèle d'accélération plus précis et général pour le même problème. En se basant sur des mesures réélles,

nous considérons que l'accélération d'une tâche se compose de trois phases. Lorsque peu de processeurs sont alloués à la tâche, l'accélération est parfaite. Lorsque beaucoup de processeurs sont alloués, l'accélération demeure constante. Lorsqu'un nombre intermédiaire de processeurs est alloué, l'accélération est linéaire mais imparfaite. Les seuils définissant ces phases, ainsi que la valeur de l'accélération maximale, dépendent de la tâche. Les mesures de temps d'exécution de noyaux d'algèbre linéaire montrent que ce modèle permet de modéliser des tâches réelles avec une grande précision. Minimiser le temps d'exécution d'un graphe dans ce contexte est malheureusement NP-dur. Deux algorithmes venant de la littérature et non conçus spécialement pour ce modèle présentent un facteur d'approximation dépendant de l'accélération des tâches. Nous proposons une nouvelle politique d'ordonnancement qui utilise la connaissance de ce modèle et possède le même facteur d'approximation. Cette heuristique se révèle être plus efficace que les autres sur des graphes synthétiques.

Chapitre 3: Utilisation efficace de plates-formes hétérogènes dans un contexte en ligne

La plate-forme étudiée dans les précédents chapitres est composée de processeurs identiques. Comme motivé plus haut, les plates-formes modernes comportent de plus en plus d'unités de calcul dédiées, comme des GPU ou des Xeon Phi, combinées avec un grand nombre de CPU traditionnels. Cette hétérogénéité dans les ressoures de calcul mène à des choix d'ordonnancement difficiles: sur quel type de processeur chaque tâche doit-elle être exécutée ? Typiquement, une tâche peut être accélérée en étant exécutée sur l'un des GPU disponibles, mais cette ressource rare pourrait être utilisée plus efficacement sur une autre tâche. Cette question est prépondérante dans la conception d'ordonnanceurs dynamiques. En effet, pour certaines applications, le graphe de tâches à ordonnancer n'est pas connu à l'avance, ou est trop grand pour permettre des calculs complexes. Ainsi, lorsqu'une tâche devient disponible, l'ordonnanceur doit décider de son allocation avec peu d'informations sur le reste du graphe. Il est parfois néanmoins possible de précaculer certaines informations sur les descendants de chaque tâche. Nous nous concentrons sur une plate-forme composée de deux types de processeurs, par exemple m CPU et k GPU, avec $m \geq k$. Chaque tâche est séquentielle (i.e., ne peut être exécutée que sur un seul processeur) et son temps d'exécution sur chaque type de ressource est connu. Notre contribution est double. Du côté théorique, nous montrons plusieurs bornes inférieures sur la compétitivité des algorithmes en ligne. Un algorithme connaissant seulement les tâches disponibles ne peut être $\sqrt{m/k}$ - compétitif. Nous étudions l'effet sur cette borne de l'ajout de flexibilité sur la manière dont les tâches sont ordonnancées et d'informations pré-calculées sur le graphe. Du côté algorithmique, nous améliorons un algorithme en ligne existant pour obtenir un algorithme $(2\sqrt{m/k}+1)$ - compétitif. Nous avons également conçu un algorithme qui se comporte mieux sur les instances non pathologiques, tout en maintenant un facteur de compétitivité en $O(\sqrt{m/k})$. Nous montrons via des simulations que cette heuristique présente des performances proches d'un algorithme hors ligne (qui connaît le graphe entier). Tous les résultats ont finalement été étendus à plusieurs types de processeurs.

Chapitre 4: Ordonnancement sous une mémoire limitée

Dans les chapitres précédents, nous n'avons pas pris en compte la consommation mémoire des ordonnancements calculés, et avons donc implicitement supposé que peu de transferts de données étaient nécessaires pour les applications visées. En effet, lorsque l'objectif est uniquement de minimiser le temps d'exécution, les ordonnancements typiques exécutent un grand nombre de tâches simultanément. Si la mémoire disponible n'est pas suffisante, cela peut mener à une pénurie de mémoire, et l'ordonnancement calculé doit donc se résoudre à des échanges de mémoire ou une exécution hors-cœur. Ces opérations ont une grande influence sur le temps d'exécution final, et doivent donc être évitées ou optimisées.

Nous nous concentrons d'abord sur le cas où il est possible d'éviter des transferts de mémoire. L'ordonnancement obtenu est ainsi généralement largement plus efficace. Pour aborder ce problème, nous considèrons un graphe dont l'exécution peut tenir dans la mémoire disponible ou non en fonction de l'ordre dans lequel les tâches sont exécutées. Un résultat intéressant établit que décider s'il existe un ordonnancement qui ne tient pas en mémoire est polynomial. Les ordonnanceurs dynamiques utilisent généralement des stratégies efficaces pour minimiser le temps d'exécution pourvu qu'il n'y ait pas de pénurie de mémoire. Ces stratégies sont typiquement adaptées de résultats théoriques comme ceux développés dans les chapitres précédents. Le problème est alors de guider ces ordonnanceurs afin de les empêcher de faire des choix qui mèneraient à une pénurie de mémoire. Cette question est difficile sur des graphes arbitraires car décider s'il existe un ordonnancement tenant en mémoire est NP-dur. Cependant, les graphes considérés sont généralement faciles à ordonnancer séquentiellement sans dépasser la mémoire disponible. Notre solution consiste à ajouter des dépendances à un graphe afin d'obtenir un graphe pour lequel chaque ordonnancement tient en mémoire. Nous utilisons le chemin critique du graphe résultant pour estimer la qualité d'une solution, i.e., le temps d'exécution qui sera obtenu par un ordonnanceur. Malheureusement, même avec la connaissance d'un ordonnancement séquentiel efficace en mémoire, calculer un tel graphe avec des dépendances factices additionnelles qui minimise le chemin critique est NP-dur. Nous nous reposons donc sur des heuristiques. Précisément, nous détectons une coupe dans le graphe correspondant à un ensemble de fichiers qui peuvent être enregistrés simultanément dans un ordonnancement, mais qui ne tient pas en mémoire. Ensuite, nous concevons plusieurs heuristiques qui ajoutent une dépendance factice afin d'empêcher l'ordonnanceur d'atteindre cette coupe. Cette procédure est répétée jusqu'à ce qu'aucune telle coupe n'existe. Les simulations sur des graphes réalistes montrent qu'une telle stratégie permet d'éviter efficacement une pénurie de mémoire tout en préservant assez de parallélisme dans le graphe.

Chapitre 5: Minimisation des transferts mémoire lors de l'ordonnancement d'un arbre

Contrairement au cas étudié dans le chapitre précédent, certaines applications peuvent demander trop de mémoire, de telle sorte qu'il est impossible de les exécuter en utilisant seulement la mémoire principale. Nous nous intéressons dans ce chapitre aux arbres d'élimination apparaissant lors des factorisations multifrontales de matrices creuses. Le graphe considéré est donc un arbre, pour lequel nous supposons qu'aucun ordonnancement ne tient en mémoire. L'objectif est alors de minimiser les transferts (ou E/S) entre la mémoire principale et un disque infini. Plusieurs problèmes reliés ont déjà été étudié. Décider si un graphe peut être exécuté sans E/S est polynomial pour un arbre mais NP-dur pour un graphe général. Si les fichiers produits par les tâches ne peuvent être partagés entre la mémoire principale et le disque, minimiser les E/S est NP-dur. Nous nous intéressons ici à des fichiers qui peuvent être divisés: il est possible de déplacer une partie de ces fichiers sur le disque. Ce modèle est pertinent lorsque les fichiers sont de taille conséquente et peuvent être découpés en plusieurs pages mémoire. Nous étudions d'abord une sous-classe d'ordonnancement, appelée *postordre*, qui exécute totalement chaque sous-arbre avant de démarrer un autre sous-arbre. Cette classe d'ordonnancements est souvent utilisée dans les ordonnanceurs pour des raisons de localité mémoire. L'ordonnancement optimal de cette classe peut être calculé efficacement. Nous montrons qu'il peut mener à un nombre d'E/S arbitrairement plus grand qu'un ordonnancement non postordre, mais qu'il est optimal si les fichiers sont de taille unitaire. Afin d'aborder le problème général, nous proposons un programme linéaire en nombres entiers et une heuristique polynomiale, qui se révèle proche de la solution optimale dans les simulations. La complexité du problème général demeure cependant ouverte.

Chapitre 6: Structures de données pour la mémoire externe

Nota Bene: Ce chapitre expose brièvement les résultats obtenus durant une visite de recherche à l'université de Stony Brook. Le sujet n'est donc pas relié au titre de ce manuscrit.

Dans ce chapitre, nous concevons plusieurs structures de données conçues pour avoir une bonne performance dans le modèle de mémoire externe. Comme dans le chapitre précédent, ce modèle prend en compte deux niveaux de mémoires, qui peuvent être appelées RAM et disque. Les opérations doivent être réalisées sur la RAM. Lorsque cette mémoire est pleine, un transfert de mémoire, ou E/S, est nécessaire afin de déplacer un ensemble d'éléments contigus de la RAM vers le disque. Ces transferts ont une grande influence sur le temps d'exécution total et doivent donc être minimisés.

Dans un premier projet, nous étudions la complexité de calculer un tableau de nombres premiers. Depuis le crible d'Eratosthène, la plupart des études se sont concentrées seulement sur la minimisation du nombre d'opérations et de l'espace mémoire nécessaire. Nous concevons des structures de données qui réduisent grandement le nombre d'E/S requis, tout en nécessitant peu d'opérations. Le second projet se concentre sur les structures de données indépendantes de l'historique. Cette propriété assure que l'état courant de la structure de données ne révèle aucune information sur les opérations passées. Nous concevons tout d'abord une liste à enjambement, ou *skip list* (une structure de données simple et indépendante de l'historique réalisant les même opérations qu'un arbre binaire de recherche auto-équilibré) atteignant les complexités optimales en mémoire externe avec forte probabilité. La deuxième structure de données mène à un arbre-B indépendant du cache (*cache-oblivious*) et de l'historique, presque optimal, i.e., une des structures de recherches classiques en mémoire externe. Cette structure de données maintient un ensemble dynamique d'éléments trié dans un tableau de taille linéaire. Nous concevons une version indépendante de l'historique avec les mêmes garanties de complexité.

Conclusion

Perspectives à court terme

Poursuite du travail théorique

Au cours de cette thèse, plusieurs questions théoriques restant à adresser ont été identifiées. Exposées dans les chapitres concernés, la plupart consiste à améliorer le rapport d'approximation d'algorithmes existants, déterminer l'existence d'algorithmes garantis lorsque seule une heuristique a été proposée, ou étendre les résultats à un contexte plus général. Deux problèmes majeurs non résolus sont détaillés ci-dessous.

- *Élaborer un algorithme hors-ligne (*offline*) pour ordonnancer un graphe de tâche sur deux types de processeurs avec un ratio d'approximation au plus 6, qui ne repose pas sur la programmation linéaire (Chapitre 3).* Comme suggéré par cet énoncé, il existe un algorithme basé sur la programmation linéaire avec un ratio d'approximation égal à 6. Un algorithme basé sur des considérations purement d'ordonnancement aurait des intérêts à la fois théoriques, car cela donne plus d'intuition sur la solution, et pratiques, car la complexité attendue est alors plus faible, donc plus proche d'un algorithme utilisable en pratique. Un tel algorithme s'inscrirait dans la continuité naturelle de la littérature existante sur les tâches indépendantes et des résulats en-ligne établis dans cette thèse.

- *Déterminer la complexité du problème de minimisation d'E/S (I/O) lors de l'ordonnancement d'un arbre, où une E/S peut concerner une fraction de fichier de données (Chapitre 5).* Ce résultat s'inscrirait dans la continuité d'une branche de l'ordonnancement théorique initiée par Liu dans

les années 1980. Ses travaux ont montré que déterminer si l'on peut ordonnancer un arbre sans E/S est polynomial, alors que ce problème est NP-dur sur un graphe général. Plus récemment, il a été prouvé que minimiser les E/S sur un arbre lorsque les fichiers ne peuvent pas être divisés entre les deux mémoires est NP-dur. La complexité de ce problème lorsque les fichiers peuvent être divisés reste donc à déterminer. Nous avons prouvé dans cette thèse que la solution optimale n'appartient pas à une classe d'ordonnancements qui, intuitivement, semblent performant.

Vers des heuristiques prêtes à être implémentées

Comme détaillé plus tôt dans l'introduction, la plupart des solutions proposées dans cette thèse n'ont pas pour vocation à être implémentées telles quelles dans des ordonnanceurs. En effet, les modèles considérés sont souvent idéaux, ce qui est nécessaire pour comprendre la complexité sous-jacente sans avoir à traiter de multiples paramètres. Par exemple, les temps de communication sont souvent négligés comme dans de nombreuses études théoriques. Ajouter ces contraintes complexifie grandement les problèmes d'ordonnancement, donc les ignorer permet de se concentrer sur les spécificités du problème étudié. En outre, les solutions proposées peuvent avoir une grande complexité, restant polynomiale, comme par exemple les heuristiques développées dans le chapitre 4.

Une manière d'utiliser le travail effectué dans cette thèse pour obtenir des algorithmes implémentables consiste à diminuer les attentes afin de réduire la complexité des algorithmes. L'algorithme proposé dans le chapitre 4 ajoute une grande quantité d'arêtes fictives pour assurer qu'aucun ordonnancement ne dépasse la limite de mémoire, ce qui est très coûteux. Une idée similaire pourrait être utilisée pour ajouter beaucoup moins d'arêtes, en ciblant seulement les points *critiques*. L'objectif pourrait être alors de limiter la consommation mémoire sans assurer qu'aucune E/S ne sera nécessaire: il peut être préférable d'autoriser quelques E/S plutôt que d'avoir un processus d'ordonnancment très lourd à exécuter. Une deuxième direction est de considérer de nouvelles contraintes primordiales, dans un contexte simplifié. Par exemple, les chapitres 1 et 2 ne prennent pas en compte les temps de communication, qui sont difficiles à modéliser correctement et empêchent des algorithmes peu coûteux d'être efficaces. Un moyen de les prendre en compte pourrait être d'améliorer l'ordonnancement *Proportional Mapping*, qui présente à la fois des performances théoriques convenables et de très bonnes propriétés de localité, en apportant quelques modifications heuristiques sur l'allocation. Un tel processus pourrait mener à un ordonnancement efficace à la fois théoriquement et en volume de communications.

Perspectives à long terme

Les travaux conduits dans cette thèse concernent une plate-forme à mémoire partagée. Une extension naturelle consiste alors à considérer des plates-formes distribuées. Dans un tel contexte, les unités de calcul sont regroupées en plusieurs *nœuds* qui sont eux-mêmes organisés selon une structure hiérarchique. Les processeurs d'un même nœud partagent typiquement une mémoire commune, donc les modèles étudiés dans ce manuscript peuvent être appliqués à l'intérieur d'un nœud. Transférer des données entre deux nœuds demande beaucoup de temps, donc ces communications sont idéalement effectuées seulement si elles sont nécessaires. Par conséquent, l'un des problèmes d'ordonnancement primordiaux sur une architecture distribuée consiste à décider sur quel nœud chaque tâche sera exécutée. De manière analogue aux travaux présentés dans cette thèse, l'étude de problèmes d'ordonnancement sur une architecture distribuée peut être décomposée en plusieurs aspects.

L'un des objectifs serait de minimiser le temps de complétion d'un graphe, sans se préoccuper de considérations relatives à la mémoire, comme étudié dans les chapitres 1 à 3. Plusieurs défis doivent être relevés dans ce contexte. L'un des principaux problèmes consiste à décider d'une allocation statique des

tâches sur les nœuds. À cause des longs temps de transfert, une solution pourrait être de diviser le graphe en utilisant des algorithmes de partitionnement (*clustering*), puis en allouant chaque groupe à un nœud. Ces algorithmes peuvent bénéficier d'un nouveau paradigme actuellement développé dans le logiciel StarPU: les tâches *hiérarchiques*, aussi appelées *bulles* ou tâches *à gros grain*. Chaque bulle représente un sous-graphe, qui est seulement révélé à l'ordonnanceur lorque cette bulle est exécutée. Ce concept, qui généralise le modèle de tâches parallèles étudié dans les chapitres 1 et 2, permet de bénéficier à la fois des avantages de l'ordonnancement à gros grain et à grain moyen. En effet, le graphe initial a une taille raisonnable, mais l'ordonnancement peut être adapté plus finement à la plate-forme une fois le sous-graphe de chaque bulle découvert. Les temps d'exécution d'algorithmes de partitionnement sur ces graphes *à bulles* ne devraient pas être prohibitifs, et chaque bulle sera donc allouée à un nœud. Comme dans cette thèse, des algorithmes permettant de traiter des tâches parallèles et des unités de calcul hétérogènes devront être développés. Afin de pallier aux erreurs de prédiction sur les temps d'exécution et de communication, il sera peut-être nécessaire d'adapter l'allocation dynamiquement.

L'utilisation mémoire des solutions proposées devra aussi être optimisée. Pour les plates-formes à mémoire partagée, nous avons proposé dans le chapitre 4 une solution pour empêcher les ordonnanceurs dynamiques d'être à court de mémoire, tout en maintenant assez de parallélisme pour éviter d'avoir des processeurs non utilisés. Une telle méthode sera difficile à appliquer dans un contexte distribué, car la mémoire elle-même est distribuée: si les calculs effectués sur un nœud demandent trop de mémoire, une partie de la charge de travail devrait être migrée vers un autre nœud, ce qui ne peut être forcé par des modifications sur le graphe d'entrée comme en mémoire partagée. Il est donc nécessaire de développer une solution pour adapter les allocations calculées dans le contexte précédent à la mémoire distribuée disponible.

Preliminaries

In this part, we introduce the model which will be used throughout this thesis, as well as some notations common to multiple chapters, summarized in Table 1.

We first define what is a task graph. A directed acyclic graph of tasks will be denoted by $G = (V, E)$, where V is the set of vertices (also called tasks or nodes), and E is the set of edges (also called dependences). We note $n = |V|$. The tasks of the graph will usually be named $T_i \in V$, for $i \in \{1, \dots, n\}$, except in Chapters 4 and 5 where we will simply use $i \in V$. If $(T_i, T_j) \in E$, we say that T_j is a ***successor*** of T_i, and T_i is a ***predecessor*** of T_j. A task may have different characteristics depending on the scope of the chapter. Nevertheless, w_i always represents the time needed to execute task T_i on one processor, which may be called the ***weight*** or the ***length*** of T_i.

In addition, a task may be executed on several processors, which decreases its computing time. The terminology of Drozdowski [57, chapter 25] distinguishes four types of tasks: *sequential* (not amenable to parallel processing), *rigid* (requesting a given number of processors), ***moldable*** (able to cope with any fixed number of processors) or even ***malleable*** (processed on a variable number of processors). When considering moldable and malleable tasks, one has to define how the processing time of a task depends on the number of allocated processors. This relation is given by the ***speedup*** function, noted s, and used in Chapters 1 and 2. Under a speedup defined by s, the ***processing time*** of a task T_i on p_i processors equals $w_i / s(p_i)$.

The platform on which the graph has to be scheduled contains p identical ***processors***, except in Chapter 3 where we consider two types of processors. Note that the term *processor* is used to describe a *computing unit*, on which a single task can be executed. It is preferred to *computing core* for readability and consistency with the scheduling literature. In Chapters 4 and 5, we also assume that the size of the available memory is M.

A schedule of a graph G will be denoted by $\mathcal{S}$. It contains among others the *starting times* of each task T_i, sometimes denoted $\sigma(i)$. We also use the ***completion time*** of a task T_i, which is equal to the starting time plus the processing time of task T_i, and is denoted by C_i. The makespan of a schedule $\mathcal{S}$ is defined as the maximal completion time among the tasks of G, and is therefore denoted by C_{max}.

We also introduce some common definitions. A ***path*** of a graph G is a list of tasks $[u_1, u_2, \dots, u_k]$ such that for every $i \in \{1, \dots, k\}$, we have $(u_i, u_{i+1}) \in E$. We then say that u_1 is an ***ancestor*** of u_k and that u_k is a ***descendant*** of u_1. The ***length*** of a path is the sum of the sequential computing times of its tasks (except in Chapter 2, where we adapt this definition to parallel tasks). A ***critical path*** of a graph G is a path of maximal length. The ***top-level*** of a task T_i is the maximal length of a path ending in T_i. Similarly, the ***bottom-level*** of a task T_i is the maximal length of a path starting at T_i. Note that in this manuscript, the computing time of T_i is counted in both its top-level and its bottom-level.

For some applications, we may have additional knowledge on the structure of the graph of tasks G. For instance, we focus in several places of this manuscript on the problem of scheduling an *elimination tree* or *assembly tree*: a workflow that arises during the factorization of sparse matrices, see Section 1.2, Chapter 1 for details. Therefore, in these cases, the graph G has an in-tree structure, also called tree: each

Graph	$G = (V, E)$	Graph of tasks, where V is the set of vertices and E the edges
	n	Number of vertices of G, i.e., $n = \|V\|$
	T_i (or i)	Task belonging to V
	w_i	Sequential processing time of task T_i
	$m_{i,j}$	Memory size of the file associated to the edge $(T_i, T_j) \in E$ (Chapter 4)
	m_i	Memory size of the file associated to the task $T_i \in V$ (Chapter 5)
	s_i or s	Speedup function of task T_i (Chapter 2) or for all tasks (Chapter 1)
Platform	t	Time ($t \in \mathbb{R}^+$)
	p or $p(t)$	Available number of processors (depends on time in Chapter 1)
	m	Number of CPUs in Chapter 3
	k	Number of GPUs in Chapter 3
	M	Maximal memory available (Chapters 4 and 5)
Schedule	$\sigma(i)$	Starting time of task T_i
	$\mathscr{S}$	Schedule solution
	C_{max}	Makespan of a schedule
	OPT	Optimal schedule or optimal makespan
	$p_i(t)$	Number of processors allocated to task T_i at time t (Chapters 1 and 2)

Table 1: Generic notations.

node has a single successor (which is then called its parent). Furthermore, we also consider slightly more general graphs, named ***series-parallel graphs*** or ***SP-graphs***. An SP-graph can be recursively defined as a single task, the series composition of two SP-graphs, or the parallel composition of two SP-graphs. A tree can easily be transformed into an SP-graph by joining the leaves according to its structure (see Figure 3).

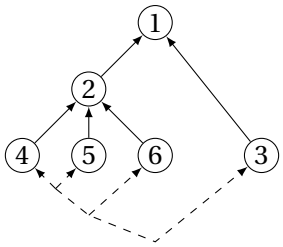


Figure 3: Example of a tree transformed into an SP-graph.

Chapter 1

The speedup model of Prasanna and Musicus for parallel tasks

« Comment puis-je faire quelque chose que j'ignore ? »

Olaf Grossebaf, *Astérix et les Normands*

As detailed in the introduction, the model of task graphs is widely used to describe the parallelism that can be exploited in an application. In this model, several tasks can be processed simultaneously as long as the given precedence constraints are respected. A challenge in expressing an application as a graph of tasks is to define the boundaries of each task. If each task contains few instructions, there will be two problems. First, the graph will be too large, and the time necessary to compute a schedule will damage the final performance. Second, the duration of each task will be short, and more time will be devoted to scheduling operations (e.g., determine the allocation, move the data) than to actual computations. On the opposite, if a task is too large, the application may lose some parallelism, as this task could have been split into smaller ones. A way to overcome these limitations is to consider parallel tasks, on which several processors can be allocated in order to reduce the computing time. In this chapter, we furthermore consider that tasks are *moldable* (can be scheduled on any number of processors) and even ***malleable*** (the number of processors allocated to a task can vary throughout the task execution). A parallel task is therefore not only defined by its sequential computing time, but also by a function, named the ***speedup***, determining the computing time given the number of processors allocated. Formally, the speedup s_i of a task T_i for a number of processors p_i is defined as the sequential computing time of T_i divided by the computing time of T_i under p_i processors. When computing a schedule minimizing the makespan, the number of processors allocated to each task has to be determined. There are therefore two conflicting types of parallelism to exploit: allocating more processors to a given task, or executing more tasks simultaneously.

The proposed algorithms to tackle this problem can be divided in two groups. The first group makes no to little assumption on the speedup function (e.g., monotonicity, concavity). The problem is then obviously more difficult, and the results obtained are either non-guaranteed heuristics or constant-factor approximation relying on complex optimization techniques, which makes them difficult to implement in a practical setting. However, for some applications, the tasks are similar (e.g., a matrix-matrix product) and therefore present comparable speedups. It is then natural to exploit this property in order to design specific low-complexity algorithms. In this chapter, we focus on task graphs coming from sparse linear algebra, and especially from the factorization of sparse matrices using the multifrontal method. Liu [104] explains that the computational dependences and requirements in Cholesky and LU factorization of

sparse matrices using the multifrontal method can be modeled as a task tree, called the *assembly tree*. We therefore focus on dependences that can be modeled as a tree.

We consider in this chapter a speedup model advocated by Prasanna and Musicus [122] for matrix operations. The objective of this study is to determine the validity and the limits of this model, as well as designing a simpler proof of the schedule minimizing the makespan on identical processors. The speedup function of each task is equal to $s_i(p_i) = p_i^\alpha$, where p_i is the number of processors allocated to the task, and $0 < \alpha \leq 1$ is a global constant. In particular, when the share of processors p_i allocated to a task T_i is constant, its processing time is given by w_i / p_i^α, where w_i is the sequential duration of T_i. The case $\alpha = 1$ represents the unrealistic case of a perfect linear speedup, and we rather concentrate on the case $\alpha < 1$ which takes into consideration the cost of the parallelization. In particular $\alpha < 1$ accounts for the cost of intra-task communications, without having to decompose the tasks in smaller granularity sub-tasks with explicit communications, which would make the scheduling problem intractable. As in [122], we also assume that it is possible to allocate non-integer shares of processors to tasks. This amounts to assume that processors can share their processing time among tasks. When task A is allocated 2.6 processors and task B 3.4 processors, one processor dedicates 60% of its time to A and 40% to B. Note that this is a realistic assumption, for example, when using modern task-based runtime systems such as StarPU [17], KAAPI [68], or PaRSEC [36]. This allows to simplify the scheduling problem and to derive optimal allocation algorithms.

The objective of such a model is not to fit exactly the actual speedup of the targeted tasks. The only way to have such a perfectly accurate model is to make no assumption on the speedup fucntion, see [85]. The objective here is to gain some insights on the tasks behavior and to understand the optimal schedule in this idealized model in order to apply the conclusions in a runtime scheduler. There is then a trade-off between the accuracy of the model and the theoretical properties it presents which can be exploited to design efficient schedules. This paragraph can be summarized by the famous quote of George E. P. Box: "All models are wrong, but some are useful".

In [121, 122], the same problem has been addressed by Prasanna and Musicus for series-parallel graphs (or SP-graphs). Such graphs are built recursively as series or parallel composition of two smaller SP-graphs. Trees can be seen as a special-case of series-parallel graphs, and thus, the optimal algorithm proposed in [121, 122] is also valid on trees. They use optimal control theory to derive general theorems for any strictly increasing speedup function. For the particular case of the speedup function p^α, Prasanna and Musicus prove some properties of the unique optimal schedule which allow to compute it efficiently. Their results are powerful (a simple optimal solution is proposed), but to obtain these results they had to transform the problem in a shape which is amenable to optimal control theory. Thus, their proofs do not provide any intuition on the underlying scheduling problem, yet it seems tractable using classic scheduling arguments.

Main contributions. In this chapter, we show that the model of malleable tasks using the p^α speedup function is justified in the context of sparse matrix factorization. We propose a new and simpler proof of the optimal schedule on series-parallel graphs, using pure scheduling arguments. We extend the previous study on distributed memory machines, where tasks cannot be split across several distributed nodes. We provide NP-completeness results and approximation algorithms. Finally, we evaluate the optimal algorithm on a set of realistic trees and estimate its improvement compared to more straightforward solutions which are unaware of the p^α speedup function.

The rest of this chapter is organized as follows. In Section 1.1, we review the related work concerning parallel task graph scheduling, including both moldable and malleable tasks. In Section 1.2, we motivate the speedup model proposed. In Section 1.4, we propose a new proof for the results of [121, 122]. In

Section 1.6, we extend the previous study on distributed memory machines. In Section 1.5, we evaluate the gain of the optimal solution via simulations.

1.1 Related work

In this section, we thoroughly review the related work on malleable task graph scheduling for models of tasks that are close or similar to our model. We also present some basic results on series-parallel graphs. Note that this survey is also relevant to Chapter 2.

1.1.1 Models of parallel tasks

The literature contains numerous models for "parallel tasks"; names and notations vary and their usage is not always consistent. The simplest model for parallel tasks is the model of *rigid* tasks, sometimes simply called *parallel tasks* [77]. A rigid task must always be executed on the same number of processors (that must be simultaneously available). In the model of *moldable* tasks, the scheduler has the freedom to chose on which number of processors to run a task, but this number cannot change during the execution. This model is sometimes called *multiprocessor tasks* [56]. The most general model is that of *malleable* tasks: the number of processors executing a task can change in any way at any time throughout the task execution. However, numerous articles use the name malleable to denote moldable tasks like, for instance, [77, 90, 101]. Depending on the variants, moldable and malleable tasks can run on any number of processors, from 1 to p, or each task T_i may have a maximum parallelism which is often denoted by δ_i [56]. Furthermore, depending on the assumptions, tasks may be preempted to be restarted later on the same set of processors, or on a potentially different one (preemption+migration). It should be noted that the model of malleable tasks is a generalization of the model of moldable tasks with preemption and migration.

An important feature of the models for moldable and malleable tasks is the task speedup functions that relate a task execution time to the number of processors it uses. Some studies, for instance [77, 85], do not make any assumption on the speedup functions. More commonly, it is assumed that the task execution time is a non-increasing function of the number of processors [63, 85, 101, 107]. Another classical assumption is that the work is a non-decreasing function [63, 85, 101] —the work is the product of the execution time and of the number of processors used— which defines the model sometimes called *monotonous penalty assumptions*. Some other works consider that the speedup function is a concave function [107]. The algorithms developed for these models, although polynomial, rely on complex optimization techniques, which makes them difficult to implement in a practical setting. Therefore, other studies have focused on specific models, for which low-complexity dedicated algorithms are designed. Several of the models considered in the literature satisfy all above assumptions: non-decreasing concave speedup function and non-decreasing work.

This is for instance the case with the model studied by Prasanna and Musicus [121] where the processing time P_i of task T_i is $P_i(k) = w_i/k^\alpha$ with α being a task-independent constant between 0 and 1 and k the number of allotted processors [121]. Another instance is the simple single-threshold model, that is, the linear model [20, 55, 114, 143, 148]: $P_i(k) = w_i/k$. Kell and Havill [94] added to that model an overhead affine in the number of processors used: $P_i(k) = (k-1)c + w_i/k$. This model is also closely related to the Amdahl's law where $P_i(k) = w_i^{(s)} + w_i^{(p)}/k$. Amdahl's law is considered in the experimental evaluation of [63].

Finally, the number of processors alloted to a task can, depending on the assumptions, either only take integer values, or can also take fractional ones [107, 121].

1.1.2 Results for moldable tasks

Du and Leung [58] have shown that the problem of scheduling moldable tasks with preemption and arbitrary speedup functions is NP-complete.

Günther et al. [77] proposed an FPTAS with no assumption on the processing times. Hunold [85] developed a heuristic for this model based on the CPA algorithm introduced in [124].

In the scope of the monotonous penalty model, Lepère, Trystram, and Woeginger [101] presented a $3+\sqrt{5} \approx 5.23606$ approximation algorithm for general DAGs, and a $\frac{3+\sqrt{5}}{2}+\epsilon \approx 2.61803+\epsilon$ approximation algorithm for series-parallel graphs and DAGs of bounded width. With the additional requirement of a concave speedup, Jansen and Zhang [91] present a 3.3-approximation. If the processing time is strictly decreasing, Chen and Chu [42] improve this factor to 2.96.

Wang and Cheng presented [143] a $3-\frac{2}{p}$-approximation algorithm to minimize the makespan while scheduling moldable task graphs with linear speedup and maximum parallelism δ_j (problem $P|prec, any, spdp\text{-}lin, \delta_j|C_{\max}$).

1.1.3 Results for malleable tasks

The problem of scheduling independent malleable tasks with linear speedups, maximum parallelism per task, and with integer allotments, that is $P|var, spdp\text{-}lin, \delta_j|C_{\max}$, can be solved in polynomial time [55, 142] using a generalization of McNaughton's wrap-around rule [111]. Drozdowski and Kubiak showed in [55] that this problem becomes NP-hard when dependences are introduced: $P|prec, var, spdp\text{-}lin, \delta_j|C_{\max}$ is NP-hard. Balmin et al. [114] present a 2-approximation algorithm for this problem. Their algorithm builds integral allotments by first scheduling the DAG on an infinite number of processors and then using the optimal algorithm for independent tasks to build an integral-allotment schedule for each interval of the previous schedule during which a constant number of processors greater than p was used.

Makarychev and Panigrahi [107] consider the problem $P|prec, var|C_{\max}$ under the monotonous penalty assumption and when allotments are rational. They provide a $(2+\epsilon)$-approximation algorithm, of unspecified complexity (their algorithm relies on the resolution of a rational linear program; this linear program is not explicitly given). Furthermore, they prove that there is no "online algorithm with sub-polynomial competitive ratio" (an online algorithm is an algorithm that considers tasks in the order of their arrival).

1.1.4 Series-parallel graphs

Series-parallel graphs can be recognized and decomposed into a tree of series and parallel combinations in linear time [141]. It is well-known that series-parallel graphs capture the structure of many real-world scientific workflows [30]. A possible way to extend algorithms designed for series-parallel graphs to general graphs is to first transform a graph into a series-parallel graph, using a process sometimes called SPization [71] before applying a specialized algorithm for SP-graphs. This was for example done in [46]. However, note that no SPization algorithm guarantees that the length of the critical path is increased by only a constant ratio.

1.2 Experimental evaluation of the model

In this section, we concentrate on evaluating the model proposed by Prasanna and Musicus in [121, 122] for our target application. The scheduling algorithm proposed by Prasanna and Musicus under

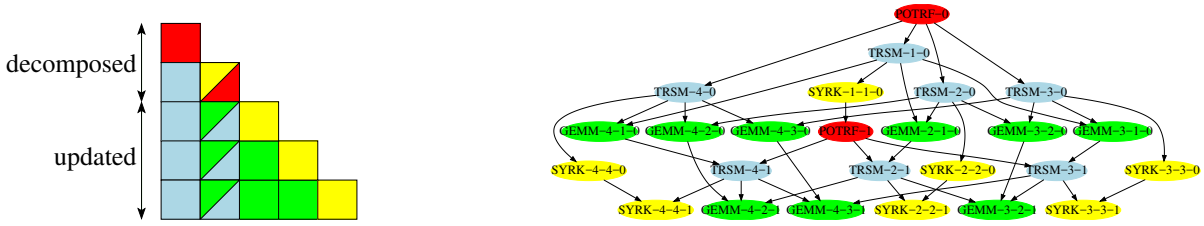
this model has already been implemented in a real multifrontal solver [23]. Due to special constraints in the task parallelization, the authors first measured a surprising super linear speedup, with $\alpha = 1.15$. Nevertheless, using the Prasanna and Musicus allocation allowed them to overtake the performance of a simple allocation proportional to the task sizes previously designed by Pothen and Sun in [120]. As in [121, 122], they assumed non-integer processor allocation, which was achieved at runtime by using time-sharing among tasks.

The model of Prasanna and Musicus states that the instantaneous speedup of a task processed on p_i processors is $s(p_i) = p_i^\alpha$. Thus, the completion time of a task T_i of size w_i which is allocated a share of processors $p_i(t)$ at time t is equal to the smallest value C_i such that

$$\int_0^{C_i} \left(p_i(t)\right)^\alpha dt \geq w_i,$$

where α is a task-independent constant. When the share of processors p_i is constant, the processing time is simplified to: w_i / p_i^α. Our goal is (i) to find whether this formula well describes the evolution of the task processing time for various shares of processors and (ii) to check that different tasks of the same application have the same α parameter. We target a modern multicore platform composed of a set of nodes each including several multicore processors. For the purpose of this study we restrict ourselves to the single node case for which the communication cost will be less dominant. In this context, $p_i(t)$ denotes the number of *cores* dedicated to task T_i at time t.

We consider applications having a tree-shaped task graph constituted of parallel tasks. This kind of execution model can be met in sparse direct solvers where the matrix is first factorized before the actual solution is computed. For instance, either the multifrontal method [59] as implemented in MUMPS [11] or qr_mumps [39], or the supernodal approach as implemented in SuperLU [103] or in PaStiX [80], are based on tree-shaped task graphs (namely the assembly tree [14]). Each task in this tree is a partial factorization of a dense sub-matrix or of a sparse panel. In order to reach good performance, these factorizations are performed using tiled linear algebra routines (BLAS): the sub-matrix is decomposed into 2D tiles (or blocks), and optimized BLAS kernels are used to perform the necessary operations on each tile. Thus, each task can be seen as a task graph of smaller granularity sub-tasks, which we call *kernels* to avoid confusion. See Figure 1.1 for an illustration.



(a) Tiled dense sub-matrix to be partially decomposed.

(b) Corresponding kernel graph.

Figure 1.1: Example of the decomposition of a task of the DAG of a Cholesky decomposition into smaller kernels.

As computing platforms evolve quickly and become more complex (e.g., because of the increasing use of accelerators such as GPUs or Xeon Phis), it becomes interesting to rely on an optimized dynamic runtime system to allocate and schedule tasks on computing resources. These runtime systems (such as StarPU [17], KAAPI [68], or PaRSEC [36]) are able to process a task on a prescribed subset of the computing cores that may evolve over time. This motivates the use of the malleable task model, where

the share of processors allocated to a task vary with time. This approach has been used and evaluated [83] in the context of the qr_mumps solver using the StarPU runtime system.

In order to assess whether tasks used within sparse direct solvers fit the model introduced by Prasanna and Musicus in [122] we conducted an experimental study on several dense linear algebra tasks. We used a test platform composed of 4 Intel E7-4870 processors having 10 cores each clocked at 2.40 GHz and having 30 MB of L3 cache for a total of 40 cores. The platform is equipped with 1 TB of memory with uniform access. We considered three dense kernels which are representative of what can be met in sparse linear algebra computations: the Cholesky and the QR factorization kernels from the Morse dense linear algebra library[1] and the standard frontal matrix factorization kernel used in the qr_mumps solver[2]. All experiments were made using the StarPU runtime.

Figures 1.2(a) to 1.2(c) present the timings obtained when computing the QR decomposition of a $M \times N$ matrix for several values of M and N, or the Cholesky factorization of a square matrix. The logarithmic scales show that the p^α speedup function models well the timings, except for small matrices when p is large. In this case, there is not enough parallelism in the task to exploit all available cores. We have performed a linear regression on the portion where $p \leq 10$ to compute the value of α for different task sizes. We performed similar experiments with a QR decomposition with $M = 1024$, and for a Cholesky factorization. The obtained values of α are gathered in Table 1.1. All these values are very close to one, which means that the parallelization is almost perfect.

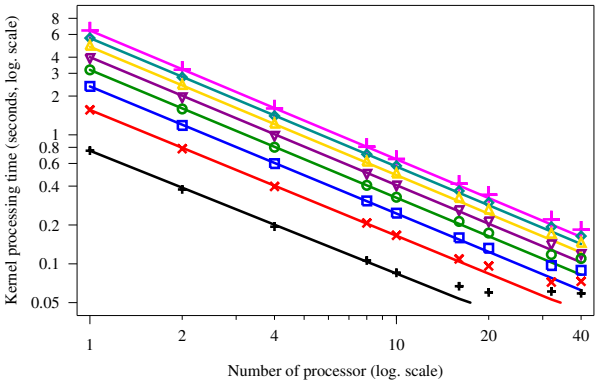
N	Value of α for QR, $M = 1024$	Value of α for QR, $M = 4096$	Value of α for Cholesky
5000	0.95	0.988	0.94
10000	0.98	0.997	0.98
15000	0.99	0.998	0.99
20000	0.99	0.999	0.99
25000	0.99	0.999	0.99
30000	0.99	0.999	1.00
35000	1.00	0.999	0.98
40000	1.00	0.999	0.98

Table 1.1: Values of α measured for dense kernels

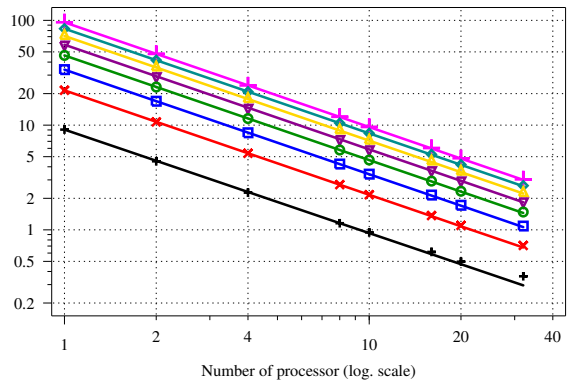
Figures 1.3(a) and 1.3(b) present the same timings for the qr_mumps frontal matrix factorization kernel, which is more relevant to this study as it is a basic block for the factorization of sparse matrices. As before, for each matrix size, the value of α was computed using linear regression on the first part of the graph ($p \leq 10$ for 1D partitioning, $p \leq 20$ for 2D partitioning). Table 1.2 gathers the results. As previously, we notice that the value of α does not vary significantly with the matrix size, which validates our model. The only notable exception is for the smallest matrix (5000x1000) with 1D partitioning: it is hard to efficiently use many cores for such small matrices (when restricting to $p \leq 4$, we compute $\alpha = 0.87$ which is very close to the other values). In all cases, for a number of processor larger than a given threshold, the performance deteriorates and stalls: using more processors is not enough to further decrease the processing time. This threshold increases with the matrix size. Our speedup model is only valid below this threshold. We claim that this is not harmful for our study, as the allocation schemes developed in the next sections allocate large numbers of processors to large tasks at the top of the tree

[1] http://icl.cs.utk.edu/projectsdev/morse/index.html

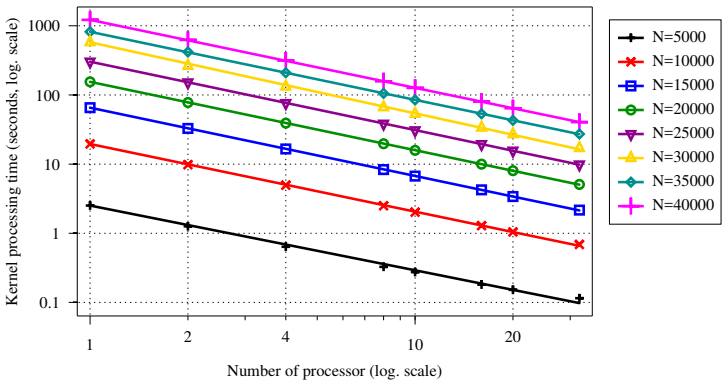
[2] Block sizes were chosen to obtain good performance: Cholesky and QR experiments use a block size of 256, qr_mumps kernel uses either block-columns of size 32 (1D partitioning) or square blocks of size 256 (2D partitioning).



(a) QR kernel for $M = 1024$.



(b) QR kernel for $M = 4096$.

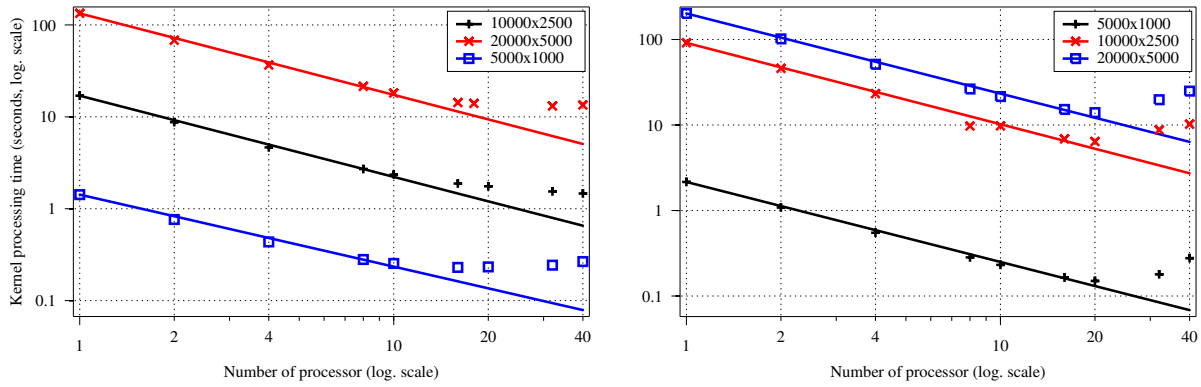


(c) Cholesky kernel ($M = N$).

Figure 1.2: Timings (points) and model (lines) of QR and Cholesky kernels.

and smaller numbers of processors for smaller tasks. Thus, our speedup model fits the timings for the range of allocations which are reasonable for each task.

The conclusions of this study should however be put into perspective. The timing points do not exactly fit the model, which is difficult to see because of the logarithmic scales here, but is obvious in the more extensive simulations conducted in Chapter 2, Section 2.2. Therefore, the model exhibits a significant behavior up to a threshold, but is not highly accurate.



(a) 1D partitioning.

(b) 2D partitioning.

Figure 1.3: Timings (points) and model (lines) of qr_mumps frontal matrix factorization kernel with two types of partitioning.

Matrix size	Value of α for 1D partitioning	Value of α for 2D partitioning
5000x1000	0.78	0.93
10000x2500	0.88	0.95
20000x5000	0.89	0.94

Table 1.2: Values of α measured for qr_mumps tasks

Finally, we notice that the value of α depends on the parameters of the problem (type of factorization, partitioning, block size, etc.). It has to be determined before the execution when considering a new kernel or new blocking parameters. The values of α obtained here are quite high thanks to the good memory performance of the considered computing platform. On other platforms, smaller values of α can be expected.

1.3 Application model

We assume that the number of available computing resources may vary with time: $p(t)$ gives the (possibly rational) total number of processors available at time t, also called the processor profile. For the sake of simplicity, we consider that $p(t)$ is a step function.

The objective is to schedule an SP-graph $G = (V, E)$ of n malleable tasks $T_1, \ldots, T_n$. The length, that is the sequential processing time, of task T_i is denoted by w_i. As motivated in the previous section, we assume that the speedup function for a task to which p processors are allocated is $s(p) = p^\alpha$, where $0 < \alpha \leq 1$ a fixed parameter. A schedule $\mathcal{S}$ is a set of nonnegative piecewise continuous functions

$\{p_i(t) \mid T_i \in V\}$ representing the time-varying share of processors allocated to each task. During a time interval Δ, the task T_i performs an amount of work equal to $\int_\Delta p_i(t)^\alpha \, dt$. Then, T_i is completed when the total work performed is equal to its length w_i. The completion time of task T_i is thus the smallest value C_i such that:

$$\int_0^{C_i} p_i(t)^\alpha dt \geq w_i.$$

We define $work_i(t)$ as the ratio of work of task T_i that is done during the time interval $[0, t]$:

$$work_i(t) = \frac{1}{w_i} \int_0^t p_i(x)^\alpha \, dx$$

A schedule is a valid solution if and only if:

1. it does not use more processors than available: $\forall t, \sum_{T_i \in V} p_i(t) \leq p(t)$);
2. it completes all the tasks: $\exists t, \ \forall T_i \in V \ work_i(t) = 1$;
3. and it respects precedence constraints: $\forall (T_i, T_j) \in E, \ \forall t,$ if $p_j(t) > 0$ then $work_i(t) = 1$.

The makespan C_{max} of a schedule is computed as $\min\{t \mid \forall i \ work_i(t) = 1\}$. Our objective is to construct a valid schedule with optimal, i.e., minimal, makespan.

Note that because of the speedup function $s(p) = p^\alpha$, the computations in the following sections will make a heavy use of the functions $s : x \mapsto x^\alpha$ and $s^{-1} : x \mapsto x^{(1/\alpha)}$. We assume that we have at our disposal a polynomial time algorithm to compute both s and s^{-1}. We are aware that this assumption is very likely to be wrong, as soon as $\alpha < 1$, since s and s^{-1} produce irrational numbers. However, without these functions, it is not even possible to compute the makespan of a schedule, and hence the problem is not in NP. Furthermore, this allows us to avoid the complexity due to number computations, and to concentrate on the most interesting combinatorial complexity, when proving NP-completeness results and providing approximation algorithms. In practice, any implementation of s and s^{-1} with a reasonably good accuracy will be sufficient to perform all the computations and, for example, compute the makespan of a schedule.

In the next section, following Prasanna and Musicus, we will not consider trees but more general graphs: *series-parallel graphs* (or SP-graphs). An SP-graph can be recursively defined as a single task, the series composition of two SP-graphs, or the parallel composition of two SP-graphs.

As already noticed in the Preliminaries section, a tree can easily be transformed into an SP-graph by joining the leaves according to its structure (see Figure 1.4). We will use $(i \parallel j)$ to represent the parallel composition of tasks T_i and T_j and $(i\,;j)$ to represent their series composition. The SP-graph of Figure 1.4 can be represented as:

$$\Big(\Big(\big((4 \parallel 5) \parallel 6\big)\,;2\Big) \parallel 3\Big)\ ;\ 1\Big).$$

Thanks to this construction, an algorithm which solves the previous scheduling problem on SP-graphs also gives an optimal solution for trees.

1.4 Optimal solution for shared-memory platforms

The purpose of this section is to give a simpler proof of the results of [121, 122] using only scheduling arguments. We consider an SP-graph to be scheduled on a shared-memory platform (each task can be distributed across the whole platform). We assume that $\alpha < 1$ and prove the uniqueness of the optimal schedule.

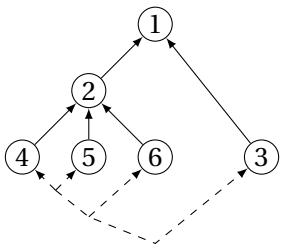


Figure 1.4: Example of a tree transformed into an SP-graph.

Our objective is to prove that any SP-graph G is *equivalent* to a single task T_G of easily computable length: for any processor profile $p(t)$, graphs G and T_G have the same makespan. We prove that the ratio of processors allocated to any task T_i, defined by $r_i(t) = p_i(t)/p(t)$, is constant from the moment at which T_i is initiated to the moment at which it is terminated. We also prove that in an optimal schedule, the two subgraphs of a parallel composition terminate at the same time and each receives a constant total ratio of processors throughout its execution. We then prove that these properties imply that the optimal schedule is unique and obeys to a *flow conservation* property: the shares of processors allocated to two subgraphs of a series composition are equal. When considering a tree, this means that the whole schedule is defined by the ratios of processors allocated to the leaves. Then, all the children of a node T_i terminate at the same time, and its ratio is the sum of its children ratios.

We first need to define the length $\mathscr{L}_G$ associated to a graph G, which will be proved to be the length of the task T_G. Then, we state a few lemmas before proving the main theorem.

Definition 1.1. *We recursively define the length $\mathscr{L}_G$ associated to an SP-graph G:*

- $\mathscr{L}_{T_i} = w_i$;
- $\mathscr{L}_{G_1;G_2} = \mathscr{L}_{G_1} + \mathscr{L}_{G_2}$;
- $\mathscr{L}_{G_1 \parallel G_2} = \left(\mathscr{L}_{G_1}^{1/\alpha} + \mathscr{L}_{G_2}^{1/\alpha}\right)^{\alpha}$.

Lemma 1.1. *An allocation minimizing the makespan uses all the processors at any time.*

Proof. The proof is established by contradiction. We assume that there exists an interval Δ throughout which some of the processors are (at least partially) idle. Without loss of generality we assume that no job completes during Δ. We distribute the unused processing power from Δ among the tasks proportionally to their allocation during Δ. The work performed during Δ is strictly increased, as the speedup function $s : p \mapsto p^{\alpha}$ is increasing, and we therefore obtain a schedule with a shorter makespan. □

We define a ***clean interval*** with regard to a schedule $\mathscr{S}$ as an interval during which no task is completed in $\mathscr{S}$.

Lemma 1.2. *When the number of available processors is constant, any optimal schedule allocates a constant number of processors per task on any clean interval.*

Proof. By contradiction, we consider an optimal schedule $\mathscr{S}$ of makespan C_{max}, and we suppose that one task j is not allocated a constant number of processors[3] on a clean interval $\Delta = [t_1, t_2]$. By definition of a clean interval, no task completes during Δ. $|\Delta| = t_2 - t_1$ denotes the duration of Δ. I denotes the set of tasks that receive a non-empty share of processors during Δ, and p the constant number of available processors. The share of processors allocated to task T_i at time $t \in \Delta$ in $\mathscr{S}$ is noted $p_i(t)$.

[3]Formally, an allocation is constant on Δ if it is equal to a given constant except, maybe, on a subset of Δ of null measure.

We want to show that there exists a valid schedule with a makespan smaller than C_{max}. To achieve this, we define an intermediate and not necessarily valid schedule $\mathscr{S}'$, which nevertheless respects the resource constraints (no more than p processors are used at time t). This schedule is equal to $\mathscr{S}$ except on Δ. The constant share of processors q_i allocated to every task T_i on Δ in $\mathscr{S}'$ is defined by:

$$q_i = \frac{1}{|\Delta|}\int_\Delta p_i(t)dt.$$

For all t, we have $\sum_{i\in I} p_i(t) = p$ because of Lemma 1.1. We get $\sum_{i\in I} q_i = p$. So $\mathscr{S}'$ respects the resource constraints. Let $W_i^\Delta(\mathscr{S})$ (resp. $W_i^\Delta(\mathscr{S}')$) denote the work done on T_i during Δ under schedule $\mathscr{S}$ (resp. $\mathscr{S}'$). We have:

$$W_i^\Delta(\mathscr{S}) = \int_\Delta p_i(t)^\alpha dt = |\Delta|\int_{[0,1]} p_i(t_1+t|\Delta|)^\alpha dt$$
$$W_i^\Delta(\mathscr{S}') = \int_\Delta \left(\frac{1}{|\Delta|}\int_\Delta p_i(t)dt\right)^\alpha dx = |\Delta|\left(\int_{[0,1]} p_i(t_1+t|\Delta|)dt\right)^\alpha.$$

As $\alpha < 1$, the function $x \mapsto x^\alpha$ is concave and then, by Jensen inequality [78], $W_i^\Delta(\mathscr{S}) \le W_i^\Delta(\mathscr{S}')$. Moreover, as $x \mapsto x^\alpha$ is *strictly* concave, this inequality is an equality if and only if the function $t \mapsto p_i(t_1+t|\Delta|)$ is equal to a constant on $[0,1[$ except on a subset of $[0,1[$ of null measure [78]. Then, by definition, p_j is not constant on Δ, and cannot be made constant by modifications on a set of null measure. We thus have $W_j^\Delta(\mathscr{S}) < W_j^\Delta(\mathscr{S}')$. Therefore, T_j is allocated too many processors under $\mathscr{S}'$. It is then possible to distribute this surplus among the other tasks during Δ, so that the work done during Δ in $\mathscr{S}$ can be terminated earlier. This remark implies that there exists a valid schedule with a makespan smaller than C_{max}; hence, the contradiction. □

We recall that $r_i(t) = p_i(t)/p(t)$ is the instantaneous ratio of processors allocated to a task T_i .

Lemma 1.3. *Let G be the parallel composition of two tasks, T_1 and T_2. If $p(t)$ is a step function, in any optimal schedule, up to the completion of G, $r_1(t)$ is constant and equal to:*

$$\pi_1 = \frac{1}{1+(w_2/w_1)^{1/\alpha}} = \frac{w_1^{1/\alpha}}{\mathscr{L}_{1\|2}^{1/\alpha}}.$$

Proof. First, we prove that $r_1(t)$ is constant on any optimal schedule. Therefore, as by Lemma 1.1 we have $r_2(t) = 1 - r_1(t)$, $r_2(t)$ will also be proved constant. This results implies in particular that both tasks terminate simultaneously as the ratios never drop to zero before the graph is completely processed.

We consider an optimal schedule $\mathscr{S}$, and two consecutive time intervals A and B such that $p(t)$ is constant and equal to p on A and q on B, and $\mathscr{S}$ does not complete before the end of B. Suppose also that $|A|p^\alpha = |B|q^\alpha$ (shorten one interval otherwise), where $|A|$ and $|B|$ are the durations of intervals A and B. By Lemma 1.2, $r_1(t)$ has constant values r_1^A on A and r_1^B on B. Suppose by contradiction that $r_1^A \neq r_1^B$.

We want to prove that $\mathscr{S}$ is not optimal, and so that we can do the same work as $\mathscr{S}$ does on $A\cup B$ in a smaller makespan. We set $r_1 = \frac{1}{2}\left(r_1^A + r_1^B\right)$. We define the schedule $\mathscr{S}'$ as equal to $\mathscr{S}$ except on $A\cup B$ where the ratio allocated to T_1 is r_1 (see Figure 1.5).

The work W_1 on task T_1 under $\mathscr{S}$ and W_1' under $\mathscr{S}'$ during $A\cup B$ are equal to:

$$W_1 = |A|p^\alpha\left(r_1^A\right)^\alpha + |B|q^\alpha\left(r_1^B\right)^\alpha$$
$$W_1' = r_1^\alpha\left(|A|p^\alpha + |B|q^\alpha\right).$$

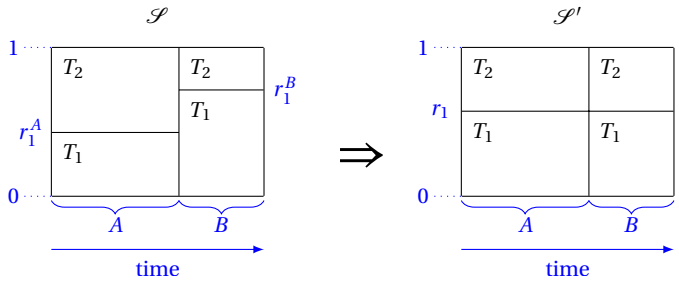


Figure 1.5: Schedules $\mathscr{S}$ and $\mathscr{S}'$ on $A \cup B$. The ordinates represent the ratio of processing power.

By the concavity of the function $s : x \mapsto x^\alpha$, we have:

$$\begin{aligned}
\frac{\left(r_1^B\right)^\alpha - (r_1)^\alpha}{r_1^B - r_1} &< \frac{(r_1)^\alpha - \left(r_1^A\right)^\alpha}{r_1 - r_1^A} \\
|B| q^\alpha \left(\left(r_1^B\right)^\alpha - (r_1)^\alpha\right) &< |A| p^\alpha \left((r_1)^\alpha - \left(r_1^A\right)^\alpha\right) \qquad \text{because } r_1^B - r_1 = r_1 - r_1^A \text{ and } |B| q^\alpha = |A| p^\alpha \\
|A| p^\alpha \left(r_1^A\right)^\alpha + |B| q^\alpha \left(r_1^B\right)^\alpha &< r_1^\alpha \left(|A| p^\alpha + |B| q^\alpha\right) \\
W_1 &< W_1'.
\end{aligned}$$

Symmetrically, we also have $W_2 < W_2'$. Therefore, $\mathscr{S}'$ performs strictly more work for both tasks during $A \cup B$ than $\mathscr{S}$. Thus, $\mathscr{S}'$ can be modified as in the proof of Lemma 1.2 to do the same work in a smaller makespan. Therefore, $\mathscr{S}$ is not optimal. So $r_1(t)$ is constant in optimal schedules.

There remains to prove that in an optimal schedule $\mathscr{S}$, $r_1(t) = \pi_1$; hence, the optimal schedule is unique. As $p(t)$ is a step function, we define the sequences (A_k) and (p_k) such that A_k is the duration of the k-th step of the function $p(t)$ and $p(t) = p_k > 0$ on A_k.[4] The sum of the durations of the A_k's is the makespan of $\mathscr{S}$. Then, as $\mathscr{S}$ completes both T_1 and T_2 with constant rates, if we note $V = \sum_k |A_k| p_k^\alpha$ and r_1 the value of $r_1(t)$, we have:

$$\begin{aligned}
w_1 &= \sum_k |A_k| r_1^\alpha p_k^\alpha = r_1^\alpha V \\
w_2 &= \sum_k |A_k| (1 - r_1)^\alpha p_k^\alpha = (1 - r_1)^\alpha V.
\end{aligned}$$

Then, $w_2 = \left(\dfrac{1 - r_1}{r_1}\right)^\alpha w_1$ and $r_1 = \dfrac{1}{1 + \left(\frac{w_2}{w_1}\right)^{1/\alpha}} = \pi_1$. □

Lemma 1.4. *Let G be the parallel composition of tasks T_1 and T_2, $p(t)$ a step function, and $\mathscr{S}$ an optimal schedule. Then, the makespan of G under $\mathscr{S}$ is equal to the makespan of the task T_G of length $\mathscr{L}_G = \mathscr{L}_{1\|2}$.*

Proof. We characterize $p(t)$ by the sequences (A_k) and (p_k) as in the proof of Lemma 1.3. Let Δ be the domain of definition of $\mathscr{S}$, so that $\Delta = [0, C_{max}]$. We define, for both tasks T_i, the function $work_i(t)$ representing the ratio of its work done during $[0, t]$:

$$work_i(t) = \frac{1}{w_i} \int_0^t p_i(x)^\alpha \, dx.$$

[4] Exceptionally, in the proofs of Lemmas 1.3 and 1.4, p_k refers to a value of $p(t)$ and not to the number of processors allocated to a task.

Hence, $work_i(0) = 0$ and $work_i(C_{max}) = 1$. For every $t \in \Delta$, let $k(t)$ be the index such that $p(t)$ is in its $k(t)$-th step at time t; hence, $p(t) = p_{k(t)}$. Formally, we have:

$$\sum_{k<k(t)} |A_k| \leq t < \sum_{k\leq k(t)} |A_k|.$$

Let $\bar{t} = t - \left(\sum_{k<k(t)} |A_k|\right)$. By Lemma 1.3, the ratio of processors allocated to T_1 is constant over Δ and equal to:

$$r_1 = \frac{w_1^{1/\alpha}}{w_1^{1/\alpha} + w_2^{1/\alpha}} = \left(\frac{w_1}{\mathscr{L}_{1\|2}}\right)^{1/\alpha}.$$

Then, we have:

$$\begin{aligned} work_1(t) &= \frac{1}{w_1}\left(\bar{t}\left(p_{k(t)}r_1\right)^\alpha + \sum_{k<k(t)} |A_k|\left(p_k r_1\right)^\alpha\right) \\ &= \frac{1}{\mathscr{L}_{1\|2}}\left(\bar{t}\,p_{k(t)}^\alpha + \sum_{k<k(t)} |A_k| p_k^\alpha\right). \end{aligned}$$

Similarly, for T_2, we have:

$$\begin{aligned} work_2(t) &= \frac{1}{w_2}\left(\bar{t}\left(p_{k(t)}(1-r_1)\right)^\alpha + \sum_{k<k(t)} |A_k|\left(p_k(1-r_1)\right)^\alpha\right) \\ &= \frac{1}{\mathscr{L}_{1\|2}}\left(\bar{t}\,p_{k(t)}^\alpha + \sum_{k<k(t)} |A_k| p_k^\alpha\right). \end{aligned}$$

We define $work(t)$ as the ratio of work that is done for the equivalent task T_G of length $\mathscr{L}_{1\|2}$ when allocated $p(t)$ processors at time t, until the task is terminated. We have:

$$work(t) = \frac{1}{\mathscr{L}_{1\|2}}\left(\bar{t}\,p_{k(t)}^\alpha + \sum_{k<k(t)} |A_k| p_k^\alpha\right) = work_1(t) = work_2(t).$$

The three ratios are identical, so they all reach 1 at time C_{max}. Then, G and T_G have the same optimal makespan under any step-function processor profile $p(t)$. □

Theorem 1.1. *For every graph G, if $p(t)$ is a step function, G has the same optimal makespan as its equivalent task T_G of length $\mathscr{L}_G$ (computed as in Definition 1.1). Moreover, there is a unique optimal schedule, and it can be computed in polynomial time.*

Proof. In this proof, we only consider optimal schedules. Therefore, when the makespan of a graph is considered, we implicitly mean its optimal makespan. We first remark that in any optimal schedule, as $p(t)$ is a step function and because of Lemma 1.2, only step functions are used to allocate processors to tasks, so Lemma 1.4 can be applied on any subgraph of G without checking that the processor profile is also a step function for this subgraph. We now prove the result by induction on the structure of G.

- G is a single task. The result is immediate.
- G is the series composition of G_1 and G_2. By induction, G_1 (resp. G_2) has the same makespan as task T_{G_1} (resp. T_{G_2}) of length $\mathscr{L}_{G_1}$ (resp. $\mathscr{L}_{G_2}$) under any processor profile. Therefore, the makespan of G is equal to $\mathscr{L}_G = \mathscr{L}_{G_1;G_2} = \mathscr{L}_{G_1} + \mathscr{L}_{G_2}$. The unique optimal schedule of G under $p(t)$ processors is the concatenation of the optimal schedules of G_1 and G_2.

- G is the parallel composition of G_1 and G_2. By induction, G_1 (resp. G_2) has the same makespan as task T_{G_1} (resp. T_{G_2}) of length $\mathscr{L}_{G_1}$ (resp. $\mathscr{L}_{G_2}$) under any processor profile. Consider an optimal schedule $\mathscr{S}$ of G and let $p_1(t)$ be the processor profile allocated to G_1. Let $\tilde{\mathscr{S}}$ be the schedule of $(T_{G_1} \parallel T_{G_2})$ that allocates $p_1(t)$ processors to T_{G_1}. $\tilde{\mathscr{S}}$ is optimal and achieves the same makespan as $\mathscr{S}$ for G because T_{G_1} and G_1 (resp. T_{G_2} and G_2) have the same makespan under any processor profile. Then, by Lemma 1.4, $\tilde{\mathscr{S}}$ (so $\mathscr{S}$) achieves the same makespan as the optimal makespan of the task T_G of length $\mathscr{L}_{G_1 \parallel G_2} = \mathscr{L}_G$. Moreover, by Lemma 1.3 applied on $(T_{G_1} \parallel T_{G_2})$, we have $p_1(t) = \pi_1 p(t)$. By induction, the unique optimal schedules of G_1 and G_2 under respectively $p_1(t)$ and $(p(t) - p_1(t))$ processors can be computed. Therefore, there is a unique optimal schedule of G under $p(t)$ processor: the parallel composition of these two schedules.

Therefore, there is a unique optimal schedule for G under $p(t)$. Moreover, it can be computed in polynomial time. We describe here the algorithm to compute the optimal schedule of a tree G, but it can be extended to treat SP-graphs. The length of the equivalent task of each subtree of G can be computed in polynomial time by a depth-first search of the tree (assuming that raising a number to the power α or $1/\alpha$ can be done in polynomial time). Hence, the values π_1 and π_2 for each parallel composition can also be computed in polynomial time. Finally, these values can be used to compute in linear time the ratios of the processor profile that should be allocated to each task after its children are completed, which describes the optimal schedule. □

1.5 Simulations

In this section, we present the results of simulations which compare the optimal allocation presented in Section 1.4 (referred to as the PM strategy, for Prasanna-Musicus) to allocations that are unaware of the speedup function p^α. Our objective is to show the potential gain in makespan obtained by taking this speedup function into account.

We compare the PM strategy to two other strategies. The first one will be referred to as the DIVISIBLE strategy. It assumes that the speedup is equal to p, which means that the parallelization of each task is perfect. Therefore, it schedules the tasks sequentially, by allocating all the processing power to one task at a time. The second one, which will be referred to as the PROPORTIONAL strategy, is known as 'proportional mapping' and has been designed in [120], as already mentioned in Section 1.2. It allocates a constant processing power to each subtree, which is proportional to the sum of the lengths of its tasks. Actually, this strategy is equal to the PM strategy when $\alpha = 1$. Both DIVISIBLE and PROPORTIONAL are optimal when $\alpha = 1$, but PROPORTIONAL is more robust to smaller values of α as it allocates smaller shares of processors to each task.

In order to compare these strategies to PM, we use a data set that contains assembly trees of a set of sparse matrices obtained from the University of Florida Sparse Matrix Collection [52]. The details concerning the computation of the data set can be found in [62]. Specifically, the trees were obtained by performing an amalgamation of elimination trees corresponding to a Cholesky factorization of 76 matrices ordered using either MeTis or amd. The data set consists in more than 600 trees each containing between 2,000 and 1,000,000 nodes with a depth ranging from 12 to 75,000. We have two models assuming that either 40 or 100 processors are available ($p(t) = 40$ or $p(t) = 100$).

A problem resides in the fact that the speedup is equal to p^α even when $p < 1$, in which case it is superlinear and so unrealistic. To avoid this issue, we modify each tree in order that each task is allocated at least one processor by the PM schedule. When we detect that a subtree of a given node u is allocated less than one processor, this subtree is processed using the whole share of processors allocated to u, right before the processing of u. This procedure mimics what is implemented in practice, where small

tasks are scheduled sequentially. Figure 1.6 presents an example of this iterative aggregation, which is described in details in the following paragraphs. Note that this process transforms the tree into an SP-graph.

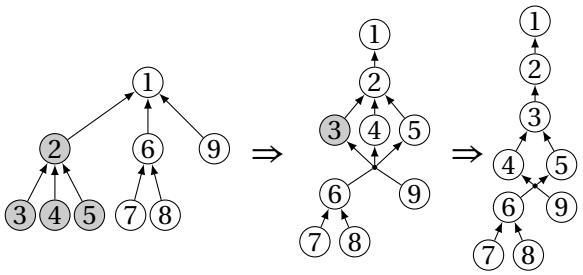


Figure 1.6: Example of the iterative aggregation of the left tree where the tasks that are allocated less than one processor in the PM schedule are shaded. The subtree rooted at task 2 is moved then modified.

More precisely, we convert each tree into an SP-graph using a recursive routine `Agreg`(G, p, α) where the parameter G is the SP-graph to modify. For each value of α, we iterate this routine on each graph until no task is allocated less than 1 processor under the PM schedule. The routine is described in the following paragraph.

On a single task, `Agreg` allocates all p processors. On a series composition, it makes recursive calls on each subgraph, with the same parameters. For a parallel composition G_0, we consider all its maximal subgraphs rooted at either a series composition or a task (this can be seen as the "true" children of G_0). `Agreg` computes the equivalent length $\mathscr{L}$ (defined in Section 1.4) of each of these subgraphs. `Agreg` then processes these subgraphs by non-decreasing equivalent length. `Agreg` computes the share of processors p_i allocated to the subgraph G_i. If $p_i < 1$, G_i is moved to be scheduled after the considered parallel composition on p processors. Then, `Agreg` transforms the subgraph G_i with parameter p into a subgraph G'_i. If $p_i \geq 1$, `Agreg` transforms the subgraph G_i with parameter p_i into a subgraph G'_i. For instance, `Agreg` launched on the parallel composition $(G_1 \parallel G_2 \parallel G_3 \parallel G_4 \parallel G_5)$ can return the graph $\left((G'_3 \parallel G'_4 \parallel G'_5); G'_2; G'_1\right)$ where G'_i is the graph returned by the corresponding call of `Agreg` on G_i. A more complex example is illustrated on Figure 1.6.

For a fixed α, $0 < \alpha \leq 1$, and an assembly tree, we compute the makespan obtained by each strategy on the tree modified by the above method. We know that PM never uses less than one processor, and computes an optimal schedule (by Theorem 1.1). Nevertheless, PROPORTIONAL builds a different allocation and may use less than one processor for some tasks. As the speedup function is not realistic in this case, we evaluate the schedule computed by PROPORTIONAL using a slightly modified and more realistic model: the speedup is equal to p^α when $p \geq 1$ and p otherwise. The criteria used for the comparison is the percentage of relative distance to PM: the percentage corresponding to the makespan overhead with respect to PM divided by the PM makespan.

We have computed this percentage for each tree in the data set and for values of α varying between 0.5 and 1. We plot in Figure 1.7 the results of the simulations for $p(t) = 40$. For both DIVISIBLE and PROPORTIONAL strategies and each value of α, the boxplot represents the first and last decile, the first and last quartile and the median. Predictably enough, the relative distance to PM decreases when α gets close to 1, as both strategies are also optimal for $\alpha = 1$. We conclude that, under these hypotheses, DIVISIBLE is not an acceptable strategy because the median relative distance approximately increases by 8% each time α decreases by 0.05, which for example gives a median relative distance of 16% for $\alpha = 0.9$. The PM strategy also offers an improvement compared to PROPORTIONAL, which is somewhat limited for large values of α: for $\alpha = 0.9$, only half of the data set results in a makespan 3% larger with

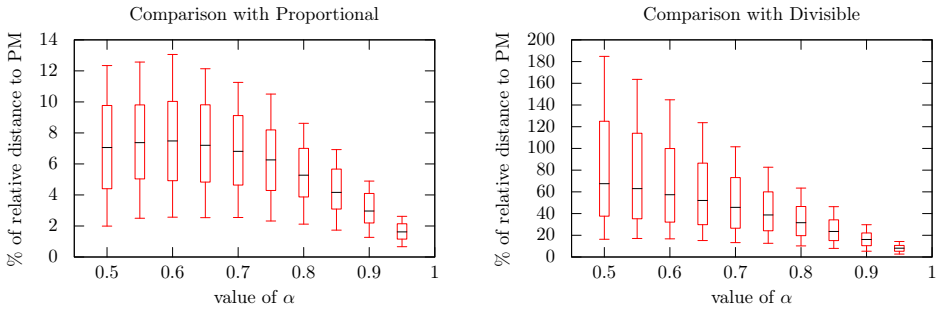


Figure 1.7: Comparison to the PM schedule with $p(t) = 40$.

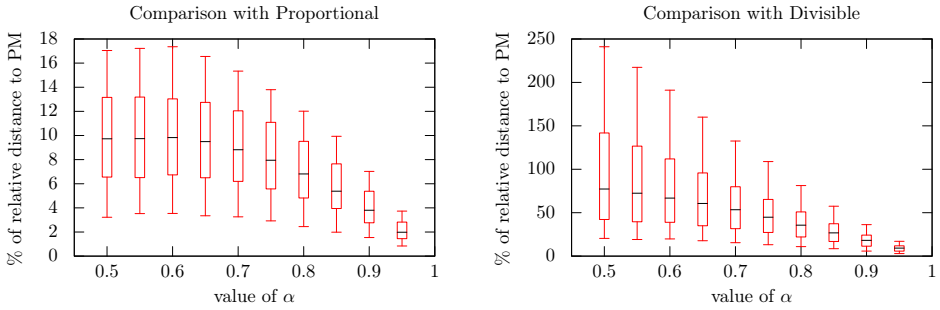


Figure 1.8: Comparison to the PM schedule with $p(t) = 100$.

PROPORTIONAL. We have also performed these tests with 100 processors, see Figure 1.8, which results on average in a 25% (resp. 10%) increase in the relative distance with PROPORTIONAL and (resp. with DIVISIBLE).

1.6 Extensions to distributed memory

The objective of this section is to extend the previous results to the case where the computing platform is composed of several nodes with their own private memory. In order to avoid the large communication overhead of processing a task on cores distributed across several nodes, we forbid such a multi-node execution: the tasks of the tree can be distributed on the whole platform but each task has to be processed on a single node. We prove that this additional constraint, denoted by $\mathscr{R}$, makes the problem much more difficult. We concentrate first on platforms with two homogeneous nodes and then with two heterogeneous nodes.

In this part, we make use of the shared-memory makespan-minimizing schedule induced by Theorem 1.1, which is referred to as the PM schedule $\mathscr{S}_{\text{PM}}$ (that stands for Prasanna and Musicus, who first depicted it).

1.6.1 Two homogeneous multicore nodes

In this section, we consider a multicore platform composed of two equivalent nodes having the same number of computing cores p. We also assume that all the tasks T_i have the same speedup function p_i^α on both nodes. We first show that finding a schedule with minimum makespan is weakly NP-complete, even for independent tasks:

Theorem 1.2. *Given two homogenous nodes of p processors, n independent tasks of sizes $w_1, ..., w_n$ and a bound T, the problem of finding a schedule of the n tasks on the two nodes that respects $\mathscr{R}$, and whose makespan is not greater than T, is (weakly) NP-complete for all values of the α parameter defining the speedup function.*

The proof relies on the PARTITION problem, which is known to be weakly (i.e., binary) NP-complete [66], and uses tasks of length $w_i = a_i^\alpha$, where the a_i's are the numbers from the instance of the PARTITION problem. We recall that we assume that functions $x \mapsto x^\alpha$ and $x \mapsto x^{1/\alpha}$ can be computed in polynomial time.

Proof. Let α be a fixed value.

Let $A = \{a_i, i \in [1, n]\}$ be an instance of the PARTITION problem. The objective is to decide whether there exists a partition of A in two sets that sum to the same value. Let $a = \sum_i a_i$. We reduce this problem to the homogeneous scheduling problem. Let $p = a/2$ and let w_i for $1 \leq i \leq n$ be equal to $w_i = a_i^\alpha$. We recall that the computation of the w_is is assumed polynomial. Let $\mathscr{J}$ be the instance of the homogeneous scheduling problem composed of p, the w_is and the bound $T = 1$. We show that there is a solution to the partition problem A if and only if $\mathscr{I}$ has a solution.

The PM schedule of the n independent tasks of size w_i on $2p$ processors has a makespan of

$$C_{max} = \left(\frac{\sum_i w_i^{1/\alpha}}{2p} \right)^\alpha = 1 = T$$

Then, by Theorem 1.1, the only schedules that achieve a makespan not greater than T on $2p$ processors are those who allocate to each task T_i the share $p_i = 2p \cdot w_i^{1/\alpha} / a = a_i$ (such schedules are not differentiated in the shared memory model of the previous section). Therefore, only such a schedule can be a solution to $\mathscr{I}$.

Such a schedule respects the $\mathscr{R}$ constraint if and only if the p_is can be partitioned between the two nodes of the platform. This is equivalent to state that a subset of the p_is sums to $p = a/2$, which is equivalent to state that A has a solution to the partition problem as for all i, $p_i = a_i$. □

We also provide a constant ratio approximation algorithm for an arbitrary tree. Note that we consider in-trees in the proof (the children are computed before their parent), but the same results apply to out-trees.

Theorem 1.3. *There exists a polynomial time $\left(\frac{4}{3}\right)^\alpha$-approximation algorithm for the makespan minimization problem when scheduling a tree of malleable tasks on two homogenous nodes.*

The proof of Theorem 1.3 consists in comparing the proposed solution to the optimal solution on a single node made of $2p$ processors, denoted $\mathscr{S}_{\text{PM}}$. Such an optimal solution can be computed as proposed in the previous section, and is a lower bound on the optimal makespan on 2 nodes with p processors. The general picture of the proposed algorithm is the following. First, the root of the tree is arbitrarily allocated to the p processors of one of the two nodes. Then, the subtrees Tr_i rooted at the root's children are considered. If none of these subtrees is allocated more than p processors in $\mathscr{S}_{\text{PM}}$, then we show how to "pack" the subtrees on the two nodes and bound the slow-down by $\left(\frac{4}{3}\right)^\alpha$ in Lemma 1.6. On the contrary, if one subtree Tr_i is allocated more than p processors in $\mathscr{S}_{\text{PM}}$, then we allocate p processors to its root, and recursively call the algorithm on its children and on the remaining subtrees. The proof of Theorem 1.3 is then done by induction, the heredity property relying on Lemmas 1.7 to 1.9. Lemma 1.5 allows the restriction to a slightly simpler class of graphs. We therefore state the necessary lemma before proving the main theorem.

Definition 1.2 ($\mathscr{S}_{\text{PM}}$)**.** *Let $\mathscr{S}_{\text{PM}}$ be the optimal schedule of G on $2p$ processors without the constraint $\mathscr{R}$.*

The makespan of $\mathscr{S}_{\text{PM}}$ is $C_{2p}^{\text{PM}} = \mathscr{L}_G/(2p)^\alpha$, which is then a lower bound of the optimal makespan with the restriction $\mathscr{R}$. One can observe that a 2^α approximation is immediate: a solution is the PM schedule of G with only p processors, whose makespan is $C_p^{\text{PM}} = \mathscr{L}_G/p^\alpha$. As the optimal makespan is not smaller than C_{2p}^{PM}, C_p^{PM} is indeed a 2^α-approximation.

The following lemma, whose proof is immediate, allows to restrict the following discussion on a slightly simpler class of graphs.

Lemma 1.5. *We can suppose without loss of generality that the length of the root of G is 0 and the root has at least two children.*

Proof. Otherwise, the chain starting at the root can be aggregated in a single task of length 0 before finding the schedule on this modified graph. It is then immediate to adapt it to the original graph, by allocating p processors to each task of this chain. Any optimal schedule would have the same allocation for this chain. □

Definition 1.3 (c_i, Tr_i and x)**.** *Let $\{c_i\}$, for $i \in \{1,\dots,n_c\}$, be the set of children of the root of G, and let Tr_i be the subtree of G rooted at c_i and including its descendants. We can suppose than the indices are ordered such that the $\mathscr{L}_{Tr_i}$'s are in decreasing order. Let $x \in [0,2]$ be such that xp processors are dedicated to Tr_1 in $\mathscr{S}_{\text{PM}}$. See Figure 1.9 for an illustration.*

Formally, we have:

$$x = \frac{2\mathscr{L}_{Tr_1}^{1/\alpha}}{\sum_{i=1}^{n_c} \mathscr{L}_{Tr_i}^{1/\alpha}}.$$

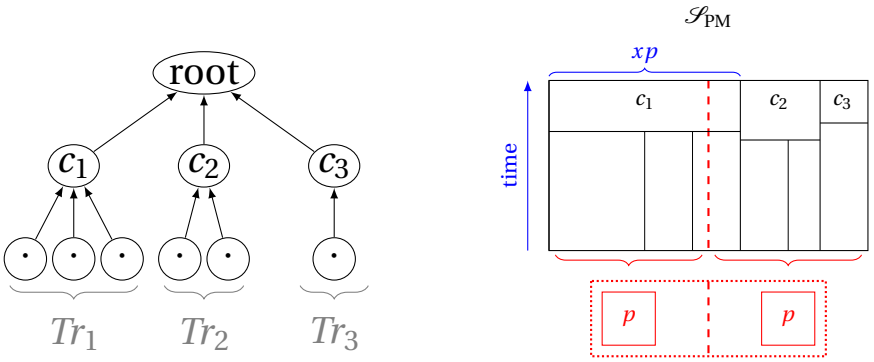


Figure 1.9: Structure of the graph G and of the schedule $\mathscr{S}_{\text{PM}}$ with $x \geq 1$.

The following lemma focuses on the simpler case of the proof, i.e., when no subtree is allocated more than p processors in $\mathscr{S}_{\text{PM}}$.

Lemma 1.6. *If we have $x \leq 1$, then a $\left(\frac{4}{3}\right)^\alpha$-approximation is computable in polynomial time.*

Proof. Let p_i be the constant share of processors allocated to Tr_i in $\mathscr{S}_{\text{PM}}$. By hypothesis, we have $p_i \leq p$ for all i, as $\mathscr{L}_{Tr_1}$ is the largest $\mathscr{L}_{Tr_i}$ and its share is equal to $xp \leq p$.

If the root has two children ($n_c = 2$), then $p_1 = p_2 = p$. Therefore, the schedule $\mathscr{S}_{\text{PM}}$ respects the restriction $\mathscr{R}$, is then optimal and so is a $\left(\frac{4}{3}\right)^\alpha$ approximation.

Otherwise, we have $n_c \geq 3$ and we partition the indices i in three sets S_1, S_2, S_3 such that the sum Σ_k of p_i's corresponding to each set S_k is not larger than p: $\forall k \in \{1,2,3\}$, $\Sigma_k = \sum_{i \in S_k} p_i \leq p$, which is always possible because no p_i is larger than p and the sum of all p_i's is $2p$. Indeed, we just have to iteratively place the largest p_i in the set that has the lowest Σ_k. If a Σ_k exceeds p, it was at least equal to $p/2$ at the previous step, and both other Σ_k also: the sum of all p_i's then exceeds $2p$, which is impossible.

Then, we define a schedule $\mathscr{S}$ in which we place the set with the largest Σ_k, say S_1, on one half of the processing power, and aggregate the two smallest, $S_2 \cup S_3$ in the other half. See Figure 1.10 for an illustration. We now compute the PM schedule of S_1 with p processors and $S_2 \cup S_3$ with p processors. The makespan of $\mathscr{S}$ is then:

$$C_{max} = \frac{\max\left(\mathscr{L}_{S_1}, \mathscr{L}_{S_2 \parallel S_3}\right)}{p^\alpha} = \frac{\mathscr{L}_{S_2 \parallel S_3}}{p^\alpha}.$$

Indeed, we have $\Sigma_1 \leq p \leq \Sigma_2 + \Sigma_3$ and $\mathscr{L}_{S_1}/\Sigma_1^\alpha = \mathscr{L}_{S_2 \parallel S_3}/(\Sigma_2 + \Sigma_3)^\alpha$, as these quantities represent the makespan of each subpart of the tree in $\mathscr{S}_{\text{PM}}$, and all subtrees Tr_i terminate simultaneously in $\mathscr{S}_{\text{PM}}$. So $\mathscr{L}_{S_1} \leq \mathscr{L}_{S_2 \parallel S_3}$.

We know that $\Sigma_1 \geq \max(\Sigma_2, \Sigma_3)$ and $\Sigma_1 + \Sigma_2 + \Sigma_3 = 2p$, so $\Sigma_1 \geq \frac{2}{3}p$, then $\Sigma_2 + \Sigma_3 \leq \frac{4}{3}p$. Therefore, in $\mathscr{S}_{\text{PM}}$, $\Sigma_2 + \Sigma_3 \leq \frac{4}{3}p$ processors are allocated to $S_2 \cup S_3$. Then, the makespan of $\mathscr{S}_{\text{PM}}$ verifies $C_{2p}^{\text{PM}} \geq \mathscr{L}_{S_2 \parallel S_3}/\left(\frac{4}{3}p\right)^\alpha$, and so $C_{max}/C_{2p}^{\text{PM}} \leq \left(\frac{4}{3}\right)^\alpha$. Therefore, $\mathscr{S}$ is indeed a $\left(\frac{4}{3}\right)^\alpha$ approximation. □

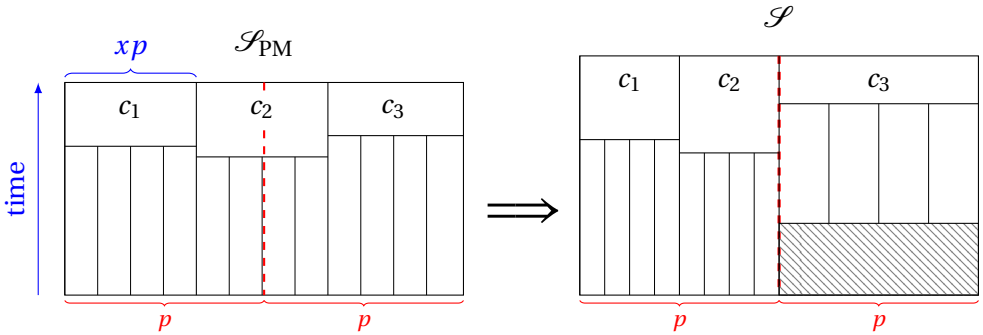


Figure 1.10: Illustration of the transformation of the schedule $\mathscr{S}_{\text{PM}}$ into $\mathscr{S}$ when $x \leq 1$, $n_c = 3$, and $S_1 = \{Tr_3\}$.

We now focus on the case not covered by Lemma 1.6, and therefore assume that $x > 1$.

Definition 1.4 ($\mathscr{R}_q$)**.** *For any $0 < q \leq p$, let $\mathscr{R}_q$ be the constraint that forces q processors to be allocated to c_1.*

We denote by B the subgraph $G \setminus \{root\} \setminus Tr_1$.

Definition 1.5 ($\mathscr{S}_u$, v_u and C_u)**.** *We define the schedule $\mathscr{S}_u$ parametrized by $u \in]0, p] \cup \{xp\}$, which respects $\mathscr{R}_u$ but not $\mathscr{R}$. It allocates a constant share $u \leq p$ of processors to c_1 until it is terminated. Meanwhile, $2p - u$ processors are allocated to schedule a part B_u of B. B_u may contain fractions of tasks. Before, the rest of the graph, which is composed of $Tr_1 \setminus \{c_1\}$ and of the potential remaining part $\bar{B}_u$ of B, is scheduled on $2p$ processors by a PM schedule, regardless of the $\mathscr{R}$ constraint. We denote by v_u the share allocated to $Tr_1 \setminus \{c_1\}$ and by C_u the makespan of the schedule.*

See Figure 1.11 for an illustration of this definition. Let $G_{u,1}$ be the graph $(Tr_1 \setminus \{c_1\}) \,\|\, \bar{B}_u$ and $G_{u,2}$ be the graph $c_1 \,\|\, B_u$. We denote by $\Delta_{u,1}$ (resp. $\Delta_{u,2}$) the time interval during which $G_{u,1}$ (resp. $G_{u,2}$) is executed in $\mathscr{S}_u$. Then, $C_u = |\Delta_{u,1}| + |\Delta_{u,2}|$. Note that the PM schedule $\mathscr{S}_{\text{PM}}$ is equal to $\mathscr{S}_{xp}$, where $u = v_u = xp$.

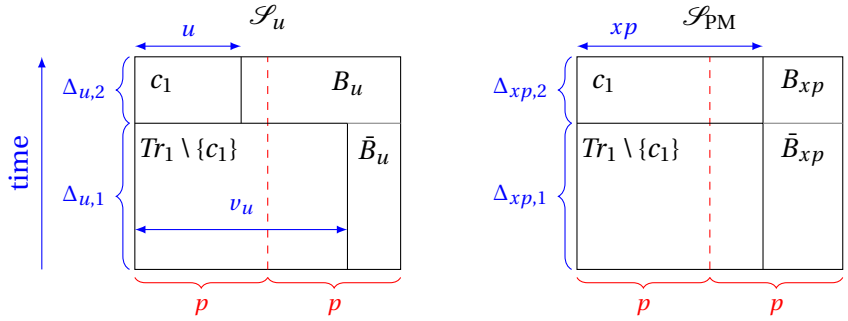


Figure 1.11: A schedule $\mathscr{S}_u$, for $u < p$ on the left and the schedule $\mathscr{S}_{\text{PM}} = \mathscr{S}_{xp}$ on the right.

Lemma 1.7. *For any $u \in]0, p]$, under the constraint $\mathscr{R}_u$, the makespan-optimal schedule is $\mathscr{S}_u$.*

Proof. Let $\mathscr{S}$ be the the makespan-optimal schedule that respects the constraint $\mathscr{R}_u$. We want to show that $\mathscr{S} = \mathscr{S}_u$.

First, suppose that c_1 terminates before B in $\mathscr{S}$. This means that $2p$ processors are dedicated to schedule B at the end of the schedule. We can slightly modify $\mathscr{S}$ by allocating $2p$ processors to B at the beginning of the schedule, for the same amount of time. This leads to the same makespan as there is no heredity constraint between B and Tr_1, and the same tasks of B can be performed in the new allocation, by a PM schedule on B. Indeed, B has the same makespan as a single task under any allocation by Theorem 1.1, and the makespan of a single task is unchanged under this new allocation. Therefore, we now assume that the schedule terminates at the execution of c_1.

Because of $\mathscr{R}_u$, $\mathscr{S}$ must allocate u processors to c_1 at the end of the schedule. In parallel to c_1, only B can be executed, and before the execution of c_1, both subgraphs B and $Tr_1 \setminus \{c_1\}$ can be executed.

Suppose that $\mathscr{S}$ and $\mathscr{S}_u$ are different during the interval $\Delta_{u,2}$. This means that, in $\mathscr{S}$, B is not scheduled according to PM ratios during $\Delta_{u,2}$. Then, the schedule $\mathscr{S}$ can be modified to schedule B in a smaller makespan by Theorem 1.1, and then to schedule the whole graph G in a smaller makespan, which contradicts the makespan-optimality of $\mathscr{S}$.

So $\mathscr{S}$ and $\mathscr{S}_u$ are equal during the time interval $\Delta_{u,2}$. Then, it remains to schedule the graph $G_{u,1} = (Tr_1 \setminus \{c_1\}) \,\|\, \bar{B}_u$, which has a unique optimal schedule, the PM schedule, that is followed by $\mathscr{S}_u$. Therefore, $\mathscr{S} = \mathscr{S}_u$. □

Lemma 1.8. *If $x > 1$, then $\mathscr{S}_p$ is the makespan-optimal schedule among the $\mathscr{S}_w$ for $w \in]0, p]$, i.e., we have $p = \operatorname{argmin}_{w \in]0,p]} (C_w)$.*

Proof. Let $u_{\text{OPT}} = \operatorname*{argmin}_{w \in]0,p]} (C_w)$. We will prove here that $u_{\text{OPT}} = p$.

For the sake of simplification, we denote in this proof u_{OPT} by u, v_u by v, $\Delta_{u,1}$ by Δ_1 and $\Delta_{u,2}$ by Δ_2. We will then consider the schedule $\mathscr{S}_u$, which is makespan-optimal among the $\mathscr{S}_w$, for $w \in]0, p]$.

Suppose by contradiction that $u < p$. We will build a schedule $\bar{\mathscr{S}}$ following the constraint $\mathscr{R}_{\bar{u}}$ for a value $\bar{u}$ such that $u < \bar{u} < p$, that will contradict the optimality of $\mathscr{S}_u$.

Proof that v is larger than p Note that this result can be intuitively deduced from an observation of the schedules.

As we have $x > 1$, we know that:

$$\mathscr{L}_{Tr_1 \setminus \{c_1\}} > \mathscr{L}_{\bar{B}_{xp}} = \mathscr{L}_B - \mathscr{L}_{B_{xp}} > \mathscr{L}_B - \mathscr{L}_{c_1}.$$

The first inequality holds because in $\mathscr{S}_{xp}$, the subgraphs $Tr_1 \setminus \{c_1\}$ and $\bar{B}_{xp}$ are scheduled in parallel, and each subgraph is scheduled according to the PM ratios. Then, each subgraph has the same makespan as its equivalent task. Moreover, xp (resp. $(2-x)p$) processors are allocated to $Tr_1 \setminus \{c_1\}$ (resp. $\bar{B}_{xp}$). Therefore, we get $|\Delta_{xp,1}| = \mathscr{L}_{Tr_1 \setminus \{c_1\}}/(xp)^\alpha = \mathscr{L}_{\bar{B}_{xp}}/((2-x)p)^\alpha$. As $x > 1$, more processors are allocated to $Tr_1 \setminus \{c_1\}$, so $\mathscr{L}_{Tr_1 \setminus \{c_1\}} > \mathscr{L}_{\bar{B}_{xp}}$. By the same reasoning between c_1 and B_{xp} in $\mathscr{S}_{xp}$, we get $\mathscr{L}_{B_{xp}} < \mathscr{L}_{c_1}$ and the second inequality holds. See Figure 1.11 for an illustration.

With similar arguments between the subgraphs c_1 and B_u in the schedule $\mathscr{S}_u$, and using the hypothesis $u < p$, we get $\mathscr{L}_{B_u} > \mathscr{L}_{c_1}$. The difference with the previous case is that the share of processors allocated to both subgraphs is not computed by the PM ratios, but as B_u is scheduled under $(2p-u)$ processors with the PM ratios, it has the same makespan as its equivalent task:

$$|\Delta_{u,2}| = \frac{\mathscr{L}_{B_u}}{(2p-u)^\alpha} = \frac{\mathscr{L}_{c_1}}{u^\alpha} \qquad \text{so} \qquad \mathscr{L}_{B_u} > \mathscr{L}_{c_1}.$$

Combining these two inequalities, we have $\mathscr{L}_{\bar{B}_u} < \mathscr{L}_B - \mathscr{L}_{c_1} < \mathscr{L}_{Tr_1 \setminus \{c_1\}}$, and by using the same reasoning in the other way with the parallel execution of $Tr_1 \setminus \{c_1\}$ and $\bar{B}_u$ in $\mathscr{S}_u$, we finally prove $v > p$.

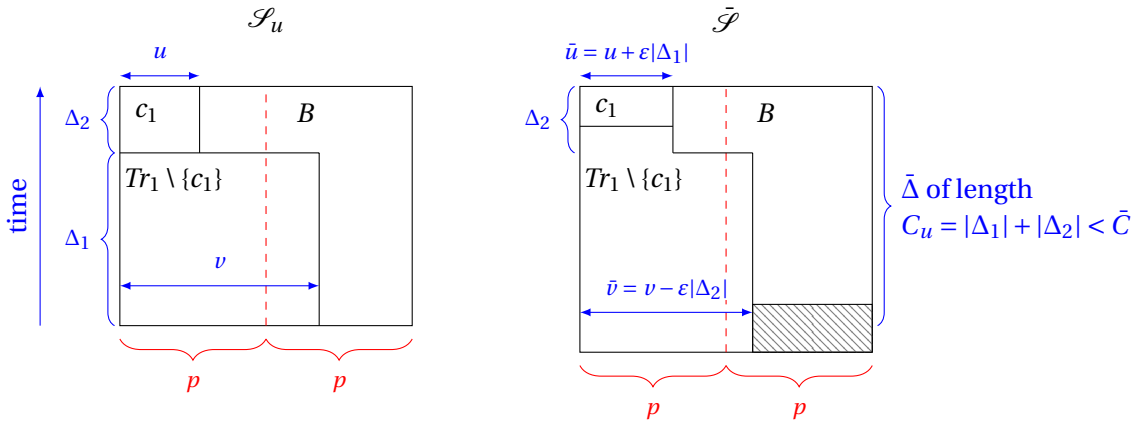


Figure 1.12: Schedules $\mathscr{S}_u$ and $\bar{\mathscr{S}}$, assuming that B begins after Tr_1 in $\bar{\mathscr{S}}$.

Definition of the schedule $\bar{\mathscr{S}}$ and exhibition of a contradiction Let $\varepsilon > 0$ small enough such that $u + \varepsilon|\Delta_1| < p$ and $v - \varepsilon|\Delta_2| > p$. Let $\bar{u} = u + \varepsilon|\Delta_1|$ and $\bar{v} = v - \varepsilon|\Delta_2|$. Note that $0 < u < \bar{u} < p < \bar{v} < v$.

Let $\bar{\mathscr{S}}$ be the schedule allocating $\bar{u}$ processors to Tr_1 during a time interval of length $|\Delta_2|$ at the end of the schedule, and $\bar{v}$ processors to Tr_1 before. The subgraph B is scheduled following PM ratios in parallel to Tr_1, in such a way that it terminates at the same time as c_1 and there is no idle time after the beginning of its execution. The subgraph Tr_1 is scheduled in the same way as B, following PM ratios as soon as its execution begins. Note that B and Tr_1 may not begin simultaneously. See Figure 1.12 for an illustration of the case where B begins after Tr_1. Let $\bar{C}$ be the makespan of $\bar{\mathscr{S}}$.

As $\bar{u} > u$, the processing time of c_1 is smaller than $|\Delta_2|$ in $\bar{\mathscr{S}}$, so $\bar{\mathscr{S}}$ respects the constraint $\mathscr{R}_{\bar{u}}$. Then, by Lemma 1.7, as $\bar{\mathscr{S}} \neq \mathscr{S}_{\bar{u}}$, we know that $\bar{C} > C_{\bar{u}}$. In addition, by the definition of C_u, we get $\bar{C} > C_{\bar{u}} \geq C_u$.

We can assume without loss of generality that Tr_1 and B are both a unique task, because both subgraphs are scheduled under the PM ratios in $\bar{\mathscr{S}}$ and $\mathscr{S}_u$, once the (time varying) share of processors allocated to each subgraph is fixed.

Let $\bar{\Delta}$ be the interval of time ending when $\mathscr{S}_u$ terminates and having a length equal to $C_u = |\Delta_1| + |\Delta_2|$. See Figure 1.12 for an illustration.

Let $\bar{W}_{Tr}$ (resp. $\bar{W}_B$) be the total work of the task Tr_1 (resp. B) that is executed in $\bar{\mathscr{S}}$ during $\bar{\Delta}$. Similarly, we define W_{Tr} (resp. W_B) the total work of the task Tr_1 (resp. B) that is executed in $\mathscr{S}_u$. These two last quantities are equal to:

$$\begin{aligned} W_{Tr} &= |\Delta_1| v^\alpha + |\Delta_2| u^\alpha \left(= \mathscr{L}_{Tr_1}\right) \\ W_B &= |\Delta_1| (2p - v)^\alpha + |\Delta_2| (2p - u)^\alpha \, (= \mathscr{L}_B) \end{aligned}$$

As $\bar{C} > C_u$, we cannot have both $\bar{W}_{Tr} \geq W_{Tr}$ and $\bar{W}_B \geq W_B$. Indeed, in this case, all the tasks of G would be completed by $\bar{\mathscr{S}}$ in a makespan smaller than C_u, which is a contradiction. For both tasks B and Tr, we consider two cases.

If Tr_1 begins in $\bar{\mathscr{S}}$ during $\bar{\Delta}$, then $\bar{W}_{Tr} = \mathscr{L}_C = W_{Tr}$ because the execution of Tr_1 would hold entirely in $\bar{\Delta}$. Otherwise, we have:

$$\bar{W}_{Tr} = |\Delta_1| \bar{v}^\alpha + |\Delta_2| \bar{u}^\alpha.$$

We know that $0 < u < \bar{u} < \bar{v} < v$. Therefore, by the concavity of the function $s : x \mapsto x^\alpha$, we conclude that $\bar{W}_{Tr}$ is larger than W_{Tr}:

$$\begin{aligned} \frac{\bar{u}^\alpha - u^\alpha}{\bar{u} - u} &> \frac{v^\alpha - \bar{v}^\alpha}{v - \bar{v}} \\ \frac{\bar{u}^\alpha - u^\alpha}{\varepsilon |\Delta_1|} &> \frac{v^\alpha - \bar{v}^\alpha}{\varepsilon |\Delta_2|} \\ |\Delta_2| \left(\bar{u}^\alpha - u^\alpha\right) &> |\Delta_1| \left(v^\alpha - \bar{v}^\alpha\right) \\ |\Delta_2| \bar{u}^\alpha + |\Delta_1| \bar{v}^\alpha &> |\Delta_2| u^\alpha + |\Delta_1| v^\alpha \\ \bar{W}_{Tr} &> W_{Tr}. \end{aligned}$$

Therefore, in any case, we have $\bar{W}_{Tr} \geq W_{Tr}$.

Then, we treat similarly the subgraph B. If B begins in $\bar{\mathscr{S}}$ during $\bar{\Delta}$, then $\bar{W}_B = \mathscr{L}_B = W_B$.

Otherwise, we have:

$$\bar{W}_B = \Delta_1 (2p - \bar{v})^\alpha + \Delta_2 (2p - \bar{u})^\alpha$$

Similarly, we know that $2p-u > 2p-\bar{u} > 2p-\bar{v} > 2p-v > 0$. Therefore, by the concavity of the function $s : x \mapsto x^\alpha$, we have:

$$\begin{aligned}
\frac{(2p-u)^\alpha - (2p-\bar{u})^\alpha}{\bar{u}-u} &< \frac{(2p-\bar{v})^\alpha - (2p-v)^\alpha}{v-\bar{v}} \\
\frac{(2p-u)^\alpha - (2p-\bar{u})^\alpha}{\varepsilon|\Delta_1|} &< \frac{(2p-\bar{v})^\alpha - (2p-v)^\alpha}{\varepsilon|\Delta_2|} \\
\frac{(2p-\bar{u})^\alpha - (2p-u)^\alpha}{\varepsilon|\Delta_1|} &> \frac{(2p-v)^\alpha - (2p-\bar{v})^\alpha}{\varepsilon|\Delta_2|} \\
|\Delta_2|\left((2p-\bar{u})^\alpha - (2p-u)^\alpha\right) &> |\Delta_1|\left((2p-v)^\alpha - (2p-\bar{v})^\alpha\right) \\
|\Delta_2|(2p-\bar{u})^\alpha + |\Delta_1|(2p-\bar{v})^\alpha &> |\Delta_2|(2p-u)^\alpha + |\Delta_1|(2p-v)^\alpha \\
\bar{W}_B &> W_B
\end{aligned}$$

Then, in any case, we have both $\bar{W}_{Tr} \geq W_{Tr}$ and $\bar{W}_B \geq W_B$, so we get the contradiction.
Therefore, we have $u \geq p$ and so $u = p$. □

Lemma 1.9. *If $x > 1$, then the makespan of $\mathscr{S}_p$ is not smaller than the minimal makespan of a schedule respecting $\mathscr{R}$.*

Proof. Let $\mathscr{S}_{\text{OPT}}$ be a schedule respecting $\mathscr{R}$ of minimal makespan. Note that in $\mathscr{S}_{\text{OPT}}$, a constant share of $u^* \leq p$ processors must be allocated to c_1 due to $\mathscr{R}$, as in $\mathscr{S}_{u^*}$. Indeed, if this share is not constant, because of the concavity of the function $s : x \mapsto x^\alpha$, it would be better to always allocate the mean value to c_1. This would allow to terminate earlier both c_1 and the tasks executed in parallel to c_1 on the same part, as proved by Lemma 1.3.

Therefore, $\mathscr{S}_{\text{OPT}}$ respects the constraint $\mathscr{R}_{u^*}$. So its makespan is not smaller than the one of $\mathscr{S}_{u^*}$ by Lemma 1.7. Therefore, it is not smaller than the one of $\mathscr{S}_p$ by Lemma 1.8, which proves the lemma. □

We are now ready to prove Theorem 1.3. The corresponding approximation algorithm is provided in Algorithm 1.

Algorithm 1: HOMOGENEOUSAPP(G,p)

```
G̃ ← G
Modify G as in Lemma 1.5
Compute the PM schedule S_PM of G on 2p processors
Determine the c_i, the Tr_i, B, and x
if x ≥ 1 and c_1 is a leaf then
    Build S: shrink from S_PM the share of processors allocated to c_1 to p processors
else if x ≤ 1 then
    Build S: schedule the Tr_i's as in Lemma 1.6, and compute the PM schedule on each part
else
    Compute the schedule S_p and partition G in G_{p,1} and G_{p,2} as in Definition 1.5
    S^r ← HOMOGENEOUSAPP(G_{p,1}, p)
    Build S: schedule G_{p,1} as in S^r then G_{p,2} as in S_p
Adapt S to the original graph G̃ if G ≠ G̃ by scheduling the additional tasks on p processors
return S
```

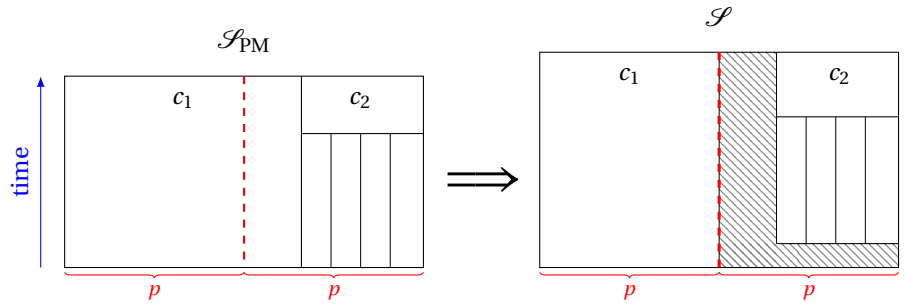


Figure 1.13: Illustration of $\mathscr{S}_{\text{PM}}$ and an optimal schedule $\mathscr{S}$ when $x \geq 1$ and c_1 is a leaf.

Proof of Theorem 1.3. We prove here by induction on the tree structure of G that Algorithm 1 is a $\left(\frac{4}{3}\right)^{\alpha}$-approximation.

If G has only one task, the result is immediate.

As stated in Lemma 1.5, we can suppose that the root has length 0 and has at least two children. Otherwise, the root and the chain rooted at it can be optimally scheduled on p processors without increasing the approximation ratio.

Then, we treat the cases that do not need the heredity property.

- if $x \geq 1$ and c_1 is a leaf, then the makespan of $\mathscr{S}_{\text{PM}}$ is equal to $\mathscr{L}_{c_1}/(xp)^{\alpha}$. Then, the schedule that differs from $\mathscr{S}_{\text{PM}}$ by reducing the allocation of c_1 to p, the maximum possible under the constraint $\mathscr{R}$, achieves the minimal makespan. See Figure 1.13 for an illustration.

- if $x \leq 1$, the result is given by Lemma 1.6.

Now, we suppose the result true for a graph G of less than n nodes. The case remaining is when c_1 is not a leaf and $x > 1$. Consider such a graph G of n nodes.

We consider the schedule $\mathscr{S}_p$, whose makespan C_p is not larger than the makespan of $\mathscr{S}_{\text{OPT}}$ as stated in Lemma 1.9.

We now build the schedule $\mathscr{S}$, which achieves a $\left(\frac{4}{3}\right)^{\alpha}$-approximation respecting $\mathscr{R}$. At the end of the schedule, $G_{p,1}$ is scheduled as in $\mathscr{S}_p$. At the beginning of the schedule, we use the heredity property to derive from $\mathscr{S}_p$ a schedule of $G_{p,2}$ that follows the $\mathscr{R}$ constraint. See Figure 1.14 for an illustration.

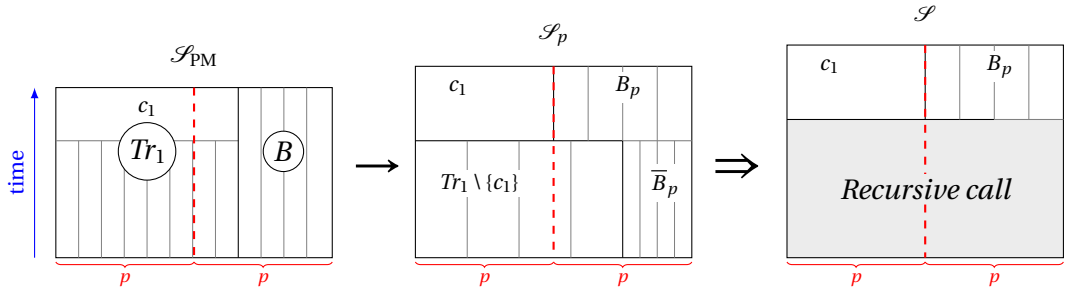


Figure 1.14: Illustration of the construction of $\mathscr{S}$, when c_1 is not a leaf and $x > 1$.

More formally, we have $G_{p,2}$, which is the parallel composition $(Tr_1 \setminus \{c_1\}) \parallel \bar{B}_p$, composed of at most $n-1$ nodes. So, by induction, a schedule $\mathscr{S}^r$ achieving a $\left(\frac{4}{3}\right)^{\alpha}$-approximation can be computed for $G_{p,2}$. This means that its makespan C^r_{max} is at most $\left(\frac{4}{3}\right)^{\alpha} \Delta_{p,2}$, as $\mathscr{S}_p$ completes $G_{p,2}$ with PM ratios in a time $\Delta_{p,2}$, which is then the optimal time.

Consider the schedule $\mathscr{S}$ of G that schedules $G_{p,2}$ as in $\mathscr{S}^r$, then schedules $G_{p,1}$ as in $\mathscr{S}_p$. The time necessary to complete $G_{p,1}$ is then equal to $\Delta_{p,1}$. The makespan C_{max} of $\mathscr{S}$ then respects:

$$C_{max} = \Delta_{p,1} + C^r_{max} \leq \Delta_{p,1} + \left(\frac{4}{3}\right)^\alpha \Delta_{p,2} \leq \left(\frac{4}{3}\right)^\alpha \left(\Delta_{p,1} + \Delta_{p,2}\right) \leq \left(\frac{4}{3}\right)^\alpha C_p \leq \left(\frac{4}{3}\right)^\alpha C_{\text{OPT}}.$$

Then, $\mathscr{S}$ is a $\left(\frac{4}{3}\right)^\alpha$-approximation, so Algorithm 1 is a polynomial-time $\left(\frac{4}{3}\right)^\alpha$-approximation. □

1.6.2 Two heterogeneous multicore nodes

We suppose here that the computing platform is made of two processors of different processing capabilities: the first one is made of p cores, while the second one includes q cores, and all cores are identical. We also assume that the parameter α of the speedup function is the same on both processors. As the problem gets more complicated, we concentrate here on n independent tasks, of lengths $w_1, ..., w_n$. The (p,q)-SCHEDULING problem consists in finding a schedule of minimal makespan. Thanks to the homogeneous case presented in Section 1.6.1, we already know that scheduling independent tasks on two nodes is NP-complete.

This problem is close to the SUBSET SUM problem. Given n numbers, the optimization version of SUBSET SUM considers a target K and aims at finding the subset with maximal sum smaller than or equal to K. There exists many approximation schemes for this problem. In particular, Kellerer et al. [95] propose a fully polynomial approximation scheme (FPTAS). Based on this result, an approximation scheme can be derived for our problem. The full proof is detailed below. We assume that each $w_i^{1/\alpha}$ is an integer in order to apply the FPTAS for SUBSET SUM, which is valid only on integers. This assumption is necessary as the complexity of the algorithm depends on the precision required. In practice, any encoding of these numbers can be interpreted as integers.

We first need a few definitions before stating Theorem 1.4 which implies the construction of an FPTAS for a restricted version of the (p,q)-SCHEDULING problem in Corollary 1.1.

Consider an instance $\mathscr{I}$ of (p,q)-SCHEDULING. We define the following quantities:

$$x_i = w_i^{1/\alpha} \quad ; \quad S = \sum_{i=1}^{n} x_i \quad ; \quad X = \{x_i, i \in \{1,\dots,n\}\} \quad ; \quad r = \max\left(\frac{q}{p}, \frac{p}{q}\right).$$

We use the notation A to represent the subset of the indices of the tasks allocated to the p-part in a given schedule, and $\bar{A}$ is the complementary of A. Then, the schedule that partitions the tasks according to the subset A and performing a PM schedule on both parts is denoted by $\mathscr{S}_A$.

For $\lambda > 1$, a λ-approximation of (p,q)-SCHEDULING returns a schedule whose makespan is not larger than λ times the optimal makespan. For $0 < \kappa < 1$, a κ-approximation of the SUBSET SUM instance composed of the set X and a target K returns a subset A of X such that the sum of its elements ranges between κOPT and OPT, where:

$$OPT = \max_{A \mid \sum_A x_i \leq K} \sum_A x_i.$$

We furthermore define $\varepsilon_\lambda = \lambda^{1/\alpha} - 1$ and $\varepsilon_\kappa = 1 - \kappa$. An approximation scheme $\mathscr{A}$ resolving SUBSET SUM is defined as follows. Given an instance $\mathscr{J}$ of SUBSET SUM and a parameter $0 < \kappa < 1$, it computes a solution to $\mathscr{J}$ achieving a κ-approximation in a time complexity $f_{\mathscr{A}}(n, \varepsilon_\kappa)$. An approximation scheme $\mathscr{B}$ resolving (p,q)-SCHEDULING is defined as follows. Given an instance $\mathscr{I}$ of the (p,q)-scheduling problem and a parameter $\lambda > 1$, it computes a solution to $\mathscr{I}$ achieving a λ-approximation in a time complexity $f_{\mathscr{B}}(\mathscr{J}, \varepsilon_\lambda)$.

Remark 1.1. *There exist an FPTAS for* SUBSET SUM*: Kellerer et al. [95] have designed an FPTAS of time complexity* $O\left(\min\left(n/\varepsilon_\kappa, n+1/\varepsilon_\kappa^2\log(1/\varepsilon_\kappa)\right)\right)$ *and space complexity* $0(n+1/\varepsilon_\kappa)$.

As previously mentioned, we design an FPTAS for a restricted version of (p,q)-SCHEDULING, where the x_i's are integer. This problem is defined in Definition 1.6 as (p,q)-SCHEDULING RESTRICTED. This allows to use algorithms designed to solve the integer problem SUBSET SUM. The proposed scheme is defined in Algorithm 2 and its complexity when using the FPTAS of [95] is given in Corollary 1.1 of Theorem 1.4.

Definition 1.6. *The* (p,q)-SCHEDULING RESTRICTED *problem is defined from the* (p,q)-SCHEDULING *problem by replacing the input* w_i *by* $x_i = w_i^{1/\alpha}$*, and is restricted to the case where the* x_i *are integers, and* p *and* q *are given in unary.*

Theorem 1.4. *Given a* κ*-approximation scheme* $\mathscr{A}$ *of* SUBSET SUM *of time complexity* $(n,\varepsilon_\kappa)\mapsto f_{\mathscr{A}}(n,\varepsilon_\kappa)$*, Algorithm 2 performs is a* λ*-approximation scheme to* (p,q)-SCHEDULING RESTRICTED *with time complexity* $(n,p,q,\alpha,\lambda)\mapsto O\left(n+f_{\mathscr{A}}\left(n,\frac{\varepsilon_\lambda}{r}\right)\right)$*, assuming that raising a number to the power* α *or* $1/\alpha$ *can be done is constant time.*

Corollary 1.1. *The* (p,q)-SCHEDULING RESTRICTED *problem admits an FPTAS of time complexity* $O\left(\min\left(\frac{nr}{\varepsilon_\lambda}, n+\left(\frac{r}{\varepsilon_\lambda}\right)^2\log\left(\frac{r}{\varepsilon_\lambda}\right)\right)\right)$ *and space complexity* $0\left(n+\frac{r}{\varepsilon_\lambda}\right)$*: use Algorithm 2 with the FPTAS of [95].*

Algorithm 2: HETEROGENEOUSAPP($w_1,\ldots,w_n$, p, q, λ, $\mathscr{A}$)

$r \leftarrow \max\left(\frac{p}{q},\frac{q}{p}\right)$; $S \leftarrow \sum_{T_i\in V} w_i^{1/\alpha}$; $\varepsilon_\lambda \leftarrow \lambda^{1/\alpha}-1$; $X \leftarrow \left\{w_i^{1/\alpha} \mid i\in\{1,\ldots,n\}\right\}$;
if $\lambda > (1+r)^\alpha$ **then**
  return *the PM schedule on the largest node*
$A \leftarrow \mathscr{A}\left(X, \frac{pS}{p+q}, \frac{\varepsilon_\lambda}{r}\right)$; $B \leftarrow \mathscr{A}\left(X, \frac{qS}{p+q}, \frac{\varepsilon_\lambda}{r}\right)$;
return *the schedule with the minimum makespan between* $\mathscr{S}_A$ *and* $\mathscr{S}_{\bar{B}}$

Proof of Theorem 1.4. Let $\mathscr{I}$ be an instance of (p,q)-SCHEDULING RESTRICTED, $\lambda>1$ and $\mathscr{A}$ be an approximation scheme of SUBSET SUM. We recall that raising a number to the power α or $1/\alpha$ is assumed feasible.

If $\lambda \geq (1+r)^\alpha$, it suffices to compute the PM schedule on the largest node of the platform. We assume in the following that $\lambda < (1+r)^\alpha$. We define $\kappa = \left(1-\frac{1}{r}(\lambda^{1/\alpha}-1)\right)$, so that $\varepsilon_\kappa = 1-\kappa = \frac{\varepsilon_\lambda}{r}$ and $0<\kappa<1$.

A lower bound on the makespan is $C_{\text{ideal}} = \left(\frac{S}{p+q}\right)^\alpha$. Indeed, it represents the makespan of the PM schedule on $p+q$ processors, which may respect the constraint for some values of the x_i. In this schedule, we have $\sum_{i\in A} x_i = \frac{pS}{p+q}$. Let Σ_{ideal} denotes this quantity.

Let $\mathscr{S}_{\text{OPT}}$ be an optimal schedule of $\mathscr{I}$, and A_{O} the subset of the tasks allocated to the p-part of the platform in $\mathscr{S}_{\text{OPT}}$. We first assume that $\mathscr{S}_{\text{OPT}}$ terminates to schedule the tasks of A_{O} before the time C_{ideal}, which is equivalent to say, with $\Sigma_{\text{OPT}} = \sum_{i\in A_{\text{O}}} x_i$, that

$$\Sigma_{\text{OPT}} \leq \Sigma_{\text{ideal}} = \frac{pS}{p+q}. \tag{1.1}$$

Therefore, the makespan of $\mathscr{S}_{\text{OPT}}$, which we name C_{OPT}, is equal to the makespan on the q-part of the schedule, so we have:

$$C_{\text{OPT}} = \left(\frac{\sum_{i\in\bar{A}_{\text{O}}} x_i}{q}\right)^{\alpha} = \left(\frac{S-\Sigma_{\text{OPT}}}{q}\right)^{\alpha}.$$

Let Λ be the set of subsets of X such that:

$$\Lambda = \left\{ A \subset X \;\middle|\; (1-\varepsilon_\kappa)\Sigma_{\text{OPT}} \le \sum_{i\in A} x_i \le \Sigma_{\text{OPT}} \right\}.$$

We prove in the following paragraph that a subset $A \in \Lambda$ is computed by the algorithm $\mathscr{A}$ launched on the set X, with the target $K = \Sigma_{\text{ideal}} = \frac{pS}{p+q}$, and the precision ε_κ. Recall that the x_i are assumed to be integers in the formulation of (p,q)-SCHEDULING RESTRICTED.

By the definition of Σ_{OPT}, we have $A_O \in \Lambda$. Then, no subset A of X verifies $\Sigma_{\text{OPT}} < \sum_{i\in A} x_i \le \Sigma_{\text{ideal}}$, because the associated schedule $\mathscr{S}_A$ would have a makespan smaller than C_{OPT}, which contradicts the optimality of $\mathscr{S}_{\text{OPT}}$. So A_O is an optimal solution of the instance submitted to $\mathscr{A}$. Therefore, $\mathscr{A}$ launched on this instance with the parameter ε_κ will return a set $A \in \Lambda$ in time $f_{\mathscr{A}}(n, \varepsilon_\kappa)$.

Let A be an element of Λ. We know that the makespan C_A of the corresponding schedule $\mathscr{S}_A$ allocating the tasks corresponding to A on the p-part is:

$$C_A = \left(\max\left(\frac{\sum_{i\in A} x_i}{p}, \frac{\sum_{i\in\bar{A}} x_i}{q}\right)\right)^{\alpha}$$

As $A \in \Lambda$, we have:

$$\sum_{i\in\bar{A}} x_i = S - \sum_{i\in A} x_i \qquad \text{and} \qquad \sum_{i\in A} x_i \ge \kappa\Sigma_{\text{OPT}} \qquad \text{so} \qquad \frac{\sum_{i\in\bar{A}} x_i}{q} \le \frac{S-\kappa\Sigma_{\text{OPT}}}{q}.$$

Because $A \in \Lambda$ and we assumed that Equation 1.1 is valid, we have:

$$\sum_{i\in A} x_i \le \Sigma_{\text{OPT}} \le \Sigma_{\text{ideal}}.$$

This last inequality implies that the makespan is equal to the makespan of the q-part of the platform: $C_A = \left(\sum_{i\in\bar{A}} x_i / q\right)^{\alpha}$. Therefore, we have:

$$\left(\frac{C_A}{C_{\text{OPT}}}\right)^{1/\alpha} \le \frac{S-\kappa\Sigma_{\text{OPT}}}{S-\Sigma_{\text{OPT}}}.$$

Then, as $0 < \kappa < 1$ and $\Sigma_{\text{OPT}} \le \Sigma_{\text{ideal}}$, we get:

$$\left(\frac{C_A}{C_{\text{OPT}}}\right)^{1/\alpha} \le \frac{S-\kappa\Sigma_{\text{ideal}}}{S-\Sigma_{\text{ideal}}} \le \frac{1-\frac{\kappa p}{p+q}}{1-\frac{p}{p+q}} \le \frac{p+q-\kappa p}{q} \le 1 + \frac{p}{q}(1-\kappa).$$

Then, r is not smaller than $\frac{p}{q}$, so:

$$\frac{C_A}{C_{\text{OPT}}} \le (1 + r\,(1-\kappa))^{\alpha} = \lambda.$$

We have supposed so far that Equation 1.1 holds. Note that otherwise, we have a symmetric inequality exchanging p and q:

$$\sum_{i\in\bar{A}_O} x_i \le S - \frac{pS}{p+q} = \frac{qS}{p+q}.$$

Then, as the problem is also symmetric in p and q, by an analogue reasoning, one we prove that $\mathscr{A}$ launched on the set X, the target $\frac{qS}{p+q}$, and the precision ε_κ returns a set B in:

$$\Lambda' = \left\{ B \subset X \;\middle|\; (1-\varepsilon_\kappa) \sum_{i \in \bar{A}_0} x_i \leq \sum_{i \in B} x_i \leq \sum_{i \in \bar{A}_0} x_i \right\}.$$

The schedule that associates B to the q-part of the processors then has a makespan smaller than λC_{OPT}. To conclude, Algorithm 2 launched with the parameter λ computes a set $A \in \Lambda$ and a set $B \in \Lambda'$, then returns the schedule that has the minimum makespan between $\mathscr{S}_A$ and $\mathscr{S}_{\bar{B}}$. Therefore, regardless of whether Equation 1.1 is valid, the returned schedule has a makespan smaller than λC_{OPT}, and so Algorithm 2 achieves a λ-approximation.

The complexity of computing the makespan of these schedules is linear if we assume that raising a number to the power α or $1/\alpha$ can be done in constant time. Adding the complexity of the approximation scheme $\mathscr{A}$, we get a total complexity equal to $O\left(n + f_{\mathscr{A}}\left(n, \frac{\varepsilon_\lambda}{r}\right)\right)$. □

1.7 Conclusion

In this chapter, we have studied how to schedule trees of malleable tasks whose speedup function on multicore platforms is p^α. We have first motivated the use of this model for sparse matrix factorizations by actual experiments. When using factorization kernels actually used in sparse solvers, we show that the speedup follows quite well the p^α model for reasonable allocations. On the machine used for our tests, α is in the range 0.85–0.95. Then, we proposed a new proof of the optimal allocation derived by Prasanna and Musicus [121, 122] for such trees on single node multicore platforms. Contrarily to the use of optimal control theory of the original proofs, our method relies only on pure scheduling arguments and gives more intuitions on the scheduling problem. Based on these proofs, we proposed several extensions for two multicore nodes: we prove the NP-completeness of the scheduling problem and propose a $\left(\frac{4}{3}\right)^\alpha$-approximation algorithm for a tree of malleable tasks on two homogeneous nodes, and an FPTAS for independent malleable tasks on two heterogeneous nodes. Finally, we have estimated the potential gain of using an optimal allocation compared to simpler allocations from the literature on a single multicore node by extensive simulations. Although the improvement over simpler allocations may seem small in the measured range of α values, it has to be noted that (i) even a 5% improvement is interesting when comparing real software implementations, which is also why the optimal allocation under this particular speed-up model has already been considered for sparse solvers [23], (ii) the value of α is expected to be smaller for machine with weaker memory bandwidth and (iii) memory bandwidth increases at a smaller pace than core computing rates [75], which is likely to make smaller values of α more relevant in the future.

The speedup model studied however suffers from several limitations. There is no lower bound on the processing time of a task, whereas every task has a sequential part that has to be processed and cannot be shortened. Moreover, the value of α must be the same for every task, which is a very strong hypothesis when dealing with highly heterogeneous tasks. The accuracy of the model even when choosing the best value for α is not totally satisfactory. Furthermore, it would be preferable to avoid using rational numbers of processors allocated to each task. These observations, combined with the fact that the observed gain is significant but not large, lead us to study a more general model, which is the subject of the next chapter.

Chapter 2

The two-threshold roofline speedup model for parallel tasks

« J'ai pas plus rapide. »

Panoramix, *Astérix et Obélix: Mission Cléopâtre*

In this chapter, we study the problem of scheduling parallel task graphs to minimize the makespan, as in Chapter 1. We target the same applications, i.e., the assembly tree arising in multifrontal factorizations of sparse matrices. Hence, we focus on scheduling task trees, but also consider series-parallel graphs and even general DAGs. The main objective of this chapter is to design and validate a more accurate and general speedup model for these applications, before designing scheduling algorithms specific for this model. We recall that the speedup function s_i of a task T_i for p_i processors is defined as the sequential processing time of T_i divided by its processing time under p_i processors. The speedup model of the previous chapters presented several limitations. For instance, the speedup could be arbitrarily large, whereas it is always bounded by a threshold in practice; and all the tasks of a graph were modeled by the same speedup function, which is very constraining. We now therefore consider a more flexible model.

The proposed speedup model, validated on linear algebra kernels benchmarks, is a continuous piecewise linear function, depending on two thresholds on the processor allotment: before a first threshold, the speedup is perfect, that is, equal to the number of processors; between the two thresholds, it is linear, but not perfect anymore; after the second threshold, it stalls and stays constant. The intuition behind this model is the following. In the first part, all the processors are fully used so the parallelism is close to perfect. In the last part, the maximum parallelism has been reached, so no further progress can be made. In the middle part, additional processors are underused but still beneficial. Therefore, the speedup of each task is parameterized by three quantities: the two thresholds and the maximal speedup, which allows accounting for different behaviors among the tasks of the graph. This model extends the well-studied simple single-threshold model, with a perfect speedup before the threshold, and constant speedup thereafter. This simplified model has been studied both in theoretical scheduling [55] and for practical schedulers [114]. Contrarily to most existing studies, we also assume that tasks are *preemptible* (a task may be interrupted and resumed later), *malleable* (the number of processors allocated to a task can vary over time) and we allow *fractional allocation* of processors. We claim that this model is reasonable based on the following two arguments. Firstly, changing the allocation of processors is easily achieved using the time sharing facilities of operating system schedulers or hypervisors: actual runtime schedulers are able to dynamically change the allocation of a task [84]. Secondly, given preemption and malleability, it is possible to transform any schedule with fractional allocation to a schedule with

integral allocation using McNaughton's wrap-around rule [111] (as shown in Section 2.1). Hence, we can consider fractional allocations that are simple to design and analyze, and then transform them into integral ones when needed.

Main contributions. In this chapter, we propose a practical piecewise linear speedup model which is divided into three parts. Up to a first threshold the speedup is perfect, equaling the number of processors. Then it grows linearly, but with a slope smaller than one until a second threshold is reached, after which the speedup remains constant. This model is validated experimentally: it closely follows speedup curves typical for linear algebra kernels. For this model, we show the NP-completeness of the decision problem associated with the minimization of the makespan for a given graph. We study previously proposed algorithms PROPMAPPING (Proportional Mapping) which is commonly used by runtime schedulers, and FLOWFLEX, and propose model-optimised variants of these algorithms. Furthermore, the novel GREEDYFILLING is proposed, designed for the new speedup model. Both GREEDYFILLING and PROPMAPPING are shown to be 2-approximation algorithms with a single threshold. Finally, we perform simulations both on synthetic series-parallel graphs and on real task trees from linear algebra applications demonstrating the general superiority of the new GREEDYFILLING and the model-optimized variants of the traditional algorithms.

The rest of this chapter is organized as follows. Section 2.1 details the model and Section 2.2 validates it experimentally. The complexity of the problem is depicted in Section 2.3. In Section 2.4, we describe and prove the performance of several algorithms, which are then compared via simulations in Section 2.5. A review of the relevant related work on malleable task graph scheduling can be found in Section 1.1, Chapter 1.

2.1 Application model

We consider a workflow of tasks whose precedence constraints are represented by a task graph $G = (V, E)$ of n nodes, or tasks: a task can only be executed after the termination of all its predecessors. We assume that G is a *series-parallel* graph. Such graphs are built recursively as series or parallel composition of two or more smaller SP-graphs, and the base case is a single task. Trees can be seen as a special-case of series-parallel graphs.

Each task $T_i \in V$ is associated with a ***weight*** w_i that corresponds to the work that needs to be done to complete the task. By extension, the weight of a subgraph of G is the sum of the weights of the tasks it is composed of. The ***start time*** of a task T_i is defined as the time when the processing of its work starts for the first time. Denoted by p is the total number of identical processors available to schedule G. Tasks are assumed to be preemptible and malleable; each task T_i may be allocated a fractional, time-varying amount $p_i(t)$ of processors at time t. The speedup of each task, illustrated in Figure 2.1, is a piecewise linear function of the number of processors allocated to the task. Task T_i is associated with two integer ***thresholds***, $\delta_i^{(1)}$ and $\delta_i^{(2)}$, on the number of processors and a maximum speedup Ω_i, which define the speedup of the task:

- for a number of processors smaller than, or equal to, the first threshold, the task is perfectly parallel;
- for a number of processors larger than the second threshold, the speedup is bounded by the maximum speedup;
- between the two thresholds, the speedup is linear but not perfect.

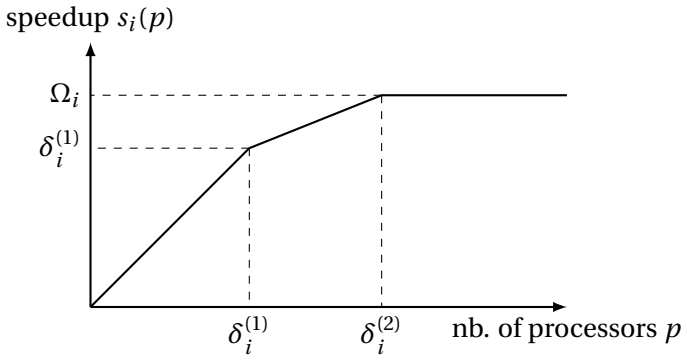


Figure 2.1: Illustration of the proposed speedup model and its notations.

Formally, the speedup function is a continuous piecewise linear function defined as

$$s_i(p) = \begin{cases} p & \text{if } p \leq \delta_i^{(1)} \\ \delta_i^{(1)} + \dfrac{(p-\delta_i^{(1)})(\Omega_i - \delta_i^{(1)})}{\delta_i^{(2)} - \delta_i^{(1)}} & \text{if } \delta_i^{(1)} \leq p \leq \delta_i^{(2)} \\ \Omega_i & \text{if } p \geq \delta_i^{(2)} \end{cases} \tag{2.1}$$

The completion or ***finish time*** of task T_i is thus defined as the smallest value C_i such that

$$\int_0^{C_i} s_i(p_i(t))\,dt = w_i.$$

The objective is to minimize the *makespan* of the application, that is the latest task finish time.

Model variants There exists a notable special case of our model: when both thresholds are equal ($\delta_i^{(1)} = \delta_i^{(2)}$) we necessarily get $\Omega_i = \delta_i^{(1)}$ and we get back to a capped perfect threshold. The problem of minimizing the makespan of a graph with this model is noted $P|prec, var, frac, spdp\text{-}lin, \delta_j|C_{\max}$ and is studied in [55, 114].

Some of the algorithms presented in this paper also apply to some restricted variants of the problem. A notable one is the case of *moldable* tasks, which prohibits any variation in the set of processors used by a task: in this case, $p_i(t)$ must be constant on some time interval, and null elsewhere.

Other notations In the following, we will often use the *length* of a path, which is defined as the minimum time needed to complete all the tasks of this path, provided that an unlimited number of processors is available. This corresponds to the definition introduced in the Preliminaries section, where the duration of each task is set to w_i/Ω_i. The *critical path CP* of the graph and the *bottom-level* of a task follow from this definition.

Extension of McNaughton wrap-around rule

When scheduling tasks with perfect speedup, it is possible to remove the assumption of fractional allocation without degrading the makespan thanks to malleability, using the so-called "MacNaughton wrap-around rule" [111]. We adapt here this result to our model with two integer thresholds. For the sake of simplicity, the proof is presented only for two tasks but easily extends to any allocation.

Lemma 2.1. *Consider two tasks T_1 and T_2 sharing an integer number of processors p : T_1 (resp. T_2) is allocated a fractional number of processors p_1 (resp. p_2). We can produce a schedule with preemption where T_1 and T_2 are allocated integer numbers of processors at all times, and in which both tasks perform the same amount of work.*

Proof. Let C_{max} be the finish time of the original schedule. We build a schedule where $\lfloor p_1 \rfloor$ (resp. $\lceil p_2 \rceil$) processors are allocated to task T_1 (resp. T_2) during t units of time, and $\lceil p_1 \rceil$ (resp. $\lfloor p_2 \rfloor$) during $C_{max} - t$ units of time. t is chosen such that the area dedicated to task T_1 is the same as in the original allocation, which means:

$$t \lfloor p_1 \rfloor + (C_{max} - t) \lceil p_1 \rceil = C_{max} p_1,$$

so $t = C_{max}(\lceil p_1 \rceil - p_1)$. The same holds correspondingly for task T_2. This ensures that we can apply this transformation to both tasks without exceeding the processor limit.

Now, we want to prove that in the new allocation, T_1 performs the same amount of work (and correspondingly for T_2). We denote by $s(\cdot)$ the speedup function of task T_1. The work done on task T_1 is given by:

$$\begin{aligned} t \times s(\lfloor p_1 \rfloor) + (C_{max} - t) \times s(\lceil p_1 \rceil) &= C_{max} \times s(\lceil p_1 \rceil) - t(s(\lceil p_1 \rceil) - s(\lfloor p_1 \rfloor)) \\ &= C_{max}\left(s(\lceil p_1 \rceil) - \frac{\lceil p_1 \rceil - p_1}{\lceil p_1 \rceil - \lfloor p_1 \rfloor}(s(\lceil p_1 \rceil) - s(\lfloor p_1 \rfloor))\right). \end{aligned}$$

We know that:

$$\frac{\lceil p_1 \rceil - p_1}{\lceil p_1 \rceil - \lfloor p_1 \rfloor} = \frac{s(\lceil p_1 \rceil) - s(p_1)}{s(\lceil p_1 \rceil) - s(\lfloor p_1 \rfloor)},$$

because s is linear between $\lceil p_1 \rceil$ and $\lfloor p_1 \rfloor$, as the thresholds are integer. Finally, we get:

$$t \times s(\lfloor p_1 \rfloor) + (C_{max} - t) \times s(\lceil p_1 \rceil) = C_{max} \times s(p_1),$$

which completes the proof. □

Note that this may add a number of preemptions proportional to the number of tasks for each interval.

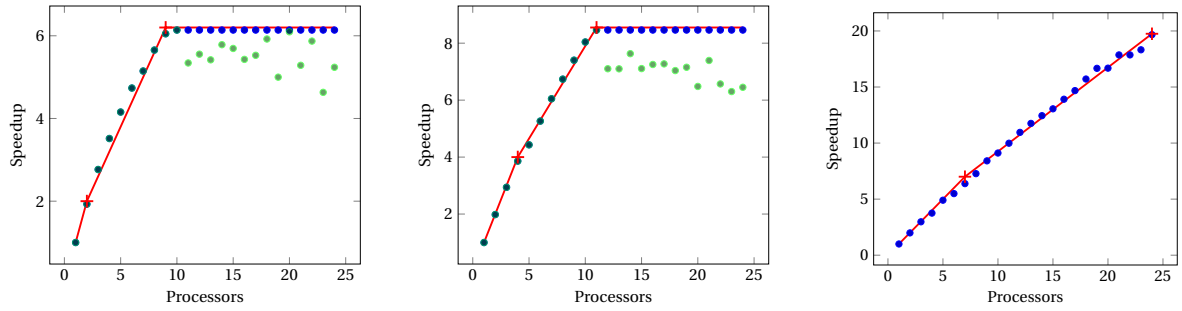
2.2 Experimental validation of the model

In this section, we show that the proposed model is realistic enough to model parallel tasks coming from an actual application. In order to do this, we ran tasks coming from linear algebra applications on a parallel platform of 24 cores. It consists of two Haswell Intel Xeon E5-2680 processors, each one containing 12 cores running at 3.30 GHz, and embeds 128 GB of DDR4 RAM (2133MHz).[1]

To compute the speedup graph as shown in Figure 2.1, we ran each task with $p = 1, \ldots, 24$ cores. Note that on this platform, the "number of processors" has to be understood as "number of cores". We first tested a dense numerical algebra routine: the dense Cholesky factorization. We noticed that the speedup of such dense tasks was perfect, i.e., equal to the number of cores used, up to using the full platform ($p = 24$). However, usual parallel applications are not made (only) of tasks with perfect parallelism, and our model precisely aims at taking these speedup limitations into account. Thus, we focused on another

[1] These experiments were carried out using the PlaFRIM experimental testbed (https://www.plafrim.fr/), being developed under the Inria PlaFRIM development action with support from Bordeaux INP, LABRI and IMB and other entities: Conseil Régional d'Aquitaine, Université de Bordeaux and CNRS (and ANR in accordance to the programme d'investissements d'Avenir).

linear algebra application, which is the multifrontal QR decomposition of sparse matrices as performed by QR_MUMPS [6], with 2D partitioning. Each task of this application is the QR decomposition of a dense rectangular matrix. Similar tasks have already been studied in the evaluation of the model used in Chapter 1, see Section 1.2. For the set of matrices described in Section 2.5, we computed the size of all the dense QR decompositions associated to its multifrontal QR decomposition. For each of the ten thousand resulting sizes, we timed such a task on $p = 1, \ldots, 24$ cores. Each timing was performed 5 times using random data and we retain the average performance.



(a) Matrix size 1984x1834 (b) Matrix size 1072x22385 (c) Matrix size 13007x15575

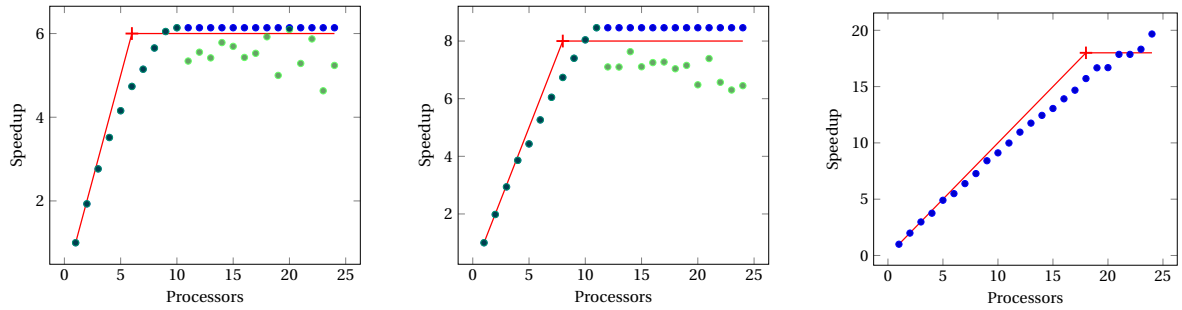
Figure 2.2: Speedup and fitted model for different matrix sizes.

In the following, we present the speedup of three tasks which are representative of all possible sizes. The first case, presented in Figure 2.2(a), corresponds to small matrix sizes. The green dots represent the actual speedup measured on the platform. Note that for up to 10 processors, the speedup increases with the number of processors and then the performance decreases and exhibits a larger variability for larger number of processors: when adding too many processors, more time is spent in communicating and synchronizing the processors, which hinders the performance. This behaviour is not unusual; a "smart" implementation of the task would be aware of this and would limit the number of processors to be used to 10, even if more processors are allocated to the task. Thus, we first transform the measured speedup so that it is never decreasing with the number of processors. Formally, this *corrected* speedup, plotted with blue dots on the figures, is given by:

$$correctedSpeedup(p) = \max_{k \leq p} measuredSpeedup(k)$$

In order to fit our speedup model to this corrected speedup, we computed the values of the parameters $(\delta^{(1)}, \delta^{(2)}, \Omega)$ that minimize the sum of the squares of the distance between the model and the corrected measurements for all p values between 1 and 24. Given the limited range of possible values for the thresholds (which are integers in $\{1, 2, \ldots, 24\}$) and maximum speedup (in $[1; 24]$), we decided to simply test all possible values of the parameters (with fixed precision for the maximum speedup) and select the ones that minimize the sum of residuals. The resulting speedup model is plotted in red.

Figures 2.2(b) and 2.2(c) plots the same measured speedup, corrected speedup and fitted model for larger matrices: a medium-size matrix on Figure 2.2(b) with one large dimension and one small, and a large matrix on Figure 2.2(c). As expected, we notice that the thresholds increase with the matrix size. For the larger matrices, the second threshold is set to 24 although it would probably be larger on a larger platform. Overall, we notice that the fitting of the model is very accurate, the median coefficient of determination being larger than 0.98 (a value of 1 means a perfect fit).

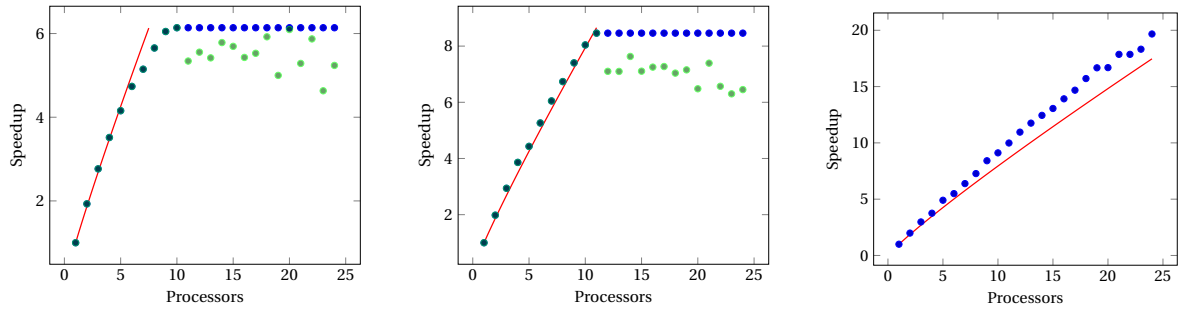


(a) Matrix size 1984x1834 (b) Matrix size 1072x22385 (c) Matrix size 13007x15575

Figure 2.3: Speedup and single-threshold model for different matrix sizes.

Some available algorithms for solving our problem consider the single-threshold variant presented in the previous section. Thus, we also fitted the previous (corrected) speedup measurements with this model, composed of a first perfect linear speedup and then a plateau where the speedup is equal to the threshold. Figure 2.3 plots the obtained speedup model using the same matrix sizes as Figure 2.2. We see that this model is less accurate, the median coefficient of determination being 0.90: it is too optimistic before the threshold, as the task is not perfectly parallel, and it is too pessimistic for many processors allocated, as better performance can be reached with a number of cores larger than the threshold.

We also studied the accuracy of the model adopted in Chapter 1, for which the speedup is equal to $p \mapsto p^{\alpha}$, where α is identical for all tasks. In Figure 2.4, we plotted the obtained model for the same matrix sizes as Figure 2.2, with a value of α equal to 0.9. This value was chosen to fit the timings of Figure 2.4(b) before the parallelism stalls. We can see that the model is accurate when few processors are allocated, but it then rapidly underestimates or overestimates the computing time for different matrix sizes. Other values of α would be more adequate, and the plateau is not correctly modeled.



(a) Matrix size 1984x1834 (b) Matrix size 1072x22385 (c) Matrix size 13007x15575

Figure 2.4: Speedup and fitted model of Chapter 1 for different matrix sizes, with $\alpha = 0.9$.

2.3 Problem complexity

Task malleability and perfect speedup make this problem much easier than most scheduling problems. However, quite surprisingly, adding limiting the possible parallelism with thresholds is sufficient to make it NP-complete. We restrict ourselves here to the model where both thresholds are equal ($\delta_i =$

$\delta_i^{(1)} = \delta_i^{(2)} = \Omega_i$ for all tasks i) as it is already NP-complete. Note that a similar result already appeared in [55], however its proof is more complex and not totally specified, which makes it difficult to check; this is why we propose this new proof.

Theorem 2.1. *The problem of minimizing the makespan is NP-complete.*

Proof. We start by proving that this problem belongs to NP. Without loss of generality, we restrict to schedules which allocate a constant share of processors to each task between any two task completions. Note that from a schedule that does not respect this condition, we can construct a schedule with the same completion times simply by allocating the average share of processors to each task in each such interval. Given a schedule that respects this restriction, it is easy to check that it is valid in time polynomial in the number of tasks.

To prove completeness, we perform a reduction from the 3SAT problem which is known to be NP-complete [66]. An instance $\mathscr{I}$ of this problem consists of a boolean formula, namely a conjunction of m disjunctive clauses, $C_1, \ldots, C_m$, of 3 literals each. A literal may either be one of the n variables $x_1 \ldots x_n$ or the negation of a variable. We are looking for an assignment of the variables which leads to a TRUE evaluation of the formula.

Instance definition From $\mathscr{I}$, we construct an instance $\mathscr{J}$ of our problem. This instance is made of $2n+1$ chains of tasks and $p = 3$ processors. The first $2n$ chains correspond to all possible literals of instance $\mathscr{I}$; they are denoted L_{x_i} or $L_{\overline{x_i}}$ and called *literal chains*. The last chain is intended to mimic a variable "processor profile", that is a varying number of available processors over time for the other chains, and is denoted by L_{pro}. Our objective is that for every *pair of literal chains* (L_{x_i} and $L_{\overline{x_i}}$), one of them starts at some time $t_i = 2(i-1)$ and the other at time $t_i + 1$. The one starting at time $t_i + 1$ will have the meaning of TRUE. We will construct the chains such that (i) no two chains of the same pair can start both at time $t_i + 1$ and (ii) at least one chain L_{x_i} or $L_{\overline{x_i}}$ corresponding to one of the three literals of any given clause starts at time $t_i + 1$.

For any chain, we consider its *critical path* length, that is, the minimum time needed to process it provided that enough processors are available. The makespan bound M of instance $\mathscr{J}$ is equal to the critical path length of the last chain L_{pro}, and will be specified later. Thus, to reach M, all tasks of L_{pro} must be allocated their threshold, and no idle time may be inserted between them.

In constructing the chains, we only use tasks whose weight is equal to their threshold, so that their minimum computing time is one. Then, a chain is defined by a list of numbers $[a_1, a_2, \ldots]$: the i-th task of the chain has a threshold and a weight a_i. As a result, the critical path length of a chain is exactly the number of tasks it contains. We define $\varepsilon = 1/4n$ and present the general shape of a literal chain L_a, where a is either x_i or $\overline{x_i}$:

$$L_a = [1, \varepsilon, \underbrace{\varepsilon, \ldots, \varepsilon}_{2(n-i)}, \underbrace{\mathit{SelectClause}(a)}_{2m \text{ tasks}}, \underbrace{\varepsilon, \ldots, \varepsilon}_{2(n-i)}, 1].$$

The leftmost and rightmost parts of the chain are dedicated to ensuring that in each pair of literal chains, one of them starts at time $t_i = 2(i-1)$ and the other at time $t_i + 1$. The central part of the chain is devoted to clauses, and ensures that for each clause, at least one chain corresponding to a literal of the clause starts at time $t_i + 1$:

$$\mathit{SelectClause}(a) = [\mathit{InClause}(C_1, a), \varepsilon, \ldots, \mathit{InClause}(C_m, a), \varepsilon],$$

where:

$$\mathit{InClause}(C_k, a) = \begin{cases} [1 - \frac{2}{3}n\varepsilon] & \text{if } a \text{ appears in } C_k \\ [\varepsilon] & \text{otherwise} \end{cases}$$

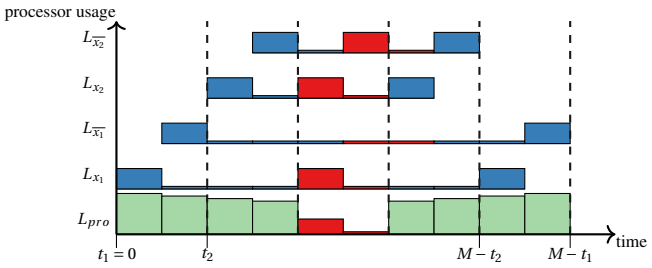


Figure 2.5: Example of a possible schedule for the instance $\mathscr{J}$ associated to the formula $x_1 \vee x_2 \vee \bar{x}_2$.

In total, the chain L_a includes $4n+2m-4i+3$ tasks. Finally, the profile chain is defined as follows:

$$L_{\text{pro}} = [\underbrace{2, 2-\varepsilon, \ldots, 2-(2n-1)\varepsilon}_{2n \text{ tasks}}, \underbrace{L_{10}, L_{10}, \ldots, L_{10}}_{2m \text{ tasks}}, \underbrace{2-(2n-1)\varepsilon, \ldots, 2-\varepsilon, 2}_{2n \text{ tasks}}],$$

where $L_{10} = [1-(\frac{2}{3}n-2)\varepsilon, 3\varepsilon]$. The critical path length of L_{pro} defines $M = 2m+4n$. Figure 2.5 presents a valid schedule for the instance corresponding to the formula $(x_1 \vee x_2 \vee \overline{x_2})$, which corresponds to the assignment $\{x_1 = x_2 = \text{FALSE}\}$.

From a truth assignment to a valid schedule We assume here that we are given a solution to $\mathscr{I}$, i.e. a truth assignment of its variables: let v_i denote the value of variable x_i in this assignment. We construct the following schedule for $\mathscr{J}$: for all chains, each task is allocated a number of processors equal to its threshold and no idle time is inserted between any two consecutive tasks. Chain L_{pro} starts at time 0 while chain L_{x_i} (respectively $L_{\overline{x_i}}$) starts at time t_i+1 if v_i is TRUE (resp. FALSE), otherwise it starts at time t_i. It is straightforward to check that this schedule is valid. Here, we only concentrate on the most critical part, namely the central part which corresponds to the clauses. We count the number of processors used during time interval $[2n+2(k-1), 2n+2k]$ which corresponds to clause C_k:

- In the first half of this interval, at most 2 literal chains can have a task of size $1-\frac{2}{3}n\varepsilon$ since at least one in the three literals of the clause is true. Together with the other literal chains and the profile, the maximum processor occupancy is at most (remember $\varepsilon = 1/4n$):

$$\underbrace{2\left(1-\frac{2}{3}n\varepsilon\right)}_{\text{FALSE literal chains}} + \underbrace{(2n-2)\varepsilon}_{\text{other literal chains}} + \underbrace{1-\left(\frac{2}{3}n-2\right)\varepsilon}_{L_{\text{pro}}} = 3.$$

  (This is maximum, because $1-\frac{2}{3}n\varepsilon > \varepsilon$.)
- In the second half of this interval, at most 3 literal chains can have a task of size $(1-\frac{2}{3}n\varepsilon)$, which may result in a maximum number of busy processors of:

$$\underbrace{3\left(1-\frac{2}{3}n\varepsilon\right)}_{\text{TRUE literal chains}} + \underbrace{(2n-3)\varepsilon}_{\text{other literal chains}} + \underbrace{3\varepsilon}_{L_{\text{pro}}} = 3.$$

The resulting schedule has a makespan of M and is thus a solution to $\mathscr{J}$.

From a valid schedule to a truth assignment We now assume that instance $\mathscr{J}$ has a valid schedule S and we aim at reconstructing a solution for $\mathscr{I}$. First we prove some properties on the starting times of chains through the following lemma.

Lemma 2.2. *In any valid schedule S for $\mathcal{J}$,*

i. each pair of chains $L_{x_i}, L_{\overline{x_i}}$ is completely processed during time interval $[t_i, M - t_i]$,
ii. one of them is started at time t_i and the other one at time $t_i + 1$,
iii. all tasks of both chains are allocated their threshold,
iv. there is no idle time between any two consecutive tasks of each chain.

Proof. The proof is done by induction on i, by carefully checking when the first and last tasks of chains $L_{x_i}, L_{\overline{x_i}}$ may be scheduled, given the resources which are not used by the previous chains and by L_{pro}.

Base case: Consider $i = 1$. The critical path of chains L_{x_1} and $L_{\overline{x_1}}$ is $4n + 2m - 4i + 3 = 4n + 2m - 1$. With $M = 4n + 2m$, both have to start in the interval $[0, 1]$.

The following discussion of the base case is written in general terms (that is for any i) to reuse it in the inductive step, but applies here for $i = 1$, with $t_1 = 0$.

We consider the first task of chain L_{x_i} and the first task of chain $L_{\overline{x_i}}$. Both tasks have weight 1. Let A denote the first of these two tasks to complete (at a time t_A) and let B be the other one (which completes at time t_B). Given the $2(i-1)$ chains already scheduled (none for $i = 1$), the number of processors available during interval $[t_i, t_i + 1]$ is 1 and during interval $[t_i + 1, t_i + 2]$ is $1 + \varepsilon$. A and B both complete at or after time $t_i + 1$. We note $t_A = t_i + 1 + \Delta_1$ and $t_B = t_i + 1 + \Delta_1 + \Delta_2$ ($\Delta_1 \geq 0$ and $\Delta_2 \geq 0$). Note that because of the critical path length of the remaining tasks of both chains and the limited time span, $\Delta_1 \leq 1$ and $\Delta_1 + \Delta_2 \leq 1$. Figure 2.6 illustrates the previous notations and the amount of processors available for tasks A and B (note that after time t_A, B may use only $\delta_B = 1$ processor).

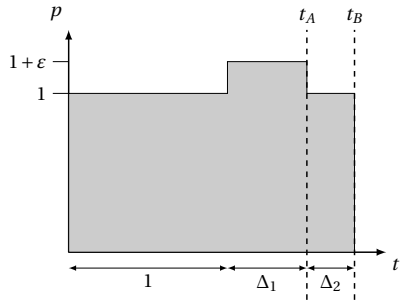


Figure 2.6: Illustration of the notations used in Lemma 2.2.

Since $w_A + w_B = 2$ work units have to be performed before time t_B, we have:

$$1 + \Delta_1(1 + \varepsilon) + \Delta_2 \geq 2,$$

and thus $\Delta_2 \geq 1 - \Delta_1(1 + \varepsilon)$ and $t_B \geq t_i + 2 - \Delta_1\varepsilon$.

We symmetrically apply the same reasoning to the last tasks C and D of these two chains, and their *starting* times t_C and t_D, assuming that C is started before D. By setting $t_D = M - t_i - 1 - \Delta_1'$, we get $t_C \leq M - t_i - 2 + \Delta_1'\varepsilon$. We distinguish between two cases, depending on the chains to which A, B, C, and D belong to:

- In the first case, we assume that A and D belong to the same chain. We consider the other chain, containing B and C. Because exactly $4(n-i) + 2m + 1$ tasks need to be processed between these two tasks, we have:

  $$t_C \geq t_B + 4(n - i) + 2m + 1,$$

  which gives:

  $$\Delta_1'\varepsilon \geq 1 - \Delta_1\varepsilon.$$

We have $\Delta_1 \leq 1$ and similarly, $\Delta'_1 \leq 1$. Together with the previous inequality, this gives $\varepsilon \geq 1/2$ which is not possible since $\varepsilon = 1/4n$. Hence B and C cannot belong to the same chain.

- In the second case, we consider that A and C belong to the same chain. Because exactly $4(n-i)+2m+1$ tasks need to be processed between A and C (and between B and D), we have:

$$t_C \geq t_A + 4(n-i) + 2m + 1 \text{ and } t_D \geq t_B + 4(n-i) + 2m + 1,$$

which gives:

$$2m + 4n - t_i - 2 + \Delta'_1\varepsilon \geq t_i + 1 + \Delta_1 + 4(n-i) + 2m + 1,$$

and

$$2m + 4n - t_i - 1 - \Delta'_1 \geq t_i + 2 - \Delta_1\varepsilon + 4(n-i) + 2m + 1,$$

which are simplified (using $t_i = 2(i-1)$) into:

$$\Delta'_1\varepsilon \geq \Delta_1 \text{ and } \Delta'_1 \leq \Delta_1\varepsilon.$$

This leads to $\Delta_1 \leq \Delta_1\varepsilon^2$. As $0 < \varepsilon < 1$, we have $\Delta_1 = 0$, so $t_A = t_i + 1$. Then, no processor can be allocated to B during $[t_i, t_{i+1}]$.

In other words, one task among the first task of L_{x_i} and the first task of $L_{\overline{x_i}}$ is fully processed during interval $[t_i, t_i + 1]$ and the other one is not processed before $t_i + 1$. Because of its critical path length, the chain starting second must be processed at full speed (each task being allocated a number of processors equal to its threshold) and without idle time in the interval $[t_i + 1, M - t_i]$. The last task of the chain starting at time t_i must then be completed at time $M - t_i - 1$ and thus this chain must also be processed at full speed and without idle time. This also implies that all available processors are used in the intervals $[t_i, t_i + 2]$ and $[M - t_i - 2, M - t_i]$.

Inductive step: Now assume that the lemma holds for $i-1$. With $t_1 = 0$ and the inductive property on the last observation we know that no processor is available for chains L_{x_i} and $L_{\overline{x_i}}$ before $2(i-1)$ and after $M - 2(i-1)$. The time span available for the remaining chains is thus $4n + 2m - 4i + 4$ while the critical path of chains L_{x_i} and $L_{\overline{x_i}}$ is $4n + 2m - 4i + 3$: these chains cannot be started after $2(i-1)+1$ to be completed within the time span. Setting $t_i = 2(i+1)$ we reuse the above argument about the scheduling of the two chains L_{x_i} and $L_{\overline{x_i}}$, which proves (i)-(iv). □

For each literal chain which starts at time $t_i + 1$, we associate the value TRUE in an assignment of the variables of $\mathscr{I}$, and we associate the value FALSE to all other literals. Thanks to the previous lemma, we know that exactly one literal in the pair $(x_i, \overline{x_i})$ is assigned to TRUE. Furthermore, not three tasks of size $1 - \frac{2}{3}n\varepsilon$ can be scheduled at time $2n + 2(k-1)$ because of the profile chain L_{pro}, as this would lead to a number of occupied processors of:

$$\underbrace{3\left(1 - \frac{2}{3}n\varepsilon\right)}_{\text{3 FALSE literal chains}} + \underbrace{(2n-3)\,\varepsilon}_{\text{other literal chains}} + \underbrace{1 - \left(\frac{2}{3}n - 3\right)\varepsilon}_{L_{\text{pro}}} = 4 - \frac{2}{3}n\varepsilon = 4 - \frac{1}{6} > 3 = p.$$

Thus, at least one literal of each clause is set to TRUE in our assignment. This proves that it is a solution to $\mathscr{I}$. □

2.4 Heuristics description and approximation analysis

We now move to the description and analysis of three heuristics. Two of them come from the literature (PROPMAPPING and FLOWFLEX) while a third one, called GREEDYFILLING is novel. For both pre-existing heuristics, we present their original version as well as some optimizations for our model.

2.4.1 Performance analysis of Proportional Mapping

A widely used algorithm for this problem, which has been implemented in the simulations of Chapter 1, is known as "proportional mapping" [120]. In this algorithm, a sub-graph is allocated a number of processors that is proportional to the ratio of its weight to the sum of the weights of all sub-graphs under consideration. Based on the structure of the considered SP-graph G, Algorithm 3 allocates a share of processors to each sub-graph and eventually each task. Any given graph G (with SP-graph characteristics) can be decomposed into its series and parallel components using an algorithm from [141], and thus be processed by Algorithm 3. Observe that thresholds are not considered in this proportional mapping.

Algorithm 3: PROPMAPPING $(G = (V, E, w), p)$

if *top-level composition is series composition of K sub-graphs* **then**
  Allocate $p_k = p$ processors to each subgraph
else top-level composition is parallel composition of K sub-graphs $G_1, \ldots, G_K$
  Allocate $p_k = \frac{w_{G_k}}{\sum_j w_{G_j}} p$ processors to subgraph G_k, $1 \leq k \leq K$, where w_{G_k} is the weight of G_k
Call PROPMAPPING (sub-graph k, p_k) for each sub-graph k

The schedule corresponding to this proportional mapping simply starts each task as soon as possible (i.e., after all its predecessors have completed), as given in Algorithm 4.

Algorithm 4: PROPSCHEDULING $(G = (V, E, w), p)$

Call PROPMAPPING (G, p) to determine p_i for each task $T_i \in V$
foreach $T_i \in V$ **do**
  Start T_i with p_i processors as soon as possible, i.e., after all its predecessors completed

Indeed, given the proportional mapping of processors, there are always enough processors available to do that. It is worth noting that the created schedule is compatible with the moldable model: each task uses the same number of processors throughout its entire execution. As such, Algorithm 4 can also be used for the moldable model.

In the case of perfect parallelism (i.e., $\delta_i^{(1)} \geq p$, $\forall T_i \in G$), there is no idle time as all tasks of a parallel composition terminate at exactly the same time (due to the proportional mapping). Hence, this schedule achieves the optimal makespan $C_\infty = \frac{1}{p}\sum_{i \in V} w_i$. For the general case the following theorem holds.

Theorem 2.2. PROPSCHEDULING *is a $(1 + r)$-approximation algorithm for makespan minimization where $r = \max_i \delta_i^{(2)}/\Omega_i$.*

Proof. We first note that the optimal makespan when all tasks have perfect parallelism, C_∞, is a lower bound on the optimal makespan with thresholds C_{OPT}. We have thus $C_\infty \leq C_{\text{OPT}} \leq C_{max}$, where C_{max} is the makespan obtained by PROPSCHEDULING, and we want to show that $C_{max} \leq (1 + r)C_{\text{OPT}}$.

The critical path cp of G, as defined in Section 2.1, is a longest path in G, where the length is defined as the sum of the work of each task on the path divided by its maximum speedup, $len(cp) = \sum_{i \in cp} \frac{w_i}{\Omega_i}$. Naturally, the critical path length is another lower bound on the optimal makespan, $len(cp) \leq C_{\text{OPT}}$.

Consider the schedule produced by PROPSCHEDULING. There is at least one path Φ in G from the entry task to the exit task, with no idle time between consecutive tasks. In other words, on Φ the execution of a task starts when the execution of the preceding task finishes. Such a path always exists because we start tasks as early as possible, so this property is always true between the tasks of a serial

composition, and it is true for at least one task in each parallel composition. The execution length of Φ is the makespan C_{max}, because it includes no idle time and that it goes from entry to exit task. It is given by:

$$C_{max} = \sum_{i \in \Phi} \frac{w_i}{s_i\left(\min\left\{p_i, \delta_i^{(2)}\right\}\right)}.$$

Let us divide the tasks of Φ into two sets: the set A of tasks executed with their threshold processors $p_i = \delta_i^{(2)}$ and the set B of tasks executed with the allocated number of processors $p_i < \delta_i^{(2)}$, with $A \cup B = \Phi$. We then have:

$$C_{max} = \sum_{i \in A} \frac{w_i}{s_i\left(\delta_i^{(2)}\right)} + \sum_{i \in B} \frac{w_i}{s_i(p_i)} = \sum_{i \in A} \frac{w_i}{\Omega_i} + \sum_{i \in B} \frac{w_i}{s_i(p_i)}.$$

The first term (called C_A) is per definition smaller than or equal to the length of the critical path $len(cp)$. The second term (C_B) consists only of tasks that are executed with their proportionally allocated processors. Therefore, these tasks are allocated as many processors as in the schedule achieving the optimal makespan C_∞ when ignoring thresholds. This means that

$$C_\infty \geq \sum_{i \in B} \frac{w_i}{p_i} = \sum_{i \in B} \frac{w_i}{s_i(p_i)} \frac{s_i(p_i)}{p_i} \geq \min_{i \in B}\left(\frac{s_i(p_i)}{p_i}\right) C_B.$$

With the definition of the s_i's given in Equation 2.1, we know that functions $x \mapsto \frac{s_i(x)}{x}$ are non-increasing, so $\min_{i \in B}\left(\frac{s_i(p_i)}{p_i}\right) \geq \frac{1}{r}$. Therefore,

$$r C_\infty \geq C_B.$$

We then get the desired inequality:

$$M \;\leq\; len(cp) + r C_\infty \;\leq\; (1+r) C_{\text{OPT}}.$$

□

The complexity of PROPSCHEDULING is $O(|V|)$, as it consists of a simple traversal of the SP-graph.

2.4.2 Optimizations of Proportional Mapping

The main drawback of PROPMAPPING is that it assumes perfect speedup. When applied to actual tasks with imperfect speedup functions, some tasks may finish later than expected by the algorithm. In some cases, sibling tasks (tasks that share the same successor) may complete earlier, thus leaving some processors idle, which induces performance loss. In order to address this issue, a natural idea is to redistribute the processors left idle by the termination of some task T_i to T_i's siblings, that is, to the tasks that share the same successor T_j and are still running. This is for example what is done in [83]. We design such an algorithm, called PROPMAPREBALSIBLINGS, which redistributes the processing power of terminated tasks to their siblings, proportionally to the weight of the target tasks.

Note that both the original PROPMAPPING or this optimization are agnostic of both thresholds. Thus, we introduce a new variant of PROPMAPPING called PROPMAPREBALTHRESHOLD that takes advantage of the speedup model introduced above. It also consists of redistributing processors left idle when a task terminates while its successor is not ready yet. The main difference with the previous variant is that idle processors are not redistributed only to siblings, but to all currently running tasks for which $p_i < \delta_i^{(2)}$. Again, the redistribution is done according to the weight of the tasks. Both variants are detailed in Algorithm 5 and Algorithm 6, and have a complexity of $O(|V|^2)$.

Algorithm 5: PROPMAPREBALSIBLINGS ($G = (V, E, w)$, p)

```
Call PROPMAPPING (G, p) to determine p_i for each task T_i ∈ G
FreeTasks ← source tasks
while FreeTasks ≠ ∅ do
    t ← time when the first task T_k ∈ FreeTasks is completed using p_k processors
    foreach task T_i ∈ FreeTasks do
        allocate p_i processors to T_i until time t
    FreeTasks ← FreeTasks \ {T_k}
    foreach task T_i ∈ FreeTasks siblings of T_k do
        p_i ← p_i + share of p_k proportional to the weight of T_i
    foreach T' ∈ successors(T_k) such that T' has no unprocessed predecessors do
        FreeTasks ← FreeTasks ∪ {T'}
```

Algorithm 6: PROPMAPREBALTHRESHOLD ($G = (V, E, w, \delta^{(2)})$, p)

```
Call PROPMAPPING (G, p) to determine p_i for each task T_i ∈ G
FreeTasks ← source tasks
while FreeTasks ≠ ∅ do
    foreach task i ∈ FreeTasks do Surplus_i ← 0
    Surplus ← p − Σ_{i∈FreeTasks} p_i
    foreach T_i such that p_i < δ_i^(2) do
        p'_i ← p_i + (share of Surplus proportional to the weight of T_i)
    t ← time when the first task T_k ∈ FreeTasks is completed with p'_k processors
    foreach task i ∈ FreeTasks do
        allocate p'_i processors to T_i until time t
    FreeTasks ← FreeTasks \ {T_k}
    foreach T' ∈ successors(T_k) such that T' has no unprocessed predecessors do
        FreeTasks ← FreeTasks ∪ {T'}
```

2.4.3 A novel algorithm: Greedy-Filling

Proportional mapping is a common approach. However, it does not make use of the malleability of tasks and it is restricted to SP-graphs. In this section we study an algorithm, called GREEDYFILLING, which may schedule any DAG and takes advantage of the tasks' malleability. It considers one task at a time and greedily allocates it the largest possible processing power.

We now detail this algorithm, presented in Algorithm 7. First, each task is given a priority. In practice, we use for each task T_i its bottom-level, as it is a lower bound on the overall completion time once task T_i has started. The algorithm builds the schedule in chronological order while maintaining the set of free tasks. The difference is that, instead of sharing the resources according to the weight of tasks, we consider them in the order defined by priorities. We allocate each task T_i up to $\delta_i^{(1)}$ processors if possible, so as to stay in the perfect parallelism zone. If there are processors in excess, we reconsider the tasks in the same order, increasing their allocation up to $\delta_i^{(2)}$.

It is interesting to note that since the total number of processors p and all thresholds are integers ($\forall T_i \in V,\ \delta_i^{(1)}, \delta_i^{(2)} \in \mathbb{N}$), all allocated processors p_i are integers too.

Algorithm 7: GREEDYFILLING ($G = (V, E, w, \delta^{(1)}, \delta^{(2)})$, p)

```
1  Assign a priority to each task T_i ∈ V
2  FreeTasks ← source tasks
3  while FreeTasks ≠ ∅ do
4      Sort FreeTasks by non-increasing priorities;
5      for each task i in FreeTasks do
6          allocate at most δ_i^(1) processors to task i without exceeding p in total
7      if some processors are not yet allocated then
8          for each task i in FreeTasks do
9              allocate at most δ_i^(2) processors to task i without exceeding p in total
10     Schedule tasks until some task T_k completes
11     Remove T_k from FreeTasks, add its successors whose predecessors have all completed
```

Theorem 2.3. GREEDYFILLING *is a* $1 + r - \frac{\delta^{(2)}_{\min}}{p}$ *approximation for makespan minimization, with* $\delta^{(2)}_{\min} = \min_{T_i \in V} \delta^{(2)}_i$ *and* $r = \max_{T_i \in V} \delta^{(2)}_i / \Omega_i$.

Proof. This proof is a transposition of the classical proof by Graham [72]. In any schedule produced by GREEDYFILLING, let T_1 be a task whose completion time is equal to the completion time of the whole task graph. We consider the last time t_1 prior to the start of the execution of T_1 at which not all processors were fully used. If the execution of T_1 did not start at time t_1 this is only because at least one ancestor T_2 of T_1 was executed at time t_1. Then, by induction we build a dependence path $\Phi = T_k \rightarrow \ldots \rightarrow T_2 \rightarrow T_1$ such that all processors are fully used during the execution of the entire schedule except, maybe, during the execution of the tasks of Φ.

We consider the execution of any task T_i of Φ. At any time during the execution interval(s) of T_i (due to malleability it might be executed in disconnected intervals), either all processors are fully used, or some processors are (partially) idle and then, because of Step 9, $\delta^{(2)}_i$ processors are allocated to T_i. Therefore, during the execution of T_i, the total time during which not all processors are fully used is at most equal to w_i/Ω_i and there are at most $p - \delta^{(2)}_i$ idle processors. Let *Idle* denote the sum of the idle areas in the schedule, i.e., idle periods multiplied by idle processors. Then we have:

$$Idle \leq \sum_{i=1}^{k} \left(\frac{w_i}{\Omega_i} \times \left(p - \delta^{(2)}_i \right) \right) \leq \left(p - \delta^{(2)}_{\min} \right) \times \sum_{i=1}^{k} \frac{w_i}{\Omega_i} \leq \left(p - \delta^{(2)}_{\min} \right) len(cp) \leq \left(p - \delta^{(2)}_{\min} \right) C_{\text{OPT}}.$$

Let *Used* denote the sum of the busy areas in the schedule. As s_i is concave (cf. Equation 2.1), the busy area dedicated to schedule the task T_i is maximized when T_i is allocated to $\delta^{(2)}_i$ processors. Then, the area is equal to $\delta^{(2)}_i w_i/\Omega_i$ and:

$$Used \leq \sum_{T_i \in V} \frac{\delta^{(2)}_i w_i}{\Omega_i}.$$

Now, let $r = \max_i \frac{\delta^{(2)}_i}{\Omega_i}$. Note that $C_{\text{OPT}} \leq \sum_i \frac{w_i}{p}$. Then we have:

$$Used \leq \sum_{T_i \in V} w_i r \leq r p C_{\text{OPT}}.$$

The makespan of the considered schedule is then equal to:

$$C_{max} = \frac{1}{p}(Idle + Used) \leq \left(1 + r - \frac{\delta^{(2)}_{\min}}{p}\right) C_{\text{OPT}}.$$

□

Note that the above proof makes little reference to how the schedule of G has been constructed. The only important characteristic is that the algorithm never leaves a processor deliberately idle if there are tasks that could be scheduled. Hence, the above approximation factor will also apply to other algorithms which adhere to that characteristic:

Corollary 2.1. *Any scheduling algorithm which never deliberately leaves a processor idle if it could benefit to any available task is a* $1 + r - \delta^{(2)}_{\min}/p$ *approximation for makespan minimization, with* $\delta^{(2)}_{\min} = \min_{T_i \in V} \delta^{(2)}_i$ *and* $r = \max_i \delta^{(2)}_i / \Omega_i$.

The proof of Theorem 2.3 can easily be adapted to the single threshold model, which gives the following result. This is particularly useful to prove that the FLOWFLEX algorithm, presented below, is an approximation algorithm.

Corollary 2.2. *In the single threshold model* ($\delta_i = \delta^{(1)}_i = \delta^{(2)}_i = \Omega_i$), *any scheduling algorithm which never deliberately leaves a processor idle if it could benefit to any available task is a* $2 - \frac{\delta_{\min}}{p}$ *approximation for makespan minimization where* $\delta_{\min}$ *is the smallest threshold among all tasks.*

The complexity of GREEDYFILLING is $O(|V|^2)$ as the main loop is iterated $O(|V|)$ times, going over $O(|V|)$ tasks each time. The total management and ordering of *FreeTasks* can be done in $O(|V|\log|V|)$, e.g. with a priority queue.

2.4.4 The FlowFlex algorithm

We now introduce FLOWFLEX, a scheduling algorithm introduced in [114] and designed for a model similar to the single threshold variant described in Section 2.1, which considers that $\delta_i = \delta^{(1)}_i = \delta^{(2)}_i = \Omega_i$ for all tasks i. FLOWFLEX first allocates to each task its maximal number of processors δ_i, as if there was an infinite number of processors available. Then, in each time interval where the allocation is constant, if the total number of allocated processors exceeds p, the allocation is scaled down proportionally. This algorithm is detailed in Algorithm 8.

In its original version, FLOWFLEX assumes a perfect speedup before the threshold. Thus, scaling down the shares of the tasks proportionally preserves the simultaneous completion of the amount of work performed in the original interval. This is no longer true with imperfect speedup functions. This is why we introduce an optimized version FLOWFLEXREBALANCE that redistributes idle processors among running tasks once a task completes the amount of work it had to process in a given interval. The redistribution is done proportionally to the thresholds δ_i, which corresponds to the original allocation before scaling. This optimized variant is described in Algorithm 9.

The complexity of FLOWFLEX is $O(|V|^2)$ as there are at most $|V|$ constant intervals, and each iteration of the main loop is linear in $|V|$. The complexity of FLOWFLEXREBALANCE $O(|V|^3)$ as the redistribution procedure is done linearly in $|V|$.

Algorithm 8: FLOWFLEX ($G = (V, E, w, \delta)$, p)

$\mathscr{S} \leftarrow$ schedule obtained by allocating δ_i processors to T_i and starting tasks as soon as they are free
Sort tasks by non-decreasing completion times t_i, with $t_0 = 0$
$t \leftarrow 0$
for $i = 0$ *to* $n-1$ **do**
  foreach T_j **do** $work_j \leftarrow$ amount of work completed by task T_j in interval $[t_i; t_{i+1}]$ of $\mathscr{S}$
  $L \leftarrow$ set of tasks T_j such that $work_j > 0$
  foreach T_j *in* L **do**
    $p_j \leftarrow p \times \delta_j / \sum_{k \in L} \delta_k$
    Starting at time t, allocate p_j processors to T_j until it completes $work_j$ at some time t'_j
  $t \leftarrow \max_{j \in L} t'_j$

Algorithm 9: FLOWFLEXREBALANCE ($G_{DAG} = (V, E, w, \delta)$, p)

$\mathscr{S} \leftarrow$ schedule obtained by allocating p_i processors to T_i and starting tasks as soon as they are available
Sort tasks by non-decreasing completion times t_i, with $t_0 = 0$
$t \leftarrow 0$
for $i = 0$ *to* $n-1$ **do**
  foreach T_j **do** $work_j \leftarrow$ amount of work completed by task T_j in interval $[t_i; t_{i+1}]$ of $\mathscr{S}$.
  $L \leftarrow$ set of tasks T_j such that $work_j > 0$
  foreach T_j *in* L **do**
    $p_j \leftarrow p \times \delta_j / \sum_{k \in L} \delta_k$
  repeat
    Starting at time t, allocate p_j processors to each $T_j \in L$ until one task T_ℓ completes a work of $work_\ell$
    $t \leftarrow$ time when task T_l has completed the work $work_\ell$
    Remove T_ℓ from L
    foreach T_j *in* L **do**
      $p_j \leftarrow p_j + p_l \times \delta_j / \sum_{k \in L} \delta_k$
    Redistribute p_k over the p_j of the tasks of L, proportionally to their threshold $\delta^{(2)}$
  until L *is empty*

2.5 Experimental comparison

In this section we compare through simulation the new heuristic (GREEDYFILLING), reference heuristics (PROPMAPPING and FLOWFLEX) and the proposed extensions of these reference heuristics (PROPMAPREBALTHRESHOLD, PROPMAPREBALSIBLINGS and FLOWFLEXREBALANCE). These simulations use either synthetic graphs of synthetic tasks, or actual task trees whose task execution times were recorded through actual executions, as detailed below. Each algorithm has been simulated in C++: given a graph of tasks, and a speedup function for each task, the schedule is computed. We compare all heuristics through their makespan (total completion time).

2.5.1 Datasets

First, we consider a set of 30 synthetic random SP-graphs composed each of 200 nodes. In order to compute a random SP-graph of $x > 1$ nodes, we follow the following recursive strategy: toss k uniformly in $[1, x-1]$; with a probability of $1/2$, build a series composition of two random SP-graphs of respectively

k and $x-k$ nodes and, otherwise, build a parallel composition of these graphs. Then, in order to generate a random task (i.e., a random graph of $x = 1$ node), we choose its weight w uniformly in $[1;1000]$. The first threshold, $\delta^{(1)}$, is defined by $\delta^{(1)} = \lceil w/100 \rceil$; hence, $\delta^{(1)} \in [1;10]$. The second threshold, $\delta^{(2)}$, is uniformly drawn in $[\delta^{(1)}, 2\delta^{(1)}]$. The slope between the thresholds is uniformly drawn in $[0.5, 1]$. Therefore, in this dataset (called SYNTH), each task perfectly follows our speedup model.

Second, we consider a set of 24 trees whose size vary from 39 to 5900 nodes. These elimination trees have been generated (with either `colamd` [51] or `scotch` [117] ordering) using QR_MUMPS [6] on matrices from the University of Florida Sparse Matrix Collection [52], such that each task of a tree corresponds to the dense QR factorization of the associated matrix. The completion time of a task solely depends on the dimensions of the matrix. In order to determine the actual behavior of such a task, we benchmarked the time necessary to perform this task for a number of processors ranging from 1 to 24, as detailed in Section 2.2. Thus, in this dataset (called TREES), a task is characterized both by its parameters in our model ($\delta^{(1)}$, $\delta^{(2)}$ and Ω) and by the set of its completion times recorded through actual executions for up to 24 processors. The actual execution times are used in the experiments to determine the finish times of the scheduled trees (makespans).

2.5.2 Results

In order to compare the performance of these algorithms, we use a generic tool called *performance profile* [54]. For a given dataset, we compute the performance of each heuristic on each graph and for each considered value for the total number of available processors (namely 1, 2, 4, 6, 8, 10, 12, 16, 20 and 24). Then, instead of computing an average above all the cases, a performance profile reports a cumulative distribution function. Given a heuristic and a deviation τ expressed in percentage, we compute the fraction of test cases in which the performance of this heuristic is at most τ% larger than the best observed performance, and plot these results. Therefore, the higher the curve, the better the method: for instance, for a deviation $\tau = 5\%$, the performance profile shows how often a given method lies within 5% of the smallest makespan obtained.

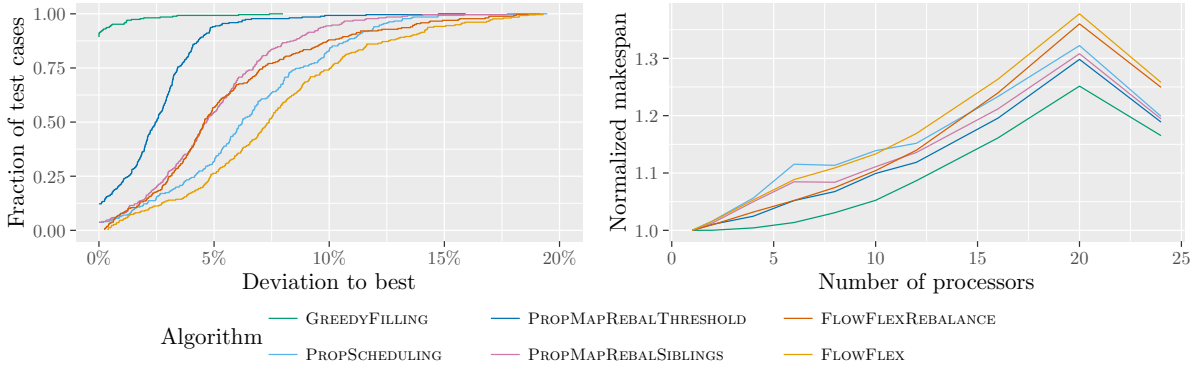


Figure 2.7: Performance profiles for up to 24 processors on SYNTH (left, where the best performance is top-left) and performance of the heuristics on a sample graph (right, where the best performance is bottom). Note that the following figures use the same legend.

In Figure 2.7, we present the performance profiles for the SYNTH dataset on the left, and the makespan obtained by each heuristic on a sample graph on the right. On this latter plot, the y-axis has been normalized by the classical lower bound on makespan: the maximum of the critical path and of the total work divided by the number of processors.

The first result is that GREEDYFILLING clearly outperforms the other algorithms: it has the best result in almost 95% of the test cases. On the other hand of the spectrum, FLOWFLEX and PROPMAPPING are the two worst heuristics. Both of them are clearly outperformed by their variants. Of all these variants, the best one is obviously PROPMAPREBALTHRESHOLD which achieves very good performance. Although the difference with GREEDYFILLING is striking, one should remark that PROPMAPREBALTHRESHOLD achieves a makespan within 5% of the best one in more than 93% of the instances. The overall hierarchy could have been expected as GREEDYFILLING is the only heuristic to be aware of both thresholds, and among the other, only PROPMAPREBALTHRESHOLD makes use of $\delta^{(2)}$. In turn these results suggest that the proposed speedup model with two thresholds can be used effectively to shorten the produced schedules.

The right-hand side of Figure 2.7 presents the typical results for a sample graph. The respective performance of heuristics is roughly independent of the number of available processors, and GREEDYFILLING presents the best results. Overall, the shape of the curves were predictable: when there are very few available processors, there is little possibility of wasting computational resources and all heuristics achieve near-perfect performance; when the number of processors is very large all heuristics that are aware of the second threshold provide similar processor allocation and achieve similar near-perfect performance. The hardest part is in the intermediate zone when the most significant differences can be observed.

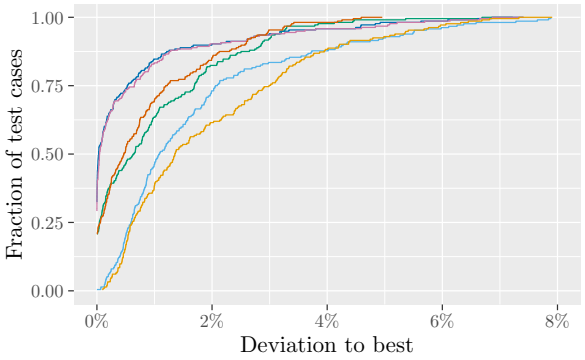
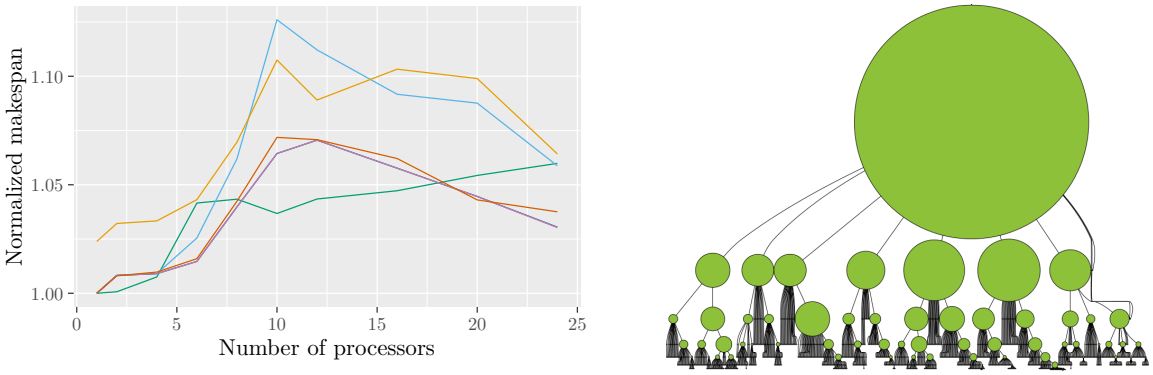


Figure 2.8: Performance profiles for up to 24 processors on TREES. Note the scale difference to Figure 2.7.

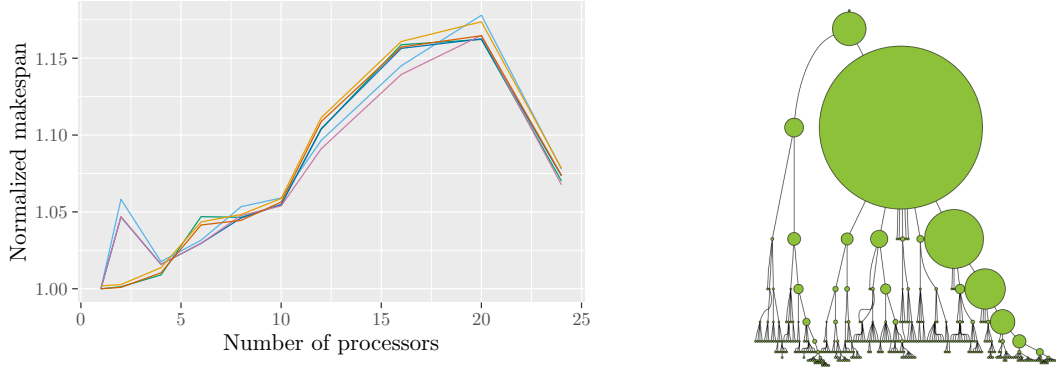
We present the performance profile for the TREES dataset in Figure 2.8, with additional representative samples in Figures 2.9(a) and 2.9(b). The legend of these graphs is the same as the one of Figure 2.7. The first observation to be made is that the difference between the graphs has significantly decreased. For each of the four heuristics GREEDYFILLING, PROPMAPREBALTHRESHOLD, PROPMAPREBALSIBLINGS, and FLOWFLEXREBALANCE, in 80% of the cases the deviation is at most 2% and in 63% of the cases it is at most 1%.

An explanation for this is that the trees of this dataset often contain a task (near the root one) whose completion time is far beyond the rest of the graph, as illustrated on the right in Figures 2.9(a) and 2.9(b).

Within the small difference between the algorihtm, the results are similar to the previous data set (ignoring GREEDYFILLING for the moment): FLOWFLEX and PROPMAPPING are the two worst heuristics; both heuristics are clearly outperformed by their variants; PROPMAPREBALTHRESHOLD achieves the best performance among these variants, but this time the performance of PROPMAPREBALSIBLINGS is almost indistinguishable from that of PROPMAPREBALTHRESHOLD. GREEDYFILLING also performs better than the previously proposed algorithms FLOWFLEX and PROPMAPPING, but its rel-



(a) Tree *lp-nug30-colamd*.



(b) Tree *GL7-d24-scotch*.

Figure 2.9: Performance of the heuristics and visual representation of two trees of the TREES dataset, where the area of a node is proportional to its sequential execution time.

ative performance compared with the proposed variants has changed: its performance on actual trees (Figure 2.8) is now slightly behind these variants, when it was clearly the best solution on synthetic ones (Figure 2.7). The better performance of PROPMAPREBALSIBLINGS compared to GREEDYFILLING may be surprising because PROPMAPREBALSIBLINGS does not have any knowledge on the computed (estimated) thresholds. This performance is actually due to the structure of the graphs, as detailed below.

One should recall that the performance profiles gather results over the whole dataset. Varying performance of an algorithm can depend upon the structure of the tree and the processing power available. GREEDYFILLING achieves very good results when the structure of the graph is well-balanced, which is generally the case in the SYNTH dataset (Figure 2.7) as the graphs are generated recursively, as well as in the actual tree of Figure 2.9(a). This remark comes from the fact that GREEDYFILLING tries to maximize the efficiency of the allocation from the beginning of the schedule: if possible, it limits the allocation to every task to its first threshold, so that the overall speedup remains perfect. This explains why GREEDYFILLING is the best heuristic for medium numbers of processors in Figure 2.9(a): the tree is well-balanced, and for this range of processors, maintaining a perfect speedup is more efficient than balancing the allocation in the way PROPMAPREBALSIBLINGS does. However, GREEDYFILLING performance degrades relatively when some branches in the tree are far from being critical and should have their execution delayed, even if this means exceeding the first threshold on other tasks and having a non-perfect speedup. Therefore it only achieves average performance on the TREES dataset (Figure 2.8),

where other heuristics frequently have slightly better performance. For instance, the tree of Figure 2.9(b) has a highly critical branch on the right side, and GREEDYFILLING does not allocate enough processors to this branch at the beginning of the schedule, which leads to performance worse than that of the simple PROPSCHEDULING for average numbers of processors. With few processors, GREEDYFILLING fully prioritizes the critical branch as the first thresholds are not reached yet, and therefore achieves very good performance. In such a tree and with sufficient processing power, PROPMAPREBALSIBLINGS and PROPMAPREBALTHRESHOLD are the better choice as they progress quicker on the critical branches.

PROPMAPREBALTHRESHOLD achieves very good performance for synthetic graphs and is then only surpassed by GREEDYFILLING. It also achieves the best performance (with PROPMAPREBALSIBLINGS) for actual graphs. Therefore, PROPMAPREBALTHRESHOLD is never a bad choice (for the tested configurations). No other heuristic has this characteristic. One can also note that if PROPMAPREBALSIBLINGS achieves rather bad performance for synthetic graphs, it represents one of the best heuristics for actual graphs. This heuristic furthermore presents a practical advantage over PROPMAPREBALTHRESHOLD, whose effect is not taken in account in our model: it preserves the locality of the computations, allocating idle processors on tasks from the same branch as the node they were executing.

2.6 Conclusion

In this chapter, we have proposed a simple, but practical speedup model for graphs of malleable tasks, which is an interesting trade-off between tractability and accuracy. We have first provided an NP-completeness proof of the makespan minimization problem under this model. This was followed by a study of heuristic solutions, where we proposed model-optimized variants of the existing algorithms PROPMAPPING and FLOWFLEX. Designed for the new speedup model, we also proposed the novel GREEDYFILLING algorithm and studied the approximation ratio of these algorithms. To evaluate the algorithms, we performed simulations both on synthetic series-parallel graphs and on real task trees from linear algebra applications. They demonstrated the general superiority of the new GREEDYFILLING and the model-optimised variants of the traditional algorithms. In general, employing the new speedup model helps to improve the scheduling results.

The studies conducted here may be extended in several directions. On the theoretical side, there obviously exist constant-factor approximations for our model, as it falls into the hypotheses of existing algorithms, see Section 1.1. However, these algorithms rely on complex optimization techniques, so it remains to design a low-complexity constant-factor approximation. On the practical side, the next step is to implement such strategies into an actual runtime scheduler, in order to evaluate their efficiency.

The main limitation of the studies conducted in Chapter 1 and in this chapter comes from the fact that the platform is idealized: there are no communication times considered between tasks, the processors are all identical, and the memory storage is not considered. Such simplifications are close to the reality for some applications. However, when handling large data files or when a hybrid platform composed of different types of processors is targeted, we need to resort to different strategies. The objective of the following chapters is precisely to address these kinds of issues.

Chapter 3

Exploiting hybrid platforms in an online setting

« Mais je ne pouvais pas le deviner... »

Prolix, *Le Devin*

In Chapters 1 and 2, we focused on a platform composed of identical processors. However, modern computing platforms increasingly use specialized hardware accelerators, such as GPUs or Xeon Phis: as of November 2017, 102 of the supercomputers in the TOP500 list include such accelerators, while several of them include different accelerator types [138]. Therefore, the methods studied in Chapters 1 and 2 cannot be used on such a platform. Furthermore, the increasing complexity of these platforms makes it hard to predict the exact execution time of computational tasks or of data movement. Thus, dynamic runtime schedulers are often preferred to static ones, as they are able to adapt to variable running times and to cope with inaccurate predictions. Indeed, with the widespread heterogeneity of computing platforms, many scientific applications now rely on runtime schedulers such as OmpSs [126], XKaapi [32], or StarPU [17]. While task graphs have been widely studied in the theoretical scheduling literature [57], most of the existing studies (as Chapters 1 and 2) concentrate on static scheduling in the offline context: both the graph and the running times of the tasks are known beforehand.

We believe that there is a crucial need for online schedulers, that is, for scheduling algorithms that rely neither on the structure of the graph nor on the knowledge of tasks' running times. First, not all graphs are fully available at the beginning of the computation: sometimes the graph itself depends on the data being processed, and different inputs may result in different task graphs. This is in particular the case when the behavior of an iterative application depends on the accuracy of the output. Second, in most existing runtimes, even if the graph does not depend on the input data, it is not fully submitted at the beginning of the computation; instead, tasks are dynamically uncovered during the computation. Third, even if part of the graph is available, schedulers usually avoid traversing large parts of the graph each time they take a decision in order to strongly limit the time needed to take decisions. Finally, tasks' processing times are not always known beforehand, and the occasionally available predictions may not be very accurate, as two successive executions of the same task may result in slightly different timings.

There has recently been an effort of the scheduling community to fill the gap between the assumptions used in theoretical studies and those underlying schedulers for runtime systems (see details in Section 3.1). Schedulers for independent tasks on hybrid platforms have first been proposed [21, 34, 40]. Recently, an online scheduler for independent tasks on hybrid platforms [87] has been adapted for task graphs [8].

In the present chapter, we concentrate on the online scheduling of task graphs on a hybrid platform composed of 2 types of processors, that we call CPU and GPU for convenience. There are m CPUs and k GPUs, where $m \geq k \geq 1$. Note that we do not make any assumptions on the CPUs and GPUs, so that these results may be symmetrically applied to the converse case with more GPUs. The objective is to schedule a DAG G of tasks, so as to minimize the total completion time, or makespan. Each task can be assigned either to a single CPU or to a single GPU. The processing time of task T_i on a CPU is noted by $\overline{w_i}$ and on a GPU by $\underline{w_i}$.

We consider the following online problem. At the beginning, the algorithm is aware of all the input tasks of the graph, and can schedule each one on either a CPU or on a GPU. A task is released and becomes available to the scheduler only when all its predecessors are terminated. At any given point in the computation, the scheduler is totally unaware of tasks that have not yet been released, but it knows the processing times $\overline{w_i}$ and $\underline{w_i}$ of all available tasks. We do not take into account the time needed for moving data and assume that there is no delay between the release of a task and the start of its processing.

The closer related work considering the very same problem is [8] which provides a $4\sqrt{m/k}$-competitive algorithm for this problem. The number $\sqrt{m/k}$ will be used throughout the chapter as it appears to be deeply connected to this problem. We will therefore use the notation $\tau = \sqrt{m/k}$.

Main contributions. In this chapter, we prove that the competitive ratio of any online algorithm on a hybrid platform is lower-bounded by $\tau = \sqrt{m/k}$. We study how the knowledge of the task graph and the flexibility of the scheduler may influence the lower bound; we especially prove that knowing the bottom level of any task or having preemptive tasks does not help much, whereas the knowledge of the number of descendants allows to reduce the lower bound to $\frac{1}{2}\sqrt{\tau}$. We propose a $(2\tau + 1)$ - competitive algorithm, where the state-of-the art algorithm was proved to be 4τ-competitive. We then propose a simple heuristic, based on the system-oriented heuristic EFT, which is both a competitive algorithm and performs well in practice, as we show with a comprehensive simulation set. Finally, we extend our results to more than two types of processors, for which we extend both the lower bounds and the online algorithms.

The rest of the chapter is organized as follows. In Section 3.1, we briefly review the related work. In Section 3.2, we study lower bounds on the competitive ratio of any online algorithm. In Section 3.3, we propose two online algorithms. In Section 3.4, we show that the main difficulty of the problem is to decide whether each task has to be allocated to CPUs or to GPUs. When these decisions are fixed, any list scheduling algorithm is then 3-competitive. In Section 3.5 we study the generalized problem with more than two types of processors. In Section 3.6, we study through simulations the behavior of several online algorithms on different datasets, composed either of actual or synthetic task graphs.

3.1 Related work

We briefly position our contributions in comparison to the existing work, starting with the offline case when the whole scheduling problem (both task dependences and running times) is known beforehand.

Offline algorithms. Several schedulers for independent tasks on hybrid platforms have been proposed. Bleuse et al. [34] designed a polynomial but expensive $\left(\frac{4}{3} + \frac{1}{3k}\right)$-approximation. Low complexity algorithms, which are closer to our work, have been studied in [21, 40] and achieve approximation ratios respectively equal to 2 and $2 + \sqrt{2}$. For tasks with precedence constraints, Kedad-Sidhoum et al. [93] provided a tight 6-approximation based on linear programming.

In a different context, Chudak and Shmoys [44] provided a $\log(p)$-approximation for the $Q \mid prec \mid C_{max}$ problem: scheduling a graph of tasks on p machines with different speeds. Considering independent moldable tasks (tasks can be assigned to multiple CPUs or one GPU), Bleuse et al. [33] provide a 2-approximation.

Online algorithms. When tasks with precedences are released over time, Graham's List Scheduling algorithm [73] is $(2-1/m)$-competitive on homogeneous processors. Svensson proved in [134] that no polynomial offline algorithm can have a better competitive ratio assuming $P \neq NP$ and a variant of the Unique Games Conjecture.

On the problem of scheduling independent tasks on two sets of processors, Imreh [87] proposed a $(4-2/m)$-competitive algorithm. The principle of this algorithm is to schedule a task T_i on GPU if $\overline{w_i}/\underline{w_i} \geq m/k$, or if T_i can be terminated on GPU given the current schedule before time $\overline{w_i}$. Chen et al. [43] later refine a similar algorithm by introducing four tunable parameters. A combination of parameters lead to an improved competitive ratio equal to 3.85. They also show that any online algorithm has a competitive ratio at least 2.

Based on this work, Amaris et al. [8] exhibited an online algorithm for precedence constraints, achieving a competitive ratio of $4\sqrt{m/k}$. The main difference with the previous algorithms is that a task is scheduled on GPU if it is accelerated by a factor at least $\sqrt{m/k}$ (and not at least m/k).

Runtime strategies. Actual runtime schedulers usually rely on low-complexity scheduling policies to limit the time needed to allocate tasks. For instance, StarPU [17] builds a performance model of tasks that enables to predict their processing times. When a new task is submitted, it is allocated to the resource that will complete it the soonest (when using the **dm** policy, previously called **heft-tm** in [16]), which corresponds to the classical Earliest Finish Time (EFT) scheduling policy [102]. Other strategies have been proposed that take into account communication times, or precomputed task priorities, depending on the descendants of each task. We include similar information in the design of the lower bounds on competitive ratios (Section 3.2).

3.2 Lower bound on online algorithms competitiveness

In this section, we provide a lower bound on the competitive ratio of any online algorithm: no online algorithm has a competitive ratio smaller than $\tau = \sqrt{m/k}$ for any values of m and k (Theorem 3.1) and smaller than $\tau + 1$ for infinitely many values of m and k (Lemma 3.1). We also study how adding flexibility to task processing or giving some knowledge of the graph to the scheduler impacts this lower bound.

Intuitively, the main difficulty for this problem arises from choosing on which type of resource (CPU or GPU) a given task should be processed, and not to come up with the final schedule. This is indeed proven in Theorem 3.6, Section 3.4: if the allocation of the tasks is fixed, any online list scheduling algorithm is $\left(3-\frac{1}{m}\right)$-competitive, which is optimal.

The proof of Theorem 3.1 heavily relies on the fact that an online algorithm has no information on the successors of each task. In practice, it is sometimes possible to get some information on the task graph, for example by pre-computing some information offline before submitting the tasks. For instance, offline schedulers usually ranks available tasks with priorities based on the dependences. On homogeneous platforms, the *bottom-level* of a task is commonly used, and is defined as the maximum length of a path from this task to an exit node, where nodes of the graphs are weighted with the processing time of the corresponding tasks. In the heterogeneous case, the priority scheme used in the standard HEFT

algorithm [139] is to set the weight of each node as the average processing time of the corresponding task on all resources.

Knowing the bottom-level does not change the lower-bounds of Theorem 3.1 and Lemma 3.1, see Theorem 3.2. The only benefit is a diminution by a factor 2 if there is exactly one GPU. An interesting component of this proof is that all the tasks are equivalent (same CPU and GPU computing times) so other heterogeneous variants of the bottom level are also captured.

When the online scheduler is given the knowledge of the number of descendants of each submitted task in addition to their bottom-level, the lower bound of Theorem 3.1 is reduced to $\frac{1}{2}\sqrt{\tau}$ when m/k is large enough (see Theorem 3.3), so no constant-factor competitive algorithm exists. Note that all the tasks are also equivalent in this proof; so it also captures, for instance, the knowledge of the CPU and GPU computing times of all the descendants; only the pattern of precedence relations remains unknown. Note that, however, no algorithm has been proposed that reaches this bound.

Another interesting question is whether adding flexibility on how tasks are processed changes this bound. Allowing task spoliation (where tasks can be canceled and restarted on another resource, as done in [21]) does not change any lower bound. Allowing task migration (where tasks can be interrupted and resumed on another resource) divides the lower bounds obtained by a factor 2.

Table 3.1 summarizes the results for all combination of knowledge given to the scheduler and flexibility on the task processing. The best known competitive ratio for every setting is smaller than $2\tau + 1$, and is achieved by the QA algorithm we design in Section 3.3.1. This algorithm does not use all the knowledge or flexibility as it does not practice spoliation or migration, does not use any information on the bottom level or the descendants, and schedules tasks one by one, without looking at other available tasks.

Flexibility	Knowledge	Lower bound	Proof	Note
None or Spoliation	None	τ	Th. 3.1	$\tau+1$ for specific instances
	Bottom Level	τ	Th. 3.2	$\frac{1}{2}\tau$ if $k=1$; $\tau+1$ for spec. inst.
	BL + descendants	$\frac{1}{2}\lfloor\sqrt{2\tau^*}\rfloor$	Th. 3.3	–
Migration	None	$\frac{1}{2}\tau$	Th. 3.1	–
	BL	$\frac{1}{2}\tau$	Th. 3.2	$\frac{1}{4}\tau$ if $k=1$
	BL + descendants	$\frac{1}{4}\lfloor\sqrt{2\tau^*}\rfloor$	Th. 3.3	–

Table 3.1: Summary of the results obtained for various versions of online models.
τ^* represents the largest triangular number such that $\tau^* \leq \tau$. If τ is large, we have $\lfloor\sqrt{2\tau^*}\rfloor \geq \sqrt{\tau}$.

First, we consider algorithms that are not aware of the bottom level of the tasks.

Theorem 3.1. *No online algorithm has a competitive ratio smaller than τ, even when spoliation is authorized. If preemption with migration is authorized, no online algorithm has a competitive ratio smaller than $\frac{\tau}{2}$.*

Proof. Consider an online algorithm $\mathscr{A}$, making use of spoliations. We assume for the moment that τ is an integer. We consider an integer n as large as we want. A large n will lead to a large graph and a competitive ratio closer to τ. We will use an adversary proof, by building a graph composed of nm tasks denoted by T_i^j, with $1 \leq j \leq n\tau$ and $1 \leq i \leq k\tau$. The CPU processing time of each task equals τ and the GPU processing time equals 1.

The procedure can be cut into $n\tau$ phases. During the jth phase, tasks T_i^j for i from 1 to $k\tau$ are independent and available. The adversary selects the task that $\mathscr{A}$ completes the latest, breaking ties

arbitrarily. Let T_*^j be this task. The $k\tau$ tasks of the next phase are then made successors of T_*^j. See Figure 3.1 for an illustration of a built graph.

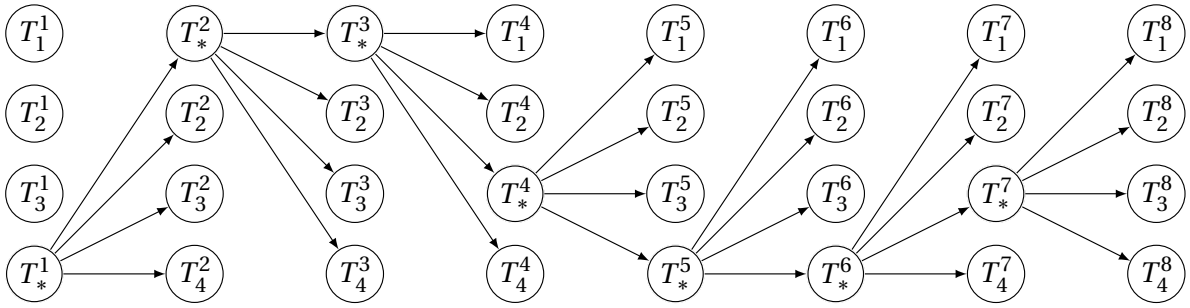


Figure 3.1: Example of built graph with $\tau = 2$, $k = 2$, $n = 4$.

We now show how to build an efficient (offline) schedule $\mathscr{S}$ of the resulting graph. A *bucket* is defined as a set of processors, a starting time and a duration time. We use buckets to book some processors for an amount of time, and schedule a set of tasks in a given bucket. We consider $n+1$ buckets, as illustrated in Figure 3.2. Buckets B_i for i from 1 to n each concerns all m CPUs, lasts a time τ, and starts at time $i\tau$. Note that m tasks fit into each bucket. The last bucket, B concerns one GPU, starts at time 0 and lasts a time $n\tau$.

$\mathscr{S}$ schedules the $n\tau$ tasks T_*^j successively on a single GPU, which fit into bucket B. In parallel, $\mathscr{S}$ schedules the remaining tasks on CPU. More precisely, it puts in bucket B_ℓ tasks T_i^j such that $(\ell-1)\tau < j \leq \ell\tau$, except for tasks T_*^j. They all fit into the bucket as there are less than $\tau \times k\tau \leq m$ such tasks. Moreover, task $T_*^{\ell\tau}$ completes at time $\ell\tau$. Therefore, every task T_i^j with $(\ell-1)\tau < j \leq \ell\tau$ can be started at time $\ell\tau$, and thus can be scheduled into bucket B_ℓ. Therefore, $\mathscr{S}$ achieves a makespan equal to $(n+1)\tau$.

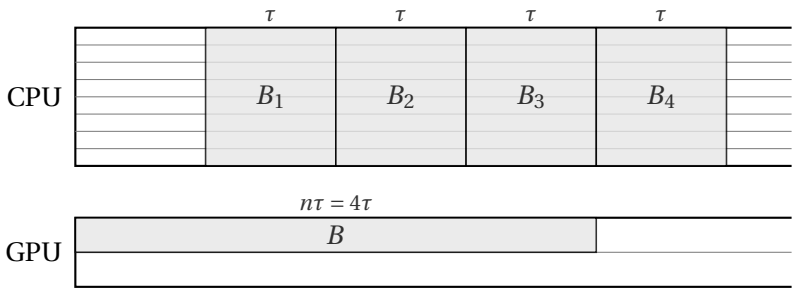


Figure 3.2: Buckets used by $\mathscr{S}$ with $n = 4$.

Now, we consider algorithm $\mathscr{A}$, and we show that the makespan obtained is at least $n\tau^2$. At each phase, the adversary reveals the next phase only when all the tasks of the current phase are completed. If one task of the phase is scheduled on CPU, it takes a time τ. Otherwise, all $k\tau$ tasks are scheduled on GPU, and the last one completes at time at least $k\tau/k = \tau$. Therefore, $\mathscr{A}$ completes each phase in time at least τ. As there are $n\tau$ phases, the whole graph cannot be scheduled in time smaller than $n\tau^2$. The competitive ratio of $\mathscr{A}$ is then at least:

$$\frac{n\tau^2}{(n+1)\tau} \quad \underset{n\to\infty}{\longrightarrow} \quad \tau.$$

Now, consider an algorithm $\mathscr{A}$' which makes use of preemption with migration. The adversary strategy and the schedule $\mathscr{S}$ is unchanged. We first prove by contradiction that $\mathscr{A}$' cannot terminate a phase in a makespan smaller than $\tau/2$. Assume that one phase is terminated in time $\tau/2$. We consider the fraction of each task performed on a CPU. All tasks have a processing time of τ on CPU, so for each task, this fraction cannot be larger than one half. Therefore, at least half of each task is executed on a GPU, which takes a time $1/2$ for each task, so it takes $k\tau/2$ units of GPU computing time. As we assumed that the phase is terminated in time $\tau/2$, there is no more than $k\tau/2$ work units available on the k GPUs, which thus cannot process more than one half of each task. Therefore, at least half of each task is processed on CPUs, from the very beginning to the very end of the phase. This requires to execute each task simultaneously on a CPU and on a GPU, which is not possible even with migration. Therefore, $\mathscr{A}$' cannot terminate one phase in time $\tau/2$ (and a fortiori in a shorter time). Thus, $\mathscr{A}$' requires a time larger than $n\tau^2/2$ to complete all $n\tau$ phases. The competitive ratio of $\mathscr{A}$' is then at least:

$$\frac{\frac{1}{2}n\tau^2}{(n+1)\tau} \quad \underset{n\to\infty}{\longrightarrow} \quad \frac{\tau}{2}.$$

In a last step, we now relax the constraint that τ is an integer. Let q be an integer as large as we want, and $r = \lfloor q(\tau - \lfloor\tau\rfloor)\rfloor$, so that $r/q \leq \tau - \lfloor\tau\rfloor \leq (r+1)/q$. A large q will lead to a greater precision. We adapt the graph in the following way: there are now $n(\lfloor\tau\rfloor + r)$ phases each containing $k\lfloor\tau\rfloor + 1$ tasks. For each phase j such that $(j \mod n) \leq \lfloor\tau\rfloor$, the tasks have a CPU computing time τ and a GPU computing time 1. In the remaining nr phases, tasks have a CPU computing time equal to τ/q and a GPU computing time equal to $1/q$. Intuitively, we split the phases corresponding to the fractional part of τ into multiple phases of smaller tasks.

We now adapt the schedule $\mathscr{S}$ which still fits inside the buckets (as previously defined). The tasks T_*^j all fit inside bucket B. Indeed, there are composed of $n\lfloor\tau\rfloor$ tasks of GPU computing time 1 and nr tasks of GPU computing time $1/q$; the time needed to process them sequentially is then equal to $n(\lfloor\tau\rfloor + r/q) \leq n\tau$. For bucket B_1, we execute the tasks T_i^j for $j = 1,\ldots,\lfloor\tau\rfloor + r$ and $i = 1,\ldots,k\lfloor\tau\rfloor + 1$, except tasks T_*^j. The corresponding tasks T_*^j are completed before the start of bucket B_1. Inside bucket B_1, we execute $k\lfloor\tau\rfloor^2$ tasks of CPU computing time τ and $kr\lfloor\tau\rfloor$ tasks of CPU computing time τ/q. The number of processors needed is then:

$$\begin{aligned} k\lfloor\tau\rfloor^2 + \left\lceil k\frac{r}{q}\lfloor\tau\rfloor\right\rceil &\leq k\lfloor\tau\rfloor^2 + \lceil k(\tau - \lfloor\tau\rfloor)\lfloor\tau\rfloor\rceil \\ &\leq k\lfloor\tau\rfloor^2 - k\lfloor\tau\rfloor^2 + \lceil k\tau\lfloor\tau\rfloor\rceil \leq \lceil k\tau^2\rceil = m. \end{aligned}$$

Therefore, the first $\lfloor\tau\rfloor + r$ phases fit into bucket B_1. The same reasoning applies to the following buckets, so $\mathscr{S}$ achieves a makespan equal to $(n+1)\tau$.

Concerning the algorithm $\mathscr{A}$ (assuming it does not make use of preemption with migration), the phases where tasks have a GPU computing time equal to 1 still need a time τ to be completed: if one task is scheduled on CPU, it takes a time τ; if all $k\lfloor\tau\rfloor + 1$ tasks are scheduled on GPUs, this takes at least a time $\lfloor\tau\rfloor + 1 \geq \tau$. Similarly, the other phases need a time τ/q to be completed. The total makespan is then at least $n\tau\left(\lfloor\tau\rfloor + \frac{r}{q}\right) \geq n\tau\left(\tau - \frac{1}{q}\right)$. When q and n tend to infinity, the competitive ratio tends to τ. Note that if $\mathscr{A}$ makes use of preemption with migration, this ratio tends to $\tau/2$, which terminates the proof. □

The proof of Theorem 3.1 heavily relies on the fact that the online algorithm has zero information on the successors of each task. As mentioned earlier, it is sometimes possible to get some information, such as the bottom level of each task. On heterogeneous platforms, several adaptations exist to the

bottom level, as discussed at the beginning of this section. We prove that using such information does not improve the bound.

Theorem 3.2. *If $k \geq 2$, no online algorithm has a competitive ratio smaller than τ, even when spoliation is authorized, and the bottom level of each task is known. If preemption with migration is authorized, no online algorithm has a competitive ratio smaller than $\frac{\tau}{2}$.*

If $k = 1$, then we obtain the same bounds divided by a factor 2.

Proof. The proof relies on the construction used to prove Theorem 3.1. We assume for the moment that $k \geq 2$. For simplification, we rely on the construction for an integer τ but the modification easily extends to a decimal τ.

We add $n\tau$ tasks U^j to the built graph, with $1 \leq j \leq n\tau$, where there is a dependence from U^j to U^{j+1} for each j. Each task has a CPU computing time equal to τ and a GPU computing time equal to 1, as tasks T_i^j. For each task T_i^j, we add a dependence from T_i^j to U^j. See Figure 3.3 for an illustration of the graph.

The longest path starting from any task T_i^j to an endpoint of the built graph then has a length equal to $n\tau - j + 2$: it is composed for instance of task T_i^j and tasks U^j to $U^{n\tau}$. Note that tasks T_*^j have multiple paths of length $n\tau - j + 2$, see Figure 3.3.

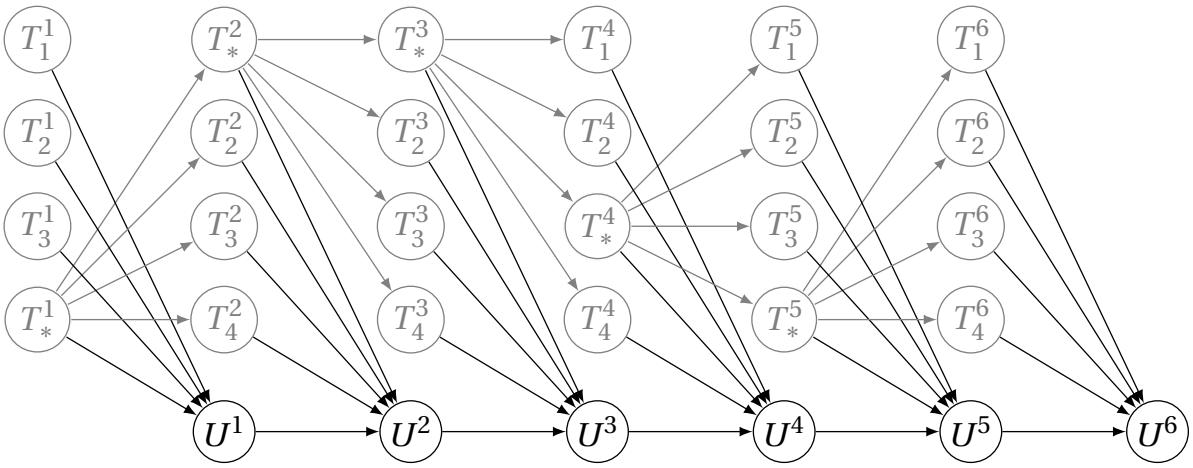


Figure 3.3: Example of built graph with $\tau = 2$, $k = 2$, $n = 3$. In gray, the tasks and dependences existing in the previous proof.

Therefore, for any j, the $k\tau$ tasks T_i^j have the same bottom level. So when $\mathscr{A}$ terminates task T_i^j, the adversary can choose whether T_*^j is equal to T_i^j or not, while respecting the bottom level furnished to $\mathscr{A}$. Then, the lower bound on the makespan reached by $\mathscr{A}$ (and $\mathscr{A}$' if preemption with migration is allowed) still holds.

It remains to define the schedule $\mathscr{S}$ with the added tasks, and to show that its makespan is at most $(n+2)\tau$. Recall that we assumed $k \geq 2$. We add another bucket B^U concerning a different GPU than B (as $k \geq 2$), starting at time 2τ and lasting $n\tau$ units of time, see Figure 3.4. Task U^j is scheduled in B^U at time $2\tau + j$. Note that for any ℓ, tasks $U^{(\ell-1)\tau}$ to $U^{\ell\tau}$ are executed after bucket B_ℓ, which contains tasks T_i^j for $(\ell-1)\tau < j \leq \ell\tau$. Therefore, every task T_i^j is terminated before task U^j is scheduled, so no precedence constraints are violated.

The lower bounds proved in the Theorem 3.1 are then unchanged.

If $k = 1$, we define the bucket B^U on the unique GPU, starting at time $n\tau$ and terminating at time $2n\tau$. The makespan obtained by $\mathscr{S}$ is then twice longer, so the lower bounds obtained are twice smaller. □

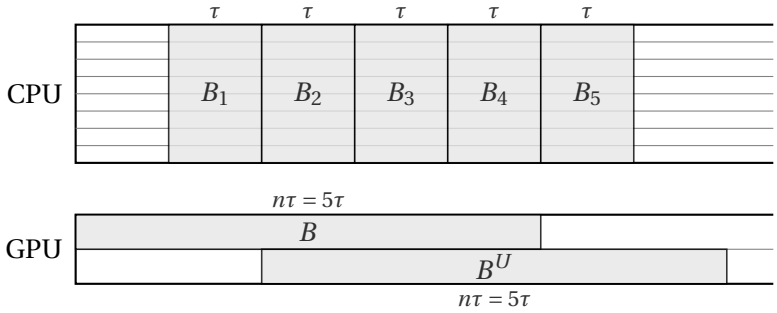


Figure 3.4: Buckets used by $\mathscr{S}$ with $n = 5$.

In the proof of Theorem 3.2, we used the fact that the bottom level is no longer useful to differentiate tasks T_*^j from other tasks T_i^j. However, the former may have many more descendants than the latter ($\Theta(nm)$ compared to $\Theta(n\tau)$ for small j). A metric that could then still be used is the total weight of the descendants of a task. We nevertheless prove in Theorem 3.3 that this knowledge cannot lead to constant-factor approximations, as we prove a lower-bound in $\Theta(\sqrt{\tau})$. As discussed at the beginning of this section, all the tasks used in the following proof are identical, so the *weight* can actually capture several functions such as the number of descendants, the average computing time...

Theorem 3.3. *No online algorithm has a competitive ratio smaller than* $\frac{1}{2}\lfloor\sqrt{2\tau^*}\rfloor$*, even when spoliation is authorized, and both the bottom level and the total weight of the descendants of each task is known. If preemption with migration is authorized, no online algorithm has a competitive ratio smaller than* $\frac{1}{4}\lfloor\sqrt{2\tau^*}\rfloor$.

In these bounds, τ^* *is the largest triangular integer not larger than* τ*. Recall that* τ *is a triangular integer if we have* $\tau = 1 + 2 + \ldots + \lfloor\sqrt{2\tau}\rfloor$.

Proof. The proof relies on the construction used to prove Theorem 3.2, but using less phases and adding several tasks. We first assume that τ is a triangular integer larger than 1, which means that there exists an integer $q > 1$ such that $\sum_{i=1}^{q} i = \tau$. The exact value of q is $\frac{1}{2}\sqrt{1+8\tau} - \frac{1}{2} = \lfloor\sqrt{2\tau}\rfloor$. The graph built in this proof contains $q+1$ phases of $k\tau$ tasks each.

We add m tasks V_i^j to the built graph, with $1 \leq j \leq q$ and for each j, with $1 \leq i \leq (q+1-j)k\tau$. By definition of q, this indeed sums to $k\tau^2 = m$ additional tasks. All these tasks have a CPU computing time equal to τ and a GPU computing time equal to 1, as tasks T_i^j.

We now build the graph so that for each j, the $k\tau$ tasks T_i^j have the same number of descendants. For each $j \leq q$, we add dependences from every task T_i^j except T_*^j to every task $V_i^{j'}$ such that $j' \geq j$. See Figure 3.5 for an illustration of the graph, where all tasks T_*^j have been set to $T_{k\tau}^j$ for simplification, and where tasks sharing the same successors and predecessors have been agglomerated. In this example, we have $q = 3$, $\tau = 6$ and $m = 36$. For instance, T_1^3 has 6 new successors (tasks $V_{1\ldots6}^3$), T_1^2 has 18 new successors and T_1^1 has $m = 36$ new successors.

Let j be fixed. In this graph, the descendants of task T_*^j are tasks $U^{j'}$ for $j' \geq j$, tasks $T_i^{j'}$ for $j' \geq j$ and tasks $V_i^{j'}$ for $j' > j$. The descendants of any task T_i^j except T_*^j are tasks $U^{j'}$ for $j' \geq j$ and tasks $V_i^{j'}$ for $j' \geq j$. The difference is that task T_*^j has the $(q+1-j)k\tau$ tasks $T_i^{j'}$ as successors but not the $(q+1-j)k\tau$ tasks V_i^j. Therefore, at j fixed, the number of successors is the same for each task T_i^j.

So when $\mathscr{A}$ terminates a task $\frac{j}{i}$, the adversary can choose whether T_*^j is equal to T_i^j or not, while respecting the bottom level and number of descendants furnished to $\mathscr{A}$. Then, the lower bound on the

makespan reached by $\mathscr{A}$ (and $\mathscr{A}$' if preemption with migration is allowed) on each phase still holds. Therefore, $\mathscr{A}$ leads to a makespan of at least $(q+1)\tau$ and $\mathscr{A}'$ to a makespan of at least $\frac{1}{2}(q+1)\tau$.

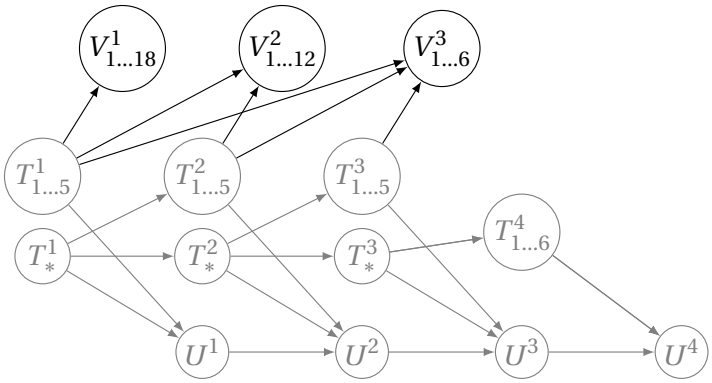


Figure 3.5: Example of built graph with $\tau = 6$, $q = 3$, $k = 1$, $m = 36$. In gray, the tasks and dependences existing in the previous proof.

It remains to define the schedule $\mathscr{S}$ with the added tasks, and to show that its makespan is at most $2\tau+q+1$. The schedule $\mathscr{S}$ is similar to the one of the previous proof, except that tasks V_i^j are executed on CPU after tasks T_i^j. As there are m such tasks, this takes a time τ. To summarize, tasks T_*^j are executed on GPU in time q, then the remaining tasks T_i^j are executed on CPU in time τ, then, in parallel, tasks U^j are executed on GPU in time $q+1$ and tasks V_i^j are executed on CPU in time τ, see Figure 3.6. The makespan obtained is then equal to $\tau + q + \max\{\tau, q+1\} = 2\tau + q$. The last equality is valid as for $\tau \geq 3$, we have $q \leq \tau - 1$.

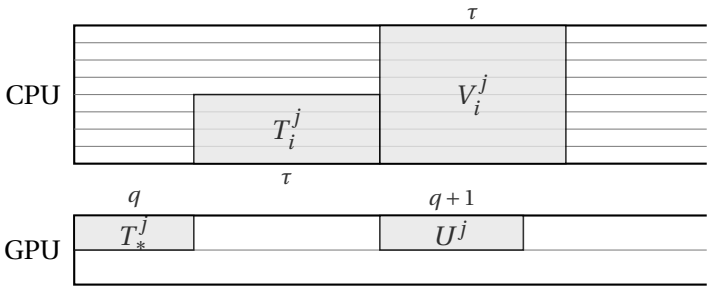


Figure 3.6: Shape of the schedule $\mathscr{S}$.

Recalling that $\tau = q(q+1)/2$, the competitive ratio of $\mathscr{A}$ is then at least:

$$\frac{(q+1)\tau}{2\tau+q} = \frac{1}{2}\frac{q(q+1)^2}{q(q+1)+q} > \frac{q}{2}\frac{(q+1)^2}{(q+1)^2} = \frac{q}{2} > \frac{1}{2}\left\lfloor\sqrt{2\tau}\right\rfloor.$$

Similarly, the competitive ratio of $\mathscr{A}$' is at least $\frac{1}{4}\left\lfloor\sqrt{2\tau}\right\rfloor$.

If τ is not a triangular integer, or even not an integer, the same proof applies where q is the maximal integer such that $\sum_{i=1}^{q} i \leq \tau$. The exact value of the computing ratio obtained is then:

$$\frac{1}{2}\left\lfloor\sqrt{2\tau+\frac{1}{4}}-\frac{1}{2}\right\rfloor = \Theta\left(\sqrt{\tau}\right).$$

□

Theorems 3.1 and 3.2 prove a lower bound valid for any value of m and k. In the following lemma, we improve the result of Theorem 3.2 for specific values of m and k. This lemma implies that no online algorithm can have a competitive ratio smaller than $\tau + 1$ for every value of m and k.

Lemma 3.1. *There exist infinitely many values of m and k for which no online algorithm has a competitive ratio smaller than $\tau + 1$, even when spoliation is authorized and the bottom level of each task is known.*

Proof. We assume throughout this proof that τ is an integer and that $k \geq 2(\tau + 1)$. This proof is adapted from the proof of Theorem 3.2 when τ is an integer, and where each occurrence of τ is replaced by $\tau + 1$.

Specifically, the graph built in this proof is composed of $n(\tau + 1)$ phases of $k\tau + 1$ tasks, along with a chain of $n(\tau + 1)$ tasks, for an arbitrarily large n. The CPU computing time of each task is equal to $\tau + 1$ and the GPU computing time is equal to 1. The dependences are built as in the proof of Theorem 3.2.

Any online algorithm needs a time $\tau + 1$ to schedule each phase. Indeed, in each phase, either $\tau + 1$ tasks are scheduled on a single GPU, or one task is scheduled on CPU. Therefore, the makespan obtained by any online algorithm on the total $n(\tau + 1)$ phases is at least $n(\tau + 1)^2$.

We now build an offline schedule similar to the one of Theorem 3.2, except that $k - 1$ tasks of each phase are scheduled simultaneously on GPUs. Therefore, with the tasks of the additional chain, the GPUs are all busy (except at the beginning and the end of the schedule). As each phase contains $k\tau + 1$ tasks, it remains to schedule $k(\tau - 1) + 2$ tasks per phase on the CPUs. The objective is to schedule $\tau + 1$ phases simultaneously in each bucket, where the length of a bucket is equal to $\tau + 1$ units of time. The number of tasks per bucket is then:

$$\begin{aligned}(k(\tau - 1) + 2)(\tau + 1) &= k\left(\tau^2 - 1\right) + 2(\tau + 1) \\ &= m - k + 2(\tau + 1) \\ &\leq m.\end{aligned}$$

The last inequality comes from the assumption on k. Each task can be scheduled on one CPU so the relevant tasks of $\tau + 1$ phases fit into each bucket. Therefore, by similar arguments to the proof of Theorem 3.2, we conclude that the optimal schedule has a makespan at most $(n + 2)(\tau + 1)$.

When n increases, we obtain the result. □

3.3 Competitive algorithms

3.3.1 The Quick Allocation (QA) algorithm

Amaris et al. [8] designed an online algorithm named ER-LS composed of two phases. For each available task, it firsts decides whether it should be allocated to CPUs or GPUs, and then schedule it on the appropriate resource type. More precisely, ER-LS can be described as follows:

1. Take any available task T_i.
   (a) If T_i can be terminated on GPU in the current schedule before time $\overline{w_i}$, allocate it to GPU.
   (b) If $\overline{w_i}/\underline{w_i} \leq \tau = \sqrt{m/k}$, then allocate T_i to CPU, otherwise assign it to GPU.
2. Schedule T_i as soon as possible on the assigned type of processor
3. If there are remaining tasks, return to Step 1.

This algorithm is proved to be 4τ-competitive. We propose here a simplified version of this algorithm, for which we prove a better competitive ratio. The improvement comes both from the simplification and a tighter analysis. We define the algorithm QA, which stands for Quick Allocation. Rule 1a of ER-LS is deleted, so the allocation phase is then simplified to:

- Take any available task T_i. If $\overline{w_i}/\underline{w_i} \leq \tau$, then allocate T_i to the CPU side, otherwise allocate it to the GPU side.

Note that this allocation phase does not take into account any precedence relation or current schedule. Once this task is allocated to the CPU or GPU side, it is scheduled on the processor of this side which has completed its tasks the soonest.

One could wonder why the ratio τ is the best choice in the allocation phase. Intuitively, there are more CPUs than GPUs, so if $\overline{w_i}/\underline{w_i} < 1$, task T_i is executed faster on CPU, which is a lesser rare resource, therefore task T_i should be allocated to CPU. On the contrary, if $\overline{w_i}/\underline{w_i} > m/k$, then not only task T_i is executed faster on GPU, but if there are many independent tasks with the same processing times to compute, they will be executed faster all on GPUs than all on CPUs. Therefore we can allocate T_i to GPU without wasting a rare resource. When this ratio is between 1 and m/k, the loss is minimized when switching resource at the geometric mean of 1 and m/k, which is equal to τ.

We now show that QA is $\left(2\tau+1-\frac{1}{k\tau}\right)$ – competitive and that this ratio is almost tight, as we provide an example on which QA leads to a makespan $\left(2\tau+1-\frac{1}{k}\right)$ times larger than the optimal solution.

Theorem 3.4. QA *is* $\left(2\tau+1-\frac{1}{k\tau}\right)$ – *competitive.*

Proof. We consider a graph, an online instance of this graph, and the schedule $\mathscr{S}$ computed by QA, of makespan C_{max}. We also consider an optimal schedule of this graph (later referred to by *the* optimal solution), and we let OPT be its makespan.

Let W_c (resp. W_g) be the total load on the CPUs (resp. GPUs). Let cp be a critical path the task graph given the allocation of $\mathscr{S}$, and CP be the sum of the processing times of the tasks of cp in $\mathscr{S}$.

The objective is to prove that:

$$C_{max} \leq \left(2\tau+1-\frac{1}{k\tau}\right) OPT.$$

We first use Lemma 3.2 to bound C_{max} using the processor loads (W_c and W_g) and the length of the critical path (CP). Then, we bound the expression obtained in function of OPT to prove the result.

Lemma 3.2.

$$C_{max} \leq \frac{W_c}{m} + \frac{W_g}{k} + \left(1-\frac{1}{m}\right) CP.$$

Proof. We consider the path p defined as being the longest path (in terms of execution time in $\mathscr{S}$) that contains a task that terminates at time C_{max} in $\mathscr{S}$. By definition, the length of p is at most CP. In order to simplify the reasoning, we assume that there is a task T_0 of null processing time that is the predecessor of every task in the graph, and that this task belongs to p.

Consider a moment t when no task of p is being executed in $\mathscr{S}$. Let T_ℓ be the last task of p to be executed before t and T_n be the next task of p to be executed after t in $\mathscr{S}$. Note that both tasks always exist because T_0 is executed at the start of the graph and a task of p is executed at the end of the schedule $\mathscr{S}$. Suppose first that T_n is executed on CPU. As T_n is not scheduled immediately after T_ℓ, and the schedule has been obtained by a list algorithm, no CPU is idling between the termination of T_ℓ and the start of T_n. Symmetrically, if T_n is executed on GPU, no GPU is idling in this period.

Therefore, when no task of p is being executed, either all the CPU are busy or all the GPU are busy. The CPU (resp. GPU) can be all busy for at most a time of W_c/m (resp. W_g/k). And the tasks of p are executed during a time at most CP.

Hence, we have:

$$C_{max} \leq \frac{W_c}{m} + \frac{W_g}{k} + CP.$$

We can further refine this inequality. Let P_c (resp. P_g) be the processing time of the tasks of p on CPU (resp. GPU). Then, these processing times can be removed from the total loads in the inequality:

$$\begin{aligned} C_{max} &\leq \frac{W_c - P_c}{m} + \frac{W_g - P_g}{k} + P_c + P_g \\ &\leq \frac{W_c}{m} + \frac{W_g}{k} + \frac{m-1}{m} P_c + \frac{k-1}{k} P_g \\ &\leq \frac{W_c}{m} + \frac{W_g}{k} + \frac{m-1}{m} \left(P_c + P_g\right) \\ &\leq \frac{W_c}{m} + \frac{W_g}{k} + \frac{m-1}{m} CP. \end{aligned}$$

□

Bounding the loads We denote by A_c (resp. A_g) the set of tasks placed on CPU (resp. GPU) both by $\mathscr{S}$ and in the optimal solution. We denote by C_c (resp. C_g) the set of tasks placed on CPU (resp. GPU) by $\mathscr{S}$ but not in the optimal solution. The lowercase denotes the sum of the processing times of these sets.

The optimal makespan OPT is at least equal to the average work on CPU (and on GPU) in the optimal solution. In this solution, the tasks executed on CPU are tasks of the sets A_c and C_g. Tasks of A_c have the same processing time in $\mathscr{S}$ and in the optimal solution, as they are executed on CPU in both cases. Tasks of C_g are completed faster in $\mathscr{S}$ than in the optimal solution. More precisely, the allocation phase ensures that any task T_i of C_g verifies $\overline{w_i} \geq \tau \underline{w_i}$. Therefore, bounding OPT by the average work on CPU gives the following inequality:

$$OPT \geq \frac{1}{m} \left(a_c + \tau c_g\right). \tag{3.1}$$

Similarly, bounding OPT by the average work on GPU gives:

$$OPT \geq \frac{1}{k} \left(a_g + \frac{c_c}{\tau}\right). \tag{3.2}$$

Using the fact that $k\tau \leq m$, we multiply by τ both sides of Equation 3.1 to get:

$$\frac{m}{k\tau} OPT \geq \frac{m}{k\tau} \frac{a_c}{m} + \frac{m}{k\tau} \frac{\tau c_g}{m} \geq \frac{a_c}{m} + \frac{c_g}{k}.$$

We then simplify Equation 3.2 using that $k\tau \leq m$:

$$OPT \geq \frac{1}{k} \left(a_g + \frac{c_c}{\tau}\right) \geq \frac{a_g}{k} + \frac{c_c}{m}.$$

Summing these two inequalities, we get:

$$\left(1 + \frac{m}{k\tau}\right) OPT \geq \frac{a_c + c_c}{m} + \frac{a_g + c_g}{k} \geq \frac{W_c}{m} + \frac{W_g}{k} \tag{3.3}$$

Bounding the critical path We now bound the length of the critical path produced: every task of this critical path is also scheduled in the optimal schedule, and forms a path. Each task can be accelerated by a factor at most τ in the optimal schedule, so the time dedicated to process this path in the optimal schedule is at least CP/τ. Therefore, we have

$$CP \leq \tau\, OPT. \tag{3.4}$$

Conclusion of the proof Finally, from Lemma 3.2 we get:

$$C_{max} \leq \frac{W_c}{m} + \frac{W_g}{k} + \left(1 - \frac{1}{m}\right) CP.$$

Equations 3.3 and 3.4 lead to:

$$C_{max} \leq \left(1 + \frac{m}{k\tau}\right) OPT + \left(\tau - \frac{\tau}{m}\right) OPT. \tag{3.5}$$

Using that $\tau^2 \geq m/k$ so $\tau/m \geq 1/k\tau$, we deduce:

$$C_{max} \leq (1+\tau)\, OPT + \left(\tau - \frac{1}{k\tau}\right) OPT \leq \left(2\tau + 1 - \frac{1}{k\tau}\right) OPT.$$

Hence, the theorem. □

Note that up to Equation 3.5, the only property of τ that has been used in the above proof is that $\tau \leq m/k$. Therefore, this equation is also valid for the following variant of QA, which we name QA', and has a slightly smaller competitive ratio.

- Take any available task T_i. If $\overline{w_i}/\underline{w_i} \leq \tau'$, then allocate T_i to the CPU side, otherwise allocate it to the GPU side, where
$$\tau' = \tau \sqrt{\frac{m}{\max(k, m-1)}}.$$

Lemma 3.3. *The competitive ratio of* QA' *is* $3 - \frac{1}{m}$ *when* $\tau = 1$ *and* $1 + 2\tau\sqrt{1 - \frac{1}{m}}$ *otherwise.*

Proof. First, if $\tau = 1$, then $m = k$ so $\tau' = \tau$ and QA' is equivalent to QA. The result is thus given by Theorem 3.4. We now assume that $\tau > 1$ so $\tau' = \tau\sqrt{\frac{m}{m-1}}$, because $k \leq m-1$.

As $\tau' \leq m/k$, we can use a proof similar to Theorem 3.4 to derive Equation 3.5, which gives the following result:

$$\begin{aligned} C_{max} &\leq \left(1 + \frac{m}{k\tau'}\right) OPT + \tau' \left(1 - \frac{1}{m}\right) OPT \\ &\leq \left(1 + \tau\sqrt{\frac{m-1}{m}}\right) OPT + \tau\sqrt{1 - \frac{1}{m}}\, OPT \\ &\leq \left(1 + 2\tau\sqrt{1 - \frac{1}{m}}\right) OPT. \end{aligned}$$

Hence, the lemma. □

We now prove that the competitive ratio of QA is almost tight in the following theorem.

Theorem 3.5. *The competitive ratio of* QA *is at least* $\left(2\tau + 1 - \frac{1}{k}\right)$.

Proof. We let ε be a small processing time and we define $q = (k-1)k$.

Consider a graph composed of $q + mk + 2$ tasks, labeled by T_i for i from 1 to $q + mk + 2$. The first $q + mk + 1$ tasks are all independent. These tasks are composed of four groups:

- The first q tasks have an infinite CPU processing time and a GPU processing time equal to $x = (k-1)/q = 1/k$.
- The next mk tasks have a CPU processing time of $(1+\varepsilon)/k$ and a GPU processing time of $1/\sqrt{mk}$.
- The next task, T_{q+mk+1} has an infinite CPU processing time and a GPU processing time of ε.
- The last task of the graph, T_{q+mk+2} is a successor of T_{q+mk+1}. Its CPU processing time is equal to τ, and its GPU time is equal to $1 + \varepsilon$.

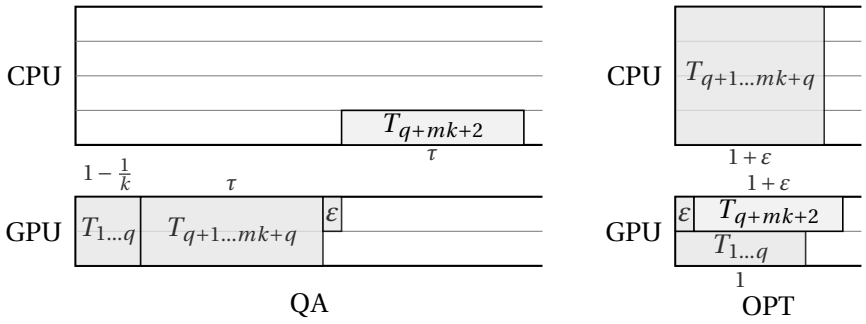


Figure 3.7: Schedule obtained by QA (left) and the optimal one (right).

We consider the online setting in which the tasks T_i arrive in the order given by i. The ratio of CPU time over GPU time is larger than τ for every task except the last one. Then, QA schedules the first q tasks on k GPUs in time $qx/k = (k-1)/k$. Then, it schedules the next mk tasks on k GPUs in time $m/\sqrt{mk} = \tau$. Task T_{q+mk+1} is then scheduled on GPU in a time ε, after which the last task is scheduled on CPU. The makespan obtained is then equal to:

$$M_{\text{QA}} = 2\tau + \frac{k-1}{k} + \varepsilon.$$

Another possibility consists in scheduling first T_{q+mk+1} then T_{q+mk+2} on a single GPU, which take a time $1+2\varepsilon$. In parallel, tasks T_1 to T_q are scheduled on the remaining $(k-1)$ GPUs, which takes a time $qx/(k-1) = 1$. In parallel, we schedule tasks T_{q+1} to T_{q+mk} on m CPUs, which are then completed at time $1 + \varepsilon$. The makespan obtained is then:

$$M = 1 + 2\varepsilon.$$

The schedules obtained are illustrated on Figure 3.7.

The ratio of the makespan obtained by QA divided by M is then equal to:

$$\frac{M_{\text{QA}}}{M} = \frac{2\tau + \frac{k-1}{k} + \varepsilon}{1 + 2\varepsilon} \quad \xrightarrow[\varepsilon \to 0]{} \quad 2\tau + \frac{k-1}{k}.$$

□

3.3.2 A tunable competitive algorithm which performs well in practice

EFT, which stands for Earliest Finish Time, is one of the most intuitive algorithm to solve this problem: it schedules each task on the resource on which it will be completed the soonest. This algorithm has good performance in practice, as the load between resources is maintained balanced. However, on some instances, it can achieve makespans $m/k+2-\frac{1}{k}$ times longer than the optimal solution or the one computed by QA, even on independent tasks, as proved in Lemma 3.4.

Lemma 3.4. *The competitive ratio of* EFT *is at least* $(m/k+2-\frac{1}{k})$*, even on independent tasks.*

Proof. Let ε be arbitrary small. We assume that k divides m and $k>1$.

We first prove a weaker result, by exposing an instance on which EFT achieves a makespan equal to m/k where the optimal result is $1+\varepsilon$.

Consider $(m+k)m/k$ tasks composed of two types. m tasks of type A have a CPU computing time equal to $1+\varepsilon$ and a GPU computing time equal to 1. The remaining m^2/k tasks, of type B, have a CPU computing time equal to 1 and a GPU computing time equal to ε.

The online instance is decomposed into m/k phases, each starting by k tasks of type A followed by m tasks of type B.

An optimal schedule allocates each task A on a single CPU, and all the tasks B on GPUs. This achieves a makespan equal to $1+\varepsilon$.

EFT allocates the first k tasks A on GPU as they complete faster (1 versus $1+\varepsilon$). Then, all the GPUs are busy until time 1, so EFT allocates the next m tasks of type B on CPU. Therefore, at the end of the first phase, all the processors are busy until time 1. Consequently, after the m/k phases, EFT achieves a makespan equal to m/k. We have then proved the first result.

This instance can now be modified to prove the lemma. Split the last phase into $k-1$ sub-phases, where each sub-phase contains the same tasks, but which computing time are divided by k. EFT schedules this phase in time $1-\frac{1}{k}$, using all processors, achieving a makespan equal to $(m-1)/k$. An optimal schedule uses $k-1$ CPUs to schedule the A tasks in time 1, and schedules the B tasks on GPUs. Now, add a new phase at the end of the instance composed of one task of type A followed by k tasks of type C, which have an infinite CPU computing time and a GPU computing time equal to 1. The schedules obtained are represented in Figure 3.8. The last A task is noted A' to differentiate it from the previous A tasks.

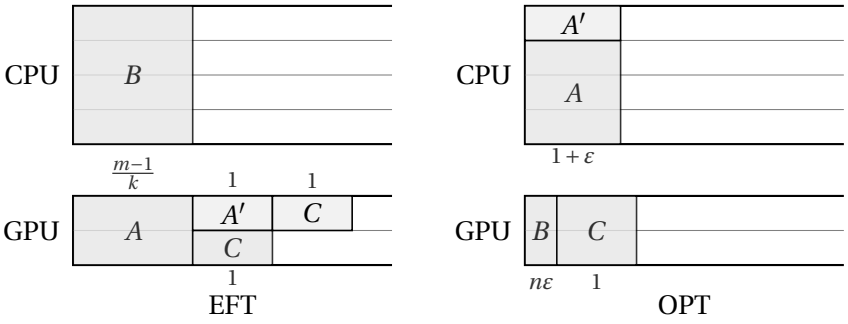


Figure 3.8: Schedule obtained by EFT (left) and an optimal one (right).

The optimal schedule executes the task A on the last idling CPU, and each task C on a GPU. The makespan obtained is then at most $1+n\varepsilon$, where n is the number of tasks in the graph.

EFT schedules the $k+1$ tasks of this phase on GPU. Its makespan is then increased by 2, to reach a value of $m/k+2-\frac{1}{k}$; hence, the lemma. □

We propose a new tunable algorithm, named MIXEFT that benefits both from the performance of EFT on most instances, and from the robustness of QA on the hardest graphs. The idea is to improve EFT by switching to a guaranteed algorithm if EFT does not perform well enough. The algorithm is composed of two phases. In the first phase, it is equal to EFT except that it also simulates the schedule that QA would have produced on the same instance. If the makespan obtained by EFT is more than λ times larger than the makespan obtained by the simulated QA (for a fixed positive parameter λ) we switch to the second phase, and MIXEFT from this point behaves as QA. A small λ leads to a smaller competitive ratio, but may degrade the performance of MIXEFT in practice. We propose to use a value of λ between 1 and 2. The pseudocode is provided in Algorithm 10.

Algorithm 10: MIXEFT (λ)

```
1  P_QA ← simulated platform
2  P_EFT ← simulated platform
3  StayEFT ← yes
4  while there is a new task T_i do
5      if StayEFT then
6          On P_EFT, Schedule T_i as soon as possible on the resource on which it completes the
             earliest
7          On P_QA, schedule T_i as soon as possible on CPU if w̄_i / w_i ≤ τ and on GPU otherwise
8          if makespan in P_EFT is λ times larger than the makespan in P_QA then
9              StayEFT ← no
10     if StayEFT then
11         Schedule T_i as soon as possible on the resource on which it completes the earliest
12     else
13         Schedule T_i as soon as possible on CPU if w̄_i / w_i ≤ τ and on GPU otherwise
```

The competitive ratio of this algorithm is in $O(\lambda\tau)$. Indeed, if OPT represents the length of the optimal schedule, QA solves the whole graph in less than $(2\tau+1)OPT$. Therefore, the time to complete the first phase is less than λ times this quantity. For the second phase, it is less than this quantity. The whole graph is then completed in less than $(\lambda+1)(2\tau+1)OPT$.

We however conjecture that the competitive ratio of MIXEFT is similar to $\max(\lambda, 2\tau+1)$. This statement is motivated by two ideas. It seems unlikely that EFT performs worse than QA on an instance in which QA is far from the optimal. So, when the switch occurs, we expect the makespan to be at most $\max(\lambda, 2\tau+1)OPT$. Secondly, when the switch occurs, it is likely that many resources are busy in the optimal solution. Therefore, we expect the makespan of the optimal solution to increase between the switch and the end of the graph. The competitive ratio is then smaller than the addition of the competitive ratio of both phases.

3.4 The allocation is more difficult than the schedule

The proofs of the lower bounds presented in Section 3.2 and the competitive algorithms designed in Section 3.3 focus substantially more on the allocation decisions (i.e., deciding whether to execute a task on CPUs or on GPUs) than on the scheduling decisions (e.g., if a task is allocated to CPUs, deciding its starting time and the CPU). In this section, we support this observation by showing that if the allocation

decisions are given by an oracle, then any online list scheduling algorithm is $(3-\frac{1}{m})$ - competitive, which is the best competitive ratio achievable.

Theorem 3.6. *If the allocation of each task is fixed, any online list scheduling algorithm is* $\left(3-\frac{1}{m}\right)$ *- competitive.*

Proof. Consider a graph where each task has a fixed alllocation, an online instance of this graph, and the schedule $\mathscr{S}$ computed by any online list scheduling algorithm, of makespan C_{max}. Let W_c (resp. W_g) be the total load on the CPUs (resp. GPUs). Let cp be a critical path of $\mathscr{S}$, and CP be the sum of the processing times of the tasks of cp in $\mathscr{S}$.

The result of Lemma 3.2 stated in the proof of Theorem 3.4 holds, therefore we have:

$$C_{max} \leq \frac{W_c}{m} + \frac{W_g}{k} + \left(1-\frac{1}{m}\right) CP.$$

Let OPT be the optimal makespan given the fixed allocation. The m CPUs have to execute tasks whose execution time sum to W_c, so $OPT \geq W_c/m$. Similarly, $OPT \geq W_g/k$, and as CP is the length of the critical path, we have $OPT \geq CP$. Therefore, we conclude that:

$$C_{max} \leq \left(3-\frac{1}{m}\right) OPT.$$

□

We now show that this upper bound is tight.

Lemma 3.5. *If the allocation of each task is fixed, no online scheduling algorithm has a competitive ratio smaller than* $\left(3-\frac{1}{m}\right)$.

Proof. We assume $m \geq 2$. Note that the result also holds for $m = 1$, with a simpler example without the second group of tasks built below. Let $\mathscr{A}$ be an online scheduling algorithm. We let n be an integer multiple of $km(m-1)$ and an adversary will build a graph G composed of the $2n+1$ following tasks:

- tasks T_1 to T_n have a GPU computing time equal to k/n and an infinite CPU computing time;
- tasks T_{n+1} to T_{2n} have a CPU computing time equal to $(m-1)/n$ and an infinite GPU computing time;
- task T_{2n+1} has a CPU computing time equal to 1 and an infinite GPU computing time.

In the graph G, there will exist $i \in [1,n]$ and $j \in [n+1,2n]$ such that the dependences of G are from task T_i to tasks $T_{n+\ell}$ for every $\ell > 0$ and from task T_j to task T_{2n+1}.

Every such graph can be scheduled in time $1+(k+m-1)/n$: schedule each task as soon as possible starting task T_i at time 0, task T_j at time k/n and task T_{2n+1} at time $(k+m-1)/n$, see Figure 3.9. Tasks $T_{1\ldots n}$ are completed on k GPUs in time $n/k * k/n = 1$, tasks $T_{n+1\ldots 2n}$ are completed on m CPUs in time $(m-1)/n * n/(m-1) = 1$, and task T_{2n+1} in time 1.

Now, consider algorithm $\mathscr{A}$. The adversary selects the last task of $T_{1\ldots 2n}$ to be terminated as the predecessor of every task $T_{n+\ell}$ for every $\ell > 0$. Similarly, it selects the last task of $T_{n+1\ldots 2n}$ to be terminated as the predecessor of task T_{2n+1}. The makespan obtained is then at least the time necessary to complete $T_{1\ldots n}$ on k GPUs, plus the time to complete $T_{n+1\ldots 2n}$ on m CPUs, plus the time to complete T_{2n+1} on 1 CPU, see Figure 3.9:

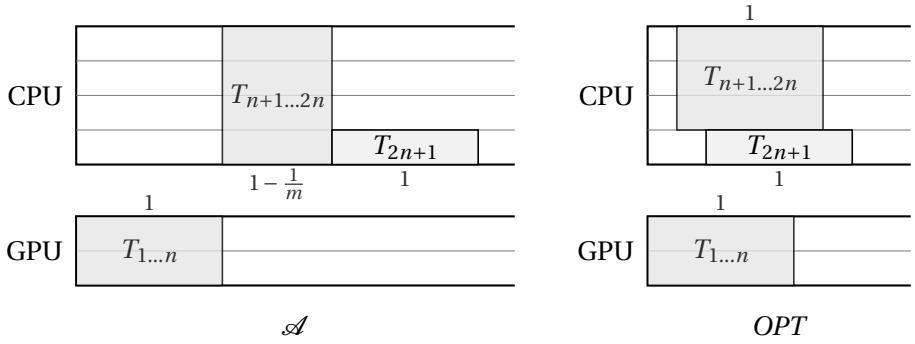


Figure 3.9: Schedules obtained by $\mathscr{A}$ and OPT.

$$1+\frac{n}{m}\,\frac{m-1}{n}+1=3-\frac{1}{m}.$$

Therefore, the competitive ratio of $\mathscr{A}$ is at least:

$$\frac{3-\frac{1}{m}}{1+\frac{1}{n}(k+m-1)} \quad \xrightarrow[n\to\infty]{} \quad 3-\frac{1}{m}.$$

□

3.5 Extension to multiple types of processors

In this section, we generalize our study to $Q \geq 2$ types of processors, which allows to model a platform composed of several accelerator types. For comparison, in the offline setting, Amaris et al. [7] provide a $Q(Q+1)$-approximation. We denote by m_q the number of processors of type q, and we assume that they are ordered such that $m_q \geq m_{q+1}$. The computing time of task T_i on processor type q is denoted by $w_{i,q}$.

Our first result directly extends the lower bounds of Section 3.2 for Q processor types. We only detail the proof of Theorem 3.7 here, but the same generalization can be done for every lower bound presented in Table 3.1, replacing τ by $\sqrt{\sum_{q=1}^{Q-1} m_q/m_Q}$.

Theorem 3.7. *No online algorithm for Q processor types has a competitive ratio smaller than $\sqrt{\sum_{q=1}^{Q-1} m_q/m_Q}$.*

Proof. Let $\mathscr{P}$ be the target platform composed of Q types of processors. Consider an alternative platform $\mathscr{P}'$ composed of 2 types of processors, m' CPUs and k' GPUs, with $m' = \sum_{q=1}^{Q-1} m_q$ and $k' = m_Q$.

Any instance G' on $\mathscr{P}'$ can be simulated by an instance G on $\mathscr{P}$, which has the same vertices and edges as G'. The processing times of the tasks of G are defined as follows: for any task T_i , $w_{i,Q}$ is equal to the GPU processing time of T_i on $\mathscr{P}'$ and $w_{i,q}$, for $q = 1,\ldots,Q-1$, is equal to its CPU processing time. Therefore, a schedule of G on $\mathscr{P}$ can be adapted as a schedule of G' on $\mathscr{P}'$ achieving the same makespan, and vice-versa: the processor types 1 to $Q-1$ are equivalent in $\mathscr{P}$ and can be mapped to the CPUs of $\mathscr{P}'$.

Suppose by contradiction that an online algorithm has a competitive ratio smaller than $\sqrt{\sum_{q=1}^{Q-1} m_q/m_Q} = \sqrt{m'/k'}$ on $\mathscr{P}$. Its competitive ratio on $\mathscr{P}'$ is then smaller than $\sqrt{m'/k'}$, which violates Theorem 3.1. □

We also adapt the QA algorithm (and thus MIXEFT) for this setting, by changing its allocation phase:

- Allocate T_i to a processor type q that minimizes $w_{i,q}/\sqrt{m_q}$.

Note that with $Q = 2$, this algorithm is equal to the original QA. Theorem 3.8 (proved below) generalizes the competitive ratio. However, the gap with the lower bound proved above increases with Q, as the lower bound is roughly the square root of the sum of ratios m_q/m_Q whereas the competitive ratio of QA is roughly the sum of the square roots of the same ratios.

Theorem 3.8. *On Q types of processors,* QA *is* $\left(\sqrt{\frac{m_1}{m_Q}} + \sum_{q=1}^{Q}\sqrt{\frac{m_q}{m_Q}}\right)$*-competitive.*

In comparison, there is an instance similar to the one of Lemma 3.4 on which EFT achieves a ratio larger than $\sum_{q=1}^{Q} m_q/m_Q$. Indeed, by setting identical computing times to processor types 1 to $Q-1$, EFT behaves as if there were $\sum_{q=1}^{Q-1} m_q$ CPUs and m_Q GPUs, which leads to this result.

The generalization of QA may seem straightforward, but it brings new insights on the underlying principles. We can see that the value $\tau = \sqrt{m/k}$ hides several concepts. Comparing the processing times ratio $\overline{w_i}/\underline{w_i}$ to τ is actually a way to select the resource type q that minimizes $w_{i,q}/\sqrt{m_q}$. The competitive ratio of QA on two types of processors, $2\tau+1$, is the sum of two terms. The first one is the maximal deceleration of a task compared to the optimal schedule, which is equal to τ on two types of processors and generalized to $\sqrt{m_1/m_Q}$. The second one is the maximal loss when scheduling too many tasks on the fastest resource type while all the others may idle, which is equal to $\tau+1$ on two types of processors, and generalizes to $\left(\sum_{q=1}^{Q}\sqrt{m_q}/\sqrt{m_Q}\right)$. The gap with the lower bound of Theorem 3.7 is explained by the fact that the proposed lower bound does not exploit the different execution times of tasks on the $Q-1$ first resource types. The construction proposed in Section 3.2 actually strongly relies on the fact that tasks have only two different processing times: either all tasks are executed on GPU, or at least one of them is not executed on GPU. Both cases must lead to the same processing time for the lower bounds to hold. It should be noted that this generalization exhibits another meaning of the value τ: it is equal to the value $\sqrt{\sum_{q=1}^{Q-1} m_q/m_Q}$ when $Q = 2$.

Proof of Theorem 3.8. This proof is similar to the one of Theorem 3.4.

We consider a graph G and the schedule $\mathcal{S}$ computed by QA, of makespan C_{max}. We consider also an optimal offline solution, to which we will refer as *the* optimal solution, of makespan OPT. Let W_q be the total load on the processors of type q for each $q \in \{1,\dots,Q-1\}$. Let cp be a critical path of $\mathcal{S}$, and CP be the sum of the processing times of the tasks of cp in $\mathcal{S}$.

First, we prove that:

$$C_{max} \leq \sum_{q=1}^{Q} \frac{W_q}{m_q} + CP. \tag{3.6}$$

As in Theorem 3.4, consider a path p of tasks of G whose execution starts the soonest and terminates exactly at time C_{max} in $\mathcal{S}$. When no task of p is being executed, one type of processor is necessarily busy because of the scheduling strategy, which always schedules an available task if one processor of each type is idle. The total amount of time during which at least one type of processor has no idle resource is at most $\sum_{q=1}^{Q} \frac{W_q}{m_q}$; hence, the result.

We now bound CP. Consider a task T_i that is executed on processor type ℓ in QA and q in the optimal solution. We have, by definition of QA, m_1 and m_Q:

$$w_{i,\ell} \leq \sqrt{\frac{m_\ell}{m_q}}\, w_{i,q} \leq \sqrt{\frac{m_1}{m_Q}}\, w_{i,q}. \tag{3.7}$$

Summing over the tasks of *cp*, we obtain:

$$CP \leq \sqrt{\frac{m_1}{m_Q}} OPT. \tag{3.8}$$

We consider the workload W_q^* on processor type q in the optimal solution, which is not larger than $m_q OPT$. For any processor type ℓ, let C_q^ℓ be the sum of the computing times on processors of type ℓ of tasks allocated to processor type ℓ in QA and to processor type q in the optimal solution. We can lower bound W_q^* in the optimal solution by the quantities C_ℓ^q, using the first inequality of Equation 3.7:

$$OPT \geq \frac{W_q^*}{m_q} \geq \frac{1}{m_q} \sum_{\ell=1}^{Q} \sqrt{\frac{m_q}{m_\ell}} C_q^\ell$$
$$\sqrt{\frac{m_q}{m_Q}} OPT \geq \sum_{\ell=1}^{Q} \frac{C_q^\ell}{\sqrt{m_Q m_\ell}} \geq \sum_{\ell=1}^{Q} \frac{C_q^\ell}{m_\ell}.$$

Now, summing over all processor types q, we get:

$$\frac{1}{\sqrt{m_Q}} \left(\sum_{q=1}^{Q} \sqrt{m_q} \right) OPT \geq \sum_{q=1}^{Q} \sum_{\ell=1}^{Q} \frac{C_q^\ell}{m_\ell} \geq \sum_{\ell=1}^{Q} \frac{1}{m_\ell} \sum_{q=1}^{Q} C_q^\ell$$
$$\geq \sum_{\ell=1}^{Q} \frac{W_\ell}{m_\ell}. \tag{3.9}$$

Finally, combining Equations 3.6, 3.8 and 3.9, we get the result:

$$C_{max} \leq \frac{1}{\sqrt{m_Q}} \left(\sqrt{m_1} + \sum_{q=1}^{Q} \sqrt{m_q} \right) OPT.$$

□

3.6 Simulations

We now provide simulations to illustrate the performance of both competitive algorithms and simple heuristic strategies on various task graphs.

3.6.1 Baseline heuristics

In addition to the four online algorithms discussed above (ER-LS from [8], QA, EFT, and MIXEFT, implemented with $\lambda = 2$ unless otherwise specified), we consider two simple strategies that follow the same scheme as QA, with a different allocation criteria: QUICKEST allocates each task to the resource type on which its computing time is smaller; RATIO allocates a task on GPUs if and only if its GPU computing time is at least m/k times smaller than its CPU computing time. Intuitively, QUICKEST should perform well on graphs on which the critical path is preponderant. On the opposite, RATIO should perform well on graphs with a high parallelism throughout the execution.

We also used the offline HEFT algorithm [139], which is known to perform well in practice, as a baseline to compare all online strategies. Moreover, backfilling is performed following HEFT insertion policy.

3.6.2 Experimental setup

We used three types of instances: realistic DAGs corresponding to the Cholesky factorization, random DAGs used in the literature, and ad hoc instances designed to be difficult for this problem and specifically for QA.

Cholesky factorization is a linear algebra application whose parallel implementation usually uses a blocked algorithm on a tiled matrix for performance issues. We consider matrix sizes ranging from 2×2 tiles to 15×15 tiles, which leads to DAGs with 4 to 680 tasks. Tasks correspond to four linear algebra kernels: GEMM, SYRK, TRSM, and POTRF. Their respective processing times on a CPU are set to 170ms, 95ms, 88ms, and 33ms, and on a GPU to 5.95ms, 3.65ms, 8.11ms, and 15.6ms, which corresponds to measures [5, 22] made using the Chameleon software [41].

The random instances come from the STG set [135], which is often used in the literature to compare the performance of scheduling strategies. The set contains instances with 50 to 5000 nodes. We report here the simulations made with 180 graphs of 300 nodes each. In these instances, 45 graphs are generated by each random DAG generator (`layrpred`, `layrprob`, `samepred` and `sameprob`). Both `layrpred` and `layrprob` generators lead to graphs with nodes structured by layers, whereas `samepred` and `sameprob` lead to more intricate graphs. In contrast to their counterpart with the suffix `-prob`, generators with the suffix `-pred` specify the average number of predecessors for each task. We consider that the cost generated by the STG random generator is the processing time of the corresponding task on a GPU. Based on the previous measures for linear algebra kernels, we assume that the average speedup between CPU and GPU is around 15 with a large variance. Thus, to obtain the processing time of a task on CPU, we multiply its cost on GPU by a random value with expected value 15 and standard deviation 15. For that, we use a gamma distribution because it has been advocated for modeling job runtimes [64], it is positive and it is possible to specify its expected value and standard deviation by adjusting its parameters.

Finally, specific random instances have been designed to test the limitations of QA. These ad hoc instances consist of a chain of tasks together with a set of independent tasks, such that all cores are expected to finish simultaneously if a GPU is dedicated to the chain and all independent tasks are load-balanced on the other cores. The expected processing time of a task on a GPU is 1 (with a standard deviation of 0.1) and the expected processing time on a CPU varies from $(m/k)^{-1/4}$ to $(m/k)^{5/4}$ (with a standard deviation equal to 10% of this expected value). For a given expected CPU cost μ, the number of tasks in the chain is $\lceil \frac{n}{m/\mu+k} \rceil$, where $n = 300$ is the total number of tasks. Therefore, the larger μ, the longer the chain.

3.6.3 Results

Figures 3.10 to 3.13 depict the performance of the six online scheduling algorithms for $m = 20$ CPUs and $k = 2$ GPUs because it best highlights the difference between the online strategies. Except when varying its parameter λ (Figure 3.13), MixEFT performs exactly as EFT (and is thus omitted for better readability).

On Cholesky DAGs (Figure 3.10), EFT (and thus MixEFT) is always the best strategy. The only difference between QA and ER-LS concerns the first tasks (as we removed Step 1a in QA), which explains why their behavior is similar for large graphs. QA, ER-LS, and Ratio all put POTRF tasks on a CPU, which leads to performance loss when the graph is small because its parallelism is limited and the GPUs are often idle. However, it is acceptable for larger graphs in which many tasks may be executed in parallel on the GPUs. On the contrary, Quickest puts all tasks on the GPUs. This is efficient for small graphs with low parallelism but it becomes worse than Ratio for large graphs.

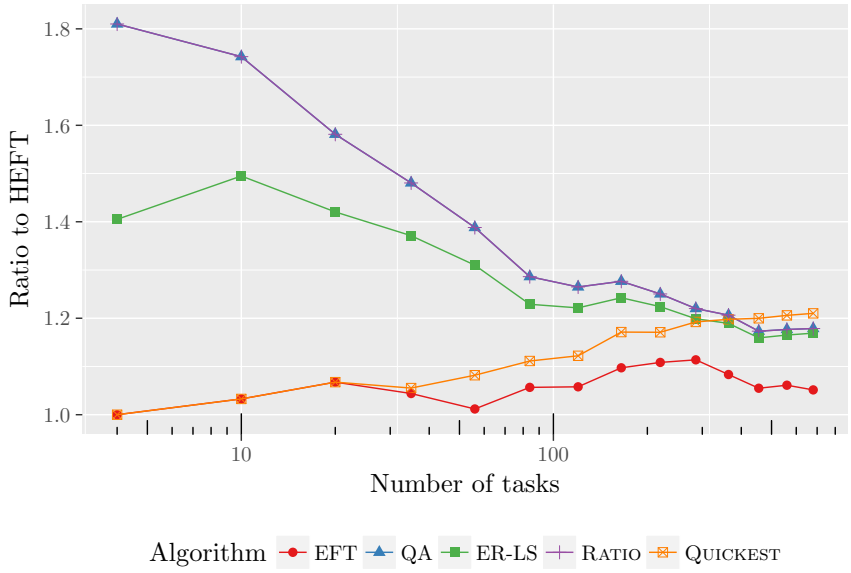


Figure 3.10: Ratios of the makespan over HEFT for EFT, QA, ER-LS, RATIO, and QUICKEST with $m = 20$ CPUs and $k = 2$ GPUs on Cholesky instances. MIXEFT is not shown because it performs exactly as EFT.

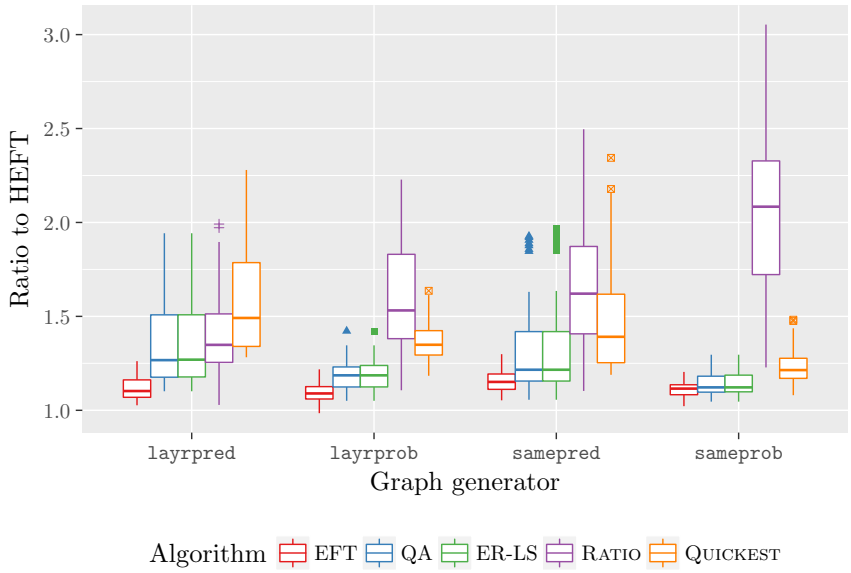


Figure 3.11: Ratios of the makespan over HEFT for EFT, QA, ER-LS, RATIO, and QUICKEST with $m = 20$ CPUs and $k = 2$ GPUs on random instances with $n = 300$ tasks from the STG data set. MIXEFT is not shown because it performs exactly as EFT.

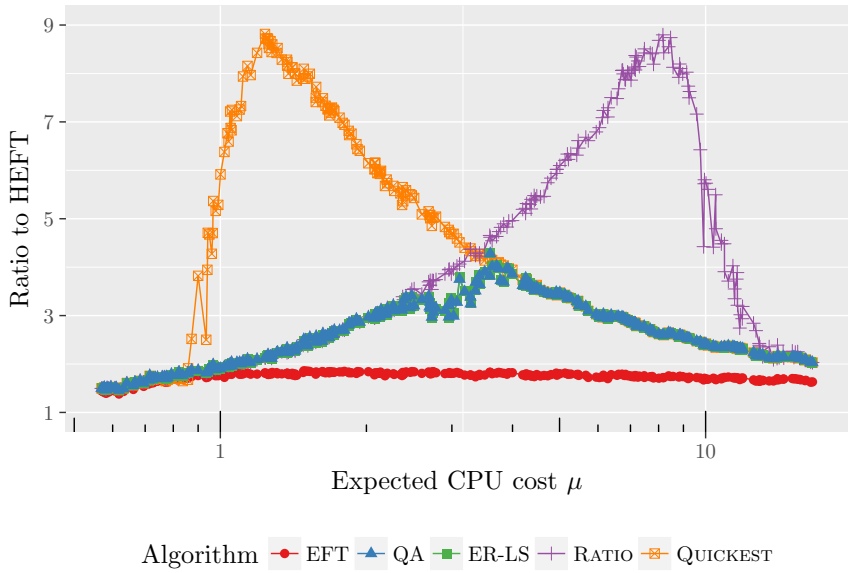


Figure 3.12: Ratios of the makespan over HEFT for EFT, QA, ER-LS, RATIO, and QUICKEST with $m = 20$ CPUs and $k = 2$ GPUs on 300 ad hoc instances with $n = 300$ tasks. MIXEFT is not shown because it performs exactly as EFT.

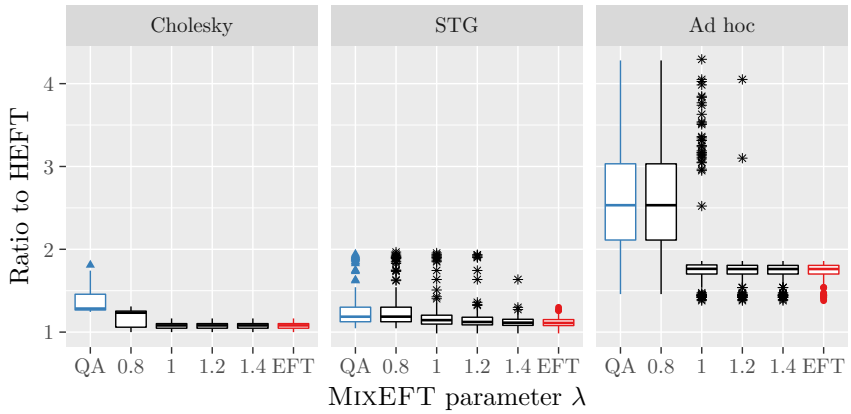


Figure 3.13: Ratios of the makespan over HEFT for QA, MIXEFT, and EFT with $m = 20$ CPUs and $k = 2$ GPUs on 14 Cholesky, 180 STG, and 300 ad hoc instances. ER-LS, RATIO, and QUICKEST are discarded.

Figure 3.11 shows that `layrpred` graphs from the STG data set yield the largest difference between QA/ER-LS and the best algorithms (HEFT and EFT). Additionally, RATIO is often better than QUICKEST. This suggests that using additional CPUs increases the efficiency and that `layrpred` graphs have some parallelism. `sameprob` graphs leads to opposite conclusions because QUICKEST performs well. Contrarily to `layrpred` graphs in which each layer becomes ready step by step, allowing the CPUs to execute some of the tasks without slowing down the GPUs, `sameprob` graphs have more intricate dependences that provide limited parallelism.

Figure 3.12 first shows that EFT (and MIXEFT) is almost always the best online heuristic for these ad hoc graphs. For extreme values of the expected CPU processing time μ (significantly smaller than 1 or larger than m/k), all four other heuristics are equivalent and perform well. Otherwise, when μ is slightly larger than 1, the instance contains many independent tasks and QUICKEST is almost m/k worst than HEFT because scheduling the independent tasks on GPUs is not efficient. Symmetrically, when μ is slightly smaller than m/k, the instance contains a large critical path and RATIO shows poor performance, because it schedules the critical path on CPUs. QA and ER-LS take the best of these two strategies, and have a worst performance $\sqrt{m/k} \approx 3$ times larger than HEFT, when μ is close to $\sqrt{m/k}$.

Figure 3.13 shows that MIXEFT behaves like QA when its parameter λ is smaller than 1, and rapidly changes to mimic EFT when the parameter increases and exceeds 1. This transition occurs for a lower λ for Cholesky instances than for STG and ad hoc ones.

Figure 3.14 shows the performance for various platform sizes for the Cholesky dataset. EFT is always the best online heuristic and its ratio to HEFT is never more than 1.2 (i.e., 20% worse than HEFT). This also applies to MIXEFT. Depending on the number of CPUs and GPUs, the other algorithms (QA, ER-LS, RATIO, and QUICKEST) follow one of the following three strategies: 1) all tasks on CPUs – this is the case for RATIO when $m/k = 20$; 2) POTRF tasks (the least accelerated tasks) on CPUs and other tasks on GPUs – this is the case for QA and ER-LS when $m/k \geq 5$, and RATIO when $3 \leq m/k \leq 10$; 3) all tasks on GPUs – all the other cases. This first strategy is the worst one except when there are many tasks and CPUs, and a single GPU. In this case, it outperforms the third strategy because the instances present a large parallelism for which CPUs can be exploited. The second strategy is often inefficient for small instances because POTRF tasks are on the critical path and benefit from being accelerated on the GPUs. Finally, the last strategy significantly deviates from EFT only for low k and large number of tasks, which suggests that it is advantageous to exploit CPUs for large graphs when there are few GPUs.

Note that in all studied instances, EFT was never far from HEFT and that there is no practical gain of using MIXEFT rather than EFT. The main advantage of MIXEFT lies in its competitive ratio whereas EFT can lead to very large makespans on specific instances.

3.7 Towards an offline approximation algorithm

In this chapter, we have focused on the online problem of scheduling a task graph on m CPUs and k GPUs, with little knowledge on the remainder of the graph. For some applications, it is nevertheless possible to know the complete graph before the execution. We therefore focus in this section on the offline problem. As stated in Section 3.1, the reference algorithm in this setting is HLP-EST, a tight polynomial-time 6-approximation based on linear programming, designed by Kedad-Sidhoum et al. in [93]. Two main challenges remain to be addressed. First, we do not know whether there exists a polynomial algorithm with a smaller approximation ratio. The related work includes a 2-approximation on homogeneous processors [73], a $\left(\frac{4}{3} + \frac{1}{3k}\right)$ - approximation for independent tasks [34], a low-complexity 2-approximation for independent tasks [40] named BALANCEDESTIMATE, and a low-complexity $(2+\sqrt{2})$-approximation algorithm named HETEROPRIO. Second, it would be interesting both theoretically and

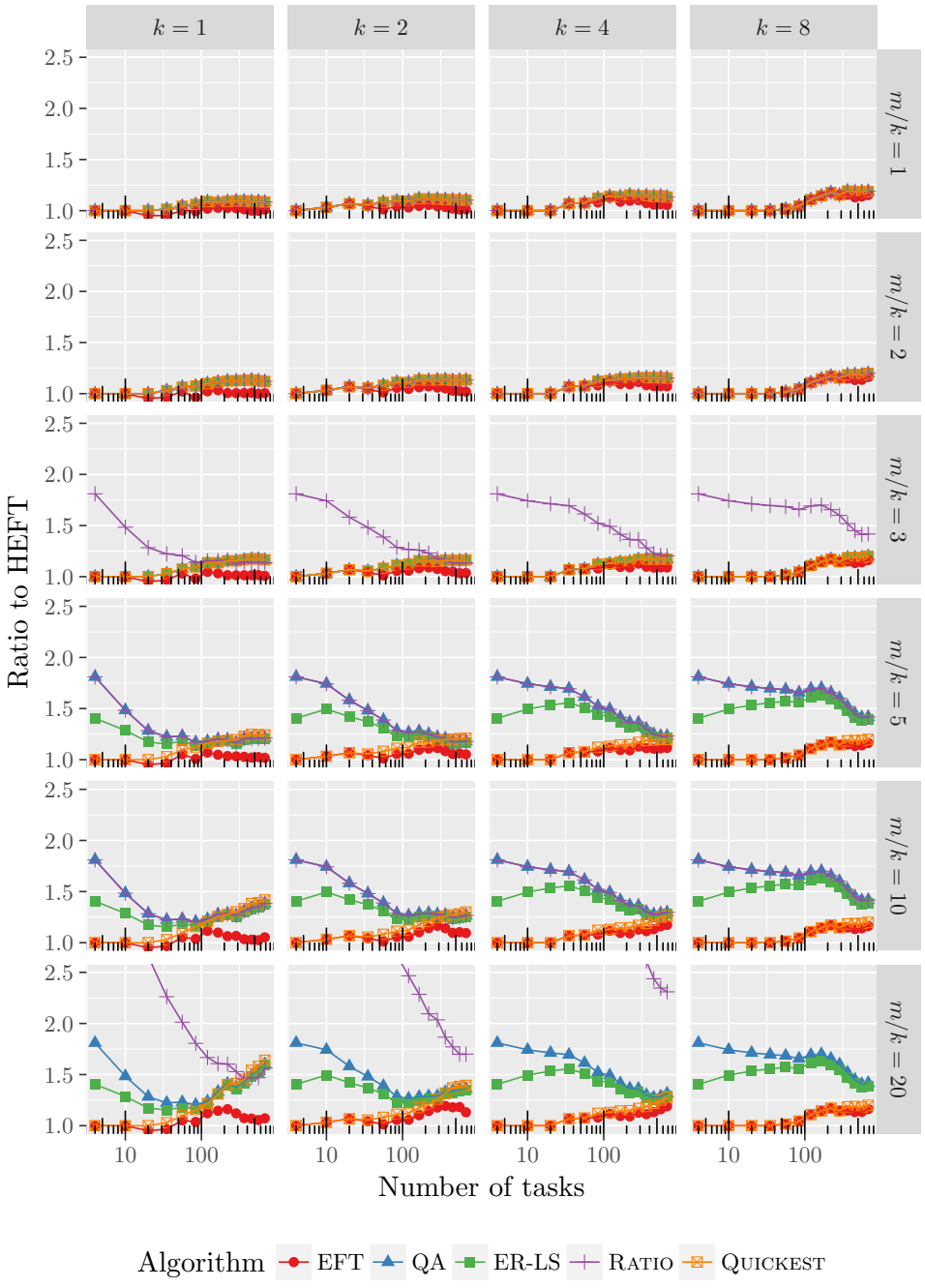


Figure 3.14: Ratios of the makespan over HEFT for EFT, QA, ER-LS, Ratio, and Quickest on Cholesky instances. MixEFT is not shown because it performs exactly as EFT. In the bottom-right plot, Ratio does not appear because its ratio is too large.

practically to design a low-complexity approximation algorithm, which does not rely on linear programming.

The main challenge when designing such an algorithm consists in determining a proper allocation (i.e., whether each task is computed on CPUs or GPUs). Indeed, as proved in Section 3.4, there is a simple 3-approximation scheduling algorithm when the allocation is fixed. In order to obtain the same guarantees as HLP-EST (a 6-approximation), it would then be sufficient that the optimal makespan of the allocation returned is at most twice the optimal makespan for any allocation. One may think that an approximation algorithm can be obtained by designing variants of BALANCEDESTIMATE or HETEROPRIO to allocate and schedule the (independent) available tasks, with additional information such as their bottom-levels. However, such an attempt cannot lead to a constant-approximation ratio as proved in Theorem 3.3.

The main concept of BALANCEDESTIMATE consists in first allocating each task on the processor type (CPUs or GPUs) on which its computing time is smaller, and then moving tasks from the most loaded processor type to the other one. The tasks that are moved first are the ones that suffer the less: the ones for which the computing time is increased by a smaller factor. Some tasks also need to be moved back on their original resource type. Typically, it may be better to substantially slow down a small task than to slightly slow down a large task.

We have tried to extend such an idea with precedence constraints, by deciding the allocation of all the tasks in the same phase, and not only of available tasks. Therefore, this does not lead to an online algorithm and the results of Section 3.2 do not apply. However, most of our attempts failed on the instance G_{ABCD} described in Example 3.1. In this instance, the optimal solution, depicted in Figure 3.15, schedules the chain composed of n tasks A on the unique GPU, followed by the n^2 tasks B scheduled each on one CPU. In parallel to tasks A, tasks C are scheduled on CPUs. In parallel to tasks B, tasks D are scheduled on the GPU. The achieved makespan is then equal to $2n + n\varepsilon$.

Example 3.1. *Let n be an integer, and $\varepsilon > 0$ be arbitrary small. Consider a platform with n^2 CPUs and 1 GPU. The graph G_{ABCD} is composed of $n^4 + 2n^2 + n$ tasks split in four types, see Table 3.2 for the details. The A tasks form a chain, and B tasks cannot be started before all tasks A are terminated. Tasks C and D have no precedence constraints.*

Task Type	Number of tasks	$\overline{w}$ (CPU)	$\underline{w}$ (GPU)	Optimal allocation
A	n	n	$1+\varepsilon$	GPU
B	n^2	n	1	CPUs
C	n^2	n	$1-\varepsilon$	CPUs
D	n^4	n	ε	GPU

Table 3.2: Tasks composing the graph G_{ABCD}, to be executed on n^2 CPUs and 1 GPU.

If we apply a concept similar to BALANCEDESTIMATE, all tasks are first allocated to the GPU. The next phase consists in selecting which tasks should be moved to CPUs, considering them by decreasing ratio of $\overline{w}/\underline{w}$. Therefore, tasks A are considered first, then tasks B, C, and D. In order to obtain an approximation algorithm, most tasks A have to be scheduled on the GPU, and most tasks B on CPUs. We first considered two parameters to determine if a task should be moved to CPUs: whether moving it to CPUs increases the critical path, and whether moving it to CPUs increases the current makespan (computed either by a list algorithm or by an oracle giving the optimal schedule). The issue is that, in this initial state, because of tasks C, moving tasks A or B to CPUs increases the critical path but decreases the current makespan. As tasks A are considered first, these parameters are not sufficient to

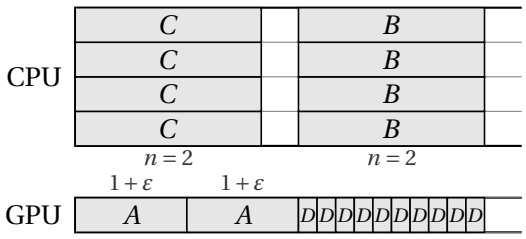


Figure 3.15: Scheme of the optimal schedule of Example 3.1 for $n = 2$.

move only tasks B to CPUs. A workaround would be for instance to initially move to CPUs every task that does not increase the critical path nor the makespan. But such a procedure would move both tasks C and D to CPUs, which leads to a high makespan. Some algorithms designed were able to schedule this example, but then either fail on independent tasks or on instances used in Theorems 3.2 and 3.3.

The main difficulty with this problem resides in the fact that, given an allocation, there does not seem to be any simple metric to determine whether a given task should be moved on a slower resource. Indeed, it can be advantageous to move a set of tasks to a slower resource whereas there is no gain in moving only one task. The challenge remains in determining whether a given task belongs to such a set. In [93], the authors successfully tackle this issue by relying on linear programming and rounding methods to design a 6-approximation named HLP-EST.

3.8 Conclusion

In this chapter, we have focused on the problem of scheduling task graphs on hybrid platforms made of two types of processors, such as CPUs and GPUs. We have studied the online case, when only the tasks whose predecessors are all completed are known to the scheduler, and the graph is thus gradually discovered. We proved that no scheduling algorithm can have a competitive ratio smaller than $\sqrt{m/k}$, and studied how this ratio varies when more knowledge on the graph is given to the scheduler and/or tasks may be migrated between processors. We have proposed a $(2\sqrt{m/k}+1)$-competitive algorithm as well as a mixed strategy, which is both $\Theta(\sqrt{m/k})$-competitive and performs as well as the best heuristics in practice. This is demonstrated through an extensive set of simulations. We have also extended the lower bounds and the competitive algorithms to the case with more than two types of processors.

Future work includes several directions. For independent tasks, there is still a gap between the best lower bound on online algorithms competitive ratio (2) and the best online algorithm (3.85-competitive) [43]. As discussed in Section 3.7, an offline approximation algorithm to schedule DAGs not relying on linear programming would have both theoretical and practical interests. The results obtained in this chapter exhibit some difficult instances to test candidate algorithms. The current best approximation ratio being 6, we can also wonder whether it can be improved. Another research direction consists in exploiting task parallelism, such as in Chapters 1 and 2. A 2-approximation has been exhibited for offline scheduling of independent tasks on hybrid platforms in [33], but this problem with precedence constraints remains unexplored. Finally, in order to model more closely realistic problems, it remains to take into account communication times when moving data from/to the GPUs, and to cope with inaccurate processing time estimates.

Chapter 4

Coping with a limited available memory

« Encore un papyrus ! Ma pauvre mémoire est pleine ! Parfois, j'ai l'impression de ne plus rien pouvoir graver ! »

Archéopteryx, *Le Papyrus de César*

One of the main objectives that have been considered in the literature (and in the previous chapters of this manuscript) concerning task graph scheduling consists in minimizing the makespan, or total completion time. However, with the increase of the size of the data to be processed, the memory footprint of the application can have a dramatic impact on the application execution time, and thus needs to be optimized [4, 125]. This is best exemplified with an application which, depending on the way it is scheduled, will either fit in the memory, or will require the use of swap mechanisms or *out-of-core* execution. There are few existing studies that take into account memory footprint when scheduling task graphs, as detailed below in the related work section.

Our focus in this chapter concerns the execution of highly-parallel applications on a shared-memory platform. Depending on the scheduling choices, the computation of a given task graph may or may not fit into the available memory. The goal is then to find the most suitable schedule (e.g., one that minimizes the makespan) among the schedules that fit into the available memory. A possible strategy is to design a static schedule before the computation starts, based on the predicted task durations and data sizes involved in the computation, similarly to what we did in Chapters 1 and 2. However, for some applications, there is little chance that such a static strategy would reach high performance: task duration estimates may be inaccurate, data transfers on the platform are hard to correctly model, and the resulting small estimation errors may accumulate and cause large delays. Thus, most practical schedulers such as the runtime systems cited above rely on *dynamic* scheduling, where task allocations and their execution order are decided at runtime, based on the system state. The risk with dynamic scheduling, however, is the simultaneous scheduling of a set of tasks whose total memory requirement exceeds the available memory, a situation that could induce a severe performance degradation.

Main contributions. In this chapter, our aim is both to enable dynamic scheduling of task graphs with memory requirements and to guarantee that at no time during the execution the available memory is exceeded. We achieve this goal by adding fictitious dependences in the graph to cope with memory constraints: these additional edges will restrict the set of valid schedules and in particular forbid the concurrent execution of too many memory-intensive tasks. This idea is inspired by [128], which applies a similar technique to graphs of smaller-grain tasks in which all the data have size 1. The quality of a

solution is measured by the length of the critical path of the graph comprising the fictitious edges. We prove that the problem of computing such fictitious edges leading to a small critical path is NP-hard. Therefore, we propose both an ILP formulation and several heuristics, which are evaluated through simulations.

Note that, contrarily to Chapters 1 and 2, we mainly target sequential tasks in this chapter, and contrarily to Chapter 3, we consider a shared-memory platform of identical processors. Therefore, we assume that the critical path of a graph is a good indicator of the makespan obtained by a dynamic scheduler for this graph on a reasonable number of processors, and that the increase of the critical path thus should be minimized by the proposed heuristics. However, the other results of this chapter only need that the memory of the platform is shared and not distributed, and can therefore be applied to parallel tasks, or even to platforms composed of different types of processors sharing the same memory.[1]

The rest of the chapter is organized as follows:

- We first briefly review the existing work on memory-aware task graph scheduling (Section 4.1).
- We propose a very simple task graph model which both accurately describes complex memory behaviors and is amenable to memory optimization (Section 4.2).
- We introduce the notion of the maximum peak memory of a workflow: this is the maximum peak memory of any (sequential or) parallel execution of the workflow. We then show that the maximum peak memory of a workflow is exactly the weight of a special cut in this workflow, called the maximum topological cut. Finally, we propose a polynomial-time algorithm to compute this cut (Section 4.3).
- In order to cope with limited memory, we formally state the problem of adding edges to a graph to decrease its maximum peak memory, with the objective of not harming too much the makespan of any parallel execution of the resulting graph. We prove this problem to be NP-hard and propose both an ILP formulation and several heuristics to solve it on practical cases (Section 4.4). Finally we evaluate the heuristics through simulations on synthetic task graphs produced by classical random workflow generators (Section 4.5). The simulations show that the two best heuristics have a limited impact on the makespan in most cases, and one of them is able to handle all studied workflows.

4.1 Related work

Memory and storage have always been a limited parameter for large computations, as outlined by the pioneering work of Sethi and Ullman [131] on register allocation for task trees. A similar model has been used in [61] to design parallel schedules using a limited number of available registers while minimizing the makespan. This model was later translated to the problem of scheduling a task graph under memory or storage constraints for scientific workflows whose tasks require large I/O data. Such workflows arise in many scientific fields, such as image processing, genomics, and geophysical simulations. The problem of task graphs handling large data has been identified by Ramakrishnan et al. [125] who introduce clean-up jobs to reduce the memory footprint and propose some simple heuristics. Their work was continued by Bharathi et al. [31] who develop genetic algorithms to schedule such workflows. This problem also arises in sparse direct solvers, as highlighted by Agullo et al. [4] who study the effect of processor mapping on memory consumption for multifrontal methods. In some cases, such as for sparse direct solvers, the task graph is a tree, for which specific methods have been proposed, both to reduce the minimum peak memory [105] and to design memory-aware parallel schedulers [19]. Directed graphs

[1] Such an architecture appears for instance when a master core is associated to slave cores, see www.netlib.org/utk/people/JackDongarra/PAPERS/sunway-report-2016.pdf.

are also used to model a different type of applications, in which the graph may be cyclic. Data stream processing in embedded systems, for instance, can be modeled as a set of several nodes performing operations infinitely often and exchanging output data via buffers. In this context, several authors have also aimed at minimizing the memory (or total buffer size) usage. Relying on the Synchronous DataFlow paradigm [99] or related models, several papers [25, 26, 144, 145] studied how to minimize the buffer usage while maintaining a given minimal throughput (which is related to the makespan in our model). It should be noted that the problem is quite different in this context as it can be simplified to deciding how much memory will be allocated to each node. Indeed, the schedule itself is then implicit as each node is executed every time it has enough input data.

As explained in the introduction, our study is inspired by the work of Sbîrlea at al. [128] from 2014. This study focuses on a different model, in which all data have the same size. They target smaller-grain tasks in the Concurrent Collections (CnC) programming model [37], a stream/dataflow programming language. Their objective is, as ours, to schedule a DAG of tasks under a limited memory. For this, they associate a color to each memory slot and then build a coloring of the data, in which two data with the same color cannot coexist. If the number of colors is not sufficient, additional dependence edges are introduced to prevent two data to coexist. These additional edges respect a pre-computed sequential schedule to ensure acyclicity. An extension to support data of different sizes is proposed, which conceptually allocates several colors to a single data, but is only suited for a few distinct sizes. This concept of adding edges to a DAG in order to ensure that no schedule will use too much memory has been previously studied by Touati in his PhD thesis conducted in the early 2000s [140, Chapter 4]. In his work, the tasks of the considered graph correspond to fine-grain instructions operating on few registers, which will be executed on an Instruction Level Parallelism processor. The different computing units share the same registers. Hence, the model is similar to the one in [128]: all data have the same size.

In the realm of runtime systems, memory footprint is a real concern. In StarPU, attempts have been made to reduce memory consumption by throttling the task submission rate [129].

Compared to the existing work, the present work studies graphs with arbitrary data sizes, and it formally defines the problem of transforming a graph to cope with a strong memory bound: this allows the use of efficient dynamic scheduling heuristics at runtime with the guarantee to never exceed the memory bound.

4.2 Problem modeling

4.2.1 Formal description

As stated before, we consider that the targeted application is described by a workflow of tasks whose precedence constraints form a DAG $G = (V, E)$. Its nodes $i \in V$ represent tasks and its edges $e \in E$ represent precedence, in the form of input and output data. The processing time necessary to complete a task $i \in V$ is denoted by w_i. In our model, the memory usage of the computation is modeled only by the size of the data produced by the tasks and represented by the edges. Therefore, for each edge $e = (i, j)$, we denote by m_e or $m_{i,j}$ the size of the data produced by task i for task j. We assume that G contains a single source node s and a single sink node t; otherwise, one can add such nodes along with the appropriate edges, all of null weight. For the sake of simplicity, we define the following sizes of inputs and outputs of a node i:

$$Inputs(i) = \sum_{j|(j,i)\in E} m_{j,i} \qquad Outputs(i) = \sum_{j|(i,j)\in E} m_{i,j}$$

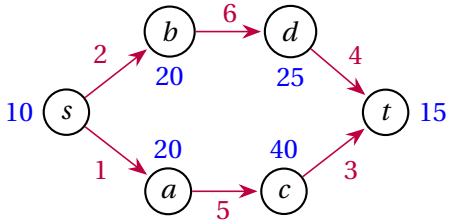


Figure 4.1: Example of a workflow, (red) edge labels represent the size $m_{i,j}$ of associated data, while (blue) node labels represent their computation weight w_i.

We propose here to use a very simple memory model, which might first seem unrealistic, but will indeed prove itself very powerful both to model complex memory behaviors and to express the peak memory usage. In the proposed model, at the beginning of the execution of a task i, all input data of i are immediately deleted from the memory, while all its output data are allocated to the memory. That is, the amount of used memory M_{used} is transformed as follows:

$$M_{\text{used}} \leftarrow M_{\text{used}} - \mathit{Inputs}(i) + \mathit{Outputs}(i)\,.$$

This model, called the SIMPLEDATAFLOWMODEL, is extremely simple, and in particular does not allow a task to have both its inputs and outputs simultaneously in memory. However, we will see right below that it is expressive enough to emulate other complex and more realistic behaviors.

Before considering other memory models, we start by defining some terms and by comparing sequential schedules and parallel execution of the graph. We say that the data associated to the edge (i, j) is ***active*** at a given time if the execution of i has started but not the one of j. This means that this data is present in memory. A ***sequential schedule*** $\mathcal{S}$ of a DAG G is defined by an order σ of its tasks. The ***memory used*** by a sequential schedule at a given time is the sum of the sizes of the active data. The ***peak memory*** of such a schedule is the maximum memory used during its execution. A ***parallel execution*** of a graph on p processors is defined by:

- An allocation μ of the tasks onto the processors (task i is computed on processor $\mu(i)$);
- The starting times σ of the tasks (task i starts at time $\sigma(i)$).

As usual, a valid schedule ensures that data dependences are satisfied ($\sigma(j) \geq \sigma(i) + w_i$ whenever $(i, j) \in E$) and that processors compute a single task at each time step (if $\mu(i) = \mu(j)$, then $\sigma(j) \geq \sigma(i) + w_i$ or $\sigma(i) \geq \sigma(j) + w_j$). Note that when considering parallel execution, we assume that all processors use the same shared memory, whose size is limited.

A very important feature of the proposed SIMPLEDATAFLOWMODEL is that there is no difference between *sequential schedules* and *parallel execution* as far as memory is concerned, which is formally stated in the following theorem.

Theorem 4.1. *For each parallel execution (μ,σ) of a DAG G, there exists a sequential schedule with equal peak memory.*

Proof. We consider such a parallel execution, and we build the corresponding sequential schedule by ordering tasks in non decreasing starting time. Since in the SIMPLEDATAFLOWMODEL, there is no difference in memory between a task being processed and a completed task, the sequential schedule has the same amount of used memory as the parallel execution after the beginning of each task. Thus, they have the same peak memory. □

This feature will be very helpful when computing the maximum memory of any parallell execution, in Section 4.3: thanks to the previous result, it is equivalent to computing the peak memory of a sequential schedule.

4.2.2 Emulation of other memory models

Classical workflow model

As we explained above, our model does not allow inputs and outputs of a given task to be in memory simultaneously. However, this is a common behavior, and some studies, such as [88], even consider that in addition to inputs and outputs, some temporary data m_i^{temp} has to be in memory when processing task i. The memory needed for its processing is then $Inputs(i) + m_i^{temp} + Outputs(i)$. Although this is very different to what happens in the proposed SIMPLEDATAFLOWMODEL, such a behavior can be simply emulated, as illustrated on Figure 4.2. For all task i, we split it into two nodes i_1 and i_2. We transform all edges (i, j) by edges (i_2, j), and edges (k, i) by edges (k, i_1). We also add an edge (i_1, i_2) with an associated data of size $Inputs(i) + m_i^{temp} + Outputs(i)$. Task i_1 represents the allocation of the data needed for the computation, as well as the computation itself, and its work is thus $w_{i_i} = w_i$. Task i_2 stands for the deallocation of the input and temporary data and has work $w_{i_2} = 0$.

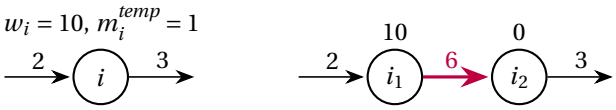


Figure 4.2: Transformation of a task as in [88] (left) to the SIMPLEDATAFLOWMODEL (right).

Shared output data

Our model considers that each task produces a separate data for every of its successors. However, it may well happen that a task i produces an output data d, of size $m_{i,d}^{shared}$, which is then used by several of its successors, and can be freed after the completion of these successors. The output data is then shared among successors, contrarily to what is considered in the SIMPLEDATAFLOWMODEL. Any task can then produce several output data, some of which can be shared among several successors. Again, such a behavior can easily be emulated in the proposed model, as illustrated on Figure 4.3.

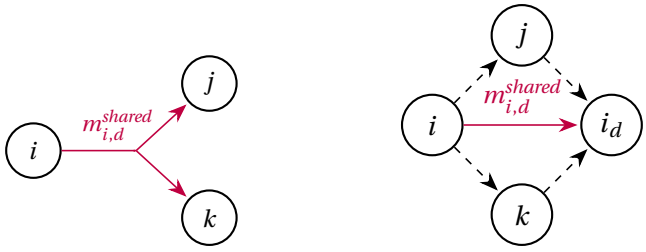


Figure 4.3: Transformation of a task i with a single shared output data into SIMPLEDATAFLOWMODEL. The plain (red) edge carries the shared data size, while dashed (black) edges have null size.

Such a task i with a shared output data will first be transformed as follows. For each shared output data d of size $m_{i,d}^{shared}$, we add a task i_d which represents the deallocation of the shared data d (and thus has null computation time w_{i_d}). An edge of size $m_{i,d}^{shared}$ is added between i and i_d: $m_{i,i_d} = m_{i,d}^{shared}$.

Data dependence to a successor j sharing the output data d is represented by an edge (i, j) with null data size ($m_{i,j} = 0$) (if it does not already exist, due to an other data produced by i and consumed by j). Finally, for each such successor j, we add an edge of null size (j, i_d) to ensure that the shared data will be freed only when it has been used by all the successors sharing it. The following result states that after this transformation, the resulting graph correctly models the memory behavior.

Theorem 4.2. *Let G be a DAG with shared output data, and G' its transformation into* SIMPLEDATAFLOWMODEL. *There exists a schedule $\mathcal{S}$ of G with peak memory M if and only if there exists a schedule $\mathcal{S}'$ of G' with peak memory at most M.*

Proof. First, consider a schedule $\mathcal{S}$ which executes the graph G with a memory of size M. We assume that $\mathcal{S}$ frees shared output data as soon as possible (otherwise we first transform it into a schedule freeing shared output data as soon as possible, which does not increase the peak memory). We transform $\mathcal{S}$ into a schedule $\mathcal{S}'$ of G. When $\mathcal{S}$ schedules a node i of G, $\mathcal{S}'$ schedules the same node i of G'. When $\mathcal{S}$ frees a shared data d output by node i, $\mathcal{S}'$ schedules node i_d. We now show by induction on $\mathcal{S}$ that $\mathcal{S}'$ is a valid schedule on G' and that both schedules use the same amount of memory at any time. Suppose $\mathcal{S}'$ valid for the first k operations of $\mathcal{S}$, and consider the following one. When $\mathcal{S}$ schedules a node i of G, $\mathcal{S}'$ schedules the same node of G'. Its predecessors are then completed. The sum of the sizes of the output data of i in G and G' are equal, as when the transformation removes a shared output data, it adds a single edge of the same size, along with null-weight edges. If a shared data d output by a task j is freed in $\mathcal{S}$, then task j_d is executed in $\mathcal{S}'$, which reduces the memory consumption by the size of the data d. Therefore, $\mathcal{S}$ and $\mathcal{S}'$ have the same memory consumption for an additional operation. By induction, we get the result.

Now, suppose there exists a schedule $\mathcal{S}'$ of G' with a peak memory equal to M. We transform $\mathcal{S}'$ into a schedule $\mathcal{S}$ of G. When $\mathcal{S}'$ schedules a node i of G', $\mathcal{S}$ schedules the same node i of G. When $\mathcal{S}'$ schedules node i_d of G', $\mathcal{S}$ frees the shared data d output by node i. As in the previous case, We now show by induction on $\mathcal{S}'$ that $\mathcal{S}$ is a valid schedule on G and that both schedules use the same amount of memory at any time. Suppose $\mathcal{S}$ valid for the first k operations of $\mathcal{S}'$, and consider the following one. If $\mathcal{S}'$ schedules a node i of G', $\mathcal{S}$ schedules the same node of G. As previously, the precedence is respected, and the memory consumed is the same in both schedules. If $\mathcal{S}'$ schedules a node i_d of G', $\mathcal{S}$ frees the shared data d output by task i. The nodes consuming this data are completed, so this operation is authorized, and the memory consumption is reduced by the size of data d in both cases. By induction, we get the result. □

Pebble game

One of the pioneer work dealing with the memory footprint of a DAG execution has been conducted by Sethi [130]. He considered what is now recognized as a variant of the PEBBLEGAME model. We now show that the proposed SIMPLEDATAFLOWMODEL is an extension of PEBBLEGAME. The pebble game is defined on a DAG as follows:

- A pebble can be placed on a node with no predecessor at any time;
- A pebble can be placed on a node if all its predecessors have a pebble;
- A pebble can be removed from a node at any time;
- A pebble cannot be placed on a node that has been previously pebbled.

The objective is to pebble all the nodes of a given graph, using a minimum number of pebbles. Note that the pebble of a node should be removed only when all its successors are pebbled. This is the main difference with our model, where a node produces a different output data for each of its successors. Thus, the PEBBLEGAME model ressembles the model with shared output data presented above, with all data of size one. We thus apply the same transformation and consider that a pebble is a shared output data used for all the successors of a node. In addition, we add a fictitious successor to all nodes without successors. Hence, the pebble placed on such a node can be considered as the data consumed by this successor. Then, we are able to prove that the memory behavior of the transformed graph under SIMPLEDATAFLOWMODEL corresponds to the pebbling of the original graph, as outlined by the following theorem.

Theorem 4.3. *Let P be a DAG representing an instance of a* PEBBLEGAME *problem, and G its transformation into* SIMPLEDATAFLOWMODEL. *There exists a pebbling scheme $\mathscr{P}$ of P using at most B pebbles if and only if there exists a schedule $\mathscr{S}'$ of G' with peak memory at most B.*

Proof. In the PEBBLEGAME model, we can consider that every node outputs a single data of size one consumed by all its successors. Recall that for a node with no successor, the transformation acts as if a fictitious successor existed for this node. Therefore, the transformation adds one node for each node u in the graph P. In order to clarify whether we consider the graph P or G, we call u_1 the node of G that corresponds to the node u of P and u_2 the node of G that corresponds to the data output by node u of P. If u has no successor in P, we denote f_u the fictitious node added in G, successor of u_1 and predecessor of u_2. See Figure 4.4 for an illustration.

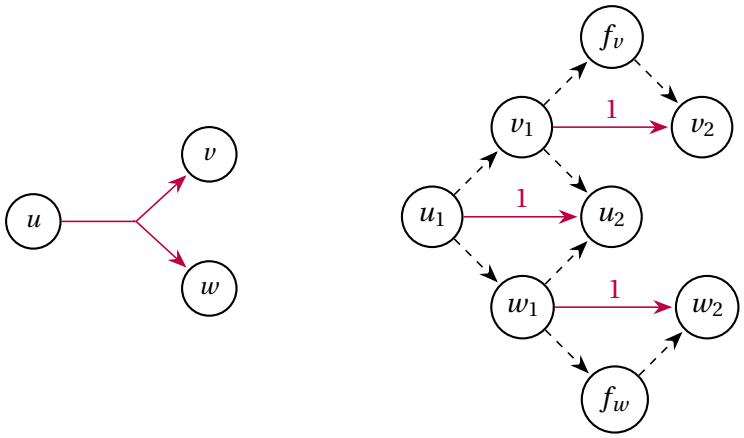


Figure 4.4: Transformation of an instance of the PEBBLEGAME problem into an instance of SIMPLEDATAFLOWMODEL.

First, we consider a traversal $\mathscr{P}$ which traverses P with B pebbles. We transform $\mathscr{P}$ into a schedule $\mathscr{S}$ of G: when $\mathscr{P}$ pebbles a node u of P, $\mathscr{S}$ executes the node u_1 of G, when $\mathscr{P}$ removes a pebble from a node u of P, $\mathscr{S}$ executes the node u_2 of G. If u has no successor in P, $\mathscr{S}$ first executes f_u then u_2. We now show by induction on $\mathscr{P}$ that $\mathscr{S}$ is a valid schedule on G. Suppose $\mathscr{S}$ valid for the first k operations of $\mathscr{P}$, and consider the following one. If $\mathscr{P}$ pebbles a new node u, this means that u_1 was not executed before by $\mathscr{S}$, as recomputations are forbidden, and that all the predecessors v^i of u have been pebbled by $\mathscr{P}$, so that the nodes v_1^i have been executed by $\mathscr{S}$ in G. These nodes v_1^i correspond to the predecessors of u_1 in G, so the execution of u_1 is valid. If $\mathscr{P}$ unpebbles a node u, this means that its successors v^i have already been pebbled. Indeed, otherwise, as recomputations are not allowed, $\mathscr{P}$ would not be a valid schedule. Therefore, as the predecessors of u_2 in G are u_1 and the v_1^i, these nodes

have been executed by $\mathscr{S}$, so the execution of u_2 is valid. If u has no successor in P, then the execution of f_u and u_2 is valid. Finally, $\mathscr{S}$ is a valid schedule. At any time, the memory used by $\mathscr{S}$ is equal to the numbers of nodes u of P such that u_1 is executed but not u_2. This is equal to the number of pebbles required by $\mathscr{P}$, so $\mathscr{S}$ is a valid schedule of G using a memory of size B.

Now, we consider a schedule $\mathscr{S}$ of G with a peak memory equal to B. We transform $\mathscr{S}$ into a traversal $\mathscr{P}$ of P: when $\mathscr{S}$ executes a node u_1, $\mathscr{P}$ pebbles the node u, and when $\mathscr{S}$ executes a node u_2, $\mathscr{P}$ removes the pebble of node u. Nothing is done when $\mathscr{S}$ executes a node f_u. We now show by induction on $\mathscr{S}$ that $\mathscr{P}$ is a valid traversal of P. Suppose $\mathscr{P}$ valid for the first k operations of $\mathscr{S}$, and consider the following one. First, suppose that $\mathscr{S}$ executes a node u_1. Let v^i be the predecessors of u in P. By the precedence constraints, we know that the nodes v_1^i have already been executed by $\mathscr{S}$, and that the nodes v_2^i have not. Therefore, $\mathscr{P}$ has pebbled the nodes v^i, but have not unpebbled them. So $\mathscr{P}$ is allowed to pebble u. Now, suppose that $\mathscr{S}$ executes a node u_2. The node u_1 has already been pebbled by the precedence constraints, so removing this pebble is a valid move. Therefore, $\mathscr{P}$ is a valid traversal of P. As above, at any time, the memory used by $\mathscr{S}$ is equal to the numbers of nodes u of P such that u_1 is executed but not u_2. This is equal to the numbers of pebbles used by $\mathscr{P}$. □

4.2.3 Peak memory minimization in the proposed model

The emulation of the PEBBLEGAME problem, as proposed above, allows us to formally state the complexity of minimizing the memory of a DAG, as expressed by the following theorem.

Theorem 4.4. *Deciding whether an instance of* SIMPLEDATAFLOWMODEL *can be scheduled with a memory of limited size is NP-complete.*

Proof. The problem of deciding whether an instance of PEBBLEGAME can be traversed with a given number of pebbles is NP-complete [130]. Then, thanks to Theorem 4.3, we know that an instance of PEBBLEGAME can be transformed into an instance of SIMPLEDATAFLOWMODEL (with twice as many nodes), which then inherits of this complexity result. □

4.3 Computing the maximal peak memory

In this section, we are interested in computing the ***maximal peak memory*** of a given DAG $G = (V, E)$, that is, the largest peak memory that can be reached by a sequential schedule of G. Our objective is to check whether a graph can be safely executed by a dynamic scheduler without exceeding the memory bound.

We first define the notion of ***topological cut***. We recall that G contains a single source node s and a single sink node t.

Definition 4.1. *A topological cut (S, T) of a DAG G is a partition of G in two sets of nodes S and T such that $s \in S$, $t \in T$, and no edge is directed from a node of T to a node of S. An edge (i, j) belongs to the cut if $i \in S$ and $j \in T$. The weight of a topological cut is the sum of the weights of the edges belonging to the cut.*

For instance, in the graph of Figure 4.1, the cut $(\{s, a, b\}, \{c, d, t\})$ is a topological cut of weight 11. In the SIMPLEDATAFLOWMODEL, the memory used at a given time is equal to the sum of the sizes of the active output data, which depends solely on the set of nodes that have been executed or initiated. Therefore, the maximal peak memory of a DAG is equal to the maximum weight of a topological cut.

Definition 4.2. *The* MAXTOPCUT *problem consists in computing a topological cut of maximum weight for a given DAG.*

We first prove that this problem is polynomial, by providing a linear program over the rationals solving it, and then propose an explicit algorithm which does not rely on linear programming.

4.3.1 Complexity of the problem

The MaxTopCut problem belongs to the family of problems in which we are interested in computing a weighted cut in a graph that optimizes some quantity.

The problem of finding a cut of *minimum* weight (when edge weights are nonnegative) has been thoroughly studied in the literature, and many polynomial-time algorithms have been proposed to solve it, both undirected and directed graphs [98]. On the opposite, computing a maximal cut is in general much more difficult. It is well-known that this problem is NP-complete on general graphs, both undirected and directed [92], and with unit weights [67]. In 2011, Lampis et al. even extend this result to DAGs [97], which are our scope of interest. However, our problem is more restrictive, as we are only interested in maximal *topological cuts* on DAGs, which means that all the edges of the cut have the same direction. This constraint actually heavily reduces the set of possible cuts. There are 2^n possible cuts for any DAG with n nodes: the number of ways to partition the nodes in two sets. However, the number of topological cuts can be much lower: only $n-1$ possibilities for a chain graph on n nodes. The problem of finding a maximal topological cut is then intuitively easier than finding a maximal cut in a DAG.

We show that MaxTopCut is actually polynomial by exhibiting a Linear Program solving it. This proof is adapted from [118].

Theorem 4.5. *The problem of finding a maximal topological cut in a DAG is polynomial.*

Proof. We consider a DAG G, where each edge (i,j) has a weight $m_{i,j}$. We assume that it has a single source vertex s and a single target vertex t (otherwise, add these nodes with null-weight edges).

We now consider the following linear program $\mathscr{P}$.

$$\max \sum_{(i,j)\in E} m_{i,j} d_{i,j} \tag{4.1}$$

$$\forall (i,j) \in E, \quad d_{i,j} = p_i - p_j \tag{4.2}$$

$$\forall (i,j) \in E, \quad d_{i,j} \geq 0 \tag{4.3}$$

$$p_s = 1 \tag{4.4}$$

$$p_t = 0 \tag{4.5}$$

Intuitively, an integer solution of $\mathscr{P}$ corresponds to a valid topological cut (S,T). The variable p_i represents the potential of vertex i: if it is equal to 1 then $i \in S$ and if it is equal to 0 then $i \in T$. Then, $d_{i,j}$ is equal to 1 if the edge (i,j) belongs to the cut (S,T) and 0 otherwise. Finally, the objective function represents the weight of the cut. However, a general solution of $\mathscr{P}$ consists of rational numbers and not integers, so does not correspond directly to a topological cut. Nevertheless, we show that for this particular program, a naive rounding algorithm exhibits a topological cut, which can then be computed in polynomial time.

Note that $\mathscr{P}$ is similar to the classic linear program computing the minimal $s-t$ cut [98]. The only differences are Equation 4.2 being an equality instead of an inequality, and the direction of the objective function.

We begin by proving that if G admits a topological cut of weight M, there is a solution of the linear program for which the objective function equals M. Let (S,T) be a topological cut of G. For every node i, we define $p_i = 1$ if $i \in S$ and $p_i = 0$ if $i \in T$. Then, for each edge (i,j) belonging to the cut, we have $p_i - p_j = 1$ and for the remaining edges (i,j), we have $p_i - p_j = 0$. Indeed, no edge can be directed from

T to S by definition. Therefore, we have for all $(i,j) \in E$, $d_{i,j} = p_i - p_j \geq 0$ so the proposed valuation satisfies $\mathscr{P}$, and the objective function is equal to the weight of (S,T).

Now, suppose that $\mathscr{P}$ admits a valid rational solution of objective function M^*. We prove that there exists a topological cut (S^*,T^*) of G of weight at least M^*. First, note that for any edge (i,j), we have $d_{i,j} \geq 0$ so $p_i \geq p_j$. Then, every node of G belongs to a directed path from s to t by definition of s and t. Therefore, every p_i belongs to $[0,1]$. Indeed, for a given $i \in V$, let $v_1, v_2, \ldots, v_k$ be the vertices of a directed path from i to t, with $i = v_1$ and $t = v_k$. Then, we deduce that $p_i = p_{v_1} \geq p_{v_2} \geq \cdots \geq p_{v_k} = p_t = 0$. A similar proof with a path from s to i shows that p_i is not larger than 1.

In order to prove the existence of (S^*,T^*), we consider a random topological cut (S,T) defined as follows: draw r uniformly in $]0,1[$, and let (S,T) be the cut (S_r,T_r), with $S_r = \{i \mid p_i > r\}$ and $T_r = \{j \mid p_j \leq r\}$. This partition is valid as for any $i \in S_r$ and $j \in T_r$, we have $p_i > p_j$, so the edge (j,i) cannot belong to E: this would imply $d_{j,i} < 0$ which violates a constraint of $\mathscr{P}$. Now, let us compute the expected cost $M(S,T)$ of (S,T). The probability for a given edge (i,j) to belong to (S,T) is exactly $d_{i,j} = p_i - p_j$, as r is drawn uniformly in $]0,1[$ and all p_i belong to $[0,1]$. Therefore, the expected cost of (S,T) is given by

$$\begin{aligned} E(M(S,T)) &= \sum_{(i,j)\in E} m_{i,j} \Pr\big((i,j) \text{ belongs to } (S,T)\big) \\ &= \sum_{(i,j)\in E} m_{i,j} d_{i,j} = M^*. \end{aligned}$$

Therefore, there exists $r \in]0,1[$ such that $M(S_r,T_r) \geq M^*$, which proves the existence of a topological cut (S^*,T^*) of weight at least M^*. Note that an algorithm could then find such a topological cut by computing $M(S_{p_i},T_{p_i})$ for every $i \in V$.

We now show that it is not necessary, as, if M^* is the optimal objective function, then the weight of any cut (S_r,T_r) is equal to M^*. First, note that no cut (S_r,T_r) can have a weight larger than M^* by definition. So, for all r, we have $M(S_r,T_r) \leq M^*$. As $E(M(S,T)) = M^*$, we conclude that $\Pr(M(S,T) < M^*)) = 0$. It remains to show that no single value of r can lead to a suboptimal cut. Assume by contradiction that there exists $r_0 \in]0,1[$ such that $M(S_{r_0},T_{r_0}) < M^*$. Let $r_1 = \min\{p_i \mid p_i > r_0\}$, which is defined as $p_t = 1 > r_0$, and consider any $r \in [r_0, r_1[$. For every $i \in V$, if $p_i > r_0$ then $p_i \geq r_1 > r$, and if $p_i \leq r_0$ then $p_i \leq r$, so, by definition of S_r and T_r, we have $(S_r,T_r) = (S_{r_0},T_{r_0})$. Therefore, we get

$$\Pr\big(M(S,T) < M^*\big) \geq \Pr\big((S,T) = (S_{r_0},T_{r_0})\big) \geq r_1 - r_0 > 0.$$

This inequality contradicts the fact that $\Pr(M(S,T) < M^*)) = 0$.

To conclude, a maximal topological cut can be computed by first solving the linear program $\mathscr{P}$ in rationals, then selecting any cut (S_r,T_r), for instance by taking $r = 1/2$. □

4.3.2 Explicit algorithm

In the previous section, we have exhibited a linear program solving the MAXTOPCUT problem. We are now interested in an explicit polynomial algorithm, which allows us to have a different approach on the problem, and to solve it without relying on a linear program solver. We first consider a problem related to the dual version of MAXTOPCUT, which we call MINFLOW:

Definition 4.3. *The* MINFLOW *problem consists in computing a flow of minimum value where the amount of flow that passes through each edge is not smaller than its weight.*

We recall that the value of a flow f is defined as $\sum_{j,\ (s,j)\in E} f_{s,j}$. In this problem the edge weights do not represent *capacities* as in a traditional flow, but rather *demands*: the minimum flow must be larger than these demands on all edges[2]. We recall that the MAXFLOW problem consists in finding a flow of maximum value where the amount of flow that passes through each edge is not larger than its weight. Its dual version, the MINCUT problem, consists in computing the st-cut (S, T) of minimum weight, where $s \in S$ and $t \in T$. Note that this cut may not be topological. See [47, Chapter 26] for more details. The MINFLOW problem is described by the following linear program.

$$\begin{aligned} \min \quad & \sum_{j \mid (s,j)\in E} f_{s,j} \\ \forall j \in V \setminus \{s,t\}, \quad & \left(\sum_{i \mid (i,j)\in E} f_{i,j}\right) - \left(\sum_{k \mid (j,k)\in E} f_{j,k}\right) = 0 \\ \forall (i,j) \in E, \quad & f_{i,j} \geq m_{i,j} \end{aligned}$$

We propose in Algorithm 11 an explicit algorithm to resolve the MAXTOPCUT problem. A similar algorithm for a very close problem has been proposed in [45]. We first need an upper bound f_{max} on the value of the optimal flow solving the dual MINFLOW problem on G. We can take for instance f_{max} equal to one plus the sum of the $m_{i,j}$'s. The algorithm builds a flow f with a value at least f_{max} on all edges. Intuitively, the flow f can be seen as an optimal flow f^* solving the MINFLOW problem, on which has been added an arbitrary flow f^+. In order to compute f^* from f, the algorithm explicitly computes f^+, by solving a MAXFLOW instance on a graph G^+. Intuitively, this step consists in maximizing the flow that can be subtracted from f^*. Finally, the maximum topological cut associated to the flow f^* is actually equal to the minimum st-cut of G^+ that can be deduced from the residual network induced by f^+. We recall that the residual network of G^+ induced by f^+ contains the edge (i, j) such that either $(i, j) \in E$ and $f^+_{i,j} < m^+_{i,j}$ or $(j, i) \in E$ and $f^+_{j,i} > 0$, as defined for instance in [45].

The complexity of Algorithm 11 depends on two implementations: how we compute the first flow f and how we solve the MAXFLOW problem. The rest is linear in the number of edges. Computing the starting flow f can be done by looping over all edges, finding a simple path from s to t containing a given edge, and adding a flow going through that path of value f_{max}. Note that this method succeeds because the graph is acyclic, so every edge is part of a simple path (without cycle) from s to t. This can be done in $O(|V||E|)$. Solving the MAXFLOW problem can de done in $O\big(|V||E|\log\big(|V|^2/|E|\big)\big)$ using Goldberg and Tarjan's algorithm [69]. Therefore, Algorithm 11 can be executed in time $O\big(|V||E|\log\big(|V|^2/|E|\big)\big)$.

Algorithm 11: Resolving MAXTOPCUT on a DAG G

1 Construct a flow f for which $\forall (i, j) \in E,\ f_{i,j} \geq f_{max}$, where $f_{max} = 1 + \sum_{(i,j)\in E} m_{i,j}$
2 Define the graph G^+ equal to G except that $m^+_{i,j} = f_{i,j} - m_{i,j}$
3 Compute an optimal solution f^+ to the MAXFLOW problem on G^+
4 $S \leftarrow$ set of vertices reachable from s in the residual network induced by f^+ ; $T \leftarrow V \setminus S$
5 **return** the cut (S, T)

Theorem 4.6. *Algorithm 11 solves the* MAXTOPCUT *problem.*

Proof. First, we show that the cut (S, T) is a topological cut. We have $s \in S$ and $t \in T$ by definition. We now show that no edge exist from T to S in G. By definition of S, no edge exist from S to T in the

[2]This must not be mistaken with the demands of vertices (i.e., the value of the consumed flow) as in the Minimum Cost Flow problem.

residual network, so if there exists an edge (j,i) from T to S in G, it verifies $f^+_{j,i} = 0$. We then show that every edge of G has a positive flow going through it in f^+, which proves that there is no edge from T to S.

Assume by contradiction that there exists an edge (k,ℓ) such that $f^+_{k,\ell}$ is null. Let $S_k \subset V$ be the set of ancestors of k, including k. Then, S_k contains s but not t nor ℓ as G is acyclic. Denoting $T_k = V \setminus S_k$, we get that (S_k, T_k) is a topological cut as no edge goes from T_k to S_k by definition. The weight of the cut (S_k, T_k) is at most the value of the flow f, which is $|f|$. As $f^+_{k,\ell} = 0$, the amount of flow f^+ that goes through this cut is at most $|f| - f_{k,\ell} \leq |f| - f_{max}$. Therefore, the value of f^+ verifies $|f^+| \leq |f| - f_{max}$.

Now, we exhibit a contradiction by computing the amount of flow f^+ passing through the cut (S,T). By definition of (S,T), all the edges from S to T are saturated in the flow f^+: for each edge $(i,j) \in E$ with $i \in S$ and $j \in T$, we have $f^+_{i,j} = m^+_{i,j} = f_{i,j} - m_{i,j}$. The value of the flow f^+ is equal to the amount of flow going from S to T minus the amount going from T to S. Let $E_{S,T}$ (resp. $E_{T,S}$) be the set of edges between S and T (resp. T and S). We have the following (in)equalities:

$$\begin{aligned}
|f^+| &= \left(\sum_{(i,j)\in E_{S,T}} f^+_{i,j}\right) - \left(\sum_{(j,i)\in E_{T,S}} f^+_{j,i}\right) \\
&\geq \left(\sum_{(i,j)\in E_{S,T}} \left(f_{i,j} - m_{i,j}\right)\right) - \left(\sum_{(j,i)\in E_{T,S}} f_{j,i}\right) \\
&\geq |f| - \left(\sum_{(i,j)\in E_{S,T}} m_{i,j}\right) \quad > \quad |f| - f_{max}
\end{aligned}$$

Therefore, we have a contradiction on the value of $|f^+|$, so no edge exists from T to S and (S,T) is a topological cut.

Now, we define the flow f^* on G, defined by $f^*_{i,j} = f_{i,j} - f^+_{i,j} \geq m_{i,j}$. We show that f^* is an optimal solution to the MinFlow problem on G. It is by definition a valid solution as $f^+_{i,j} \leq m^+_{i,j} = f_{i,j} - m_{i,j}$ so $f^*_{i,j} = f_{i,j} - f^+_{i,j} \geq f_{i,j} + m_{i,j} - f_{i,j} = m_{i,j}$. Let g^* be an optimal solution to the MinFlow problem on G and g^+ be the flow defined by $g^+_{i,j} = f_{i,j} - g^*_{i,j}$. By definition, $g^*_{i,j} \geq m_{i,j}$ so $g^+_{i,j} \leq f_{i,j} - m_{i,j} = m^+_{i,j}$. Furthermore, we know that $g^*_{i,j} \leq f_{max}$ because there exists a flow, valid solution of the MinFlow problem, of value $\sum_{(i,j)\in E} m_{i,j} \leq f_{max}$: simply add for each edge (i,j) a flow of value $m_{i,j}$ passing through a path from s to t containing the edge (i,j). Then, we have $g^*_{i,j} \leq f_{max} \leq f_{i,j}$ so $g^+_{i,j} \geq 0$ and g^+ is therefore a valid solution of the MaxFlow problem on G^+, but not necessarily optimal.

So the value of g^+ is not larger than the value of f^+ by optimality of f^+, and therefore, the value of f^* is not larger than the value of g^*. Finally, f^* is an optimal solution to the MinFlow problem on G.

Now, we show that (S,T) is a topological cut of maximum weight in G. Let (S_0,T_0) be any topological cut of G. The total amount of flow of f^* passing through the edges belonging to (S_0,T_0) is equal to the value of f^*. As for all $(i,j) \in E$ we have $f^*_{i,j} \geq m_{i,j}$, the weight of the cut (S_0,T_0) is not larger than the value of f^*. It remains to show that this upper bound is reached for the cut (S,T). By the definition of (S,T), we know that for $(i,j) \in (S,T)$, we have $f^+_{i,j} = m^+_{i,j} = f_{i,j} - m_{i,j}$. Therefore, on all these edges, we have $f^*_{i,j} = f_{i,j} - f^+_{i,j} = m_{i,j}$, so the value of the flow f^* is equal to the weight of (S,T).

Therefore, (S,T) is an optimal topological cut. □

4.4 Lowering the maximal peak memory of a graph

In Section 4.3, we have proposed a method to determine the maximal topological cut of a DAG, which is equal to the maximal peak memory of any (sequential or parallel) traversal. We now move to the problem

of scheduling such a graph within a bounded memory M. If the maximal topological cut is at most M, then any schedule of the graph can be executed without exceeding the memory bound. Otherwise, it is possible that we fail to schedule the graph within the available memory. One solution would be to provide a complete schedule of the graph onto a number p of computing resources, which never exceeds the memory. However, using a static schedule can lead to very poor performance if the task duration are even slightly inaccurate, or if communication times are difficult to predict, which is common on modern computing platforms. Hence, our objective is to let the runtime system dynamically choose the allocation and the precise schedule of the tasks, but to restrict its choices to avoid memory overflow.

In this section, we solve this problem by transforming a graph so that its maximal peak memory becomes at most M. Specifically, we aim at adding some new edges to G to limit the maximal topological cut. Consider for example the toy example of Figure 4.1. Its maximal topological cut has weight 11 and corresponds to the output data of tasks a and b being in memory. If the available memory is only $M = 10$, one may for example add an edge (d, a) of null weight to the graph, which would result in a maximal topological cut of weight 9 (output data of a and d). Note that on this toy example, adding this edge completely serializes the graph: the only possible schedule of the modified graph is sequential. However, this is not the case of realistic, wider graphs. We formally define the problem as follows.

Definition 4.4. *A* **partial serialization** *of a DAG $G = (V, E)$ for a memory bound M is a DAG $G' = (V, E')$ containing all the edges of G (i.e., $E \subset E'$), on which the maximal peak memory is bounded by M.*

In general, there exist many possible partial serializations to solve the problem. In particular, one might add so many edges that the resulting graph can only be processed sequentially. In order to limit the impact on parallel performance of the partial serialization, we use the critical path length as the metric. The critical path is defined as the path from the source to the sink of the DAG whose total processing time is maximum. By minimizing the increase in critical path when adding edges to the graph, we expect that we limit the impact on performance, that is, the increase in makespan when scheduling the modified graph.

We first show that finding a partial serialization of G for memory M is equivalent to finding a sequential schedule executing G using a memory of size at most M. On the one hand, given a partial serialization, any topological order is a valid schedule using a memory of size at most M. On the other hand, given such a sequential schedule, we can build a partial serialization allowing only this schedule (by adding edge (i, j) if i is executed before j). Therefore, as finding a sequential schedule executing G using a memory of size at most M is NP-complete by Theorem 4.4, finding a partial serialization of G for a memory bound of M is also NP-complete.

However, in practical cases, we know that the minimum memory needed to process G is smaller than M. Therefore, the need to find such a minimum memory traversal adds an artificial complexity to our problem, as it is usually easy to compute a sequential schedule not exceeding M on actual workflows. We thus propose the following definition of the problem, which includes a valid sequential traversal to the inputs.

Definition 4.5. *The* MINPARTIALSERIALIZATION *problem consists, given a DAG $G = (V, E)$, a memory bound M, and a sequential schedule $\mathcal{S}$ of G not exceeding the memory bound, in computing a partial serialization of G for the memory bound M that has a minimal critical path length.*

4.4.1 Complexity analysis

We now show that the MINPARTIALSERIALIZATION problem is NP-complete. As explained above, this complexity does not come from the search of a sequential traversal with minimum peak memory. To prove this result, we first propose the following lower bound on the makespan.

Lemma 4.1. *Let $G = (V, E)$ be a DAG. Any schedule $\mathcal{S}$ of peak memory $M_{\mathcal{S}}$ and makespan $T_{\mathcal{S}}$ verifies:*

$$T_{\mathcal{S}} M_{\mathcal{S}} \geq \sum_{i \in V} Outputs(i)\, w_i.$$

As a corollary, if G has a maximal peak memory of $M_{\max}$, then the length T_∞ of its critical path satisfies:

$$T_\infty M_{\max} \geq \sum_{i \in V} Outputs(i)\, w_i.$$

Proof. To prove this result, we consider the function which associates to each time step the memory usage using schedule $\mathcal{S}$ at this time. Its maximum is $M_{\mathcal{S}}$ and it is defined between $t = 0$ and $t = T_{\mathcal{S}}$, so the area under the curve is upper bounded by $T_{\mathcal{S}} M_{\mathcal{S}}$. Now, for each task, its output data must be in memory for at least the execution time of this task, hence $\sum_{i \in V} Outputs(i)\, w_i$ is a lower bound of the area under the curve, which proves the result. □

We now consider the decision version of the MINPARTIALSERIALIZATION problem, which amounts to finding a partial serialization of a graph G for a memory M with critical path smaller than CP, and prove that it is NP-complete.

Theorem 4.7. *The decision version of the* MINPARTIALSERIALIZATION *problem is NP-complete, even for independent paths of length two.*

Proof. First, this problem is in NP as given a partial serialization of a graph G for a memory bound M, one can check in polynomial time that it is valid: simply compute its maximum peak memory (using Algorithm 11) and the length of its critical path.

To prove the problem NP-hard, we perform a reduction from 3-PARTITION, which is known to be NP-complete in the strong sense [66]. We consider the following instance $\mathcal{I}_1$ of the 3-PARTITION problem: let a_i be $3m$ integers and B an integer such that $\sum a_i = mB$. We consider the variant of the problem, also NP-complete, where $\forall i, B/4 < a_i < B/2$. To solve $\mathcal{I}_1$, we need to solve the following question: does there exist a partition of the a_i 's in m subsets $A_1, \ldots, A_m$, each containing exactly 3 elements, such that, for each A_k, $\sum_{i \in A_k} a_i = B$. We build the following instance $\mathcal{I}_2$ of our problem. We define a DAG G with $6m$ vertices denoted by u_i and v_i for $1 \leq i \leq 3m$. G contains $3m$ edges, each pair (u_i, v_i), which have weights equal to a_i. Each vertex u_i has a unit work and v_i has a null work. The memory bound is equal to B and the problem asks whether there exists a partial serialization of G for B with critical path length at most m. A schedule $\mathcal{S}$ executing sequentially the pairs u_i, v_i does not exceed the memory bound B (not even $B/2$), so the instance $(G, B, \mathcal{S})$ is a valid instance of the MINPARTIALSERIALIZATION problem.

Assume first that $\mathcal{I}_1$ is solvable, let $A_1, \ldots, A_m$ be a solution. We build a solution to $\mathcal{I}_2$. Define the graph G' from the graph G with the following additional edges. For $i \in [1, m-1]$, add edges of null weight between every v_j for $a_j \in A_i$ and every u_k for $a_k \in A_{i+1}$. The critical path of G' is then equal to m. Let $S, \bar{S}$ be a topological partition of the graph G', with no edge from $\bar{S}$ to S, and C be the set of edges between S and $\bar{S}$. Assume that C contains an edge (u_j, v_j): $u_j \in S$ and $v_j \in \bar{S}$. Then let k be such that a_j and a_k do not belong to the same set A_i. There is a directed path connecting either v_j to u_k or v_k to u_j, so $(u_k, v_k) \notin C$. Therefore, as $A_1, \ldots, A_m$ is a solution to $\mathcal{I}_1$, the weight of the cut C is equal to B, so G' solves $\mathcal{I}_2$.

Now, assume that $\mathcal{I}_2$ is solvable, let G' be a partial serialization of G for B whose critical path has length T_∞ at most m. Note that the following bound due to Lemma 4.1 is tight on G':

$$T_\infty M_{\max} \geq \sum_{i \in V} Outputs(i)\, w_i.$$

Indeed, the length of the critical path verifies $T_\infty \leq m$, the maximal peak memory $M_{\max}$ verifies $M_{\max} \leq B$, for any $i \in [1, m]$ u_i has a unit weight and v_i a null one and $Outputs(u_i) = a_i$. Therefore, $\sum_{i \in V} Outputs(i)\, w_i = mB$ so $T_\infty = m$ and $M_{\max} = B$.

Let U_1 be the set of nodes u_i without predecessors in G'. There cannot be more than three nodes in U_1 because the cut $(U_1, \bar{U}_1)$ would have a weight larger than B. Assume by contradiction that its weight is less than B. Consider the graph G'_1 equal to G' except that the nodes in U_1 have a null work. The critical path of G'_1 is equal to $m-1$ and in G'_1, we have $\sum_{i \in V} Outputs(i)\, w_i > mB - B = (m-1)B$, so the bound of Lemma 4.1 is violated. Therefore, the weight of the cut $(U_1, \bar{U}_1)$ is equal to B, so U_1 is composed of three vertices that we will denote by $u_{i_1}, u_{j_1}, u_{k_1}$, and we have $a_{i_1} + a_{j_1} + a_{k_1} = B$.

Suppose by contradiction that there exists a node u_i not in U_1 such that there is no path from v_{i_1}, v_{j_1} or v_{k_1} to u_i. Then, $(U_1 \cup \{u_i\}, V \setminus (U_1 \cup \{u_i\}))$ is a topological cut of G'_1 of weight strictly larger than B, which is impossible by definition of $\mathscr{I}_2$. Therefore, in G'_1, the nodes that have no ancestors are $U_1 = \{u_{i_1}, u_{j_1}, u_{k_1}\}$, and the nodes whose ancestors belong in U_1 are $\{v_{i_1}, v_{j_1}, v_{k_1}\}$.

We can then apply recursively the same method to determine the second set U_2 of three vertices $u_{i_2}, u_{j_2}, u_{k_2}$ without ancestors of positive work in G'_1. We now define G'_2 as equal to G'_1 except that nodes of U_2 have a null work, and continue the induction.

At the end of the process, we have exhibited m disjoint sets of three elements a_i that each sum to B, so $\mathscr{I}_1$ is solvable. □

4.4.2 Finding an optimal partial serialization through ILP

We present in this section an Integer Linear Program solving the MinPartialSerialization problem. This formulation combines the linear program determining the maximum topological cut and the one computing the critical path of a given graph.

We consider an instance of the MinPartialSerialization problem, given by a DAG $G = (V, E)$ with weights on the edges, and a memory limit M. The sequential schedule $\mathscr{S}$ respecting the memory limit is not required. First, for any $(i, j) \notin E$, we set $m_{i,j} = 0$. We furthermore assume that there is a single source vertex s and a single target vertex t, as explained above.

We first consider the $e_{i,j}$ variables, which are equal to 1 if edge (i, j) exists in the associated partial serialization, and to 0 otherwise.

$$\forall (i, j) \in V^2, \quad e_{i,j} \in \{0, 1\} \tag{4.6}$$

$$\forall (i, j) \in E, \quad e_{i,j} = 1 \tag{4.7}$$

We need to ensure that no cycle has been created by the addition of edges. For this, we compute the transitive closure of the graph: we enforce that the graph contains edge (i, j) if there is a path from node i to node j. Then, we know that the graph is acyclic if and only if it does not contain any self-loop. This corresponds to the following constraints:

$$\forall (i, j, k) \in V^3, \quad e_{i,k} \geq e_{i,j} + e_{j,k} - 1 \tag{4.8}$$

$$\forall i \in V, \quad e_{i,i} = 0 \tag{4.9}$$

Then, we use the flow variables $f_{i,j}$, in a way similar to the formulation of the MinFlow problem. If $e_{i,j} = 1$, then $f_{i,j} \geq m_{i,j}$, and $f_{i,j}$ is null otherwise. Now, the flow going out of s is equal to the maximal cut of the partial serialization, see the proof of Theorem 4.6, so we ensure that it is not larger than M. Now, note that each $f_{i,j}$ can be upper bounded by M without changing the solution space. Therefore, Equation 4.11 ensures that $f_{i,j}$ is null if $e_{i,j}$ is null, without adding constraints on the others

$f_{i,j}$. This leads to the following inequalities:

$$\forall (i,j) \in V^2, \quad f_{i,j} \geq e_{i,j} m_{i,j} \tag{4.10}$$

$$\forall (i,j) \in V^2, \quad f_{i,j} \leq e_{i,j} M \tag{4.11}$$

$$\forall j \in V \setminus \{s,t\}, \quad \sum_{i \in V} f_{i,j} - \sum_{k \in V} f_{j,k} = 0 \tag{4.12}$$

$$\sum_{j \in V} f_{s,j} \leq M \tag{4.13}$$

This set of constraints defines the set of partial serializations of G with a maximal cut at most M. It remains to compute the length of the critical path of the modified graph, in order to formalize the objective. We use the p_i to represent the top-level of each task, that is, their earliest completion time in a parallel schedule with infinitely many processors. The completion time of task s is w_s, and the completion time of another task is equal to its processing time plus the maximal completion time of its predecessors:

$$p_s \geq w_s$$
$$\forall (i,j) \in V^2, \quad p_j \geq w_j + p_i e_{i,j}$$

The previous equation is not linear, so we transform it by using W, the sum of the processing times of all the tasks and the following constraints.

$$\forall i \in V, \quad p_i \geq w_i \tag{4.14}$$

$$\forall (i,j) \in V^2, \quad p_j \geq w_j + p_i - W(1 - e_{i,j}) \tag{4.15}$$

If $e_{i,j}$ is null, then Equation 4.15 is less restrictive than Equation 4.14 as $p_i < W$, which is expected as there is no edge (i,j) in the graph. Otherwise, we have $e_{i,j} = 1$ and the constraints on p_j are the same as above.

Finally, we define the objective as minimizing the top-level of t, which is the critical path of the graph.

$$\text{Minimize} \quad p_t \tag{4.16}$$

We denote $\mathscr{P}$ the resulting ILP. We now prove that there exists a solution to $\mathscr{P}$ of objective at most L if and only if there exists a partial serialization PS of G with memory bound M of critical path length at most L.

Consider a solution of $\mathscr{P}$ of objective cost at most L. Let PS be the directed graph composed of the edges (i,j) for every $i,j \in V^2$ such that $e_{i,j} = 1$. The weight of such edges is $m_{i,j}$. First, PS is acyclic. This can be shown by induction on the size of a potential cycle. No self-loop can exist as all $e_{i,i}$ are null. If a cycle contains more than one edge, Equation 4.8 ensures the existence of a strictly smaller cycle, while Equation 4.9 forbids self-loops. Then, the equations concerning $f_{i,j}$ model the MinFlow problem already studied, and ensure that the minimum flow is smaller than M. The only difference being that each $f_{i,j}$ is bounded by M, which is already the case in any solution. Finally, consider a critical path $(s, i_1, i_2, \ldots, i_k, t)$ of PS. The equations concerning the variables p_i ensure that $p_t \geq w_s + w_{i_1} + \cdots + w_{i_k} + w_t$. Therefore, L is not smaller than the critical path length. Therefore, PS is a partial serialization for M of critical path length at most L.

Now, consider a partial serialization PS of G for M, of critical path length at most L. We set $e_{i,j} = 1$ if and only if there exists a path from i to j in PS. This respects the acyclicity constraints as PS is a DAG

by definition. The maximum peak memory of PS is at most M, therefore the maximum cut of the graph induced by the variables $e_{i,j}$ is at most M, so there exists a valuation of the variables $f_{i,j}$ satisfying the flow constraints. Finally, we set the variables p_i equal to the top-level of task i in PS:

$$\forall i \in V, \quad p_i = w_i + \max_{j \in V} \{e_{j,i} p_j\}.$$

This valuation satisfies the last constraints and the objective function is then equal to L.

4.4.3 Heuristic strategies to compute a partial serialization

We now propose several heuristics to solve the MINPARTIALSERIALIZATION problem. These heuristics are based on the same framework, detailed in Algorithm 12. The idea of the algorithm, inspired by [128], is to iteratively build a partial serialization G' from G. At each iteration, the topological cut of maximum weight is computed via Algorithm 11. If its weight is at most M, then the algorithm terminates, as the obtained partial serialization is valid. Otherwise, another edge has to be added in order to reduce the maximum peak memory. We rely on a subroutine in order to choose which edge to add. In the following, we propose four possible subroutines. If the subroutine succeeds to find an edge that does not create a cycle in the graph, we add the chosen edge to the current graph. Otherwise, the heuristic fails. Such a failure may happen if the previous choices of edges have led to a graph which is impossible to schedule without exceeding the memory.

Algorithm 12: Heuristic for MINPARTIALSERIALIZATION

Input: DAG G, memory bound M, subroutine $\mathscr{A}$
Output: Partial serialization of G for memory M

```
while G has a topological cut of weight larger than M do
    Compute a topological cut C = (S, T) of maximum weight using Algorithm 11
    if the call A(G, M, C) returns (u_T, u_S) with no path from node u_S to node u_T then
        Add edge (u_T, u_S) of weight 0 to G
    else
        return Failure
return the modified graph G
```

We propose four possibilities for the subroutine $\mathscr{A}(G, M, C)$, which selects an edge to be added to G. They all follow the same structure: two vertices u_S and u_T are selected from the maximum cut $C = (S, T)$, where $u_S \in S$ and $u_T \in T$ and no path exists from u_S to u_T. The returned edge is then (u_T, u_S). For instance, in the toy example of Figure 4.1, only two such edges can be added: (c, b) and (d, a). Note that adding such an edge prevents C from remaining a valid topological cut, thus it is likely that the weight of the new maximum topological cut will be reduced.

We first recall some classical attributes of a graph, already defined in the Preliminaries section:

- The *length* of a path is the sum of the work of all the nodes is the path, including its extremities;
- The *bottom-level* of an edge (i, j) or a node i is the length of the longest path from i to t (the sink of the graph);
- The *top-level* of an edge (i, j) or a node j is the length of the longest path from s (the source of the graph) to j.

We now present the four subroutines. The MINLEVELS heuristic, as well as the two following ones, generates the set P of vertex couples $(j, i) \in T \times S$ such that no path from i to j exist. Note

that P corresponds to the set of candidate edges that might be added to G. Then, it returns the couple $(u_T, u_S) \in P$ that optimizes a given metric. MINLEVELS tries to minimize the critical path of the graph obtained when adding the new edge, by preventing the creation of a long path from s to t. Thus, it returns the couple $(j,i) \in P$ that minimizes $top\_level(j) + bottom\_level(i)$.

The MAXSIZE heuristic aims at minimizing the weight of the next topological cut. Thus, it selects a couple (j,i) such that outgoing edges of i and incoming edges of j contribute a lot to the weight of the current cut. Formally, it returns the coupe $(j,i) \in P$ that maximizes $\sum_{k \in T} m_{i,k} + \sum_{k' \in S} m_{k',j}$ (considering that $m_{i,j} = 0$ if there is no edge from i to j).

The MAXMINSIZE heuristic is a variant of the previous heuristic and pursues the same objective. However, it selects a couple of vertices which both contribute a lot to the weight of the cut, by returning the couple $(j,i) \in P$ that maximizes $\min\left(\sum_{k \in T} m_{i,k}\ ,\ \sum_{k' \in S} m_{k',j}\right)$.

Finally, the last heuristic is the only one that is guaranteed to never fail. To achieve this, it relies on a sequential schedule $\mathscr{S}$ of the graph that does not exceed the memory M. $\mathscr{S}$ is defined by a function σ, where $\sigma(i)$ equals the starting time of task i in $\mathscr{S}$. Such a sequential schedule needs to be precomputed, and we propose a possible algorithm below.

Given such a sequential schedule $\mathscr{S}$, this heuristic, named RESPECTORDER, always adds an edge (j,i) which is compatible with $\mathscr{S}$ (i.e., such that $\sigma(j) \leq \sigma(i)$), and which is likely to have the smallest impact on the set of valid schedules for the new graph, by maximizing the distance $\sigma(i) - \sigma(j)$ from j to i in $\mathscr{S}$. Let u_T be the node of T which is the first to be executed in $\mathscr{S}$, and u_S be the node of S which is the last to be executed in $\mathscr{S}$. First, note that u_S must be executed after u_T in $\mathscr{S}$, because otherwise, the peak memory of $\mathscr{S}$ will be at least the weight of C which is a contradiction. The returned couple is then (u_T, u_S). Note that no path from u_S to u_T can exist in the graph if all the new edges have been added by this method. Indeed, all the added edges respect the order $\mathscr{S}$ by definition. Then, no failure is possible, but the quality of the solution highly depends on the input schedule $\mathscr{S}$.

4.4.4 Computing a sequential schedule for MINLEVELS

In this section we discuss the generation of the schedule $\mathscr{S}$, which is used as an input for heuristic RESPECTORDER. By definition, this sequential schedule executes the DAG G using a memory at most M. As proven in Theorem 4.4, deciding if such a schedule exists is NP-complete. However, most graphs describing actual workflows exhibit a high level of parallelism, and the difficulty is not in finding a sequential schedule fitting in memory. As a consequence, we assume that a Depth First Search (DFS) schedule, which always completes a parallel branch before starting a new one, never exceeds the memory bound.

The problem with a DFS schedule is that applying RESPECTORDER using such a schedule is likely to produce a graph with a large critical path. For this objective, a Breadth First Search (BFS) schedule is more appropriate, but it is not likely to respect the memory bound.

As proposed in [128], a way to solve this problem is to "mix" DFS and BFS schedules, and tune the proportion of each one to get a schedule respecting the memory bound but still offering good opportunities for parallelism. Formally, we define the λ-BFSDFS schedule, which depends on the parameter $\lambda \in [0,1]$ and two schedules, a DFS and a BFS. A 0-BFSDFS schedule is equal to the BFS and a 1-BFSDFS schedule is equal to the DFS. For a given task i, we note $DFS(i)$ and $BFS(i)$ the rank of task i according to each schedule (i.e., the number of tasks executed before task i). Then, the λ-BFSDFS schedules the tasks of G in non-decreasing order of

$$\lambda DFS(i) + (1-\lambda) BFS(i).$$

	DAGGEN dense	DAGGEN sparse	LIGO	MONTAGE	GENOME
Nb. of test cases	572	572	220	220	220
MINLEVELS	1	12	20	1	0
RESPECTORDER	0	0	0	0	0
MAXMINSIZE	2	5	3	0	0
MAXSIZE	6	12	13	0	17
ILP	26	102			

Table 4.1: Number of failures for each dataset

The λ-BFSDFS schedule respects the precedence constraints: indeed, if task i has a successor j, then i is scheduled before j in both BFS and DFS. Then, as λ and $1-\lambda$ are non-negative, λ-BFSDFS schedules i before j.

The idea consists in starting from the 0-BFSDFS schedule, and then to increase the λ parameter until the memory of the resulting schedule is not larger than M. As we assumed that DFS (1-BFSDFS) does not exceed M, this process is guaranteed to success. In practice, we chose in the experiments to increment λ by step of 0.05 until we find an appropriate schedule.

4.5 Simulation results

We now compare the performance of the proposed heuristics through simulations on synthetic DAGs. All heuristics are implemented in C++ using the igraph library.

We generated the first dataset, named DAGGEN, using the DAGGEN software [133]. Five parameters influence the generation of these DAGs. The number of nodes belongs to $\{25, 50, 100\}$. The width, which controls how many tasks may run in parallel, belongs to $\{0.2, 0.5, 0.8\}$. The regularity, which controls the distribution of the tasks between the levels, belongs to $\{0.2, 0.8\}$. The density, which controls how many edges connect two consecutive levels, belongs to $\{0.2, 0.8\}$. The jump, which controls how many levels an edge may span, belongs to $\{1, 2, 4\}$. Combining all these parameters, we obtain a dataset of 108 DAGs. This dataset has already been used to model workflows in the scheduling literature [53, 85]. We split it in two parts in the representations: the sparse DAGGEN dataset contains the DAGs with a density of 0.2 and the dense DAGGEN dataset contains the DAGs with a density of 0.8. Indeed, this parameter leads to significant differences in the results, hence the distinction.

The three other datasets represent actual applications and have been generated with the Pegasus Workflow Generator [49]. We consider three different datasets, named LIGO, MONTAGE, and GENOME, each containing 20 graphs of 100 nodes

The sizes given for each file are incoherent, as their value changes if the file is read by several nodes. Hence, we assumed that the size of a file is the one given by the node that produced it. Several nodes can produce data which share the same name. In this case, we assumed that these data are different, which is coherent with the precedence relations. We assumed that the memory needed during the execution of a node is negligible compared to the size of the input and output data, which must be kept in memory during this process. We then apply the transformation presented in Section 4.2 to treat data that are shared among several tasks, and duplicate the nodes to cope with the memory model of SIMPLEDATAFLOWMODEL.

The heuristics have been simulated for eleven memory bounds per DAG, evenly spread between two bounds. The smallest bound corresponds to the memory required for a DFS schedule, while the largest bound corresponds to the maximal peak memory of the DAG. In the results, a normalized memory of 0 corresponds to the lowest bound, while 1 corresponds to the largest bound.

One may argue that the range of memory considered can be small for some graphs, and will then be of little interest. We therefore computed the ratio of the largest memory considered divided by the lowest for each graph, and we present the statistic summary in Table 4.2. We can see that this ratio is very high for the LIGO and GENOME dataset: finding a partial serialization achieving the lowest memory bound means that the maximal memory consumption is divided by more than 20 for most of these graphs. This ratio has a median of 6 for the MONTAGE, which is also a high potential improvement. It is lower for the sparse DAGGEN dataset, with a median of 2, and especially for the sparse DAGGEN dataset, with a median of 1.3. Note that 4 DAGs of the DAGGEN dataset have been discarded because the minimum memory equals the maximum memory.

	DAGGEN		LIGO	MONTAGE	GENOME
	dense	sparse			
First quartile	1.2	1.7	21.2	5.5	20.1
Median	1.3	2	21.7	6.2	21.5
Third quartile	1.4	2.5	22.1	6.8	22

Table 4.2: Statistic summary of the ratio maxmem/minmem for each dataset.

In order to assess the performance of the heuristics, we first examine the critical path length of the obtained partial serialization. We first normalize each critical path by the critical path of the original graph. Therefore, for the largest memory bounds, the original graph being itself a valid partial serialization, all the normalized critical paths equal 1. When a method fails to find a solution, we say that the critical path achieved is infinite. As we focus on the statistical summary of the results, this allows to fairly compare two heuristics with different success rate, as only the outlier points are not displayed. Failure rates are reported in Table 4.1.

We plot the results obtained for the sparse and dense DAGGEN dataset in Figures 4.5 and 4.6 respectively. For each heuristic and memory bound, we display the 108 results as a Tukey boxplot. The box presents the median, the first and third quartiles. The whiskers extend to up to 1.5 times the box height, and points outside are plotted individually. The first trend that can be observed, is that, as expected, the lower the memory bound, the larger the critical path. The difference between the minimal and the maximal memory bound is smaller for dense graphs. Therefore, it is logical that the heuristics lead to a larger increase of the critical path in sparse graphs. Comparing the heuristics, we can see that MINLEVELS clearly outperforms the other ones for any value of the memory bound. Then, RESPECTORDER obtains better performance than MAXMINSIZE and MAXSIZE, except when the memory bound is the lowest, where these three heuristics are comparable. Note that no significant difference appears when restricting the dataset to specific values of the generation parameters. The results are widely spread as the graphs differ in several parameters. We remark therefore that MINLEVELS is highly robust considering the variety of the graphs. On this dataset, we have also computed the optimal solution by using the Integer Linear Program presented in Section 4.4. We implemented the ILP using CPLEX with a time limit of one hour of computation on a standard laptop computer (8 cores Intel i7). When it was unable to provide a solution within the time limit, we assume a failure. This happens on sparse graphs, especially for low memory bounds, which is why it is omited on Figure 4.5.

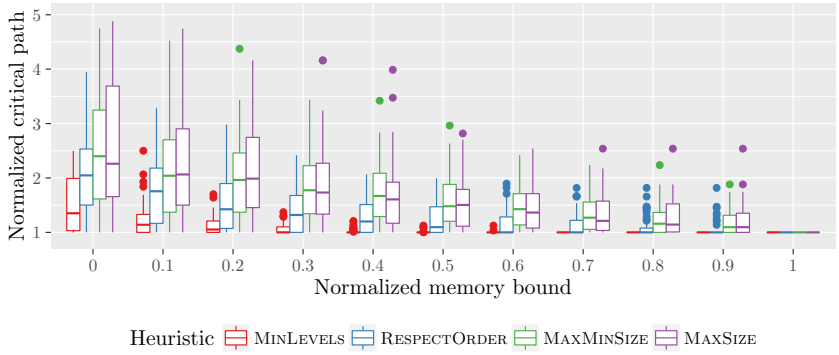


Figure 4.5: Critical path length obtained by each method for the sparse DAGGEN dataset.

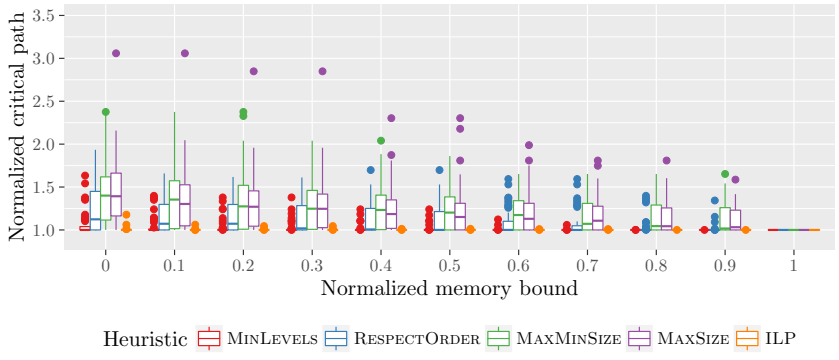


Figure 4.6: Critical path length obtained by each method for the dense DAGGEN dataset.

The second criterion we use to compare the heuristics consists in evaluating the makespan achieved by a simple scheduling heuristic on the partial serialization returned by each heuristic on a simulated platform. The chosen scheduling heuristic is the traditional list-scheduling algorithm, in which whenever a task terminates, the available task with the highest bottom level is executed. This corresponds to the well-known HEFT scheduler [139] when adapted to dynamic schedulers, as for example done in the *dmda* scheduler of StarPU [17].

We simulated a platform of 2 processors for the dataset DAGGEN, and the results are presented in Figures 4.7 and 4.8. We can notice that the differences between the heuristics are smaller than previously, while the hierarchy is not modified. On Figure 4.9, we plotted for each DAG of the DAGGEN dataset and for each memory bound, the makespan obtained by each heuristic in function of the critical path obtained.

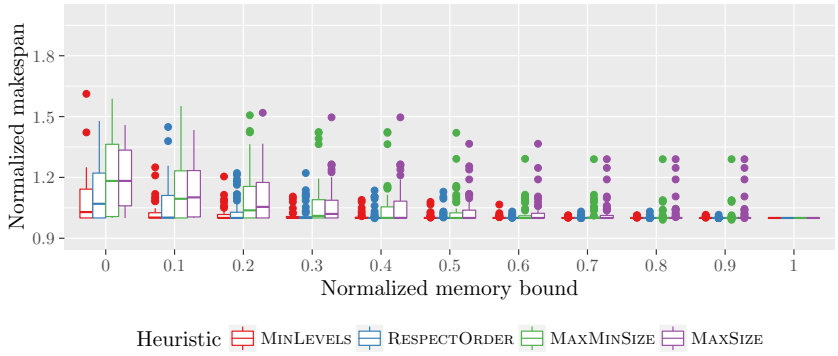


Figure 4.7: Makespan obtained by each method for the sparse DAGGEN dataset.

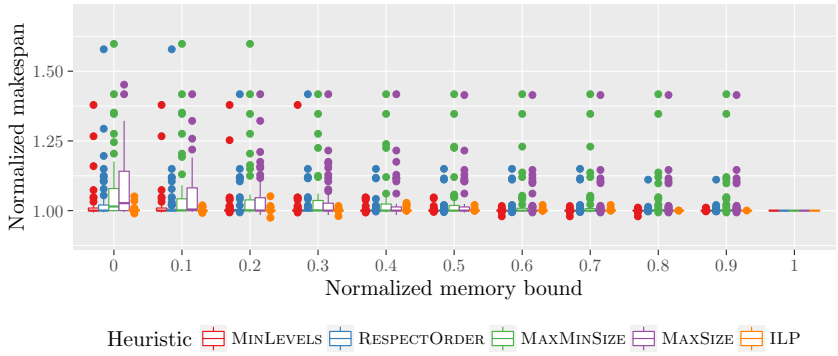


Figure 4.8: Makespan obtained by each method for the dense DAGGEN dataset.

We plot the results obtained for the LIGO dataset on Figure 4.10, showing the critical path lengths achieved by each heuristic for each memory bound. The similar structure of all graphs in this dataset explains that the results lie in a smaller interval. The hierarchy of the heuristics is the same as in the DAGGEN dataset: MINLEVELS presents the best performance, RESPECTORDER leads to slightly longer critical paths, and MAXSIZE and MAXMINSIZE achieve similar results, several times higher than the first two heuristics. Note that for the lowest memory bound, MINLEVELS never succeeds in

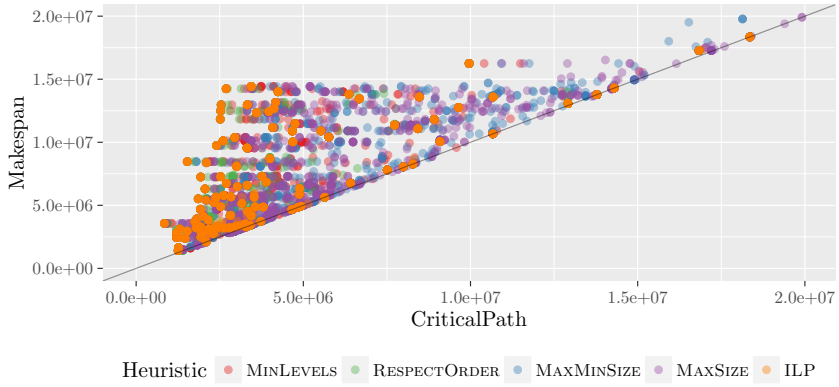


Figure 4.9: Makespan in function of the critical path length obtained by each method for the DAGGEN dataset.

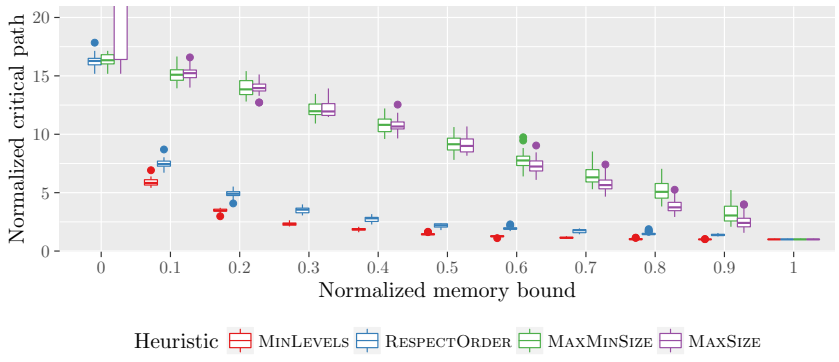


Figure 4.10: Critical path length obtained by each method for the LIGO dataset.

this dataset (hence, it does not appear in the plot), MAXSIZE also presents a high failure rate, whereas RESPECTORDER and MAXMINSIZE have comparable results.

Figure 4.11 presents the simulation on 5 processors. Except the slightly more scattered results, the ranking of the heuristics is very similar than the ones obtained with the critical path. Therefore, even if the final objective is to obtain a graph that we can schedule within a small makespan, our objective of minimizing the critical path is completely relevant.

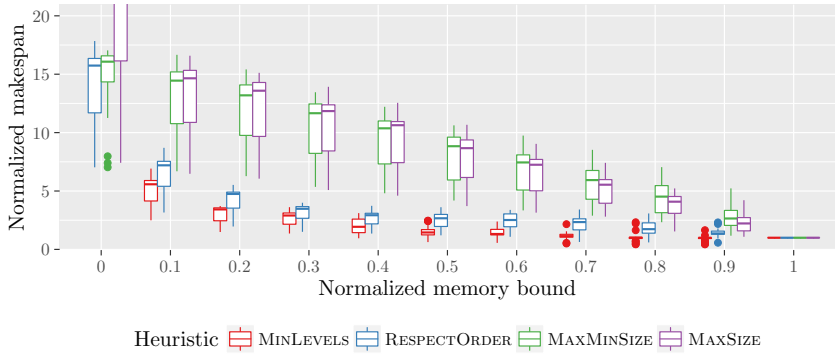


Figure 4.11: Makespan obtained by each method for the LIGO dataset.

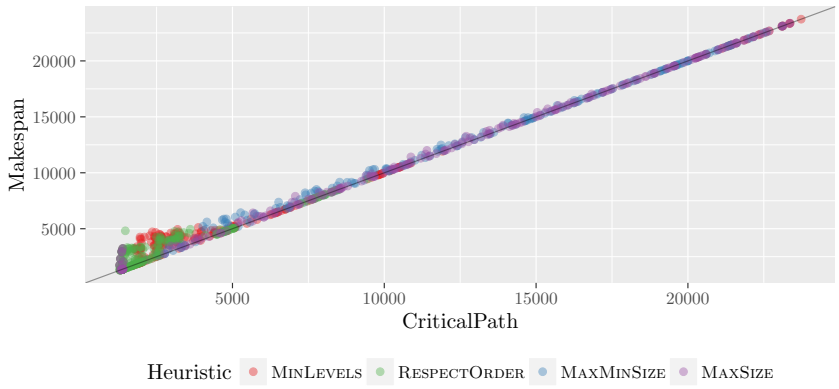


Figure 4.12: Makespan in function of the critical path length obtained by each method for the LIGO dataset.

On Figure 4.12, we plotted for each DAG of the LIGO dataset and for each memory bound, the makespan obtained by each heuristic in function of the critical path obtained. We can see that when the critical path achieved is large, the makespan obtained is very close to the critical path length. On the opposite, for smaller values of the critical path length, we can obtain a makespan several times higher, because the partial serialization kept more parallelism in the graph.

In Figures 4.13 to 4.15, we present the same results for the GENOME dataset. We observe a trend similar to the results on the LIGO dataset, except that MINLEVELS never fails, even for the lowest memory bound.

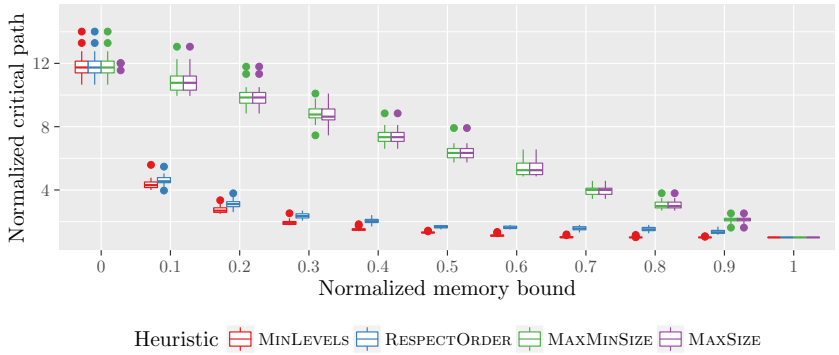


Figure 4.13: Critical path length obtained by each method for the GENOME dataset.

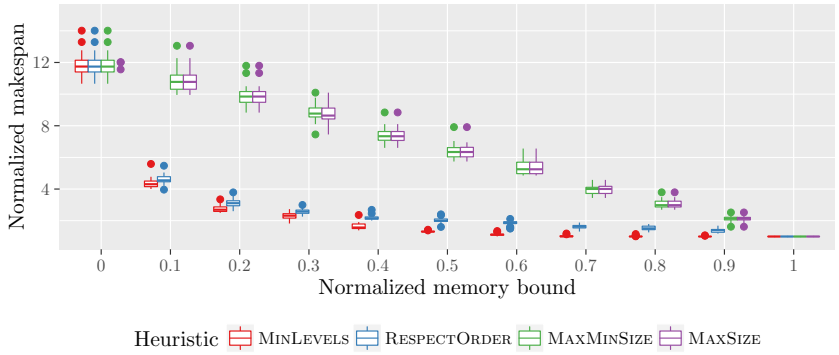


Figure 4.14: Makespan obtained by each method for the GENOME dataset.

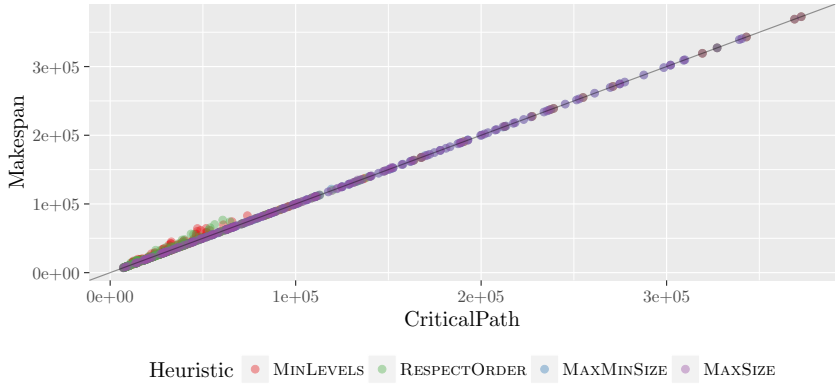


Figure 4.15: Makespan in function of the critical path length obtained by each method for the GENOME dataset.

In Figures 4.16 to 4.18, we present the same results for the MONTAGE dataset. We observe a trend similar to the results on the LIGO dataset, except that MINLEVELS and RESPECTORDER always present better results that the other heuristics, even for the lowest memory bound.

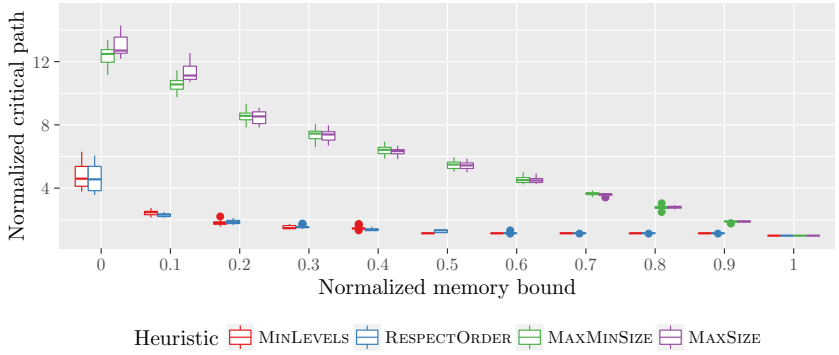


Figure 4.16: Critical path length obtained by each method for the MONTAGE dataset.

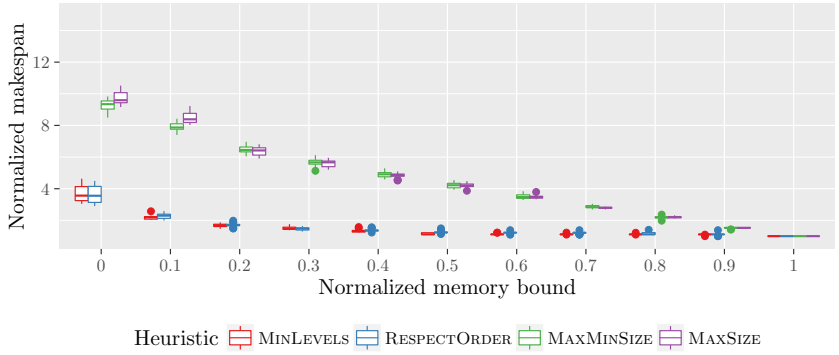


Figure 4.17: Makespan obtained by each method for the MONTAGE dataset.

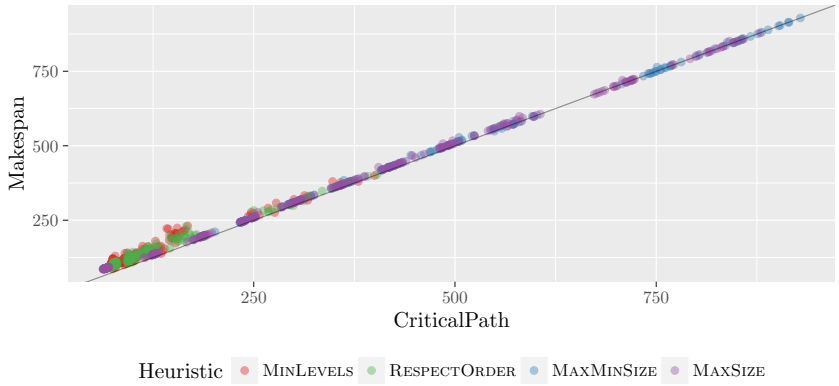


Figure 4.18: Makespan in function of the critical path length obtained by each method for the MONTAGE dataset.

We have shown in these experiments that we can partially serialize realistic graphs so that any schedule fits a given memory bound, for a reasonable cost in terms of the critical path and makespan augmentation. One may argue that the maximal peak memory considered does not reflect the actual memory consumption of a traditional algorithm. In order to address this problem, we measured the peak memory achieved by the scheduling heuristic on every graph of the datasets (without fictitious edges). Then, we normalized it in the same way as in the plots above: a value of 1 means that the maximal peak memory is actually achieved, and a value of 0 means that the peak memory reached is the same as the Depth First Search considered. Note that we can obtain negative values, which happened only for some graphs of the DAGGEN datasets, if the DFS requires a larger memory. The statistical summary is presented in Table 4.3. We note that the scheduling heuristic uses the maximal peak memory for most of the graphs of the LIGO and GENOME datasets, and a normalized memory larger than 0.88 for most of the graphs of the MONTAGE dataset. Therefore, on these realistic graphs, the gain in memory is high. On the DAGGEN dataset, the partial serialization is not as beneficial, as we obtain a median of 0.53. However, note that the MINLEVELS heuristic does not lead to a high augmentation of makespan: less than a 5% increase for the lowest memory bound for 75% of the graphs. Therefore, it is logical that the memory consumption can not be reduced by a larger factor.

	DAGGEN		LIGO	MONTAGE	GENOME
	dense	sparse			
First quartile	−0.03	0.39	0.99	0.88	1
Median	0.31	0.6	1	0.9	1
Third quartile	0.71	0.75	1	0.93	1

Table 4.3: Normalized memory used by EFT

4.6 Conclusion

In this chapter, we have focused on lowering the memory footprint of task graphs representing computational workflows. As we target dynamic schedules (such as in runtime systems), we have focused on

the transformation of the graphs prior to the scheduling phase. Adding fictitious edges that represent "memory dependences" prevents the scheduler to run out of memory. After formally modeling the problem, we have shown how to compute the maximal peak memory of a graph, we have proven the problem of adding edges to cope with limited memory while minimizing the critical path to be NP-complete, and proposed both an ILP formulation of the problem and several heuristics. Simulations show that our best heuristics, RESPECTORDER and MINLEVELS, either never fail, or are able to limit the memory footprint with limited impact on the parallel makespan for most task graphs.

The most interesting theoretical question that remains to be addressed in this project is to determine whether the problem of adding edges that prevent the scheduler to exceed the memory limit while minimizing the resultant critical path length can be approximated. We have shown that computing the optimal solution is NP-hard, and we have proposed an ILP and polynomial-time heuristics to address this problem, but we do not know whether computing a constant-factor approximation in polynomial time is NP-hard.

Although the proposed algorithms are polynomial, their complexity prevents them from begin directly implemented in a runtime system dealing with large graphs. The most obvious way to improve the complexity would be to avoid recomputations at each iteration: determining the maximal cut and the priority of edges for each heuristic may be sped up using previous computations. Similarly, the strategies developed include many iterations, leading to many redundant fictitious edges. For instance, if two chains of n tasks must not be run in parallel, we may end with $O(n^2)$ edges (and iterations), where one edge would be sufficient. Even if redundant edges can be easily deleted before the execution of the graph, it would be interesting to develop a low-complexity algorithm, which directly targets *effective* edges. We can also wonder whether the critical path is the best metric to optimize. For instance, with p processors available, a graph G_1 composed of p chains of n tasks of unit processing times can be scheduled in a time n, equal to its critical path length. A graph G_2, composed of a chain of $n-1$ unit tasks followed by a fork of $(p-1)n+1$ unit tasks has the same task set as G_1, the same critical path, but its optimal makespan is almost twice longer (for large values of p and n). Following this idea, it would be interesting to design a metric aiming at minimizing the idle times on a specific platform.

Finally, another direction consists in dealing with more *clever* schedulers. The objective of this chapter is to prevent any schedule to exceed a memory bound. However, modern schedulers may detect that starting a task will directly lead to a memory shortage, and will therefore postpone it until enough memory becomes available. In this context, far less fictitious edges would need to be added. Nevertheless, this setting also increases the difficulty of computing whether a given graph may lead to a deadlock or memory shortage.

Chapter 5

Minimizing I/Os when processing a tree

« Aussi, elle est mal entretenue cette forêt: il y a des arbres partout ! »

Obélix, *Astérix légionnaire*

In Chapter 4, we focused on graphs for which the computation may or may not fit into the available memory, depending on the scheduling choices. In this chapter, we consider that the data files manipulated by the tasks are so large that the main memory is insufficient for any schedule. In this case, we have to resort to using disk as a secondary storage, which is sometimes known as *out-of-core* execution. The cost of the I/O operations to transfer data from and to the disk is known to be several orders of magnitude larger than the cost of accessing the main memory. Thus, in the case of out-of-core execution, it is a natural objective to minimize the total volume of I/O.

One of the motivation for this work comes from numerical linear algebra, and especially the factorization of sparse matrices using direct multifrontal methods [50], as seen in previous chapters. During the factorization, the computations are organized as a tree workflow called an elimination tree, and, for large matrices, the huge size of the data involved makes it necessary to reduce the memory requirement of the factorization. In this chapter, we restrict the study to such rooted in-trees. Each task uses all the data produced by its children to output new data for its parent. In particular, a task must have enough available memory to fit the input from all its children.

It is known that the problem of minimizing the peak memory, M_{peak}, of a tree traversal, that is, the minimum amount of memory needed to process a tree, is polynomial [88, 105]. However, it may well happen that the available amount of memory M is smaller than the peak memory M_{peak}. In this case, we have to decide which data, or part of data, have to be written to disk. The case when the data cannot be partially written to disk has been studied and proved NP-complete in [88]. However, it is usually possible to split data that reside in memory, and write only part of it to the disk if needed. This is for instance what is done using *paging*: all data are divided in same-size *pages*, which can be moved from main memory to secondary storage when needed. Since all modern computer systems implement paging, it is natural to consider it when minimizing the I/O volume.

Note that as in [88], the present chapter does not directly focus on parallel algorithms. However, parallel processing is the ultimate motivation for this work: complex scientific applications using large data such as multifrontal sparse matrix factorization always make use of parallel platforms. Most involved scheduling schemes combine data parallelism (a task uses multiple processors) and tree parallelism (several tasks are processed in parallel), such as in [62] and in Chapters 1 and 2. However, one cannot hope to achieve good results for the minimization of I/O volume in a parallel settings until the sequential prob-

lem is well understood, which is not yet the case. We present therefore a step towards understanding the sequential version of this problem.

Main contributions. In this chapter, we provide both theoretical and algorithmic results to the I/O volume minimization problem. We first formalize in a common framework the results scattered in the literature. Then, we prove the dominance of postorder traversals when trees are homogeneous (all output data have the same size), knowing that an algorithm to compute the best postorder traversal has been proposed by E. Agullo [3]. Concerning heterogeneous trees, we prove that neither the best postorder traversal nor the memory-peak minimization algorithms are approximation algorithms for minimizing the I/O volume. We provide an Integer Linear Programming (ILP) formulation that allows us to compute an optimal solution for small scale problems, and we design a new heuristic that takes advantage of peak-memory-optimizing algorithms. Finally, we conducted an extensive experimental comparison of all available strategies (including the ILP for small test cases) through simulations on both synthetic and realistic trees built from actual sparse matrices. These simulations show the very good performance of the proposed solution.

The rest of this paper is organized as follows. We give an overview of the related work in Section 5.1. Then in Section 5.2 we formalize our model and present elementary results. Existing solutions are studied in Section 5.3. An optimal ILP is presented in Section 5.4. We introduce a new heuristic in Section 5.5 and evaluate its performance through simulations in Section 5.6. We finally conclude and present future directions in Section 5.7.

5.1 Related work

As stated above, rooted trees are commonly used to represent task dependences for scientific applications. This is, for example, the case for some computational physics codes modeling the electronic properties of semiconductors and metals [96, 100, 109], and for the accurate modeling of the electronic structure of atoms and molecules in quantum chemistry [18, 86]. In the domain of sparse linear algebra, Liu [104] gives a detailed description of the construction of the elimination tree already introduced in Chapter 1, its use for Cholesky and LU factorizations, and its role in multifrontal direct methods. Note that peak memory minimization is still a crucial question for direct solvers, as highlighted by Agullo et al. [4], who study the effect of processor mapping on memory consumption for multifrontal methods.

Memory and storage have always been a limited parameter for large computations, as outlined by the pioneering work of Sethi and Ullman [131] on register allocation for task trees. In the realm of sparse direct solvers, the problem of scheduling a tree so as to minimize peak memory has first been investigated by Liu [106] in the sequential case: he proposed an algorithm to find a peak-memory-minimizing traversal of a task tree when the traversal is required to correspond to a postorder traversal of the tree. A *postorder* traversal requires that each subtree of a given node must be fully processed before the processing of another subtree can begin. A follow-up study [105] presented an optimal algorithm to solve the general problem, without the postorder constraint on the traversal. Postorder traversals are known to be arbitrarily worse than optimal traversals for memory minimization [88]. However, they are very natural and straightforward solutions to this problem, as they allow us to fully process one subtree before starting a new one. Therefore, they are widely used in sparse matrix software like `MUMPS` [9, 10], and achieve good performance on actual elimination trees [88].

As mentioned in the introduction, the problem of minimizing the I/O volume has been studied in [88] with the constraint that each data either stays in the memory or has to be written wholly to disk. We study here the case when we have the option to store part of the data, which is also the topic of E. Agullo's

PhD. thesis [3]. In his thesis, Agullo exhibited the best postorder traversal for minimizing I/O volume, which we adapt to our model in Section 5.3.1. He also studied numerous variants of the model that are important for direct solvers, as well as other memory management issues—both for sequential and parallel processing. Based on these preliminaries, he presented an out-of-core version of the MUMPS solver.

Out-of-core execution is a well-known approach for computing on large data, especially (but not exclusively) in linear algebra [127, 136].

Our problem can also be related to studies of cache misses, which are conducted in an obviously different context. A CPU cache allows to speed-up memory access times, by saving previously accessed memory blocks in a smaller but faster memory. Following the terminology of Hill and Smith [81], a CPU cache is composed of multiple sets of block frames. Each block of the secondary storage can only be stored in the block frames of a unique set, given by a mapping function. A cache is fully-associative if it contains a single set, i.e., each memory block can be stored to any cache block frame. A fully-associative cache leads to less cache misses, but each cache access is then slower. Cache misses, i.e., accesses to data which is not present in the cache, can be categorized into three types: *compulsory misses* concern data which is accessed for the first time; *conflict misses* occur when data has been evicted because all the block frames of a set are full; *capacity misses* occur when data has been evicted because all the block frames of the cache are full. In this chapter, we don't have compulsory misses as data is not present on secondary storage initially, nor conflict misses as the model we consider is similar to a fully-associative cache. Therefore, only capacity misses are relevant here. Other studies consider a fully-associative cache, but focus on the cache replacement policy given a sequence of operations. The difficulty in this chapter is rather to decide the order of the operations, the optimal replacement policy being easily computed, similarly to the study in [110], see Theorem 5.1. Indeed, in our study, we are aware of the whole set of operations when deciding the replacement policy, whereas most cache misses studies have to decide the eviction procedure without knowing future operations, see [113] for some mathematical results in this field. Typical cache-eviction heuristics include LRU, which replaces the Least Recently Used block, or MRU, which replaces the Most Recently Used block. Jain and Lin [89] review many cache replacement algorithms and propose an original solution using past operations to predict the future, before applying an adapted version of the optimal clairvoyant replacement policy of Belady [24] to improve the efficiency on repetitive workflows.

5.2 Problem modeling and basic results

5.2.1 Model and notation

As introduced above, we assume that we have an available memory (or primary storage) of limited size M, and a disk (or secondary storage) of unlimited size.

We consider a workflow of tasks whose precedence constraints are modeled by a tree of tasks $G = (V, E)$. Its nodes $v \in V$ represent tasks and its edges $e \in E$ represent dependences. All dependences are directed toward the root (denoted by *root*): a node can only be executed after the termination of all its children. The output data of a node i (also named the ***weight*** of node i) occupies a size m_i in the main memory. This data may be written totally or partially to the disk after task i produces it. In order for a node to be executed, the output data of all its children must be entirely stored in the main memory. An amount of memory m can be moved between the memory and the disk at a cost of m I/O operations, regardless of which data it corresponds to. We assume that all memory values (M, m_i) are given in an appropriate unit (such as kilobytes) and are integers. We divide the main memory into *slots*, where each slot holds one such unit of memory.

During the computation of a task i, we adopt a memory behavior similar to Chapter 4: the input data are replaced by the output data, so that they never coexist in memory. In other words, at the beginning of the computation of task i, the output data of i's children must be *in memory*, while at the end of its computation, its own output data must be in memory. The amount of memory needed in order to execute node i is thus:

$$\bar{m}_i = \max\left(m_i, \sum_{(j,i)\in E} m_j\right).$$

Again, we choose this model as it is convenient to study and more complex behaviors can be emulated, see Section 4.2.2, Chapter 4. We furthermore assume that M is at least as large as every $\bar{m}_i$, as otherwise the tree cannot be processed.

Our objective is to find a solution minimizing the total I/O volume. A solution needs to give the order in which nodes should be executed, and how much of each node should be written out during I/O operations. In particular, for a tree of n tasks, we define a solution to our problem as a *permutation* σ of $[1 \dots n]$ and an ***I/O function*** f_{IO}. We call such a solution a ***traversal***. The permutation σ represents the ***schedule*** of the nodes, that is, $\sigma(i) = t$ means that task i is the t-th task computed. The function f_{IO} represents the amount of I/O for each task: $f_{IO}(i) = m$ means that m units of the output data of task i are written to disk (see below). Note that we do not need to clarify which part of the data is written to disk, as our cost function only depends on the volume. We assume without loss of generality that when $f_{IO}(i) \neq 0$, the *write* operation on the output data of task i is performed right after task i completes (and produces the data), and the *read* operation is performed just before the use of this data by task i's parent. Finally, since there are exactly the same number of *read* and *write* operations, we only count the *write* operations.

In order for a traversal to be valid, it must respect the following conditions:

- Tasks are processed in topological order:

  $$\forall (i,j) \in E,\ \sigma(i) < \sigma(j).$$

  We say that a node i of parent j is considered ***active*** at step t under the schedule σ if $\sigma(i) < t < \sigma(j)$. This means that its output data is either partially in memory and/or partially written to disk at time t.
- The amount of data written to disk never exceeds the size of the data:

  $$\forall i \in V,\ 0 \leq f_{IO}(i) \leq m_i;$$

- Enough memory remains available for the processing of each task (taking into account active nodes):

  $$\forall i \in G, \sum_{\substack{(k,p)\in E \\ \sigma(k)<\sigma(i)<\sigma(p)}} \left(m_k - f_{IO}(k)\right) \leq M - \bar{m}_i. \tag{5.1}$$

The problem we are considering in this paper, called MinIO, is to find a valid traversal that minimizes the total amount of I/O, given by $\sum_{i\in G} f_{IO}(i)$.

We formally define a ***postorder*** traversal as a traversal σ such that, for any node i and for any node k outside the subtree Tr_i rooted at i, we have either $\forall j \in Tr_i,\ \sigma(k) < \sigma(j)$ or $\forall j \in Tr_i,\ \sigma(j) < \sigma(k)$.

5.2.2 Towards a compact solution

Although a traversal is described by both the schedule σ and the I/O function f_{IO}, the following results show that one can be deduced from the other. The first result is adapted from [3, Property 2.1], which

has the same result limited to postorder traversals (see Section 5.1). It states that given a schedule σ, it is easy to derive an I/O scheme f_{IO} which minimizes the I/O volume of the traversal (σ, f_{IO}).

Theorem 5.1. *Consider a tree G, a memory bound M, and a schedule σ. The I/O function f_{IO} following the* Furthest in the Future *policy achieves the best performance under σ.*

The I/O function f_{IO} following the *Furthest in the Future* (FiF) policy is defined as follows: during the execution of σ, whenever the memory exceeds the limit M, I/O operations are performed on the active nodes which will remain active the furthest in the future, i.e., whose execution come last in the schedule σ. This result is similar to Mattson et al.'s rule which states that this cache replacement policy is optimal [110].

Proof. Given a tree G, a memory bound M, a schedule σ, and an I/O function f_{IO} that does not respect the FiF policy, it is straightforward to transform f_{IO} into another I/O function f'_{IO} following the rule. Consider the first step when an I/O is performed on data i that is not the last to be used among active data. Let j denote the last-used among active data (so FiF would evict j). We can safely increase $f'_{IO}(j)$ and decrease $f'_{IO}(i)$ until either $f'_{IO}(j) = m_j$ or $f'_{IO}(i) = 0$. As j is active longer than i is, the memory freed by f'_{IO} is available for a longer time than the one freed by f_{IO}, which keeps the traversal valid. Repeating this transformation, we produce an I/O function which respects the FiF policy. □

On the other hand, if we have an I/O function f_{IO} describing how much of each node is written to disk, we can compute a schedule σ such that (σ, f_{IO}) is a valid traversal (if such a schedule exists).

Theorem 5.2. *Consider a tree G, a memory bound M, and an I/O function f_{IO} for which there exists a valid schedule. Such a schedule can be computed in polynomial time.*

The proof of this result is delegated to Section 5.5 where we use a similar method to derive a heuristic: once we know where the I/O operations take place, we may transform the tree by *expanding* some nodes to make these I/O operations explicit within the tree structure. If a valid traversal using f_{IO} exists, the resulting tree may be completely scheduled without any additional I/O, and such a schedule can be computed using an optimal scheduling algorithm for memory minimization.

Both previous results allow us to describe solutions in a more compact format (as either a schedule or an I/O function). However, this does not make the problem less combinatorial: there are $n!$ possible schedules and already 2^n functions f_{IO} if we restrict only to functions such that $f_{IO}(i) = 0$ or m_i.

5.2.3 Related algorithms

As mentioned in Section 5.1, the problem of minimizing peak memory, denoted MinMem, is closely related to our problem, and has been extensively studied. In this problem, the available memory is unbounded (which means no I/Os are required) and we look for a schedule that minimizes the peak memory, i.e., the maximum amount of memory used at any time during the execution. There are at least two important algorithms for this problem, which we use in the present paper:

- It is possible to compute a schedule minimizing the peak-memory in polynomial time, as proven by Liu [105]. We refer to this algorithm as OptMinMem.
- The best postorder traversal for peak-memory minimization can also be computed in polynomial time [106]. We refer to this algorithm as PostOrderMinMem.

5.3 Existing solutions are not satisfactory

We now detail two existing solutions for the MinIO problem. The first one is the best postorder traversal for MinIO proposed by Agullo [3]. The second uses the optimal traversal for MinMem proposed by Liu [105], and then applies Theorem 5.1 to obtain a valid traversal. After presenting these algorithms, we prove that neither of them is constant-factor competitive compared to the optimal traversal.

5.3.1 Computing the best postorder traversal

For the sake of completeness, we present the algorithm computing the best postorder traversal for MinIO from [3] and adapt it to our model. Recall that in a postorder traversal, when a node is processed, its whole subtree must be processed before any other external node may be started. Given a node i and a postorder schedule σ, we first recursively define S_i as the storage requirement of the subtree Tr_i rooted at i. Let $Chil(i)$ be the children of i. Then:

$$S_i = \max\left(m_i, \max_{j \in Chil(i)}\left(S_j + \sum_{\substack{k \in Chil(i) \\ \sigma(k)<\sigma(j)}} m_k\right)\right).$$

This expression represents the maximum memory peak reached during the execution. If the peak is obtained at the end of the execution, it is then equal to m_i. Otherwise, it appears during the execution of the subtree of some child j. In this case, the peak is composed of the weights of the children already processed, plus the peak S_j of Tr_j.

We may now consider $A_i = \min(M, S_i)$, which represents the amount of main memory used for the out-of-core execution of the subtree Tr_i by σ. We recursively define V_i as the volume of I/Os performed by σ during the execution Tr_i when I/O operations are chosen using the FiF policy:

$$V_i = \max\left(0, \max_{j \in Chil(i)}\left(A_j + \sum_{\substack{k \in Chil(i) \\ \sigma(k)<\sigma(j)}} m_k\right) - M\right) + \sum_{j \in Chil(i)} V_j.$$

The expression of V_i has a similar structure to the expression of S_i. No I/Os can be incurred when only the root i is in memory, hence m_i has no effect here. The second term accounts for the I/Os incurred on the children of i. Indeed, during the execution of node j, some parts of children of i must be written to disk if the memory peak exceeds M, and this quantity is at least $A_j + \sum_{k \in Chil(i)}^{\sigma(k)<\sigma(j)} m_k - M$. The last term accounts for the I/Os occurring inside the subtrees. Note that such I/Os can only happen if the memory peak of the subtree exceeds M.

It remains to determine which postorder traversal minimizes the quantity V_{root}. Note that the only term sensitive to the ordering of the children of i in the expression of V_i is:

$$\max_{j \in Chil(i)}\left(A_j + \sum_{\substack{k \in Chil(i) \\ \sigma(k)<\sigma(j)}} m_k\right).$$

Theorem 5.3 states that sorting the children of i in decreasing order of $A_j - m_j$ achieves the minimum V_i.

Theorem 5.3 (Lemma 3.1 in [106]). *Given a set of values $(x_i, y_i)_{1 \le i \le n}$, the minimum value of $\max_{1 \le i \le n}\left(x_i + \sum_{j=1}^{i-1} y_j\right)$ is obtained by sorting the sequence (x_i, y_i) in decreasing order of $x_i - y_i$.*

Therefore, the postorder traversal that processes the children nodes by decreasing order of $A_i - m_i$ minimizes the I/O cost among all postorder traversals. This traversal is described in Algorithm 13, initially called with $r = root$, and will be referred to as PostOrderMinIO. Note that in the algorithm $\oplus$ refers to the concatenation operation on lists.

Algorithm 13: PostOrderMinIO (G, r)

Output: a tree G and a node r in G
Output: an ordered list ℓ_r of the nodes in the subtree rooted at r, corresponding to a postorder

```
1 foreach i child of r do
2     ℓ_i ← PostOrderMinIO(G, i)
3     Compute the A_i value using postorder ℓ_i
4 ℓ_r ← ∅
5 for i child of r in decreasing order of A_i − m_i do
6     ℓ_r ← ℓ_r ⊕ ℓ_i
7 ℓ_r ← ℓ_r ⊕ {r}
8 return ℓ_r
```

5.3.2 PostOrderMinIO is optimal on homogeneous trees

In this section we focus on ***homogeneous trees***—that is, on trees where all nodes have output data of size one. We show that PostOrderMinIO is optimal on these homogeneous trees, i.e., that it performs the minimum number of I/Os. This somehow generalizes a result of Sethi and Ullman [131], which considers binary trees from arithmetic expressions and aim at minimizing the number of store/load operations when evaluating these expressions with a limited number of registers. They considered different variants, and the one with commutative operators closely resembles our problem, where the registers are replaced by memory slots and load/store by read/write. However, in our work, we do not limit the model to binary trees, but consider any tree with homogeneous data sizes. In the case that the heterogeneity in data sizes is limited, our result provides a good strategy of minimizing the amount of I/O operations.

Theorem 5.4. PostOrderMinIO *is optimal for homogeneous trees.*

In order to prove this theorem, we need first to define labels on the nodes of a tree. Let Tr be any homogeneous tree ($m_v = 1$ for all nodes v of Tr). In the following definitions, whenever v is a node of Tr with k children, $v_1, \ldots, v_k$ will be its children.

Memory bound $\ell(v)$. For each node v of Tr, we recursively define a label $\ell(v)$ which represents the minimum amount of memory necessary to execute the subtree $Tr(v)$ rooted at v without performing any I/Os:

$$\ell(v) = \begin{cases} 0 & \text{if } v \text{ is a leaf} \\ \max_{1 \le i \le k} (\ell(v_i) + i - 1) & \text{otherwise} \\ & \text{when ordering the children such that} \\ & \ell(v_i) \ge \ell(v_{i+1}) \text{ for } 1 \le i \le k-1 \end{cases}$$

Let PostOrder be a postorder schedule that executes the children of any node by non-increasing ℓ-labels (ties being arbitrarily broken). Intuitively, under PostOrder, while computing the i-th child, we have $i - 1$ extra nodes in memory, each of size one, so we need $\ell(v_i) + (i - 1)$ memory slots in total.

I/O indicator $c(v)$. If v_i is a child of v, intuitively, $c(v_i)$ represents the number of children of v written to disk by PostOrder during the execution of $Tr(v_i)$. This number can be either 0 or 1. We set $c(v_1) = 0$ and

$$c(v_i) = \begin{cases} 0 & \text{if } \ell(v_i) + \sum_{1 \le j < i} (1 - c(v_j)) \le M \\ 1 & \text{otherwise.} \end{cases}$$

We set $c(root) = 0$ where *root* is the root of Tr. To ease the writing of some proofs, we use the notation

$$m(v_i) = \sum_{1 \le j < i} (1 - c(v_j)).$$

Thus $m(v_i)$ represents the number of children of v in memory right before v_i is executed. Note that $m(v_1) = 0$ and $m(v_i) = (1 - c(v_1)) + \sum_{2 \le j < i} (1 - c(v_j)) \ge (1 - c(v_1)) = 1$ for $2 \le i \le k$.

I/O volumes $w(v)$ and $W(Tr(v))$. $w(v)$ represents the total number of children of v stored by PostOrder:

$$w(v) = \sum_{i=1}^{k} c(v_i) = \sum_{i=2}^{k} c(v_i).$$

Finally, for a given node v, we define $W(Tr(v))$ on the subtree rooted at v:

$$W(Tr(v)) = c(v) + \sum_{\mu \in Tr(v)} w(\mu).$$

$W(Tr(v))$ intuitively represents the total volume of communications performed during the execution of the tree $Tr(v)$ by PostOrder.

We first state the correctness of the ℓ-labels and the optimality of PostOrder for the MinMem problem.

Lemma 5.1. *With infinite memory,* PostOrder *uses $\ell(n)$ slots to compute the subtree rooted at node n.*

Proof. The result follows from the definition of $\ell(v)$. □

Lemma 5.2. *With infinite memory, any schedule uses at least $\ell(v)$ slots to compute the subtree rooted at v.*

Proof. We prove this result by induction on the size of $Tr(v)$. If v is a leaf, the result holds ($\ell(v) = 1$).

Otherwise, we assume the lemma to be true for the subtrees rooted at the children $v_1, \ldots, v_k$ of v. We consider the schedule returned by MinMem. The memory peak inherent to the execution of a subtree $Tr(v_i)$ is equal to $\ell(v_i)$ by the induction hypothesis. Assume without loss of generality that the children of v are ordered such that MinMem first computes a node of $Tr(v_1)$, then the next executed node not in $Tr(v_1)$ is in $Tr(v_2)$, then the next executed node neither in $Tr(v_1)$ nor in $Tr(v_2)$ is in $Tr(v_3)$, and so on. Then, the memory peak reached during the execution of $Tr(v_i)$ is at least $\ell(v_i) + (i - 1)$ because, in addition to $Tr(v_i)$, at least $i - 1$ subtrees have been partially executed: $Tr(v_1)$, ..., $Tr(v_{i-1})$. Finally, the total memory peak is at least equal to $\max_{1 \le i \le k}(\ell(v_i) + i - 1)$. By Theorem 5.3, this quantity is minimized when the nodes are ordered by non-increasing values of $\ell(v_i)$. Hence, the total memory peak is at least $\ell(v)$. □

We now state the performance of PostOrder for the MinIO problem (I/Os are performed using the FiF policy).

Lemma 5.3. POSTORDER *computes a given tree Tr using at most $W(Tr)$ I/Os.*

Proof. We prove this result by induction on the size of Tr. We introduce a new notation: for any node v of Tr we define $\mathscr{W}(v)$ as $\mathscr{W}(v) = W(Tr(v)) - c(v)$. In other words, $\mathscr{W}(v)$ represents the total volume of communications performed during the execution of the tree $Tr(v)$ *if* we had nothing to execute but $Tr(v)$ (in practice $Tr(v)$ may be a strict sub-tree of Tr and, therefore, the execution of $Tr(v)$ in the midst of the execution of Tr can induce more communications). Note that $\mathscr{W}(v) = W(Tr(v))$ if v is the root of Tr. We prove by induction on the size of $Tr(v)$ that at most $\mathscr{W}(v)$ I/Os are performed during the execution of $Tr(v)$.

Let us assume that v is a leaf. Because we have assumed (in Section 5.2.1) that M was large enough for a single node to be processed without I/Os, $c(v) = 0$ and thus $W(Tr(v)) = 0 = \mathscr{W}(v) + c(v)$.

Now assume that v is not a leaf. By the induction hypothesis, for any $i \in [1; k]$, POSTORDER executes the tree $Tr(v_i)$ *alone* using at most $\mathscr{W}(v_i)$ I/Os. We prove that to process the tree $Tr(v_i)$, after the trees $Tr(v_1)$ through $Tr(v_{i-1})$ were processed, we need to perform at most $W(Tr(v_i)) = \mathscr{W}(v_i) + c(v_i)$ I/Os.

Let us consider the $(i+1)$-th child of v. If $c(v_{i+1}) = 0$, then $\ell(v_{i+1}) + \sum_{1 \leq j < i+1}(1 - c(v_j)) \leq M$. Then, according to Lemma 5.1, no I/Os are required to execute $Tr(v_{i+1})$ under POSTORDER even after the processing of $Tr(v_1)$ through $Tr(v_i)$. Indeed, before the start of the processing of $Tr(v_{i+1})$ the memory contains exactly $\sum_{1 \leq j < i+1}(1 - c(v_j))$ nodes. Therefore $\mathscr{W}(v_{i+1}) = c(v_{i+1}) = W(Tr(v_{i+1})) = 0$.

We are now in the case $c(v_{i+1}) = 1$; thus $\ell(v_{i+1}) + \sum_{1 \leq j < i+1}(1 - c(v_j)) > M$. Recall that for $l \in [1; i]$, $\ell(v_l) \geq \ell(v_{i+1})$. Thus, if $\ell(v_{i+1}) \geq M$, then for $l \in [2; i]$, $\ell(v_l) \geq M$ and $c(v_l) = 1$ (because $m(v_l) \geq (1 - c(v_1)) = 1$). Therefore, after the completion of $Tr(v_i)$ there is only one node remaining in the memory: v_i. Then with a single I/O POSTORDER writes v_i to disk, the memory is empty, and $Tr(v_{i+1})$ can then be processed with at most $\mathscr{W}(v_{i+1})$ I/Os, giving a total of at most $\mathscr{W}(v_{i+1}) + c(v_{i+1}) = W(Tr(v_{i+1}))$ I/Os. The only remaining case is the case $\ell(v_{i+1}) < M$. The processing of $Tr(v_i)$ requires at least $\ell(v_{i+1})$ empty memory slots because $\ell(v_i) \geq \ell(v_{i+1})$. Hence, after the completion of $Tr(v_i)$ there are at least $\ell(v_{i+1}) - 1$ empty memory slots (the memory including the node v_i itself). Then with a single I/O POSTORDER can write v_i to disk and there are then enough empty memory slots to process $Tr(v_{i+1})$ without any additional I/Os. Therefore $W(Tr(v_{i+1})) = 1 = \mathscr{W}(v_{i+1}) + c(v_{i+1})$. This concludes the proof. □

Lemma 5.5 relies on the following intermediate result.

Lemma 5.4. *Consider a node v of a tree Tr with a child, a, whose label $\ell(a)$ satisfies $\ell(a) > M$. Now, consider any tree Tr' identical to Tr, except that the subtree rooted at a has been replaced by any tree whose new label $\ell'(a)$ satisfies $\ell'(a) \leq \ell(a)$ and $\ell'(a) \geq M$. Then $w'(v) = w(v)$.*

Proof. Let $v_1, \ldots, v_k$ be the children of v, ordered so that $\ell(v_1) \geq \cdots \geq \ell(v_k)$. Let j be the index of a: $a = v_j$. As the label of a in Tr', $\ell'(a)$, is not larger than $\ell(a)$, we can have $\ell'(a) < \ell'(v_{j+1})$. Therefore, we define another ordering of the children of v denoted by $v'_1, \ldots, v'_k$ such that $\ell'(v'_1) \geq \cdots \geq \ell'(v'_k)$. Let j' be the index of a in this ordering: $v'_{j'} = a = v_j$.

Note that $j' \geq j$. For $i \in [j+1; j']$, we have $v_i = v'_{i-1}$; at j, we have $v_j = v'_{j'}$; and for $i \notin [j; j']$, we have $v_i = v'_i$.

If $j' = 1$ then $j = 1$. This case means that a remains the node with the largest label. The labels of the other children of v remain unchanged. Because $c(v_1) = c'(v_1) = 0$ by definition, then $c'(v_i) = c(v_i)$ for any child v_i of v and, thus, $w(v)$ is equal to $w'(v)$.

Let us now consider the case $j' > 1$. From what precedes, $v'_{j'-1} = v_{j'}$. Then $\ell(v_{j'}) = \ell'(v'_{j'-1}) \geq \ell'(v'_{j'}) = \ell'(a) \geq M$. However, for any $i \in [1; j']$, $\ell'(v'_i) \geq \ell'(v'_{j'}) \geq M$ and $\ell(v_i) \geq \ell(v_{j'}) \geq M$. Therefore, for any $i \in [2; j']$, $\ell'(v'_i) + m'(v'_i) > M$ (because $m'(v'_i) \geq 1 - c'(v'_1) = 1$) and, thus, $c'(v'_i) = 1$. Similarly, for any $i \in [2; j']$, $\ell(v_i) + m(v_i) > M$ (because $m(v_i) \geq m(v_1) = 1$) and, thus, $c(v_i) = 1$. Therefore,

for $i \in [1; j']$, $c(v_i) = c'(v'_i)$. Then, for $i \in [j'+1; k]$, $v_i = v'_i$, $m(v_i) = m'(v'_i)$, and $c(v_i) = c'(v'_i)$ by an obvious induction. Therefore, $w'(v) = \sum_{i=2}^{k} c'(v'_i) = \sum_{i=2}^{k} c(v_i) = w(v)$. □

The following lemma gives a lower bound on the I/Os performed by any schedule.

Lemma 5.5. *No schedule can compute a tree Tr performing strictly less than $W(Tr)$ I/Os.*

As the proof needs to focus on deep details, we first provide a short summary. The result is proven by induction on the size of the tree. The case where no I/O is required is deduced from Lemma 5.2.

We then consider a tree Tr for which any schedule performs at least one I/O, and an optimal schedule $\mathscr{S}$ on this tree. We focus on the first node s to be stored under this schedule, and define the tree Tr' in which $Tr(s)$ is replaced by s. Using the induction hypothesis, we know that any schedule on Tr', including the restriction of $\mathscr{S}$ on Tr', performs at least $W(Tr')$ I/Os. Therefore, we deduce that $\mathscr{S}$ performs at least $W(Tr')+1$ I/Os on Tr. Thus, it remains to prove that $W(Tr') \geq W(Tr) - 1$.

To do so, we focus on the closest ancestor of s to have a label ℓ larger than M, and denote it as μ. We first prove that in the new tree Tr', we have $\ell(\mu) \geq M$. This means, by Lemma 5.4, that the w labels of the ancestors of μ are unchanged in Tr'. Then, we prove through an extensive case study that $w(\mu)$ in Tr' cannot be smaller than $w(\mu)$ in Tr minus one. Finally, we conclude that all the other w labels are equal in Tr and in Tr'; therefore, $W(Tr') \geq W(Tr) - 1$.

Proof. We proceed by induction on the number of nodes of Tr.

The base case consists of a tree Tr that can be scheduled without any I/O. For contradiction, assume that $W(Tr) > 0$. Then there exists a node v of Tr such that $w(v) > 0$ and a child v_i of v such that $c(v_i) = 1$. Then, by definition of $c(v_i)$ and of $\ell(v)$, $\ell(v) > M$. However, according to Lemma 5.2, "any schedule uses at least $\ell(v)$ slots to compute $Tr(v)$", so $Tr(v)$, and thus Tr, cannot be scheduled without I/Os. Hence, a contradiction; thus $W(Tr) = 0$.

Consider a tree Tr that cannot be scheduled without I/Os, and a schedule $\mathscr{S}$ on Tr that minimizes the total volume of I/Os.

First, by Lemma 5.1, there exists a node v such that $\ell(v) > M$. Otherwise, PostOrder would be able to schedule Tr without I/Os, which would violate our assumption on Tr. Then, the label of the root r of Tr also satisfies $\ell(r) > M$.

Let s be the first node to be stored under $\mathscr{S}$. Then, the subtree $Tr(s)$ has been scheduled without I/Os so, by Lemma 5.2, we have $\ell(s) \leq M$ and, hence, no node of $Tr(s)$ has a label larger than M. Let μ be the closest ancestor of s to have a label larger than M. μ exists as $\ell(r) > M$ and $\ell(s) \leq M$. Let $\mu_1, \dots, \mu_k$ be the children of μ, ordered such that $\ell(\mu_i) \geq \ell(\mu_{i+1})$. Let j be such that μ_j is either s or one of its ancestors. Let $t = \min\{i \in [1; k] \mid \ell(\mu_i) + i - 1 > M\}$ (t exists because, by definition, $\ell(\mu) > M$). See Figure 5.1 for an illustration of the tree.

Let Tr' be the tree obtained from Tr by replacing s by a leaf, therefore replacing the subtree $Tr(s)$ by a single node s. As $Tr(s)$ cannot be empty, Tr' contains fewer nodes than Tr. Consider a schedule $\mathscr{S}'$ on Tr' that executes the same operations as $\mathscr{S}$ on Tr and in the same order, except for the ones concerning $Tr(s)$.

We use the following notation: as above, ℓ, m, c, w are defined on nodes of the tree Tr, whereas ℓ', m', c', w' refer to the same values on the tree Tr'. The nodes in Tr' share the same names as their equivalent in Tr.

We define, as in the proof of Lemma 5.4, an ordering $\mu'_1, \dots, \mu'_k$ on the children of μ, $\ell'(\mu'_i) \geq \ell'(\mu'_{i+1})$. Furthermore, we assume that this order is consistent with the original one, which means the following. Let j' be such that $\mu_j = \mu'_{j'}$. Note that $j' \geq j$. For $i \in [j+1; j']$, we have $\mu_i = \mu'_{i-1}$; at j, $\mu_j = \mu'_{j'}$; for $i \notin [j; j']$, we have $\mu_i = \mu'_i$. In particular, we have $\mu_{j'} = \mu'_{j'-1}$ if $j' > j$ and $\mu_{j'} = \mu'_{j'}$ if $j' = j$.

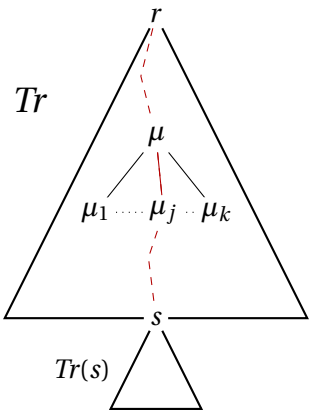


Figure 5.1: Scheme of the composition of the tree Tr.

Note that, except s and its ancestors, every node v of Tr' satisfies $\ell(v) = \ell'(v)$ and $w(v) = w'(v)$. Our objective is to prove that $W(Tr') \geq W(Tr) - 1$. We first prove that $\ell'(\mu) \geq M$. We split into cases based on the value of t defined above:

1. $t < j$. The labels of $\mu_1, \ldots, \mu_t$ are left unchanged so $\ell'(\mu) \geq \ell'(\mu_t) + t - 1 > M$.

2. $t = j$. By definition of μ, we have $\ell(\mu_j) \leq M$, so we cannot have $t = j = 1$. The labels of $\mu_1, \ldots, \mu_{t-1}$ are left unchanged, and $\ell(\mu_{t-1}) \geq \ell(\mu_t)$, so

$$\ell'(\mu) \geq \ell'(\mu_{t-1}) + t - 2 \geq \ell(\mu_t) + t - 1 - 1 > M - 1.$$

3. $t > j$. Among $\mu_1, \ldots, \mu_t$, the only label that changed is μ_j. Therefore there are $t-2$ nodes that have a label ℓ' larger than that of μ_t. Hence,

$$\ell'(\mu) \geq \ell'(\mu_t) + t - 2 > M - 1.$$

Now, we prove that $w'(\mu) \geq w(\mu) - 1$, by showing that there exists at most one index i such that $c(\mu_i) = 1$ and $c'(\mu_i) = 0$. Let I be the set of such indexes. Note that no index strictly smaller than j can be in I as the relevant labels are identical in both trees.

The following studies how the labels c and c' can differ. We consider two cases:

1. $c(\mu_j) = 0$. Thus $j \notin I$. Let $a = \min\{i \in [j+1, k] \mid c(\mu_i) = 1\}$. There are several cases; in each we show that I contains at most one element.

   (a) First, a does not exist. Then I is empty.

   (b) Assume $a > j'$. No index in $[1; j]$ can be in I, and thus no index in $[1; j']$. In particular, $c(v_l) = 0$ for $l \in [j; j']$ by definition of a. Because node μ_j appears right after node $\mu_{j'}$ in Tr', then $m'(\mu_j) = m'(\mu_{j'}) + (1 - c(\mu_{j'})) = (m(\mu_{j'}) - (1 - c(\mu_j))) + (1 - c(\mu_{j'})) = m(\mu_{j'}) + c(\mu_j) - c(\mu_{j'}) = m(\mu_{j'})$. Therefore, we have $m'(\mu_j) = m(\mu_{j'})$. As $\ell'(\mu_j) \leq \ell'(\mu_{j'})$, we get $m'(\mu_j) + \ell'(\mu_j) \leq m(\mu_{j'}) + \ell'(\mu_{j'})$. Then, because $\ell'(\mu_{j'}) = \ell(\mu_{j'})$, and by the definition of c, we conclude that $c'(\mu_j) \leq c(\mu_{j'})$.

   By definition, $j' \geq j$. Because $a > j'$, if $j' > j$, then $c(\mu_{j'}) = 0$ by definition of a. Otherwise $j' = j$ and we use the assumption $c(\mu_j) = 0$ to conclude that in all cases $c(\mu_{j'}) = 0$. Combined with $c'(\mu_j) \leq c(\mu_{j'})$ this gives us $c'(\mu_j) = 0$.

Recall that the labels in $[1; j-1]$ are left unchanged, so $c(\mu_i) = c'(\mu_i)$ for $i \in [1; j-1]$. From what precedes, $c'(\mu_j) = c(\mu_j) = 0$. By definition of a and because $j' < a$, $c(\mu_i) = 0$ for $i \in [j+1; j']$. Thus, all these nodes have the same label ℓ in Tr' and Tr, and all of them have $m'(\mu_i) \leq m(\mu_i)$ (by definition of m: they are preceded by the same nodes so their sums have the same terms, except node μ_j). Therefore, for all these nodes $c'(\mu_i) = 0$ and thus $c'(\mu_i) = c(\mu_i)$. Hence, $m(\mu_{j'+1}) = m'(\mu_{j'+1})$. Because $\ell(\mu_{j'+1}) = \ell'(\mu_{j'+1})$ we conclude that $c(\mu_{j'+1}) = c'(\mu_{j'+1})$. We then proceed by a simple induction on the nodes with a larger index to prove that I is empty.

(c) Now, assume $a \leq j'$. Once again, because the labels in $[1; j-1]$ are left unchanged, and because $c(\mu_j) = 0$, no index in $[1; j]$ can be in I, and thus no index in $[1; a-1]$ can be in I.
We have two cases to consider, depending on whether a is equal to 2 (recall that by definition $a \geq j+1 \geq 2$).

i. $a = 2$. Then $j = 1$. Therefore, in Tr', μ_a is the first child and, by definition of c, $c'(\mu_a) = 0$.

ii. $a > 2$. By definition of a, $c(\mu_a) = 1$. Then, either $a = j+1$ and then $a-1 = j$ and $c(\mu_{a-1}) = c(\mu_j) = 0$, or $a > j+1$ and then $c(\mu_{a-1}) = 0$ by definition of a. In all cases, $c(\mu_{a-1}) = 0$. Therefore, $\ell(\mu_{a-1}) + m(\mu_{a-1}) \leq M$. Because $\ell(\mu_{a-1}) \geq \ell(\mu_a)$ and $m(\mu_a) = m(\mu_{a-1}) + 1$, $\ell(\mu_a) + m(\mu_a) \leq M+1$. Because $c(\mu_a) = 1$ by definition of a, $\ell(\mu_a) = m(\mu_a) \leq M+1$.
Recall (for the third time) that the labels in $[1; j-1]$ are left unchanged, so $c(\mu_i) = c'(\mu_i)$ for $i \in [1; j-1]$. Moreover, by definition of a, $c(\mu_i) = 0$ for all $i \in [j+1; a-1]$. Therefore, because $c(\mu_j) = 0$, for all $i \in [j+1; a-1]$ $m'(\mu_i) = m(\mu_i) - 1$ and thus $c'(\mu_i) = c(\mu_i) = 0$. Also, $m'(\mu_a) = m(\mu_a) - 1$. Then $\ell'(\mu_a) + m'(\mu_a) = \ell(\mu_a) + m(\mu_a) - 1 = M$ from what precedes. Therefore, $c'(\mu_a) = 0$.

Because $c'(\mu_a) = 0$, $m'(\mu_{a+1}) = m(\mu_{a+1})$. Then, by an immediate induction, $m'(\mu_i) = m(\mu_i)$ for $i \in [a+1; j']$. Therefore $[a+1; j'] \cap I = \emptyset$. In order to prove that $[j'+1; k] \cap I = \emptyset$, we have two cases to consider:

i. $c'(\mu_j) = 1$. Here, we have $m'(\mu_{j'+1}) = m(\mu_{j'+1})$. Indeed, the only nodes with an index not larger than j' that have different values for c and c' are μ_j and a. Therefore $c'(\mu_{j'+1}) = c(\mu_{j'+1})$. We can then proceed by induction to show that no index larger than j' belongs to I.

ii. $c'(\mu_j) = 0$. Here, we have $m'(\mu_{j'+1}) = m(\mu_{j'+1}) + 1$, and therefore $c'(\mu_{j'+1}) \geq c(\mu_{j'+1})$. We can then proceed by induction to show that for any index i larger than j' we have $m'(\mu_i) \geq m(\mu_i)$ and $c'(\mu_i) \geq c(\mu_i)$.

Therefore, we have $I = \{a\}$.

2. $c(\mu_j) = 1$. Recall that the labels in $[1; j-1]$ are left unchanged, so no index in $[1; j-1]$ can be in I. We now want to show that no index in $[j+1; k]$ can be in I. By definition of m and since $c(\mu_j) = 1$, we have $m(\mu_{j-1}) = m(\mu_j)$. Then for all $i \in [j+1; j']$, we have $\ell(\mu_i) = \ell'(\mu_i)$, and we get by an immediate induction that for all $i \in [j+1; j']$, we have $c(\mu_i) = c'(\mu_i)$. In order to prove the result on the interval $[j'+1; k]$, we have two cases to consider:

(a) $c'(\mu_j) = 1$. Here, we have $m'(\mu_{j'+1}) = m(\mu_{j'+1})$, and therefore $c'(\mu_{j'+1}) = c(\mu_{j'+1})$. We can then proceed by induction to show that no index larger than j' belongs to I.

(b) $c'(\mu_j) = 0$. Here, we have $m'(\mu_{j'+1}) = m(\mu_{j'+1}) + 1$, and therefore $c'(\mu_{j'+1}) \geq c(\mu_{j'+1})$. We can then proceed by induction to show that for any index i larger than j' we have $m'(\mu_i) \geq m(\mu_i)$ and $c'(\mu_i) \geq c(\mu_i)$.

Therefore, $I \subseteq \{j\}$.

Putting things together, no node of $Tr(s)$ has a label ℓ larger than M, so none has a positive label w. Between μ and s, no node had a label ℓ larger than M. Therefore, except μ and its ancestors, all the nodes satisfy $w'(v) = w(v)$.

As $\ell'(\mu) \geq M$, all the ancestors v of μ satisfy $\ell'(v) \geq M$, so by Lemma 5.4, they also satisfy $w'(v) = w(v)$. Then, as $w'(\mu) \in \{w(\mu) - 1, w(\mu)\}$, we have $W(Tr') \geq W(Tr) - 1$.

By the induction hypothesis, $\mathscr{S}'$ executes at least $W(Tr') = W(Tr) - 1$ I/Os, so $\mathscr{S}$ executes at least $W(Tr)$ I/Os, which proves the lemma. □

We are now ready to prove Theorem 5.4.

Proof of Theorem 5.4. From Lemma 5.3 and Lemma 5.5, PostOrder is optimal for homogeneous trees. Moreover, PostOrderMinIO is a postorder that minimizes the volume of I/O operations. Hence, it is also optimal for homogeneous trees. □

The PostOrder algorithm designed in this proof is actually equivalent to PostOrderMinMem, the postorder algorithm minimizing the peak memory, when applied to homogeneous trees. The only difference with PostOrderMinIO is that the latter sorts the children by non-increasing $A_i = \min(M, l_i)$ whereas PostOrder sorts them by non-increasing l_i. PostOrderMinIO is then less specific, as it does not specify the order among subtrees with $l_i \geq M$: there are more ties that can be arbitrarily broken. This difference also implies that the schedule given by PostOrder does not depend on the value of M. It is thus *cache-oblivious* [65] and optimal (on homogeneous trees) for any memory size. Therefore, if we consider several levels of memory (e.g., a cache memory connected to a RAM, itself connected to a disk), PostOrder minimizes the memory transfers between every level (e.g., both cache-RAM and RAM-disk transfers). Note that with heterogeneous trees, this result does not hold anymore, as the optimal traversal depends on the memory size. Therefore, no algorithm can simultaneously minimize transfer between all levels of the memory hierarchy.

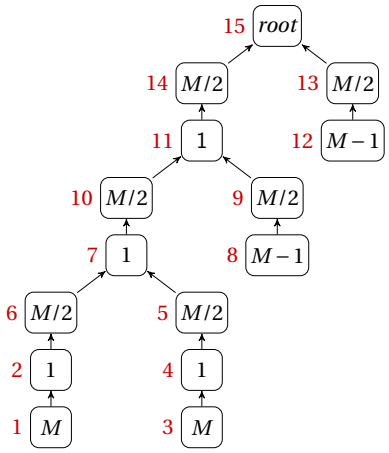
5.3.3 Postorder traversals are not competitive

Previous research has shown that the best postorder traversal for the MinMem problem is arbitrarily far from the optimal traversal [88]. We prove here that postorder traversals may also have bad performance for the MinIO problem. More specifically, we prove that there exist problem instances on which PostOrderMinIO performs arbitrarily more I/O than the optimal I/O amount. We could exhibit an example where the optimal traversal does not perform any I/O and PostOrderMinIO performs some I/O, but we rather present a more general example where the optimal traversal performs some I/O: in the following example, the optimal traversal requires 1 I/O, when PostOrderMinIO requires $\Omega(nM)$ I/Os . The tree used in this instance is depicted on Figure 5.2(a).

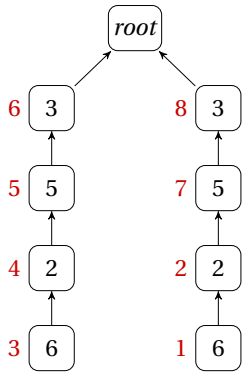
It is possible to traverse the tree of Figure 5.2(a) with a memory of size M using only a single I/O, by executing the nodes in increasing order of the (red) labels next to the nodes. After processing the minimal subtree including the two leftmost leaves, our strategy is to process leaves from left to right. Before processing a new leaf, we complete the previous subtree up to a node of weight 1; this way the leaf and the actives nodes can both fit in memory.

On the other hand, the best postorder traversal must perform a volume of I/O equal to $M/2-1$ before processing any leaf, except for the very first processed leaf. This is because the least common ancestor of any two leaf nodes has two children of size $M/2$, and all leaves have size at least $M-1$. Thus, any postorder traversal performs at least $M/2-1$ I/Os for all but one leaf node, leading to at least $3M/2-3$

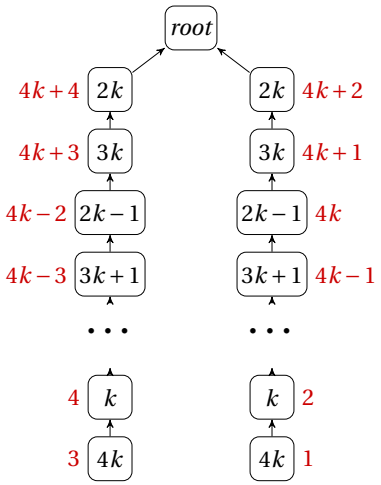
I/Os for the tree in Figure 5.2(a) (a best postorder starts from any of the two M leaves and performs $3M/2-2$ I/Os). We can extend this tree as follows: we replace *root* by a node of size 1, add to it a parent of size $M/2$ which is the left child of the new root; the right child of the new root is then a chain containing a leaf of size $M-1$ and its parent of size $M/2$. Doing this repeatedly until n nodes are used gives the lower bound of $\Omega(nM)$. Therefore, PostOrderMinIO is not constant-factor competitive.



(a) Example of a tree showing that PostOrderMinIO is not an approximation algorithm.



(b) Example of a tree where OptMinMem is not optimal for MinIO ($M=6$).



(c) Example of a tree showing that OptMinMem is not an approximation algorithm ($M=4k$).

Figure 5.2: The black label inside node i represents m_i. The red label next to the nodes indicates in (a) the optimal schedule, and in (b) and (c) the OptMinMem schedule.

5.3.4 OptMinMem is not competitive

Minimizing the amount of I/O in an out-of-core execution seems similar to minimizing the peak memory when the memory in unbounded. Thus, in order to derive a good solution for MinIO, it seems reasonable to use an optimal algorithm for MinMem, such as the OptMinMem algorithm presented by Liu [105], to compute a schedule σ and then to perform I/Os using the FiF policy. In the following, we also use OptMinMem to denote this strategy for MinIO. We prove here that there exist problem instances on which this strategy will also perform arbitrarily more I/Os than the optimal traversal.

We first exhibit in Figure 5.2(b) a tree showing that OptMinMem does not always lead to minimum I/Os in our model. Let $M=6$. The tree of Figure 5.2(b) can be completed with 3 I/Os, by doing one chain after the other. This corresponds to a peak memory of 9. But OptMinMem achieves a peak memory of 8 at the cost of 4 I/Os by executing the nodes in increasing order of the labels next to the nodes.

This example can be extended to show that OptMinMem may perform arbitrarily more I/Os than the optimal strategy. The extended tree is illustrated on Figure 5.2(c). It contains two identical chains of length $2k+2$, for a given parameter k, and the memory size is set to $4k$. The weights of the tasks in each chain (in order from root to leaf) are defined by interleaving two sequences: $\{2k, 2k-1, \ldots, k\}$ and $\{3k, 3k+1, \ldots, 4k\}$. As above, it is possible to schedule this tree with only $2k$ I/Os, but with a memory peak of $6k$, by computing one entire chain, then the other. However, OptMinMem achieves a memory

peak of $5k$ by alternating between chains, each time processing the chain until reaching a node with a weight smaller than $2k$, as represented by the labels besides the nodes. OPTMINMEM performs k I/Os on each of the $k+1$ smallest nodes, leading to a cost of $k(k+1)$ I/Os. The competitive ratio is then larger than $k/2$, and OPTMINMEM is not constant-factor competitive for the MINIO problem.

5.3.5 Unknown complexity

NP-hardness conjecture

As shown above, polynomial-time approaches based on similar problems fail to even give a constant-competitive ratio. The main issue facing a polynomial approach is the highly nonlocal aspect of the optimal solution. For example, since postorder traversals are not optimal, it may be highly useful to stop at intermediate points of a subtree's execution in order to process entirely different subtrees.

We conjecture that this problem is NP-hard due to these difficult dependences. As mentioned above, if we require entire data files to be written to disk, the problem has been shown to be NP-hard by reduction to Partition [88]. However, this proof highly depends on indivisible data, rather than on the recursive structure of trees. Taking advantage of the structure of our problem to give an NP-hardness result could lead to an interesting understanding of optimal solutions, and possibly further heuristics. We leave this as an open problem. We nevertheless prove in this section that schedules respecting an intuitively reasonable property can lead to an I/O cost far from the optimal.

Lowcut schedules are not constant-factor approximation

In this section, **we slightly change the model** in order to simplify the explanations: we consider that the execution of a tree starts from the root, and is terminated at the leaves. Therefore, each node can be executed as soon as its parent is terminated. The weight m_i of node i corresponds to the size of its *input* file (instead of its *output* file). The (unique) input file of a node must be in memory at the start of its execution, and its output files must be in memory at the end of its execution. Note that this model is equivalent to the original one, as we can transform a schedule from one model to the other by reversing the *arrow of time*. The tasks scheduled first in one model are then scheduled last in the other; the number of I/Os required is not modified by this transformation. Before explaining what we call a lowcut schedule, we need some definitions.

The ***amplification*** of a node i in a schedule $\mathscr{S}$ is a maximal series of computations on descendants of i. In other words, let $u_1, u_2, \dots, u_k, u_{k+1}$ be the nodes executed right after i in $\mathscr{S}$, such that u_{k+1} is the first one not being a descendant of i (thus $k \geq 0$). Let C be the set of nodes $\{u \mid \exists j \in [1,k],\ u \text{ is a son of } u_j\} \setminus \{u_1 \dots u_k\}$. Then we say that i is *amplified* to the cut C in $\mathscr{S}$, as C is the set of available descendants of i after the execution of u_k. Note that C can be empty if all the descendants of i are executed.

Consider a node i of weight m_i and let $Tr(i)$ be the subtree rooted at i. We define a ***cut*** of $Tr(i)$ as a set of nodes of $Tr(i)$ where no node is a descendant of another one. Note that such a cut is a set of nodes and not edges, and that it may contain only leaves. Then, a cut C of G is a ***low cut*** if there exists a node i such that C is a cut of $Tr(i)$ where $\sum_{j \in C} m_j \leq m_i$. In other words, amplifying node i to the cut C leads to a smaller memory footprint. Note that for each i, there always exists a cut of $Tr(i)$ which is a low cut: the empty set, which corresponds to executing the whole subtree $Tr(i)$, is obviously a low cut. A cut that is not low is called ***high***.

We define ***lowcut schedules*** as schedules that amplify each node to a low cut. As a side note, there always exists a schedule minimizing the peak memory that is actually a lowcut schedule: the algorithms of [88, 105] solving the memory minimization problem return lowcut schedules. Intuitively, one can be

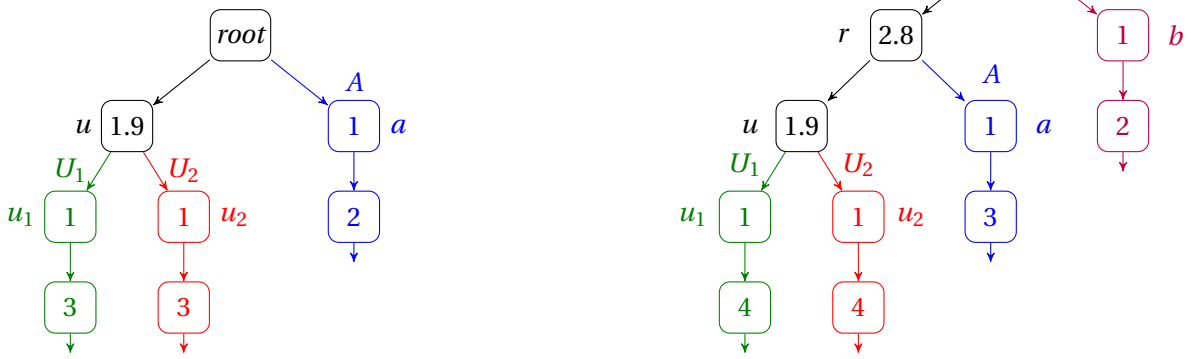
tempted to think that forbidding to amplify a node to a high cut should not harm the I/O performance very much. Indeed, it seems peculiar to execute a subtree, stop at a point where we need to store more data than before, and process another part of the workflow. Why not processing this part before, when more memory was available? Actually, it is possible that some I/Os are *enforced* by the structure of the tree, and so every schedule puts on disk some part of a given high cut C. Then, amplifying to this high cut C from a node that had a slightly smaller weight might actually lead to a smaller memory footprint after performing the unavoidable I/Os. This gain of memory can then lead to a schedule with arbitrarily less I/Os, see Lemma 5.6.

We provide below two examples which give insights on the problem, then prove in Lemma 5.6 that an algorithm returning only lowcut schedules cannot be a constant-factor approximation.

First, Remark 5.1 shows on a small example that the best lowcut schedule is not always optimal and Remark 5.2 shows on a slightly more complex tree that the best lowcut schedule can lead to twice as many I/Os as the optimal solution.

Remark 5.1. *On some trees, no optimal schedule is a lowcut schedule.*

Proof. The targeted tree is depicted on Figure 5.3(a), where the weight m_i of each node is written inside it. The subtrees rooted at nodes a, u_1, and u_2 are respectively denoted as A, U_1, and U_2. The available memory is equal to $M = 3$. Note that the subtrees U_1 and U_2 are identical, so we only consider without loss of generality schedules that compute node u_1 before u_2. After the completion of U_1, the main memory is empty, so we furthermore consider only schedules that compute U_2 straight after U_1.



(a) Tree to schedule with a memory $M = 3$

(b) Tree to schedule with a memory $M = 4$

Figure 5.3: Instances on which no lowcut schedule is optimal. The computation starts at the root and is terminated at the leaves.

Table 5.1 lists an optimal schedule and all lowcut schedules. Before computing any node except the root, the input files of nodes a and u are in main memory, which occupies 2.9 out of 3 units of memory. An optimal solution consists in scheduling u first to get to the (high) cut $\{u_1, u_2\}$ which entirely fills the memory, then performing one I/O on node u_2. Performing an I/O on this high cut is unavoidable because given the trees U_1 and U_2, any schedule must perform an I/O on u_2. Then, the freed memory slot allows to schedule A, and only one memory unit remains occupied, for the input file of u_1. Finally, U_1 and U_2 are scheduled without I/O. On the opposite, lowcut schedules either execute A first or A last, and therefore perform unnecessary I/Os on u or a. □

Class of schedule	I/O cost	Schedule	I/Os on node		
			u	u_2	a
Optimal	1	u, A, U_1, U_2		1	
Lowcut	1.9	A, u, U_1, U_2	0.9	1	
	2	u, U_1, U_2, A		1	1

Table 5.1: Optimal and lowcut schedules of the tree of Figure 5.3(a).

Remark 5.2. *There exists a tree in which no lowcut schedule is a 2-approximation.*

Proof. The targeted tree is depicted in Figure 5.3(b). The structure of this example is similar to the one of Remark 5.1, where we add a root over the tree and another branch B in parallel, and the memory weights are adapted. We have a total memory equal to $M = 4$. As previously, we consider only schedules where the node u_1 is computed before u_2.

In this tree, the optimal schedule first computes r and u. At this point, the memory is full and only 1 I/O on u_2 is needed to complete the processing of the whole tree. Indeed, after this I/O, B can be completed, which frees 1 more units of memory. Then, 2 units of memory are available, so A can be completed. Finally, only u_1 is in memory so both U_1 and U_2 can be completed.

Class of schedule	I/O cost	Schedule	I/Os on node				
			r	u	u_2	a	b
Optimal	1	u, r, B, A, U_1, U_2			1		
Lowcut	2.7	$B, A, Tr(u)$	0.8	0.9	1		
	2.8	$B, Tr(u), A$	0.8		1	1	
	2.9	$A, B, Tr(u)$		1.9	1		
	2.9	$A, Tr(u), B$		0.9	1		1
	3	$Tr(u), A, B$			1	1	1
	3	$Tr(u), B, A$			1	1	1

Table 5.2: Optimal and lowcut schedules of the tree of Figure 5.3(b).

As in Remark 5.1, without loss of generality, we only consider schedules that compute U_2 straight after U_1. Given the configuration of the tree, the lowcut schedules *must* compute the trees A, B and $Tr(u) = \{u, U_1, U_2\}$ contiguously. Moreover, a lowcut schedule cannot compute A or B directly after u nor B directly after r. We present the six lowcut schedules in Table 5.2. Therefore, any lowcut schedule performs at least 2.7 I/Os, when the optimal is 1, which proves the remark. □

Lemma 5.6. *For any integer k, there exists a tree containing $O(k)$ nodes for which no lowcut schedule is a k-approximation.*

Proof. Let $k \in \mathbb{N}$ and $\varepsilon < 1/k^2$. We consider the tree depicted in Figure 5.4, which has a structure similar to the one of Figure 5.3(b), with k branches. The size of the main memory is equal to $M = k + 2$.

Following the same idea as in Remark 5.2, an optimal schedule will perform only one I/O on node u_2, by computing first nodes $r_k, r_{k-1}, \ldots, r_1$. The memory used before this I/O always lies between $k + 2 - \frac{1}{k}$ and $k + 2$. Performing this I/O frees one slot of memory, which allows computing the branch whose leaf has a weight 2, then the one whose leaf has a weight 3, etc., and then compute the whole tree

without additional I/O. Similarly to Remark 5.2, the best lowcut schedule will perform $1 - i\varepsilon$ I/Os on each node r_i, plus one I/O on u_2, so will perform at least $k+1-\sum_{i=1}^{k} i\varepsilon \geq k+1-k^2\varepsilon > k$ I/Os, which completes the lemma. □

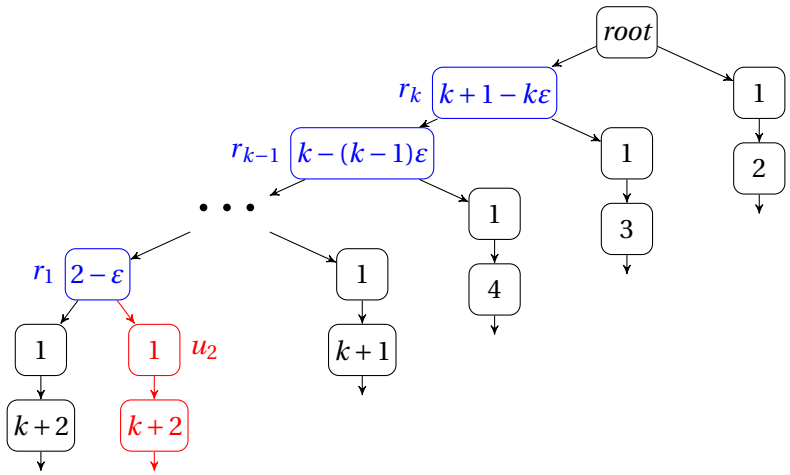


Figure 5.4: Tree on which all lowcut schedules perform k times more I/Os than the optimal, with $M = k+2$. The computation starts at the root and is terminated at the leaves.

Although the class of lowcut schedules contains schedules that minimize the peak memory, the I/O minimization problem therefore cannot be approximated by a constant factor through a restriction to this class of schedules.

5.4 ILP formulation of the problem

We now present an Integer Linear Program solving the MinIO problem.

The linear program relies on the boolean variables δ_{ij} to express the schedule constraints. δ_{ij} is equal to 1 if node i precedes node j in the corresponding schedule and 0 otherwise, as used previously in [48] for instance. All the variables considered in this linear program are nonnegative. The first constraints represent the antisymmetric (Equation 5.2), acyclic (Equation 5.3) and reflexive (Equation 5.4) properties of the order, and the consistence with the precedence constraints (Equation 5.5).

$$\forall i,j \in G,\ \delta_{ij} + \delta_{ji} = 1 \tag{5.2}$$
$$\forall i,j,k \in G,\ \delta_{ij} + \delta_{jk} + \delta_{ki} \geq 1 \tag{5.3}$$
$$\forall i \in G,\ \delta_{ii} = 1 \tag{5.4}$$
$$\forall (i,j) \in E,\ \delta_{ij} = 1 \tag{5.5}$$
$$\forall i,j \in G,\ \delta_{ij} \in \{0,1\} \tag{5.6}$$

We introduce the variable $\alpha_i \in [0,1]$ which represents the fraction of node i written to disk.

$$\forall i \in G,\ 0 \leq \alpha_i \leq 1 \tag{5.7}$$

The memory constraint is then equivalent to the following nonlinear inequality (that will be linearized ultimately). Indeed, Equation 5.8 is the transposition of the memory constraint defined as Equation 5.1 in Section 5.2.1, noting that $\delta_{ki}\delta_{ip} = 1$ if and only if node k is active during the execution of node i.

$$\forall i \in G, \quad \max\left(w_i, \sum_{(j,i)\in E} w_j\right) + \sum_{\substack{(k,p)\in E,\\ k\neq i, p\neq i}} \delta_{ki}\delta_{ip}(1-\alpha_k)w_k \leq M \tag{5.8}$$

Finally, the objective function is to minimize the I/O cost:

$$\text{Minimize} \sum_{i\in G} \alpha_i w_i \tag{5.9}$$

We have now formalized the MinIO problem described in Section 5.2.1 through a quadratic program $\mathscr{P}_{quad}$ composed of Equations 5.2 to 5.8. It then remains to linearize Equation 5.8.

We define the variables x_{ik} and y_{ik}. They are constrained to satisfy the following: if node k is active during the execution of node i, then x_{ik} is equal to 1 and y_{ik} is not larger than α_k; otherwise they are both null. Note that in the special case when i is either k or its parent, both variables are forced to be 0.

$$\forall (k,p) \in E, i \notin \{k,p\}, \quad x_{ik} = \delta_{ki} + \delta_{ip} - 1 \tag{5.10}$$

$$\text{and} \quad 0 \leq y_{ik} \leq \min\left(\delta_{ki}, \delta_{ip}, \alpha_k\right) \tag{5.11}$$

$$\forall (k,p) \in E, \quad x_{pk} = y_{pk} = x_{kk} = y_{kk} = 0 \tag{5.12}$$

$$\forall i \in G, \quad \max\left(w_i, \sum_{(j,i)\in E} w_j\right) + \sum_{(k,p)\in E} x_{ik}w_k - \sum_{(k,p)\in E} y_{ik}w_k \leq M \tag{5.13}$$

The final integer linear program $\mathscr{P}_{lin}$ is then composed of Equations 5.2 to 5.7 and 5.10 to 5.13, with the objective function described by Equation 5.9. It requires $O(n^2)$ variables and $O(n^3)$ constraints. We now prove that the linearization is correct: for any value X of the objective function, there exists a solution to $\mathscr{P}_{lin}$ of objective value X if and only if there exists a solution to $\mathscr{P}_{quad}$ of objective value X.

First, as an intermediate step, we show that if there exists a valuation of variables that satisfies Equations 5.2 to 5.7 and 5.10 to 5.12, then for $(k,p) \in E$ and $i \in G$ with $i \notin \{k,p\}$, we have $x_{ik} - y_{ik} \geq \delta_{ki}\delta_{ip}(1-\alpha_k)$. We consider such a valuation of the variables. By the precedence constraint (5.5), we have $\delta_{kp} = 1$. Hence, thanks to the antisymmetric (5.2) and acyclic (5.3) constraints, we deduce that we cannot have both $\delta_{ki} = 0$ and $\delta_{ip} = 0$. Therefore thanks to Equation 5.10, we have $x_{ik} = \delta_{ki}\delta_{ip}$. From Equation 5.11, we deduce $y_{ik} \leq \alpha_k$ and $y_{ik} = 0$ if $\delta_{ki}\delta_{ip} = 0$. Thus, we have $y_{ik} \leq \delta_{ki}\delta_{ip}\alpha_k$ and finally $x_{ik} - y_{ik} \geq \delta_{ki}\delta_{ip}(1-\alpha_k)$.

Assume that $\mathscr{P}_{lin}$ allows a feasible valuation of variables $\mathcal{V}$. $\mathcal{V}$ respects the conditions of the above paragraph, so that $x_{ik} - y_{ik} \geq \delta_{ki}\delta_{ip}(1-\alpha_k)$. Therefore, the left hand side of Equation 5.13 is not smaller than the left hand side of Equation 5.8. As Equation 5.13 is satisfied as part of $\mathscr{P}_{lin}$, Equation 5.8 is satisfied. $\mathcal{V}$ (restricted to the δ and α variables) is then a solution of $\mathscr{P}_{quad}$.

On the contrary, let us assume now that $\mathscr{P}_{quad}$ allows a feasible valuation of variables $\mathcal{V}$. Then, we complete $\mathcal{V}$ by setting, for $(k,p) \in E$ and $i \in G$ with $i \notin \{k,p\}$, $x_{ik} = \delta_{ki}\delta_{ip}$, $y_{ik} = \delta_{ki}\delta_{ip}\alpha_k$, and for $i \in \{k,p\}$, $x_{ik} = y_{ik} = 0$. Let $\mathcal{V}'$ be the completed valuation. In $\mathcal{V}'$, Equation 5.13 is then equivalent to Equation 5.8, which is thus also satisfied. We now show that $\mathcal{V}'$ satisfies Equations 5.10 and 5.11. Let $(k,p) \in E$ and $i \in G$ with $i \notin \{k,p\}$. By the precedence constraint (5.5), we have $\delta_{kp} = 1$. Hence, thanks to the antisymmetric (5.2) and acyclic (5.3) constraints, we deduce that we cannot have both $\delta_{ki} = 0$ and

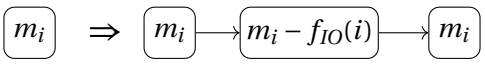


Figure 5.5: Example of node expansion.

$\delta_{ip} = 0$. Therefore, $x_{ik} = \delta_{ki}\delta_{ip} = \delta_{ki} + \delta_{ip} - 1$, so Equation 5.10 is satisfied. Then, $y_{ik} = \delta_{ki}\delta_{ip}\alpha_k$ is equal to 0 if δ_{ki} or δ_{ip} is null, and to α_k otherwise, so Equation 5.11 is satisfied. Therefore, $\mathcal{V}'$ is a solution of $\mathscr{P}_{lin}$, achieving the same objective value as $\mathcal{V}$ in $\mathscr{P}_{quad}$.

5.5 Heuristic

We now move to the design of a novel heuristic, FullRecExpand, whose goal is to improve the performance of OptMinMem for the MinIO problem. The main idea of this heuristic is to run OptMinMem several times: when we detect that some I/O is needed on some node, we force this I/O by transforming the tree. This way, the following iterations of OptMinMem will benefit from the knowledge of this I/O. We continue transforming the tree until no more I/Os are necessary.

In order to enforce I/Os, we use the technique of *expanding* a node (illustrated on Figure 5.5). Under an I/O function f_{IO}, we define the ***expansion*** of a node i as the substitution of this node by a chain of three nodes i_1, i_2, i_3 of respective weights m_i, $m_i - f_{IO}(i)$ and m_i. The expansion of a node actually mimics the action of executing I/Os; the weight of the three tasks represent which amount of main memory is occupied by this node:

1. when it is first completed ($m_{i_1} = m_i$),
2. when part of it is moved to disk ($m_{i_2} = m_i - f_{IO}(i)$),
3. when the whole data is transferred back to main memory ($m_{i_3} = m_i$).

This technique first allows us to prove Theorem 5.2, which states that given an I/O function f_{IO}, we can find a schedule σ such that (σ, f_{IO}) is a valid traversal if there exists one.

Proof of Theorem 5.2. Consider the tree G' obtained from G by expanding all the nodes for which f_{IO} is not null. Then, consider the schedule σ' obtained by OptMinMem on G', and let σ be the corresponding schedule on G. Then, the memory used by σ on G during the execution of a node i is the same as the one used by σ' on G' during the execution of the same node i, or of i_1 if i is expanded. Then, as OptMinMem achieves the optimal memory peak on G', we know that σ uses as little main memory as possible under the I/O function f_{IO}. Then, (σ, τ) is a valid traversal of G. □

The heuristic FullRecExpand is described in Algorithm 14. The main idea of the heuristic is to expand nodes in order to obtain a tree that can be scheduled without any I/O, which is equivalent to building an I/O function.

First, the heuristic recursively calls itself on the subtrees rooted at the children of the root, so that each subtree can be scheduled without any I/O (but using expansions). Then, the algorithm computes OptMinMem on this new tree, and if I/Os are necessary, it determines which node should be expanded next. This selection is the only part where FullRecExpand can deviate from an optimal strategy. Our choice is to select a node on which the FiF policy would perform I/Os; if there are several such nodes, we choose the one whose parent is scheduled the latest. After the expansion, the algorithm recomputes OptMinMem on the modified tree, and proceeds until no more I/Os are necessary.

At the end of the computation, the returned schedule is obtained by running OptMinMem on the final tree computed by FullRecExpand, and by transposing it on the original tree. The I/O performance of this schedule is then equal to the sum of the expansions.

Algorithm 14: FULLRECEXPAND (G, r, M)

Input: tree G, root of exploration r
Output: Return a tree G_r which can be executed without any I/O, obtained from G by expanding several nodes

```
foreach child i of r do
    G_i ← FULLRECEXPAND(G, i, M)
G_r ← tree formed by the root r and the G_i subtrees
while OPTMINMEM(G_r, r) needs more than a memory M do
    f_IO ← I/O function obtained from OPTMINMEM(G_r, r) using the FiF policy
    i ← node for which f_IO(i) > 0 whose parent is scheduled the latest in OPTMINMEM(G_r, r)
    modify G_r by expanding node i according to f_IO(i)
return G_r
```

FULLRECEXPAND is only a heuristic: it may give suboptimal results but also may achieve better performance than OPTMINMEM.

We illustrate the behavior of FULLRECEXPAND on two small examples. The left-hand side of Figure 5.6 provides an example where FULLRECEXPAND performs better than OPTMINMEM. OPTMINMEM computes the left branch first until node a, then the right branch until node b, before completing the left branch. The memory peak reached is 12, but this schedule incurs 4 I/Os with a memory limit of 10: 2 on node a and 2 on node b. On this example, FULLRECEXPAND expands node b as specified on the middle diagram. With this expansion, OPTMINMEM schedules the right branch until b_2 first, then the whole left branch, using one more I/O on b_2. This node is expanded a second time on the right diagram, without changing the schedule obtained by OPTMINMEM, yielding to 3 I/Os on the original tree, all on b. On this instance, FULLRECEXPAND therefore performs 3 I/Os whereas OPTMINMEM and POSTORDERMINIO perform 4 I/Os.

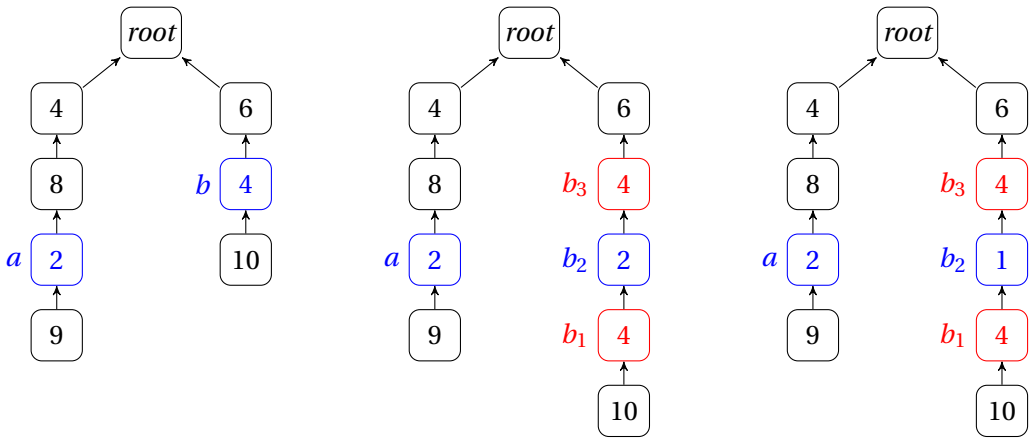


Figure 5.6: Example on which FULLRECEXPAND is optimal whereas OPTMINMEM and POSTORDERMINIO are not, with $M = 10$. The left tree is the original one, and the others are obtained during the execution of FULLRECEXPAND after the expansion of b then b_2.

Figure 5.7 provides an example where FULLRECEXPAND does not improve OPTMINMEM. On this instance, OPTMINMEM performs 4 I/Os, 2 on node a then 2 on node b, where POSTORDERMINIO executes first the left subtree and consumes only 3 I/Os on node c. This instance shows an example

where no optimal solution performs an I/O on a node where OPTMINMEM performs an I/O. Therefore, the strategy of FULLRECEXPAND cannot be optimal, even if we used a different priority at Line 6.

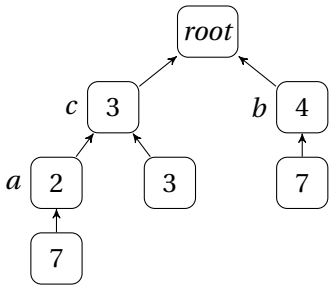


Figure 5.7: Example on which FULLRECEXPAND and OPTMINMEM perform 4 I/Os whereas POSTORDERMINIO performs 3 I/Os, with $M = 7$.

Unfortunately, the complexity of FULLRECEXPAND is not polynomial, as the number of iterations of the while loop at Line 4 cannot be bounded by the number of nodes, but may depend also on their weights. We therefore propose a simpler variant, named RECEXPAND, where the while loop at Line 4 is exited after 2 iterations. In this variant, the resulting tree G might need I/Os to be executed. The final schedule is computed as in FULLRECEXPAND, by running OPTMINMEM on this tree G. We show in the next section that this variant gives results which are very similar to the original version.

5.6 Numerical results

In this section, we compare the performance of the two existing strategies, OPTMINMEM and POSTORDERMINIO, and the two proposed heuristics, FULLRECEXPAND and RECEXPAND. All algorithms are compared through simulations on two datasets described below. Because of its high computational complexity, FULLRECEXPAND is only tested on the first smaller dataset.

5.6.1 Datasets

The first dataset, named SYNTH, is composed of 330 instances of synthetic binary trees of 3000 nodes, generated uniformly at random among all binary trees. As we considered small trees, we used half-Catalan numbers in order to draw a tree, similarly to the method described at the beginning of [108]. The memory weight of each task is uniformly drawn from $[1;100]$.

The second dataset, named SMALLSYNTH, is composed of 30,000 synthetic binary trees of 30 nodes, generated with the same method as the trees of SYNTH. This dataset contains trees small enough to allow the determination of the optimal solution by solving the ILP directly.

The last dataset, named TREES, is composed of 329 elimination trees of actual sparse matrices from the University of Florida Sparse Matrix Collection [52] already used in Chapter 1 (see [62] for more details on elimination trees and the data set). Our dataset corresponds to the 329 smallest of the 640 trees presented in [62], with trees ranging from 2000 to 40000 nodes.

For each tree of each dataset, we first computed the minimal memory size necessary to process the tree nodes: $LB = \max_i \bar{m}_i$. We also computed the minimal peak memory for an incore execution $Peak_{incore}$ (using OPTMINMEM). We eliminated all trees from the TREES dataset where $Peak_{incore} = LB$ (i.e., trees for which out-of-core execution is useless whatever the memory bound M), leaving us with 133 remaining trees in this dataset. In all other cases, note that the possible range for the memory bound M such that some I/Os are necessary is $[LB, Peak_{incore} - 1]$. The main memory bound we use

in our simulation is the middle of this interval $M_{mid} = (LB + Peak_{incore} - 1)/2$. For a more complete analysis, we also perform the same simulations with the two extreme memory bounds $M_{min} = LB$ and $M_{max} = Peak_{incore} - 1$.

5.6.2 Results

Our objective in this study is to minimize the total amount of I/Os needed to process the tree. In order to summarize and compare the performance of the different strategies we choose here to consider the number of I/Os and the memory bound M: performing 10 I/Os when the optimal only needs 1 does not have the same significance if the main memory consists of $M = 10$ slots vs. $M = 1000$ slots. Therefore, in this section, if a schedule performs k I/Os, we define its *relative I/O volume* as $(M + k)/M$. Then, a schedule with no I/O operations has a relative I/O volume of 1 while a schedule needing M I/Os has a relative I/O volume of 2.

In order to compare the performance of these algorithms, we use a generic tool called *performance profile* [54], which has already been used in Chapter 2. For a given dataset, we compute the relative I/O volume of each algorithm on each tree and for each memory limit. Then, rather than computing an average above all the cases, a performance profile reports a cumulative distribution function. We define the *deviation* of a heuristic on a given instance as the relative I/O volume of this heuristic divided by the best relative I/O volume achieved for this instance. We then use the deviation to the best heuristic for the datasets SYNTH and TREES, and the deviation to the optimal solution for the dataset SMALLSYNTH, which is computed with the ILP. Given a heuristic and a deviation τ expressed in percentage, we compute the fraction of test cases for which the heuristic has a deviation not larger than τ, and plot these results. Therefore, the higher the curve, the better the method: for instance, for a deviation $\tau = 5\%$, the performance profile shows how often a given method lies within 5% of the smallest relative I/O volume obtained.

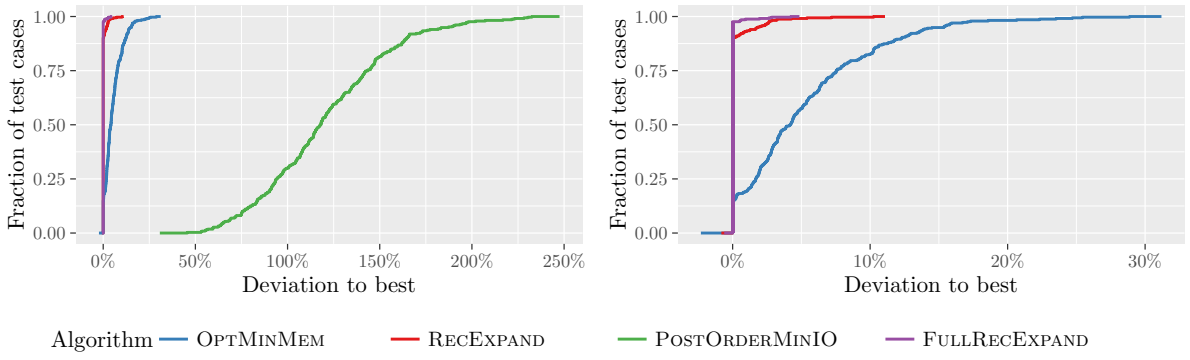


Figure 5.8: Performance profiles of FULLRECEXPAND, RECEXPAND, OPTMINMEM and POSTORDERMINIO on the SYNTH dataset with the M_{mid} memory bound (right: same performance profiles without POSTORDERMINIO).

The left plot of Figure 5.8 presents the performance profile of the four heuristics for the complete dataset SYNTH using the memory bound M_{mid}. The first result is the poor performance of POSTORDERMINIO in this dataset: it almost always has a deviation of at least 50%, and even of 100% in 75% of the cases. Thus, the right plot of the figure presents the performance profiles of exclusively OPTMINMEM, RECEXPAND, and FULLRECEXPAND. RECEXPAND performs far better than OPTMINMEM: it produces strictly less I/Os than OPTMINMEM on 90% of the instances, and on half of them, OPTMINMEM

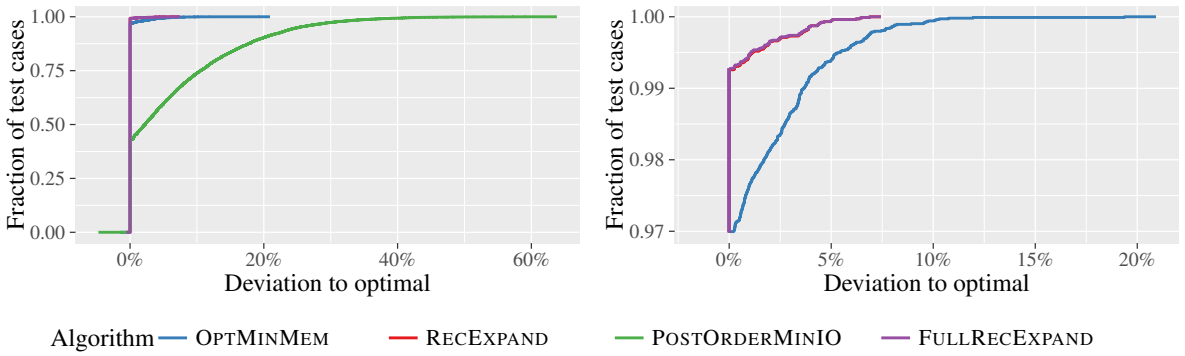


Figure 5.9: Performance profiles for the SmallSynth dataset with the M_{mid} memory bound (right: same performance profiles without PostOrderMinIO; note that we zoom in on the top part of the y-axis to better display instances where the heuristics are non-optimal).

has a deviation of at least 4%. We can also note that FullRecExpand performs only slightly better than RecExpand, but both heuristics are far ahead of OptMinMem, so the gain in the complexity of the algorithm is only balanced by a small loss of performance. For instance, RecExpand has a deviation larger than 2% over FullRecExpand on only 3% of the instances.

We present the performance profiles for the dataset SmallSynth and the memory bound M_{mid} on Figure 5.9. In this figure, the deviation is computed using the optimal solution obtained via the ILP, which allows us to analyze the quality of the solutions returned by FullRecExpand and RecExpand. As the left graph shows, the heuristics observe the same hierarchy as in the Synth dataset with larger trees, but as one could expect, the differences of performance are less significant: PostOrderMinIO has a deviation of less than 10% over the optimal solution in 75% of the instances. OptMinMem is non-optimal in 3% of the instances. FullRecExpand and RecExpand achieve better performance as they are non-optimal in respectively 0.72% and 0.74% on the instances. The performance profile on the subset of trees where at least one of these heuristics is non-optimal is presented on the right graph.

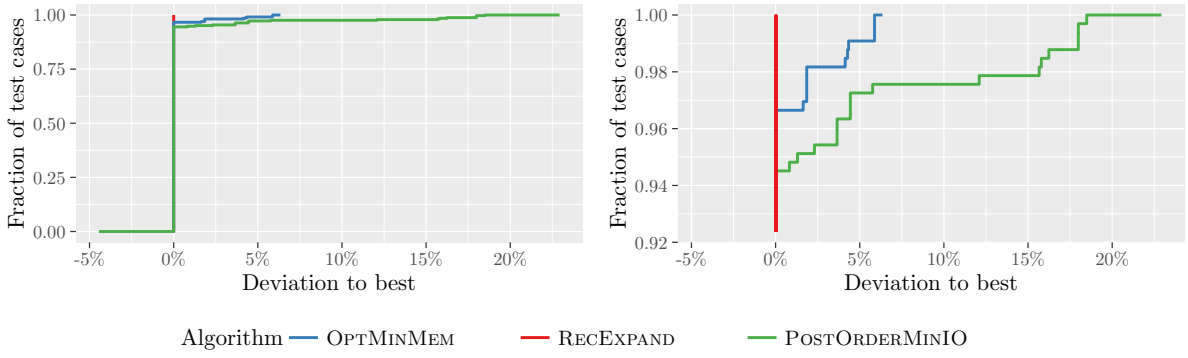


Figure 5.10: Performance profiles for the complete Trees dataset with the M_{mid} memory bound (left) and zoom on the top part corresponding to instances where the heuristic results differ (right).

The left plot of Figure 5.10 presents the performance profiles of the three heuristics PostOrderMinIO, RecExpand and OptMinMem for the complete dataset Trees using the memory bound M_{mid}. The first remark is that the three heuristics are equal on more than 90% of the 329 instances. Therefore, we now focus on the right plot, which presents the top part of the same performance profile,

corresponding to the 25 cases where the heuristics do not all give equal performance. We can see that the hierarchy is the same as in the previous dataset (RECEXPAND is never outperformed, and OPTMINMEM performs better than POSTORDERMINIO) but with smaller discrepancies between the heuristics. We observe a deviation larger than 5% on only 3% of the instances for POSTORDERMINIO and 1% of the instances for OPTMINMEM.

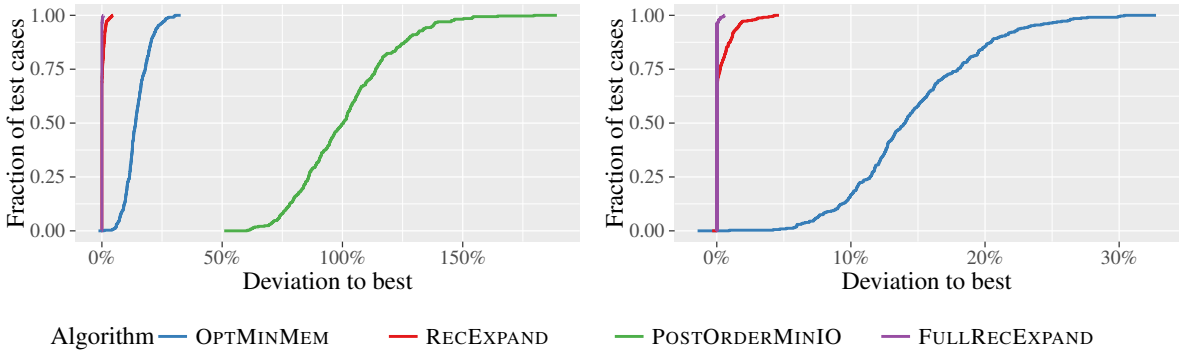


Figure 5.11: Performance profiles of FULLRECEXPAND, RECEXPAND, OPTMINMEM and POSTORDERMINIO on the SYNTH dataset with the M_{min} memory bound (right: same performance profiles without POSTORDERMINIO).

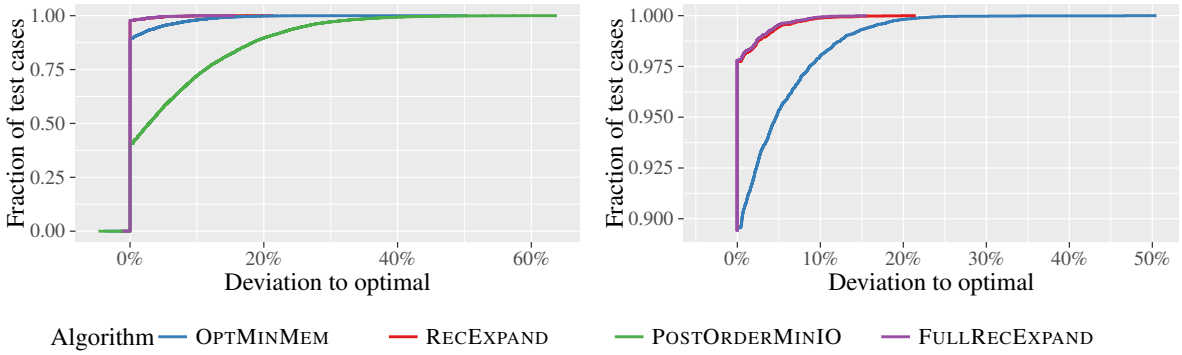


Figure 5.12: Performance profiles for the SMALLSYNTH dataset with the M_{min} memory bound (right: same performance profiles without POSTORDERMINIO, zoom on the top part corresponding to instances where the heuristic results differ).

We now consider the memory bound $M_{min} = LB$, which represents the minimum memory bound for which it is possible to compute a given tree. We plot the corresponding performance profiles for the SYNTH dataset in Figure 5.11, the SMALLSYNTH dataset in Figure 5.12, and the TREES dataset in Figure 5.13. The main conclusion that can be made in comparison to the previous results is that the difference between OPTMINMEM and RECEXPAND is significantly larger with this memory bound. Indeed, in the SYNTH dataset, there is a deviation of 10% for OPTMINMEM in 90% of the cases whereas such a deviation was reached in only 15% of the cases previously. This can be explained by the fact that the memory bound considered here is further from the memory required by MINMEM. On the other hand, the difference between POSTORDERMINIO and RECEXPAND is smaller in this case: there is a deviation of 100% for POSTORDERMINIO in half of the cases whereas we had this property in 75% of the cases with a higher memory bound. The same tendency can be observed for the TREES dataset

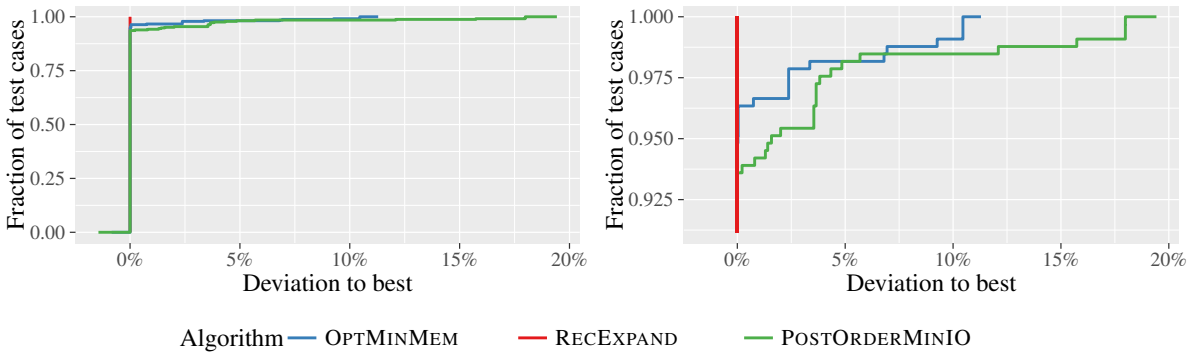


Figure 5.13: Performance profiles for the complete Trees dataset with the M_{min} memory bound (left) and zoom on the top part corresponding to instances where the heuristic results differ (right).

in Figure 5.13, even if it is less significant. For the SmallSynth dataset, the proportion of non-optimal cases is around 3.5 times larger than with the previous memory bound for the three heuristics FullRecExpand, RecExpand, and MinMem, so they are also further from the optimal, but the modification of the memory bound did not significantly modify the behavior of PostOrderMinIO.

For the sake of completeness, we have also considered the memory bound $M_{max} = Peak_{incore} - 1$, which is the opposite case: the largest memory bound for which I/Os are required in order to compute a tree. With this memory bound, OptMinMem, RecExpand, and FullRecExpand are always equal (and even optimal for the SmallSynth dataset), and only PostOrderMinIO achieves worse performance. This can be explained by the fact that M_2 is right below the memory required by OptMinMem to compute a tree without I/Os. Therefore, we can argue that it is closer to the optimal algorithm and FullRecExpand does not improve the few I/Os performed by MinMem. Nevertheless, the deviation of PostOrderMinIO is smaller than with the other memory bounds.

5.7 Conclusion

In this chapter, we revisited the problem of minimizing I/O operations in the out-of-core execution of task trees. We proved that existing solutions allow us to optimally solve the problem when all output data have identical size, but that, in the general case, none of them has a constant competitive factor compared to the optimal solution. In addition to an ILP formulation of the problem, which allows us to compute an optimal solution for small trees, we proposed a novel heuristic solution. Through simulations, we show that this new heuristic is very efficient in practice, achieves better performance than existing solutions, and achieves near optimal performance on small trees. Despite our efforts, the complexity of the problem remains open. Determining this complexity would definitely be a major step, although our findings already lay the basis for more advanced studies. These include moving to parallel out-of-core execution (as was already done for parallel incore execution [62]) as well as designing competitive algorithms for the sequential problem.

Chapter 6

Data structures for external memory

« - La production a augmenté, mais tu as toujours un problème de livraison. Tes circuits de distribution sont à revoir.
- EH ?
- Ah oui, pardon... Toi y en a pas apporter assez de menhirs à la fois. »

Caïus Saugrenus, *Obélix et compagnie*

Nota Bene: This chapter briefly exposes the results obtained during a research visit at the Stony Brook University, NY USA, in the team of Michael Bender. The subject is therefore not related to the title of this manuscript. The full details can be found in the published papers [C3, C5] which are attached in the appendix of this manuscript.

In Chapters 4 and 5, we studied several techniques to cope with a limited main memory while scheduling an application dealing with large data. The same problem arises in many domains where the computations cannot be described by a graph of task. A classic instance is the maintenance of a large database, which can be modeled by the ***dictionary problem***. In this setting, the objective is to maintain a set of elements while supporting four standard operations: insertion of a new element, deletion of an existing element, lookup whether an element exists, and range query (lookup of k consecutive elements). Most data structures used to address this problem (which are then named dictionaries) belong to self-balancing search trees, and perform each operation in a logarithmic number of operations. However, when dealing with a large database, the time spent on memory transfers is far from negligible. These transfers can occur between different parts of the memory hierarchy. As in Chapter 5, we focus on the transfers of blocks between the main memory, named RAM, and a secondary storage, named disk. The same model can be applied to the transfer of cache lines between RAM and cache memory. An I/O operation consists in the transfer of a block of contiguous memory slots from secondary storage to primary storage. The size of a block is set by the system and is therefore a parameter of the problem. Continuing on the dictionary problem, implementing a lookup on a search tree consists in repetitively bringing a block into main memory and performing few comparisons in order to decide which block will be brought in next. Therefore, the computing time is completely negligible compared to the memory transfer delays. This observation is the main motivation of the ***Disk Access Model***, introduced by Aggarwal and Vitter [1] under which we will conduct the studies in this chapter: the complexity of an algorithm is solely defined by the number of I/Os performed.

Main contributions. In this chapter, we first study the I/O complexity of computing prime number tables. Since the sieve of Eratosthenes, most studies have focused on the number of operations and the space usage. We design data structures which dramatically reduce the required I/Os, while still performing few operations. We next focus on history-independent data structures. This property ensures that the current state of the structure reveals no information on past operations. We first design a skip list (a history-independent and simple dictionary) matching the optimal external memory bounds with high probability. The second data structure yields a near-optimal history-independent cache-oblivious B-tree, i.e., one of the standard dictionaries in external memory. This data structure maintains a dynamic set of elements in sorted order in a linear-size array. We design a history-independent version with the same complexity guarantees.

6.1 Introduction to the computational model

As mentioned in the introduction, we use in this chapter the DAM model of Aggarwal and Vitter [1]. We consider two levels of memory: the ***RAM*** of size M and the ***disk*** which is arbitrarily large. A block of $B < M$ contiguous memory slots can be written from disk to RAM (or vice-versa) at the cost of one ***I/O***. The complexity of an algorithm is then equal to the number of I/Os performed.

The parameters M and B are known to the algorithms, which generally helps to decrease the I/O complexity. This model is extended by the ***cache-oblivious*** model, introduced by Frigo et al. [65], in which the parameters M and B are unknown. Remarkably, several problems, which can often be expressed recursively, have asymptotically optimal (and practical) cache-oblivious solutions, including the dictionary problem. Such a solution is then asymptotically optimal for any value of M and B, and therefore for multi-levels memory hierarchies.

The primary indexing data structure used in databases is the ***B-tree***. The B-tree is a search tree in which each node has B elements and thus fits within one I/O, and B children. Self-balancing mechanisms allow to keep the depth of each leaf in $\Theta(\log_B N)$, where N is the number of elements currently present in the data structure. Therefore, the lookups, insertions and deletions cost $O(\log_B N)$ I/Os, and the complexity of a range query of k elements is in $O(\log_B N + k/B)$. A cache-oblivious version of the B-tree has been designed by Bender et al. in [27], and achieves almost the same guarantees. The insertion and deletion costs are instead in $O\big(\log_B N + \log^2 N/B\big)$. Another relevant external-memory result is a sorting algorithm with an I/O complexity of $\textsc{sort}(N) = O\big(\frac{N}{B}\log_{M/B}\frac{N}{B}\big)$ provided by Aggarwal and Vitter [1], which they prove optimal.

6.2 The I/O complexity of computing prime tables

In this study, we revisit the classical problem of computing ***prime tables***: a list of all primes from 2 to N, for any given large N. Such prime-table-computation problems have a rich history, dating back 23 centuries to the sieve of Eratosthenes [82]. Until recently, all efficient prime-table algorithms were ***sieves***, which use a partial (and expanding) list of primes to find and disqualify composites [15, 76]. For example, the sieve of Eratosthenes maintains an array representing $2,\ldots,N$ and works by crossing off all multiples of each prime up to $\sqrt{N}$ starting with 2. The surviving numbers, those that have not been crossed off, comprise the prime numbers up to N. Polynomial-time primality testing [2] makes another approach possible: independently test each $i \in \{2,\ldots,N\}$ for primality. The approaches can be combined; sieving steps can be used to eliminate many candidates cheaply before relatively expensive primality tests are performed. This is a feature of the sieve of Sorenson [132].

Prime-table algorithms are generally compared according to two criteria. One is the standard run-time complexity, that is, the number of RAM operations. However, when computing very large prime tables that do not fit in RAM, such a measure may be a poor predictor of performance. Therefore, there has been a push to reduce the working-set size, that is, the size of memory used other than the output itself [15, 60, 132]. The hope is that if the working-set size gets small enough to fit in memory for larger N, larger prime tables will be efficiently computable. Sieves and primality testing offer a trade-off between the number of operations and the ***working-set*** size of prime-table algorithms. For example, the sieve of Eratosthenes performs $O(N \log\log N)$ operations on a RAM but has a working-set size of $O(N)$, or $O(\sqrt{N})$ for the segmented variant[1]. The fastest primality tests take polylogarithmic time in N, and so run in $O(N\text{polylog}N)$ time for a table but enjoy polylogarithmic working space. This run-time versus working-set-size analysis has lead to a proliferation of prime-table algorithms that are hard to compare.

A small working set does not guarantee a fast algorithm for two reasons. First, even slowly growing working sets may become too big for RAM. But more importantly, even if a working set is small, an algorithm can still be slow if the output table is accessed with little locality of reference.

We have designed data structures based on recent external-memory algorithms (the buffer tree, a B-tree variant [12], and a specific priority queue [13]) for efficient implementation of the sieve of Eratosthenes [82], the linear sieve of Gries and Misra [76] (called GM Linear), the sieve of Atkin [15], and the sieve of Sorenson [132]. Our algorithms work even when $N \gg M$. The complexity analysis consists in the number of I/Os each algorithm induces, in addition to the number of operations. Indeed, contrarily to the DAM model, the number of operations must not be occulted by the I/O complexity: running a simple primality test on all the table leads to the optimal I/O complexity (i.e., $O\left(\frac{N}{B \log N}\right)$, in order to output the table) but with far too many operations.

Table 6.1 summarizes our main results (recall that $\text{SORT}(N) = O\left(\frac{N}{B} \log_{M/B} \frac{N}{B}\right)$). The GM Linear sieve and the sieve of Atkin both slightly outperform the classical sieve of Eratosthenes. The sieve of Sorenson on the other hand induces far fewer I/O operations, but the RAM complexity is dependent on some number-theoretic unknowns, and may be far higher. Its complexity is thus not reported here, see [C5] for details. Note that the working-set sizes are not substantially modified by the new implementations, so the (segmented) sieve of Eratosthenes and the sieve of Atkins use $O(\sqrt{N})$ working space, whereas GM Linear uses $O(N)$ working space. This is consistent with our observation that the working space is not predictive of the I/O complexity of an algorithm.

Sieve	Original complexity	Obtained complexity		Space
	Operations	I/Os	Operations	
Eratosthenes	$N \log\log N$	$\text{SORT}(N)$	$B\text{SORT}(N)$	$\sqrt{N}$
GM Linear	N	$\text{SORT}\left(\frac{N}{\log\log N}\right)$	$B\text{SORT}\left(\frac{N}{\log\log N}\right)$	N
Atkin	$\frac{N}{\log\log N}$	$\text{SORT}\left(\frac{N}{\log\log N}\right)$	$B\text{SORT}\left(\frac{N}{\log\log N}\right)$	$\sqrt{N}$

Table 6.1: Summary of the results for three studied sieves. The obtained (asymptotic) complexities are simplified under the assumption that N is large compared to M and B, see [C5].

[1] We assume that operations are performed on machine words, and that a machine word has $\Omega(\log N)$ bits. Therefore, comparing, multiplying or adding machine words cost $O(1)$, and the main memory contains M machine words.

6.3 History-independent sparse tables and dictionaries

A data structure is ***history-independent*** if its internal representation reveals nothing about the sequence of operations that led to its current state [112, 115]. History independence in a database can have major advantages, depending on the kind of data that is being stored and the security requirements. History-independent data structures naturally support information-theoretically-secure delete. In contrast, with more standard secure delete (where the file system overwrites deleted data with zeros), information about deleted data can leak from the memory representation. For example, it reveals how much data was deleted and where in the keyspace it might have been. In fact, one of the original motivations for history independence comes from data retrieved from public documents in which it was supposed to be erased [79]. However, history independence has rarely been explored in the external-memory model, which is surprising as this model is the most suited for databases.

In this project, we study history independence for persistent, disk-resident dictionary data structures. Specifically, we give a history-independent external-memory skip list and a history-independent cache-oblivious B-tree, which are both history-independent alternatives to the B-tree. One of the main contributions is a data structure we build on the way: a history-independent packed memory array. The ***packed memory array*** (PMA) [27, 29] was actually an unlikely candidate data structure to be made history-independent, since traditional PMAs rely fundamentally on history. The history-independent PMA is one of the primary building blocks in the history-independent cache-oblivious B-tree.

As our data structures are randomized, we do not study their complexity in the worst case, but ***with high probability***. An operation has a complexity in $O(f(N))$ with high probability if for every constant c, there exists a constant d such that the complexity is larger than $d \cdot f(N)$ with probability at most n^{-c}.

6.3.1 External memory skip list

The ***skip list*** is an elegant history-independent dictionary introduced by Pugh in [123], which handles lookups, insertions and deletions in $O(\log N)$ operations, and range queries in $O(\log N + k)$ operations with high probability. A skip list consists in maintaining a sorted list containing all the elements, which allows a fast range query, insertion, and deletion once the pointer to an element is known. In order to improve the lookup complexity, each element gets promoted to a second level with probability $1/2$. These elements form a second list which contains pointers to the previous list, and are again promoted one level higher with probability $1/2$. This process continues until no element remains. With high probability, the skip list obtained contains $\Theta(\log N)$ levels, and any lookup starting at the highest level costs $O(\log N)$ operations.

The objective of this section is to design a history-independent skip list achieving the I/O guarantees of the B-tree, amortized and with high probability. Several studies such as Golovin's in [70] modify the promotion probability to $1/B$, in order to obtain $\Theta(\log_B N)$ levels. The sublists between two promoted elements are stored consecutively so that the lookups cost one I/O per level on average. However, with high probability, there are $\Omega(B \log N)$ consecutive non-promoted elements at the main level, which leads to lookup operations costing $\Omega\left(\log \frac{N}{B}\right)$ I/Os. In order to resolve this issue, our method consists in increasing the promotion probability to $1/B^{0.7}$ for instance (any constant between 0.5 and 1 also works). As this damages the range query complexity, we also group the sublists between two doubly-promoted elements at the main level. Details on how to maintain this structure with the desired complexity while staying history-independent can be found in [C3].

6.3.2 History-independent packed-memory array

One of the classic data-structural problems is called sequential file maintenance: maintain a dynamic set of elements in sorted order in a linear-sized array. If there are N elements, then the array has $\Theta(N)$ empty array positions or *gaps* interspersed among the elements to accommodate future insertions. The gaps allow some elements to shift left or right to open slots for new elements—like shifting books on a bookshelf to make room for new books.

Remarkably, there are data structures for these problems that are efficient even for *adversarial* inserts and deletes. Indeed, the number of element moves per update is only $O(\log^2 N)$ in the worst case [146], which is optimal [38]. In external memory, this data structure is called a packed-memory array [27, 29], as already introduced. It supports inserts, deletes, and range queries. Given the location where we want to insert or delete (which can be found using a separate indexing structure, e.g., [28]), it takes only $O(1 + (\log^2 N)/B)$ amortized I/Os to shift the elements. Given the starting point, a range query costs $O(1 + k/B)$ I/Os (so only $O(1)$ gaps can separate two consecutive elements). The objective of this study is to build a history-independent PMA achieving these bounds with high probability.

Prior PMAs operate as follows. To insert a new element after an existing element or to delete an element, find an enclosing subarray or range, and perform a rebalance. This spreads out the elements (and gaps) within that range. The rebalance range is chosen based upon the density within the range, where ranges have minimum and maximum allowed densities. The larger a range is, the less variability is allowed in its density. The algorithmic subtlety has to do with choosing the right rebalance ranges and the right minimum and maximum density thresholds for each range size.

However, range densities are very history *dependent*. If you repeatedly insert towards the front of an array or if you repeatedly delete from the back of the array, then the front of the array will be denser than the back. The challenge is to make a version of this data structure that is history independent—that is, where newly inserted (or deleted) elements do not seem to increase (or decrease) some local density.

The solution we provide relies on randomness to balance the elements and guarantee the following *with high probability* amortized complexity bounds: $O(\log^2 N)$ operations and $O(1 + (\log^2 N)/B)$ I/Os per update, and $O(1 + k/B)$ additional I/Os for a range query of k elements. Our version of the PMA follows a randomized recursive structure. Schematically, the array is divided in two subarrays, where the split element is randomly chosen among a set of $\Theta(N/\log N)$ elements in the middle of the array. When an insertion or a deletion occurs, this element may change, following a history-independent process. In this case, which is quite rare, the two subarrays are rebuilt from scratch. Each subarray is recursively maintained with the same method.

The cache-oblivious B-tree introduced in [27] heavily relies on a PMA implementation. Replacing the original implementation by the history-independent variant, along with other modifications of the algorithm, we obtain a history-independent cache-oblivious B-tree, matching the bounds exposed in Section 6.1, amortized and with high probability.

6.4 Conclusion

In this chapter, we have designed several data structures in the external memory model. We have proposed an implementation of several sieves which allow to compute large prime tables while performing both few I/Os and few operations. In a second project, we have designed an external-memory skip list with high probability bounds. The main contribution of this chapter may be the history-independent PMA: a data structure maintaining a set of elements in sorted order into a linear-size array. Indeed, the existence of a history-independent implementation of the PMA with the same complexity guarantees seemed unlikely, as the state of the data structure strongly depends on history in the original version.

Conclusion

In this thesis, we have studied three main aspects of task graph scheduling on modern platforms. We have developed models and algorithms allowing the efficient exploitation of task parallelism, especially for linear algebra applications. When several types of processors are available, we have designed guaranteed algorithms for online scheduling. In the context of a limited available memory, we have proposed a method to prevent dynamic schedulers from exceeding the memory limit whenever possible, and have studied the problem of minimizing memory transfers otherwise. The main contributions of each chapter are stated in the following paragraphs.

The speedup model of Prasanna and Musicus for parallel tasks

We have first studied the problem of scheduling parallel task graphs under the speedup model $p \mapsto p^\alpha$, for $0 < \alpha < 1$. This model has been previously introduced and used to represent linear algebra operations, and more specifically the workflow corresponding to the elimination tree arising in multifrontal factorization of sparse matrices. Benchmarks showed that this model is reasonable for some cases but still has a limited accuracy. We proposed a new and simpler proof of the optimal schedule on identical processors, and designed approximation algorithms for two nodes of identical cores.

The two-threshold roofline speedup model for parallel tasks

In order to correct the limitations of the model of the previous chapter, we have designed and studied a more general and accurate speedup model for the same problem. It is composed of three phases. Up to a first threshold the speedup is perfect, equaling the number of processors. Then it grows linearly, but with a slope smaller than one until a second threshold is reached, after which the speedup remains constant. These thresholds depend on the tasks, so that this model leads to a high accuracy for the conducted benchmarks on linear algebra kernels. As minimizing the makespan on identical processors is NP-complete under this model, we studied two algorithms from the literature and proposed a new scheduling policy specifically designed for this model. These algorithms have the same approximation ratio, but our new policy is more competitive on synthetic graphs.

Exploiting hybrid platforms in an online setting

We have then considered online task graph scheduling on a hybrid platform composed of m CPUs and k GPUs. We proved that no online algorithm can have a competitive ratio smaller than $\sqrt{m/k}$, even when some information on the remainder of the graph or additional scheduling power is allowed. We improved an existing online algorithm to obtain a competitive ratio smaller than $2\sqrt{m/k}+1$, and we designed a $O(\sqrt{m/k})$-competitive algorithm which performs well on conducted simulations. Finally, we extended these results on multiple types of processors.

Coping with a limited available memory

The third main theme of this thesis concerns task graph scheduling under a limited memory. In this context, we have designed a method to prevent dynamic runtime schedulers from using too much parallelism and exceeding a given memory limit. This method consists in adding fictitious dependences in order to ensure that any schedule will not exceed the memory limit, while maintaining the critical path as small as possible. The first step was to propose a simple yet powerful model to describe the memory operations. In this framework, we designed an algorithm detecting at which point of the graph the memory may be exceeded, by computing a maximal-weight *topological cut* of a DAG. Then, we proposed an integer linear program and several heuristics to select the new dependences that will be added; the problem of finding the best dependences is NP-hard, even given a known memory-efficient schedule. Simulations on realistic graphs showed that two heuristics present good performance, one of them is better on average but may fail on difficult cases.

Minimizing I/Os when processing a tree

The second studied problem dealing with limited memory is the minimization of I/Os while scheduling a tree. This problem is NP-hard when files cannot be split between the main memory and the secondary storage, but its complexity when tasks can be split remains open. Minimizing the peak memory on trees can be done in polynomial time, but the corresponding schedule may lead to arbitrarily many unnecessary I/Os. We showed that the best *postorder* schedule is optimal when files have a unit size but can also be arbitrary far from the optimal in the general setting. In order to address the general problem, we proposed an integer linear program and a polynomial heuristic, which appeared to be close to the optimal solution in simulations.

Data structures for external memory

In this additional project, we designed several data structures which target external memory efficiency. First, we studied the complexity of computing prime number tables. Since the sieve of Eratosthenes, most studies have focused only on the number of operations and the space usage. We designed data structures which dramatically reduce the required I/Os, while performing few operations. In a second part, we focused on history-independent data structure. We designed a skip list matching the optimal external memory bounds with high probability. We then designed a history-independent packed memory array (a data structure that maintains a dynamic set of elements in sorted order in a linear-size array), which yields a near-optimal history-independent cache-oblivious B-tree.

The work conducted in this thesis can be extended in several directions. We review here some short-term and long-term perspectives.

Short-term perspectives

Pursuit of the theoretical work

In this thesis, we have identified several theoretical questions that remain to be answered. They have been exposed in the appropriate chapters, and many of them consist in improving the approximation ratios of existing algorithms, determining the existence of a guaranteed algorithm where we only proposed a heuristic, or extending the results to a more general framework. In this conclusion, we would like to stress two major unsolved problems:

- *Designing an offline algorithm to schedule task graphs on two types of processors with an approximation ratio at most 6, not relying on linear programming (see Chapter 3).* As suggested by this question, an existing algorithm based on linear program rounding achieves a tight approximation ratio of 6. An algorithm based on *pure* scheduling considerations instead of linear programming would be interesting both theoretically, as it provides more insights on the solution, and practically, as its complexity is generally lower, so is closer to an implementable algorithm. Such an algorithm would be the natural continuity of existing literature on independent tasks, and of the online results we provide.

- *Determining the complexity of the problem of minimizing I/Os while scheduling a tree, when an I/O may contain only a fraction of a file (see Chapter 5).* This result would be the continuity of a branch of theoretical scheduling initiated by Liu back in the 1980s. He proved that determining whether scheduling a tree requires no I/Os is polynomial, whereas this problem is NP-hard on general DAGs. More recently, it has been proved that minimizing I/Os on a tree when files may not be split between the main memory and the secondary storage is NP-hard. Therefore, it remains to determine the complexity of minimizing I/Os on a tree when files may be split. We have proved in this thesis that the optimal solution may not belong to an intuitive class of schedule.

Towards implementable heuristics

As already emphasized in the introduction, most solutions proposed in this thesis are not intended to be implemented as such in an actual scheduler. Indeed, the considered models are often ideal, which is necessary to understand the underlying complexity without coping with multiple parameters. For instance, communication times are neglected as in many theoretical studies. Indeed, it is notorious that adding these constraints complexifies scheduling problems, so ignoring them allows to focus on the specific complexity of the studied problem. In addition, the solution proposed may have an important complexity, although polynomial, as for instance the heuristic proposed in Chapter 4.

A way to derive implementable algorithms from our studies is to lower the expectations in order to decrease the complexity of the algorithms. The algorithm proposed in Chapter 4 adds many fictitious dependences to ensure that no schedule exceeds the memory limit, at the cost of an expensive process. A similar idea could be used to add far fewer dependences, by targeting only *critical* points. The objective would then be to limit the memory consumption without ensuring that no I/O will be performed: it may be preferable to allow some I/Os rather than to have a very complex scheduling process. A second direction consists in taking into account new and important constraints, under a simplified framework. For instance, Chapters 1 and 2 do not include communication times, which are difficult to correctly model

and which prevent low-complexity algorithms to be efficient. A way to take them into account could be to improve the proportional mapping algorithm, which presents both an acceptable theoretical performance and very good locality properties, by performing some heuristic modifications on the allocation. Such a process would lead to both an efficient theoretical schedule and few communications.

Long-term perspectives

The studies conducted in this thesis focused on a shared-memory platform. A natural extension is therefore to model distributed platforms. In such a setting, the computing units are divided into *nodes* which are themselves organized following a hierarchical structure. Processors inside a given node typically share a common memory, so the models considered in this thesis can be applied to a single node. Transferring data between two nodes is highly time-consuming, so such communications are ideally performed only when necessary. Therefore, one of the crucial scheduling problems on distributed architecture is to decide on which node each task should be executed. Similarly to the work conducted in this thesis, the study of scheduling problems on a distributed architecture can be decomposed into several aspects.

First, one objective could be to minimize the theoretical makespan without focusing on memory constraints, as we did in Chapters 1 to 3. Several challenges have to be addressed in this setting. One of the main problems is to decide a static allocation of the tasks to the nodes. Because of the large transfer times, one solution would be to divide the graph relying on clustering algorithms, and then allocate each cluster to a node. These algorithms could benefit from a new paradigm currently developed in the StarPU [17] software: *hierarchical tasks*, also called *bubbles* or *coarse-grain tasks*. Each bubble represents a subgraph, which is revealed to the scheduler only at the execution of this bubble. This concept, which generalizes the parallel tasks considered in Chapters 1 and 2, allows to have benefits from both coarse-grain and medium-grain task scheduling. Indeed, the initial graph has a reasonable size, but the schedule can be finely adapted to the platform once the subgraphs inside each bubble are discovered. The running time of clustering algorithms on this *bubble graph* should not be prohibitive, and each bubble will be allocated on a node. As in this thesis, algorithms to handle parallel tasks and hybrid computing units in this context will need to be developed. In order to correct the computing and communication time estimates, it may be also necessary to resort to dynamic adaptations of the allocation.

The memory usage of the proposed solutions also needs to be optimized. In shared-memory platforms, we proposed in Chapter 4 a solution to prevent dynamic schedulers from running out of memory, while maintaining enough parallelism to avoid idle processors. Such a method would be difficult to apply in a distributed setting, as the memory itself is distributed: if the computations of one node is memory-bound, some workload should be migrated to another node, which cannot be enforced by graph modifications as in the shared-memory case. There is therefore a need for a solution to adapt the allocations computed in the previous context to the available distributed memory.

Appendix A

The I/O complexity of computing prime tables [LATIN 2016 conference]

The I/O Complexity of Computing Prime Tables

Michael A. Bender[1], Rezaul Chowdhury[1], Alex Conway[2],
Martín Farach-Colton[2], Pramod Ganapathi[1], Rob Johnson[1],
Samuel McCauley[1], Bertrand Simon[3], and Shikha Singh[1]

[1] Stony Brook University, Stony Brook, NY 11794-2424, USA.
{bender,rezaul,pganapathi,rob,smccauley,shiksingh}@cs.stonybrook.edu
[2] Rutgers University, Piscataway, NJ 08854, USA.
{farach,alexander.conway}@cs.rutgers.edu
[3] LIP, ENS de Lyon, 46 allée d'Italie, Lyon, France.
bertrand.simon@ens-lyon.fr

Abstract. We revisit classical primes sieves and analyze their performance in the external-memory model. Most prior sieves are analyzed in the RAM model, where the focus is on minimizing both the total number of operations and the size of the working set. One reason for parameterizing by working-set size is that if the working set fits in RAM, then there is a better chance that the sieve has good I/O performance.

We analyze our algorithms directly in terms of I/Os and operations. Unlike in the RAM model, where permutation is trivial, in the external-memory model, permutation can be the most expensive aspect of sieving. We show how to implement classical sieves so that they have both good I/O performance and good RAM performance, even when the problem size N becomes huge—superpolynomially larger than RAM. Towards this goal, we give two I/O-efficient priority queues that are optimized for the number of operations incurred by these sieves.

Keywords: External Memory Algorithms, Prime Tables, Sorting, Priority Queues

1 Introduction

According to Fox News [20], "Prime numbers, which are divisible only by themselves and one, have little mathematical importance. Yet the oddities have long fascinated amateur and professional mathematicians." Indeed, finding prime numbers has been the subject of intensive study for millennia.

Prime-number-computation problems come in many forms, and in this paper we revisit the classical (and Classical) problem of computing prime tables: how efficiently can we compute the table $P[a, b]$ of all primes from a to b and the table $P[N] = P[2, N]$. Such prime-table-computation problems have a rich history, dating back 23 centuries to the sieve of Eratosthenes [17, 27].

Until recently, all efficient prime-table algorithms were ***sieves***, which use a partial (and expanding) list of primes to find and disqualify composites [6, 8, 15, 27]. For example, the sieve of Eratosthenes maintains an array representing $2, \ldots, N$ and works by crossing off all multiples of each prime up to $\sqrt{N}$ starting with 2. The surviving numbers, those that haven't been crossed off, comprise the prime numbers up to N.

Polynomial-time primality testing [2, 18] makes another approach possible: independently test each $i \in \{2, \ldots, N\}$ (or any subrange $\{a, \ldots, b\}$) for primality. Nevertheless, sieving steps can be used to cheaply eliminate many candidates before the relatively

expensive tests are performed, thus improving their performance. This is a feature of the sieve of Sorenson [28] (discussed in Section 5), and can also be used to improve the efficiency of AKS [2] when implemented over a range.

Prime-table algorithms are generally compared according to two criteria [6, 23, 24, 27, 28]. One is the standard run-time complexity, that is, the number of operations such algorithms take in RAM. However, when computing very large prime tables that do not fit in RAM, such a measure may be a poor predictor of performance. Therefore, there has been a push to reduce the working-set size, that is, the size of memory used other than the table itself [6, 12, 28]. The idea is that if the working-set size is smaller, it will fit in memory for larger N, thus allowing larger prime tables to be computed efficiently.

Sieves and primality testing offer a tradeoff between the number of operations and the working-set size of prime-table algorithms. For example, the sieve of Eratosthenes performs $O(N \log\log N)$ operations on a RAM but uses a working space of size $O(N)$. The fastest primality tests take polylogarithmic time in N, and so run in $O(N\mathrm{polylog}N)$ time, but enjoy polylogarithmic working space. Sieves are also less effective at computing $T[a, b]$. For primality-test algorithms, one simply checks the $b - a + 1$ candidate primes, whereas sieves generally require computing many primes smaller than a.

A small working set does not guarantee a fast algorithm for two reasons. First, eventually even slowly growing working sets will be too big for RAM. But more importantly, even if a working set is small, an algorithm can still be slow if the output table is accessed with little locality of reference. This run-time versus working-set-size analysis has lead to a proliferation of prime-table algorithms that are hard to compare.

In this paper, we analyze a variety of algorithms in terms of the number of block transfers they induce, in addition to the number of operations. We use the standard ***disk access machine*** (***DAM***) model [1] (also called the ***external-memory model*** or ***I/O model***). For out-of-core computations, these block transfers are page faults, and for smaller computations, they are cache misses. The DAM model is often more predictive of the efficiency of an algorithm than the size of the working set or of the instruction count, since it directly counts all I/Os, both on the working set and the output array.

Let's begin by analyzing the sieve of Eratosthenes. Each prime is used in turn to eliminate composites, so the ith prime p_i touches all multiples of p_i in the array. If $p_i < B$, every block is touched. As p_i gets larger, every $\lceil p_i/B \rceil$th block is touched. We bound the I/Os by $\sum_{i=2}^{\sqrt{N}} N/(B\lceil p_i/B \rceil) \leq N \log\log N$. In short, this algorithm exhibits essentially no locality of reference and for large N, most instructions induce I/Os.

As a lead-in to our work in Section 2, we can improve the I/O complexity of the sieve of Eratosthenes as follows. Compute the primes up to $\sqrt{N}$ recursively. Then for each prime, make a list of all its multiples. The total number of elements in all lists is $O(N \log\log N)$. Sort, using an I/O-optimal sorting algorithm, and remove duplicates: this is the list of all composites. Take the complement of this list. The total I/O-complexity is dominated by the sorting step, so the time is $O(\frac{N}{B}(\log\log N)(\log_{M/B} \frac{N}{B}))$. Although this is a considerable improvement in the number of I/Os, it represents a slowdown in the number of operations, which increases by a log factor to $O(N \log N \log\log N)$.

In our analysis of the I/O complexity of diverse prime-table algorithms in this paper, one thing becomes clear. All known fast algorithms produce prime numbers, or equivalently composite numbers, out of order. Indeed, it seems to be the careful

representation of integers according to some order other than by value that allows for sublinear sieves.

Consequently, the resulting primes or composites need to be permuted. In RAM, permuting values (or equivalently, sorting small integers) is trivial. In external memory, permuting values is essentially as slow as sorting [1]. Therefore, our results will involve sorting bounds. Until an in-order sieve is produced, all fast external-memory algorithms are likely to involve sorting.

Our main result is a collection of data structures based on buffer trees [3] and external-memory priority queues [3–5] that allow prime tables to be computed quickly, with less computation than sorting implies.

1.1 Background and Related Work

In this section we discuss some previous work on prime sieves. For a more extensive survey on prime sieves, we refer readers to [27].

Much of previous work on sieving has focused on optimizing the sieve of Eratosthenes. Recall that the original sieve uses $O(N)$ working space and performs $O(N \log\log N)$ operations. The notion of chopping up the input into intervals and sieving on each of them, referred to as the ***segmented sieve of Eratosthenes*** [8], is frequently used in practice [6, 10, 12, 26, 27]. It performs the same number of operations as the original but with only $O(\sqrt{N})$ working space. On the other hand, linear variants of the sieve [9, 15, 19, 25] improve the operation count by a $\Theta(\log\log N)$ factor to $O(N)$, but also require a working set of $\Theta(N)$; see Section 3.

Recent advances in sieving use new approaches to achieve better performance. We discuss the sieves of Atkin and Sorenson in Sections 4 and 5.

Alternatively, a primality testing algorithm such as AKS [2] can be used to test the primality of each number directly. Using AKS leads to very small working set size but a large computation cost. On the other hand, the sieve of Sorenson uses a hybrid sieving approach, including elements of both sieving and direct primality testing. This results in polylogarithmic working space, but a larger RAM complexity (see Section 5 for details).

A common technique to increase sieve efficiency is a ***wheel sieve***. A wheel sieve preprocesses a large set of potential primes, quickly eliminating composites with small divisors. Specifically, a wheel sieve begins with a number W, which is a product of the first p primes (for some p). All multiples less than W of the first p primes are marked. Note that if $x < W$ is composite, then $x + W$ is composite as well (since x and W must share a divisor). Thus we iterate through each interval of W consecutive potential primes, marking off certain composites. Since this is just a scan, it takes at most linear work and I/Os, and marks off all composites divisible by one of the first p primes. We will use this technique in Sections 3 and 4. See, for example, [6] for more details.

Previously, Arge and Thorup created a priority queue that is simultaneously efficient in RAM and external memory [5]. We use this data structure as a black box in Sections 2 and 4. Their results also provide an alternative to our priority queue in Section 3.

Specifically, the bounds in Theorem 3 can be achieved by both Arge and Thorup's priority queue, and the priority queue presented in Section 3; however, there are several distinctions. The data structure in [5] requires $M < N/2$ (an upper bound on M) whereas ours requires $\sqrt{M/B} > \log_{M/B} N/B$ (a lower bound on M). Thus, the approaches are complimentary, covering different ranges of M while achieving the same bounds.

Arge and Thorup's priority queue also differs substantially in structure. Their priority queue is based on integer sorting techniques to lower the RAM complexity, whereas ours uses properties of our specific sequence of inserts. Thus our priority queue avoids the heavy machinery of integer sorting, but is only applicable in this specific context. It would be interesting to further explore the relationship between these techniques.

1.2 External-Memory Model and Prime Tables

We analyze our sieves using the ***external memory*** or ***disk-access machine*** (***DAM***) model of Aggarwal and Vitter [1]. The DAM model focuses on the block transfers between any two levels of the memory hierarchy. In this paper, we denote the smaller level by ***RAM*** or ***main memory*** and the larger level by ***disk*** or ***external memory***.

We use the RAM model for counting operations. It costs $O(1)$ to compare, multiply, or add machine words. As in the standard RAM, a machine word has $\Theta(\log N)$ bits.

The prime table $P[N]$ is represented as a bit array that is stored on disk and needs to be filled in. We say that $P[i] = 1$ means that i is prime and $P[i] = 0$ means that i is composite. We are interested in values of N, such that $P[N]$ is too large to fit in main memory. Thus, the prime table fills $\Theta(N/\log N)$ words.

RAM is divided into M *words*. Disk is modeled as arbitrarily large. Data is transferred between RAM and Disk in blocks of size B words ($\Theta(B \log N)$ bits). On a DAM (rather than a RAM), performance is measured in terms of block transfers, and computation is modeled as free [1, 29].

In this paper, we are interested in both the I/O complexity $\mathcal{C}_{\mathrm{I/O}}$ and the RAM complexity $\mathcal{C}_{\mathrm{RAM}}$. We indicate an algorithm's performance using the notation $\langle \mathcal{C}_{\mathrm{I/O}},\ \mathcal{C}_{\mathrm{RAM}} \rangle$. For example, the array implementation of the sieve of Eratosthenes can be shown to run in $\langle \Theta(N),\ \Theta(N \log\log N) \rangle$.

If the problem size is large ($N = \Omega(M^2)$), other sieves perform poorly as well. In this case, segmenting the sieve of Eratosthenes does not lead to any improvements, and the sieve of Atkin requires $\langle O(N/\log\log N),\ O(N/\log\log N) \rangle$.

In contrast, a primality-checking sieve based on AKS runs in $\langle \Theta\left(N/(B\log N)\right),\ \Theta(N \log^c N) \rangle$, as long as $M = \Omega\left(\log^c N\right)$, a factor of $B \log N$ better in memory transfers but nearly $\log^c N$ worse in terms of operations.[4]

1.3 Our Contributions

We present data structures for four main sieves with the objective of optimizing both the number of I/Os and the operation count. Our algorithms work even when $M \ll N$. We consider the sieve of Eratosthenes [17], the linear sieve of Eratosthenes [15], the sieve of Atkin [6], and the sieve of Sorenson [28].

We use the notation $\textsc{sort}\,(x) = O(\frac{x}{B} \log_{M/B} \frac{x}{B})$. Thus, the I/O lower bound of permuting x elements can be written as $\min(\textsc{sort}\,(x), x)$ [1].

We summarize our main results below.

1. *Sieve of Eratosthenes.* We show that the standard sieve of Eratosthenes can be implemented to run in $\langle \textsc{sort}\,(N),\ O(N \log_{M/B} N + N \log\log N \log\log M) \rangle$

[4] Here the representation of $P[N]$ matters most, because the I/O complexity depends on the size (and cost to scan) $P[N]$. For most other sieves in this paper, $P[N]$ is represented as a bit array and the I/O-cost to scan $P[N]$ is a lower-order term.

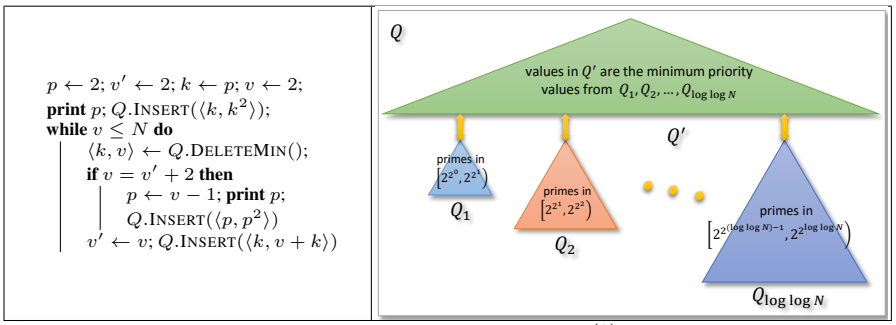


Fig. 1. (a) Original sieve of Eratosthenes using a priority queue, (b) A key-sensitive priority queue.

cost, under the assumption that $M = \Omega\left(B \log^{1+\varepsilon} \log N\right)$ for any constant $\varepsilon > 0$. We achieve these bounds using a new priority queue data structure in which the cost of any operation on an item with key k depends only on k instead of the total number of items in the data structure (as in standard priority queues).

2. *Linear sieve of Eratosthenes.* We implement the linear sieve of Eratosthenes using a buffer-tree-like data structure in $\langle \text{SORT}\,(N/\log\log N)\,,\ O((N/\log\log N)\log_{M/B}(N))\rangle$ under the assumption that $\sqrt{M/B} = \Omega(\max\{\log_{M/B}(\frac{N}{B})\,,\ \log^2_{M/B}(\frac{N}{B})/\log\log N\})$.
3. *Sublinear sieve of Atkin.* We show that the sublinear sieve of Atkin can be implemented using I/O- and RAM- efficient priority queues [5] to run in $\left\langle \text{SORT}\,(N/\log\log N)\,,\ O\left(\left(\frac{N}{\log\log N}\right)(\log_{M/B}(N) + \log\log M)\right)\right\rangle$.
4. *Sieve of Sorenson.* We analyze the sieve of Sorenson in external memory and show that it runs in $\langle O(N/B),\ O(N\pi(p))\rangle$, where $\pi(p)$ denotes the smallest i such that the pseudosquare $L_{p_i} > N/(i\log^2 N)$, where p_i is the ith prime. We also show that given the availability of pseudosquare tables, this sieve can be adapted to sieve the interval $[a, b]$ in $\langle O(1 + (b - a + \pi(p)\log^2 b)/B),\ O((b-a)\pi(p) + \pi(p)\log^2 b)\rangle$.

2 Sieve of Eratosthenes

In this section we show that the sieve of Eratosthenes can be implemented to achieve I/O- and RAM-efficiency simultaneously using sublinear space. We start with a standard priority queue based implementation of the sieve (shown in Figure $1(a)$), and show that by using a new data structure which we call a *key-sensitive priority queue*, we can achieve sorting bound in I/Os without sacrificing RAM performance.

We start by analyzing the performance of the sieve using the recently proposed and only known RAM-efficient external-memory priority queue from [5]. We then observe that the smaller the prime the larger the number of priority queue operations performed on it, and so we can potentially improve the performance of the algorithm by reducing the cost of operations on smaller primes.

Sieve of Eratosthenes using a RAM-efficient external-memory priority queue. The sieve of Eratosthenes can be implemented efficiently using the priority queue of Arge and Thorup as a black box [5]. We describe this in detail in Appendix C.1. This achieves a performance of $\langle \text{SORT}\,(N\log\log N)\,,\ O(N\log\log N(\log_{M/B} N + \log\log M))\rangle$. However, we can shave off the $\log\log N$ factor using a new type of priority queue.

Sieve of Eratosthenes using a key-sensitive priority queue. In a ***key-sensitive priority queue*** the amortized access cost of an operation on an item with key k depends on k instead of the size of the data structure. This property is useful in improving the performance of the folklore priority-queue-based implementation of sieve of Eratosthenes (given in Figure $1(a)$). In that implementation, the number of priority queue operations performed on items with a given prime k as the key varies inversely with k. Thus, a reduction in the cost of operations on smaller primes has the potential of reducing the total cost of all operations. Indeed, we use such a priority queue to achieve sorting bound on I/Os in the sieve of Eratosthenes.

A key-sensitive priority queue Q has two parts—the *top part* consisting of a single internal-memory priority queue Q', and the *bottom part* consisting of $\lceil \log \log N \rceil$ external-memory priority queues $Q_1, Q_2, \ldots, Q_{\lceil \log \log N \rceil}$. Priority queues store $\langle key, value \rangle$ pairs where key is an integer in $[1, N]$ and $value$ is the priority of the item. For our sieving application, key will be a prime in $[1, \sqrt{N}]$ and $value$ will be a multiple of that prime. For any given key there will be at most one $\langle key, value \rangle$ pair in the entire data structure.

Each Q_i in the bottom part of Q is a RAM-efficient external-memory priority queue [5] that stores $\langle k, v \rangle$ pairs such that k is a prime in $[2^{2^i}, 2^{2^{i+1}})$. Hence, Q_i will contain fewer than $N_i = 2^{2^{i+1}}$ items. Then with a cache of size M, Q_i will support insert and delete-min operations in $\langle O((\log_{M/B} N_i)/B),\ O(\log_{M/B} N_i + \log \log M) \rangle$ amortized cost [5]. But note that in Q_i, each key satisfies $\log k = \Theta(\log N_i)$. Thus the cost reduces to $\langle O((\log_{M/B} k)/B),\ O(\log_{M/B} k + \log \log M) \rangle$ for an item with key k. Though we divide the cache equally among all Q_i's, the asymptotic cost per operation remains unchanged assuming $M > B(\log \log N)^{1+\varepsilon}$ for some constant $\varepsilon > 0$.

The queue Q' in the top part will include only the item with the smallest value from each Q_i. So the size of Q' will be $\Theta(\log \log N)$. We use the dynamic integer set data structure from [21] to implement Q' so that insert, delete and delete-min operations on Q' can be supported in $O(1)$ time each using only $O(\log n)$ space (in words). We also maintain an array $A[1 : \lceil \log \log N \rceil]$ such that $A[i]$ stores Q_i's contributed item to Q' so that we can access it constant time.

The priority queue Q only needs to support insert and delete-min operations. To perform an delete-min we extract the smallest item from Q', check its key to find the Q_i it came from, extract the smallest item from that Q_i and insert it into Q'. To insert an item we first check its key to determine its destination Q_i, compare it with the item in $A[i]$, and depending on the result of the comparison we either insert the new item directly into Q_i or move Q_i's current item in Q' to Q_i and insert the new item into Q'. The following lemma summarizes the performance bounds of the operations on Q.

Lemma 1. *Let each Q_i be a RAM-efficient external-memory PQ as described in [5], and let Q' be a priority queue based on the dynamic integer set data structure given in [21]. Then in the resulting data structure, the amortized cost of insert on an item with key k is $\langle O((\log_{M/B} k)/B),\ O(\log_{M/B} k) \rangle$ and delete-min is $\langle O((\log_{M/B} k)/B),\ O(\log_{M/B} k + \log \log M) \rangle$, assuming $M > \log N + B(\log \log N)^{1+\varepsilon}$ for any constant $\varepsilon > 0$.*

We use this key-sensitive priority queue to efficiently implement the sieve of Eratosthenes. The following theorem follows from the observation that a prime p will be

involved in $\Theta(N/p)$ priority queue operations in Q, and because it is known that there are approximately $\sqrt{N}/(\ln(\sqrt{N})-1)$ prime numbers in $[1,\sqrt{N}]$ [7], and the i-th such prime number is approximately $i \ln i$ [16].

Theorem 1. *Using the priority queue from Lemma 1, the sieve of Eratosthenes costs* $\langle \text{SORT}(N),\ O(N(\log_{M/B} N + \log\log M \log\log N))\rangle$ *and uses* $O(\sqrt{N})$ *space, provided* $M > \log N + B(\log\log N)^{1+\varepsilon}$ *for some constant* $\varepsilon > 0$.

3 Linear Sieve of Eratosthenes

There are several variants of the sieve of Eratosthenes [9, 14, 15, 19] that perform $O(N)$ operations by only marking each composite exactly once. See [25] for a survey.

Even though each composite is marked exactly once, resulting in $O(N)$ operations, many of these algorithms have poor data locality. The marking requires large jumps around the array, leading to $O(N)$ I/Os—very poor locality.

In this section, we improve the locality of such linear sieves, while also taking advantage of the bit-complexity of words to improve the performance further. We use a buffer-tree-like data structure (adapted from Arge [3]) to improve the locality, resulting in a cost of $\langle \text{SORT}(N/\log\log N),\ O((N\log_{M/B} N/\log\log N)\rangle$.

We focus on one of the linear variants, the linear sieve algorithm by Gries and Misra [15], henceforth referred to as *the linear sieve of Eratosthenes*.[5] The linear sieve of Eratosthenes is based on the following fundamental property of composite numbers.

Theorem 2 ([15]). *Each composite* C *can be represented uniquely as* $C = p^r q$ *where* p *is the smallest prime factor of* C*, and* p *does not divide* q *(unless* $p = q$*).*

Thus, each composite has a unique normal form based on p, q and r. Crossing off the composites in a lexicographical order based on these (p, q, r) ensures that each composite is marked exactly once. Thus the RAM complexity is $O(N)$.

$\mathcal{C} \leftarrow \{1\};\ p \leftarrow 1;$
while $p \le \sqrt{N}$ **do** $p \leftarrow \texttt{InvSucc}_{\mathcal{C}}(p);\ q \leftarrow p;$
  while $q \le N/p$ **do**
    for $r = 1, 2, \ldots, \log_p(N/q)$ **do**
      $\texttt{InsertIn}_{\mathcal{C}}(p^r q);$
    $q \leftarrow \texttt{InvSucc}_{\mathcal{C}}(q);$
return $[1; N] \setminus \mathcal{C};$

Algorithm 1: Linear SoE

Algorithm 1 describes the linear sieve in terms of subroutines. It builds a set $\mathcal{C}$ of composite numbers, then returns its complement.

The subroutine $\texttt{InsertIn}_{\mathcal{C}}(x)$ inserts x in $\mathcal{C}$. Inverse successor ($\texttt{InvSucc}_{\mathcal{C}}(x)$) returns the smallest element larger than x that is not in $\mathcal{C}$.

While the RAM complexity is an improvement by a factor $\log\log N$ over the classic sieve of Eratosthenes, the algorithm (thematically) performs poorly in the DAM model. The overall complexity of this algorithm is $\langle O(N),\ O(N)\rangle$. In the rest of the section we improve the I/O complexity while maintaining good RAM performance.

Using a buffer-tree-like structure. As a first step, we introduce the classical buffer tree of Arge [3]; we will then modify the structure to improve the bounds of the linear sieve. We give a high-level overview of the data structure here; for details see Appendix A.

The classical buffer tree has branching factor M/B, with a buffer of size M at each node. We assume a complete tree for simplicity, so its height is

[5] Note that the other linear-sieve variants, such as [9, 14, 19] share the same underlying data-structural operations as the sieve of Gries and Misra.

$\lceil \log_{M/B} N/M \rceil = O(\log_{M/B} N/B)$. Newly-inserted elements are placed into the root buffer. If the root buffer is full of M elements, all of its elements are flushed: sorted, and then placed in their respective children; this takes $O(M/B)$ I/Os and $O(M \log M)$ RAM complexity. This process is repeated recursively for any newly-full buffers. Since each element is only flushed to one node at each level, and the amortized cost of a flush is $\langle O(1/B),\ O(\log M)\rangle$, the cost to flush all elements is $\langle O(N/B \log_{M/B} N/B),\ O(N \log N)\rangle$.

Inverse successor can be performed by searching within the tree. However, these searches are very expensive, as we must search every level of the tree—it may be that a recently-inserted element changed the inverse successor. Thus it costs at least $\langle O(M/B \log_{M/B} N/B),\ O(M \log_{M/B} N/B)\rangle$ for a single inverse successor query.

To achieve better bounds, we need to improve the inverse successor time to match the insert time. At the same time, we improve the computation time considerably; we only do $O(B)$ computations per I/O, the best possible for a given I/O bound.

As a first step, we perform a wheel sieve using the primes up to $\sqrt{\log N}$. By an analogue of Merten's Therem, this leaves only $N/\log\log N$ candidate primes. This reduces the number of insertions into the buffer tree.

To avoid the I/Os along the search path for the inverse successor queries, our buffer tree has branching factor $\sqrt{M/B}$ rather than M/B, doubling the height. We partition each buffer into $\sqrt{M/B}$ subarrays of size $\sqrt{MB}$; one for each child. Then as we scan the array, we can store the path from the root to the current leaf in $\sqrt{MB} \log_{M/B} N/B$ words. If $\sqrt{M/B} > \log_{M/B} N/B$ this path fits in memory. Thus the inverse successor queries can avoid the path-searching I/O cost, without affecting the amortized insert cost.

Next, since the elements of the leaves are consecutive integers, each can be encoded using a single bit, rather than an entire word. Since we use the word RAM model (Section 1.2), we can read $\Theta(B \log N)$ of these bits in a single block transfer.

Storing the elements in a bit array could potentially speed up queries, but only if we can guarantee that the inverse successor can always be found by scanning *only* the bit array. During an inverse successor scan, we maintain the path in memory; thus, we can flush all elements along the path without any I/O cost. This guarantees that we can get the correct inverse successor by scanning the array, maintaining the path as we scan.

Finally, since our array is static, we can improve the computation required during a flush. Specifically, since the leaves divide the array evenly, we can calculate the child being flushed to using modular arithmetic (see Appendix A for details).

In total, we insert $N/\log\log N$ elements into the buffer tree. Each must be flushed through $O(\log_{M/B} N/B)$ levels, where a flush costs $O(1/B)$ amortized I/Os and $O(1)$ computation. The inverse successor queries must scan through $N \log\log N$ elements (by the analysis of the sieve of Eratostheses), but due to our bit array representation this only takes $\langle O(N \log\log N/B \log N),\ O(N \log\log N/\log N)\rangle$, a lower-order term.

Theorem 3. *The linear sieve of Eratosthenes implemented using buffer trees, assuming* $M > B^2$, $\sqrt{M/B} > \log_{M/B}(N/B)$, *and* $\sqrt{M/B} > \log^2_{M/B}(N/B)/\log\log N$, *uses* $O(N)$ *space and has a complexity of* $\langle \textsc{sort}\,(N/\log\log N)\,,\ O((N \log_{M/B} N/B)/\log\log N)\rangle$.

4 Sieve of Atkin

The sieve of Atkin [6, 13] is one of the most efficient known sieves in terms of RAM computations. It can compute all the primes up to N in $O(N/\log\log N)$ time using $O(\sqrt{N})$ memory. We first describe the original algorithm from [6] and then use various priority queues, including the key-sensitive priority queue from Section 2, to improve its I/O efficiency.

The algorithm works by exploiting the following characterization of primes using binary quadratic forms. Note that every number that is not trivially composite must satisfy one of the three congruences. For an excellent introduction to the underlying number theoretic concepts, see [11].

Theorem 4 ([6]). *Let k be a square-free integer with $k \equiv 1 \pmod 4$ (resp. $k \equiv 1 \pmod 6$, $k \equiv 11 \pmod{12}$). Then k is prime if and only if the number of solutions to $x^2 + 4y^2 = k$ (resp. $3x^2 + y^2 = k$, $3x^2 - y^2 = k$) is odd.*

For each quadratic form $f(x, y)$, the number of solutions can be computed by brute force, iterating over the set $L = \{(x, y) \mid 0 < f(x, y) \leq N\}$. This requires $O(N)$ memory; however, by "tracing" the level curves of f, this can be reduced to $O(\sqrt{N})$ (see Appendix B). Then, the number of solutions that occur an even number of times are removed. Then by precomputing the primes less than $\sqrt{N}$, the numbers that are not square-free can be sieved out leaving only the primes as a result of Theorem 4.

The algorithm as described above requires $O(N)$ operations, as it must iterate through the entire domain L. This can be made more efficient by first performing a wheel sieve. If we choose $W = 12 \cdot \prod_{p^2 \leq \log N} p$, then by an analog of Merten's theorem, the proportion of (x, y) pairs with $0 \leq x, y < W$ such that $f(x, y)$ is a unit mod W is $1/\log\log N$. By only considering the W-translations of these pairs we obtain $L' \subseteq L$, with $|L'| = O(N/\log\log N)$ and $f(x, y)$ composite on $L \setminus L'$. The algorithm can then proceed as above.

Using priority queues. The above algorithm and its variants require that $M = O(\sqrt{N})$. By utilizing a priority queue to store the multiplicities of the values of f over L, as well as one to implement the square-free sieve, we can trade this memory requirement for I/O operations. In what follows we use an analog of the wheel sieve optimization described above, however we note that the algorithm and analysis can be easily adapted to omit this. See appendix B.3 for a more detailed algorithm description.

Having performed the wheel sieve as described above, we insert the values of each quadratic form f over each domain L into an I/O- and RAM-efficient priority queue Q [5]. This requires $|L|$ such operations (and their subsequent extractions), and so this takes $\langle \textsc{sort}(|L|),\ O(|L| \log_{M/B} |L| + |L| \log\log M/\log\log N)\rangle$. Because we have used a wheel sieve, $|L| = O(N/\log\log N)$, and so this reduces to

$$\left\langle \textsc{sort}\left(\frac{N}{\log\log N}\right),\ O\left(\frac{N \log_{M/B} N}{\log\log N} + \frac{N \log\log M}{\log\log N}\right)\right\rangle . \tag{1}$$

The remaining entries in Q are now either primes or squareful numbers. In order to remove the squareful numbers, we sieve the numbers in Q and for every prime we find, we maintain a record of the multiples of its square. We will track these as pairs $\langle p, v\rangle$ in

another I/O+RAM efficient priority queue Q'. With each value v we pull from Q, we repeatedly extract the min value $\langle p, w\rangle$ from Q' and insert $\langle p, w + p^2\rangle$ until either v is found in which case it is not square-free and thus not a prime, or exceeded, in which case it is prime.

For each prime p less than $\sqrt{N}$ that was not sieved by the wheel, this part of the algorithm will perform $\langle O(N(\log_{M/B} N)/Bp^2),\ O(N(\log_{M/B} N + \log\log M)/p^2)\rangle$ operations. Integrating over p, the total number of operations in this phase of the algorithm is less than $\langle O\left(\text{SORT}\left(N\right)/(B\log N)\right),\ O\left((\text{SORT}\left(N\right) + \log\log M)/\log N\right)\rangle$.

Theorem 5. *The sieve of Atkin implemented with a wheel sieve, as well as I/O and RAM efficient priority queues runs in* $\langle \text{SORT}\left(N/\log\log N\right),\ O((N\log_{M/B} N)/\log\log N + N\log\log M/\log\log N)\rangle$*, using* $O(N)$ *space.*

See Appendix B.1 for a description of how to reduce the space usage to $O(\sqrt{N})$.

5 Sieve of Sorenson

The sieve of Sorenson [28] uses a hybrid approach. It first uses a wheel sieve to remove multiples of small primes. Then, it eliminates nonprimes using a test based on pseudosquares. Finally it removes composite prime powers with another sieve.

The ***pseudosquare*** L_p is the smallest non-square integer with $L_p \equiv 1 \pmod 8$ that is a quadratic residue modulo every odd prime $q \leq p$. The sieve of Sorenson is based around the following lemma—its steps satisfy each requirement of the lemma explicitly. Following the theorem, we set p so that L_p is the smallest pseudosquare satisfying $L_p > N/(\pi(p)\log^2 N)$, and $s = \lfloor N/L_p\rfloor + 1$.

Theorem 6. [28] *Let* x *and* s *be positive integers. If the following hold:* (i) *All prime divisors of* x *exceed* s*,* (ii) $x/s < L_p$*, the* p^{th} *pseudosquare for some prime* p*, and* (iii) $p_i^{(x-1)/2} \equiv \pm 1 \pmod x$ *for all primes* $p_i \leq p$*, and* (iv) $2^{(x-1)/2} \equiv -1 \pmod x$ *when* $x \equiv 5 \pmod x$ *and* $p_i^{(x-1)/2} \equiv -1 \pmod x$ *for some prime* $p_i \leq p$ *when* $x \equiv 1 \pmod 8$*, then* x *is a prime or a prime power.*

To begin, the algorithm must calculate L_p. We refer to the original paper for a method that performs this calculation in $o(N)$, but which further points out that the first 73 pseudosquares (available online at https://oeis.org/A002189/b002189.txt) are sufficient for any $N < 2.9 \times 10^{24}$. Next, the algorithm calculates the first s primes.

The algorithm divides all N integers into segments of size $\Delta = \pi(p)\log N$. For each such segment, it goes through the following three phases. We assume that $M >> \pi(p)$.

In the first phase, the algorithm performs a (linear) wheel sieve to eliminate multiples of the first s primes. All remaining numbers satisfy the first requirement of Theorem 6.

In the second phase, the algorithm considers each remaining integer k in turn. It first performs a base-2 pseudoprime test, determining if $2^{(k-1)/2} \equiv -1 \pmod k$. If k does satisfy this, then for each $p_i \leq p$, it determines if $p_i^{(k-1)/2} \equiv \pm 1 \pmod k$, with $p_i^{(k-1)/2} \equiv -1 \pmod k$ for some $p_i \leq p$, as well as if $k \equiv 1 \pmod 8$. Note that this is testing the remaining requirements of Theorem 6.

To analyze the RAM complexity, first note that only $O(N/\log N)$ numbers up to N pass the base-2 pseudoprime test (mentioned in [22, 28], among other places).

Furthermore, in a single segment, only $O(\Delta/\log s)$ elements remain after the wheel sieve. Performing the base 2 pseudoprime test takes $O(\log N)$ time, for a total time of $O(N \log N/\log s) = o(N \log N)$. Performing the remaining tests, if required, takes $\pi(p)$ exponentiations, costing $O(\log N)$ operations each, leading to a total cost of $O(N\pi(p))$ over all segments.

In the third phase, the algorithm must remove all prime powers. If $N \leq 6.4 \times 10^{37}$, only primes remain and this phase is unnecessary [28, 30]. Otherwise the algorithm explicitly removes all perfect powers as follows. First, the algorithm constructs by brute force a list of all the perfect powers less than N by repeatedly exponentiating every element of the set $\{2, \ldots, \lfloor\sqrt{N}\rfloor\}$ until it passes N. This list has $O(\sqrt{N}\log N)$ elements, so these can be sorted and removed from the list of prime candidates in $\langle O(N/B), O(N)\rangle$. Therefore the complexity of the algorithm is dominated by the second phase, leading to the following theorem.

Theorem 7. *The sieve of Sorenson runs in* $\langle O\left(\frac{N}{B}\right),\ O(N\pi(p))\rangle$.

We can phrase the complexity in terms of N alone by bounding p. The best known bound for p leads to a running time of $O(N^{1.132})$. On the other hand, the Extended Riemann Hypothesis implies $p < 2\log^2 N$, and Sorenson conjectures that $p \sim \frac{1}{\log 2}\log N \log\log N$ [28]; under these conjectures the RAM complexity is $O(N\log^2 N/\log\log N)$ and $O(N \log N)$ respectively.

Sieving an interval. Similar analysis shows that we can efficiently sieve an interval with Sorenson as well. See Appendix C.2 for details.

Acknowledgments

We thank Oleksii Starov for suggesting this problem to us.

References

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

[2] M. Agrawal, N. Kayal, and N. Saxena. Primes is in P. *Annals of Mathematics*, pages 781–793, 2004.

[3] L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[4] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proc./ of the 34th Annual Symposium on Theory of Computing*, pages 268–276, 2002.

[5] L. Arge and M. Thorup. Ram-efficient external memory sorting. In *Algorithms and Computation*, volume 8283, pages 491–501. 2013.

[6] A. Atkin and D. Bernstein. Prime sieves using binary quadratic forms. *Mathematics of Computation*, 73(246):1023–1030, 2004.

[7] J. Barkley Rosser and L. Schoenfeld. Approximate formulas for some functions of prime numbers. *Illinois J. Math*, 6:64–94, 1962.

[8] C. Bays and R. H. Hudson. The segmented sieve of Eratosthenes and primes in arithmetic progressions to 1012. *BIT Numerical Mathematics*, 17(2):121–127, 1977.

[9] S. Bengelloun. An incremental primal sieve. *Acta informatica*, 23(2):119–125, 1986.

[10] R. P. Brent. The first occurrence of large gaps between successive primes. *Mathematics of Computation*, 27(124):959–963, 1973.

[11] D. A. Cox. *Primes of the form $x^2 + ny^2$: Fermat, Class Field Theory, and Complex Multiplication*. Wiley, 1989.

[12] B. Dunten, J. Jones, and J. Sorenson. A space-efficient fast prime number sieve. *IPL*, 59(2):79–84, 1996.

[13] M. Farach-Colton and M. Tsai. On the complexity of computing prime tables. In *Algorithms and Computation - 26th International Symposium, ISAAC'15*, 2015.

[14] R. Gale and V. Pratt. CGOL–an algebraic notation for MACLISP users, 1977.

[15] D. Gries and J. Misra. A linear sieve algorithm for finding prime numbers. *Communications of the ACM*, 21(12):999–1003, 1978.

[16] G. H. Hardy and E. M. Wright. *An introduction to the theory of numbers*. Oxford University Press, 1979.

[17] S. Horsley. KOΣKINON EPATOΣΘENOYΣ. or, The Sieve of Eratosthenes. Being an Account of His Method of Finding All the Prime Numbers, by the Rev. Samuel Horsley, FRS. *Philosophical Transactions*, pages 327–347, 1772.

[18] H. W. Lenstra Jr and C. Pomerance. Primality testing with gaussian periods. *Lecture Notes in Computer Science*, pages 1–1, 2002.

[19] H. G. Mairson. Some new upper bounds on the generation of prime numbers. *Communications of the ACM*, 20(9):664–669, 1977.

[20] F. News. World's largest prime number discovered – all 17 million digits. `https://web.archive.org/web/20130205223234/http://www.foxnews.com/science/2013/02/05/worlds-largest-prime-number-discovered/`, February 2013.

[21] M. Patrascu and M. Thorup. Dynamic integer sets with optimal rank, select, and predecessor search. In *FOCS*, pages 166–175, 2014.

[22] C. Pomerance, J. L. Selfridge, and S. S. Wagstaff. The pseudoprimes to $25 \cdot 10^9$. *Mathematics of Computation*, 35(151):1003–1026, 1980.

[23] P. Pritchard. A sublinear additive sieve for finding prime number. *Communications of the ACM*, 24(1):18–23, 1981.

[24] P. Pritchard. Linear prime-number sieves: A family tree. *Science of computer programming*, 9(1):17–35, 1987.

[25] P. Pritchard. Linear prime-number sieves: A family tree. *Science of computer programming*, 9(1):17–35, 1987.

[26] R. C. Singleton. Algorithm 357: An efficient prime number generator. In *Communications of the ACM*, pages 563–564, 1969.

[27] J. Sorenson. An introduction to prime number sieves. Technical Report 909, University of Wisconsin-Madison, Computer Sciences Department, 1990.

[28] J. P. Sorenson. The pseudosquares prime sieve. In *Algorithmic number theory*, pages 193–207. 2006.

[29] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing surveys (CsUR)*, 33(2):209–271, 2001.

[30] H. C. Williams. Edouard lucas and primality testing. *Canadian Mathematics Society Series of Monographs and Advanced Texts*, (22), 1998.

A Linear Eratosthenes's sieve with buffer trees

In this section, we provide further details for the algorithm given in Section 3 and prove its correctness.

We first recall the version of the linear sieve of Gries and Misra [15].

Note that we begin by pre-sieving the interval by the $\sqrt{\log N}$ smallest primes. This operation speeds up the execution without violating its correctness, as the resulting candidate primes are exactly those encountered by the algorithm after the loop where $p = p_{1+\sqrt{\log N}}$. This has a cost of $\langle O\left(N/B \log N\right),\ O\left(N/\log N\right)\rangle$, using the algorithm in Appendix C and leaves $\bar{N} = N/\log\log N$ potential primes. We will use $\bar{N}$ in the following to refer to the number of elements inserted.

Data: $S = \{2, 3, \cdots, N\}$
Result: $S = \{p \mid p \in \mathbb{P}, p \leq N\}$
`// p and q are global variables, accessible in any function`
`// C represents the set of numbers known as composites. The operations Insert and InverseSuccessor are implicitly on C`
1 $\mathcal{C} \leftarrow$ integers less than N multiple of any of the first $\sqrt{\log N}$ primes;
2 $\mathcal{T} \leftarrow$ bitarray where $\mathcal{T}[i] = 1$ iff $i \in \mathcal{C}$;
3 $p \leftarrow p_{1+\sqrt{\log N}}$;
4 **while** $p \leq \sqrt{N}$ **do**
5 $q \leftarrow p$;
6 **while** $q \leq N/p$ **do**
7 **for** $r = 1, 2, \ldots, \log_p N/q$ **do**
8 `Insert(`$p^r q$`);`
9 $q \leftarrow$ `InverseSuccessor(`q`);`
10 $p \leftarrow$ `InverseSuccessor(`p`);`
11 **return** `GetSet();` `// this is` $[2; N] \setminus \mathcal{C}$

Algorithm 2: Linear Sieve with Buffer Tree

We now present how we implement these subroutines to achieve an efficient algorithm in both I/O and RAM complexity, then prove its correctness and complexity.

A.1 Implementation

We expose in this part how the subroutines `Insert` and `InverseSuccessor` are actually implemented in our algorithm. First, we give a global description of the data structure used. Then, after setting some preliminary definitions and notations, we present the actual implementation of the subroutines.

Description of the data structure. We use a modified buffer tree which can efficiently handle the two necessary operations to maintain the set $\mathcal{C}$, `Insert` and `InverseSuccessor`. The original structure has been introduced by Arge [3], but our implementation is significantly different to achieve a lower RAM complexity.

The buffer tree is a complete tree with branching factor $\sqrt{M/B}$ and N/M leaves. Its depth is then $d = 2\left\lceil \log_{M/B} \frac{N}{M} \right\rceil$. We will assume for simplicity that the tree is complete even at the leaf level.

Each node has an associated buffer of size M. This buffer consists of $\sqrt{M/B}$ pages of size $\sqrt{MB}$, one for each child, which are internally unsorted. These pages contain the elements in that buffer that are intended for the corresponding child; see Figure 2.

Each leaf corresponds to M consecutive elements between 1 and N. Similar to the internal nodes, the buffer of a leaf is separated in $\sqrt{M/B}$ pages, each associated with exactly $\sqrt{MB}$ elements.

In addition to the buffer tree, the data structure used contains a boolean array $\mathcal{T}$, indexed from 1 to N, where, at the end of the algorithm, $\mathcal{T}[i] = 1$ if and only if i is composite. Intervals of $\mathcal{T}$ corresponding to a leaf page are considered *linked* to this page: when the entire page is brought into memory, this interval is too. This array is saved as a bit-array, which means that one machine word contains at least $\log N$ bits, and operations on a machine word can be done in constant time.

Each page P of an internal (non-leaf) node is partitioned into $\sqrt{M/B} + 1$ unsorted lists. The first one, called P^*, is the list where the new elements are appended. Each other list corresponds to a page Q of the child linked to P, and is denoted by P_Q. Elements moved to a page Q of the next level are either moved directly from P^*, or first moved to the list P_Q then later to Q. See Figure 3 for an illustration; an element can follow blue or red arrows to go to the next level. For consistency, the unique list of a leaf page L will be denoted L^*.

At each level, the numbers present in a node are smaller than the numbers present in the next node. Therefore, inside each node, the numbers present in a page are also smaller than the numbers present in the next page. Each page of the level k, counting the root as the level 0, can then contain $\sqrt{MB}$ numbers among a fixed interval of length $N/\left(MB\right)^{(k+1)/2}$. Therefore, for example, the ith leave, which is at level d (starting the count at 0), consists in M slots to store a subset of $[iM + 1\ ;\ (i + 1)M]$.

Note that no element is ever moved to an upper level nor removed from the tree, except to be inserted in $\mathcal{T}$. Elements can only be moved deeper or to $\mathcal{T}$. In addition, no duplicates are possible.

See Figure 2 for an illustration of the data structure.

Preliminaries We expose here some notations and definitions used throughout the explanation of the algorithm and the proof.

The nodes are indexed by the letter N, the pages by the letter P, and the leaf pages by the letter L.

The least common ancestor page of two leaf pages L and L' will be noted $\mathbf{LCA_P}(L,\ L')$ and its level $\mathbf{LCA}(L,\ L')$.

Above means closer to the root level and *deeper* means closer to the leaf level.

$\texttt{LCA-ABOVE}(L)$ is the property: For all leaves L', no element of L' is above $\mathbf{LCA}(L,\ L')$. In addition, no element of L is in L^*: they are all in $\mathcal{T}$.

$\texttt{LCA-ABOVE}(L, k)$ is the property: For all leaf L', no element of L' is above $\min(\mathbf{LCA}(L,\ L'),\ k)$. Note that $\texttt{LCA-ABOVE}(L, d)$ allows elements of L to be in L^*, and, by convention, $\texttt{LCA-ABOVE}(d + 1)$ is equivalent to $\texttt{LCA-ABOVE}(L)$.

Note that if $k' \geq k$, $\texttt{LCA-ABOVE}(L, k')$ implies $\texttt{LCA-ABOVE}(L, k)$.

Access functions: Due to the static structure of the tree, the following operations can be implemented with a complexity of $\langle 0,\ O(1)\rangle$. When a page P is passed in argument or

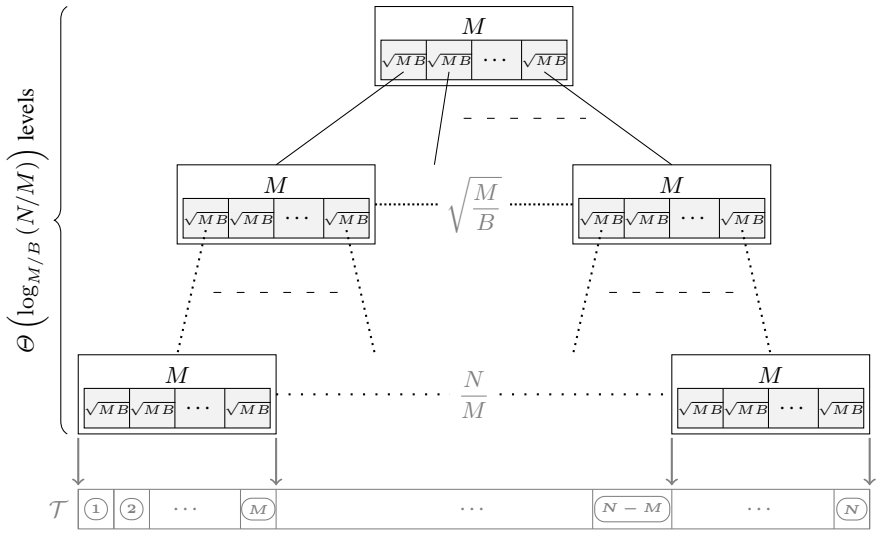


Fig. 2. Illustration of the buffer tree and $\mathcal{T}$.

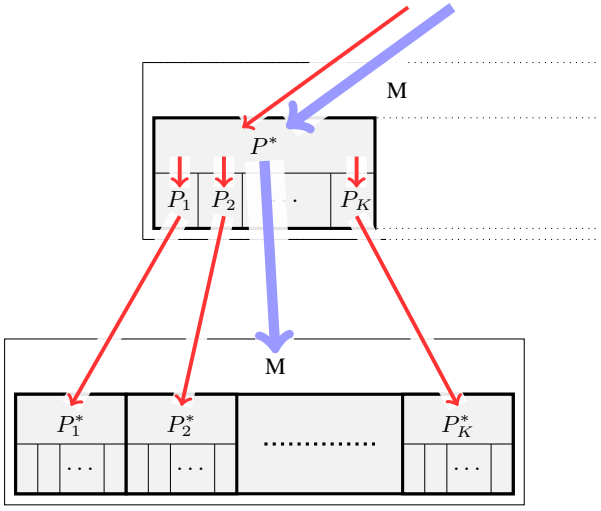


Fig. 3. Illustration of a page of the buffer tree: the list P^* can be flushed to any page of the corresponding child node in `Flush`. The other lists can only be flushed to one page in `PartialFlush` or in `Flush`. An insertion is always executed on the list P^* of each page. We have $K = \sqrt{M/B}$.

returned by a function, only an identifier is implied, and not all the numbers contained, hence the null I/O cost.

- `GetPage(x, k)` : returns the page associated to the number x at level $k \leq d$
- `GetPage(L, k)` : returns the page associated to numbers of leaf page L at level $k \leq d$

For the sake of simplicity, for any number x, we will note $L_x = \texttt{GetPage}(x, d)$. For instance, the leaf pages $L_p = \texttt{GetPage}(p, d)$ and $L_q = \texttt{GetPage}(q, d)$ can be computed in any function.

We define the set $\mathcal{P}_{p,q}$ by the set of pages associated to p or q plus the pages of the root. This set will be assumed to be in memory in the design of the algorithms. This assumption in proved later, together with the complexity proof. Note that $\mathcal{P}_{p,q}$ is modified during the execution of the algorithm. More formally, the definition of $\mathcal{P}_{p,q}$ is:

$$\begin{aligned}\mathcal{P}_{p,q} = &\{P \mid \exists k \leq d,\ P = \texttt{GetPage}(p, k) \text{ or } P = \texttt{GetPage}(q, k)\} \\ &\cup \{P \mid P \text{ is of level } 0\}\end{aligned}$$

Note that saying that $\mathcal{P}_{p,q}$ is in memory includes the relevant slots of $\mathcal{T}$.

Subroutines We expose here a detailed explanation on how both subroutines are implemented, along with the associated pseudo-code.

Management of $\mathcal{T}$: The implementations of basic operations performed on $\mathcal{T}$ are detailed in Algorithm 6. An insertion in $\mathcal{T}$ simply modifies the corresponding bit. The function $\texttt{NextIn}\mathcal{T}(x)$ computes the next candidate to be prime. All integers skipped are confirmed composites. This function uses the bit-array structure of $\mathcal{T}$ to gain a $\log N$ speedup, as we will show later.

Insertions: The algorithm `Insert` is presented in Algorithm 2.

Basically, an element x is inserted at a given level k by computing the appropriate page P and appending it to the list P^*. If this page is full, i.e., it already contains $\sqrt{MB}$ elements, these elements are moved to the next level via Procedure `Flush`. In addition, if x is associated to a page of $\mathcal{P}_{p,q}$ in the next level, it is inserted to the deepest page of $\mathcal{P}_{p,q}$ possible.

These moves are done directly from the list P^*, and the appropriate page of the next level is computed for each element. This process follows the blue arrows on Figure 3.

A call to `Insert` in Algorithm 2 triggers a call to insert the element at level 0. It can be inserted deeper according to $\mathcal{P}_{p,q}$ as illustrated in Figure 4, and trigger some flushes.

At the end of the algorithm, the call to `GetSet` flushes all the tree into the array $\mathcal{T}$, then returns $\mathcal{T}$.

Inverse Successor: The algorithm `InverseSuccessor` is presented in Algorithm 13.

The objective of `InverseSuccessor` is to compute the next element that is not in $\mathcal{C}$, which means that is not in the tree or in $\mathcal{T}$. A naive algorithm would be to check in each page if the element is present or not, but this achieves a very high complexity. The strategy of `InverseSuccessor` is to ensure that the next elements cannot be in

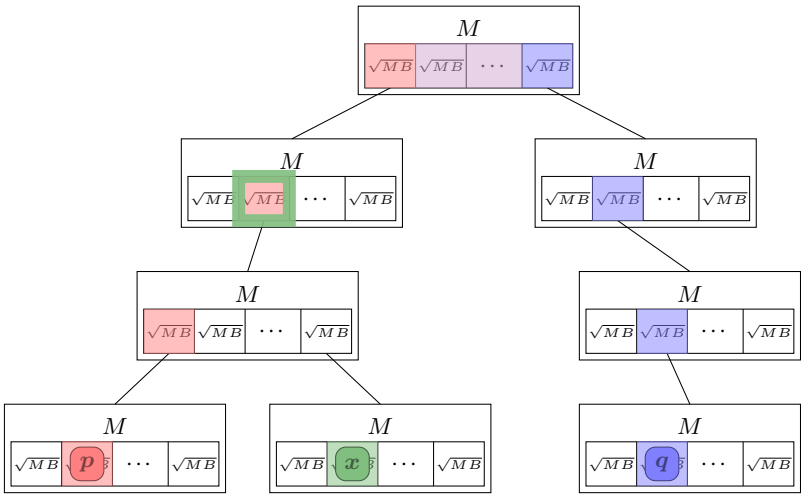


Fig. 4. Illustration of the insertion of x. The leaves in which p, q and x belong are drawn. The green rectangle is the page where x will be inserted. The colored pages are pages of $\mathcal{P}_{p,q}$. $\mathcal{T}$ is not represented.

Procedure `InsertIn`$\mathcal{T}(x)$
  Input: A multiple x with $\mathcal{T}[x] = 0$
  Result: Add x in $\mathcal{T}$
  $\mathcal{T}[x] \leftarrow 1$; `// This only modifies one bit`

Procedure `NextIn`$\mathcal{T}(x)$
  Input: An integer x with $\mathcal{T}[x] = 0$
  Result: An integer $z > x$ such that $\forall t \in (x, z)$, $\mathcal{T}[t] = 1$
  `// We even have` $\mathcal{T}[t] = 0$ `in this version but it is not required`
1 $w' \leftarrow$ machine word containing $T[x]$;
2 $w \leftarrow w'$ with all bits not after $T[x]$ set to 1;
3 **while** *w has no bit equal to 0* **do**
4 $w \leftarrow$ machine word representing the elements of T after w;
5 $z \leftarrow$ index of $\mathcal{T}$ corresponding to the first bit of w equal to 0;
6 **return** z;

Algorithm 3: Operations on $\mathcal{T}$

Algorithm `Insert` (x)
Input: A number $x \notin \mathcal{C}$
Invariant: `LCA-ABOVE`(L_p) and `LCA-ABOVE`(L_q)
Result: Insertion of x in the buffer tree or in $\mathcal{T}$, so in $\mathcal{C}$
`Insertion` (x, 0) ;

Procedure `Insertion` (x, k)
Input: A number x and a level $k \leq d$
Data: x is not in the buffer tree nor in $\mathcal{T}$
Result: Insertion of x at level $\max(\mathbf{LCA}(L_x,\ L_p),\ \mathbf{LCA}(L_x,\ L_q),\ k)$, deeper, or in $\mathcal{T}$
1 **if** L_x *equals* L_p *or* L_q **then**
2 `InsertIn`$\mathcal{T}$ (x) ;
3 **return**;
4 $P \leftarrow$ `GetPage` (x, k) ;
`// if the page of the next level is in` $\mathcal{P}_{p,q}$, `insert deeper`
5 $P_p \leftarrow$ `GetPage` (p, $k+1$) ; $P_q \leftarrow$ `GetPage` (q, $k+1$) ;
6 **if** $k < d$ *and* `GetPage` (x, $k+1$) *is equal to* P_p *or to* P_q **then**
7 `Insertion` (x, $k+1$) ; `// `$k < d$
8 **else**
9 **if** $k < d$ *and* $|P| = \sqrt{MB}$ **then**
10 `Flush` (P, k) ;
11 append x to P^* ;

Procedure `Flush` (P, k)
Input: A full page P of level $k < d$
Result: P is empty
`// Note that all the elements will be inserted in the same node`
1 **foreach** $x \in P$ **do**
2 remove x from P;
3 `Insertion` (x, $k+1$) ;

Procedure `GetSet` ()
Result: A bit array caracterising the set $[2; N] \setminus \mathcal{C}$ by the value 0
1 Call `Flush` on every page for any pre-ordering (children after parents);
2 **return** $\mathcal{T}$;

Algorithm 4: Insertion algorithm and sub-functions

a node: they are in $\mathcal{T}$. This is done by a call to the procedure `PartialFlush`. Then, it is efficient to compute the next element that is not in $\mathcal{C}$ by scanning $\mathcal{T}$.

It is still inefficient if `PartialFlush` has to scan each level of the tree to move the appropriate elements to $\mathcal{T}$, so this function uses the fact that the inserts are done at the deepest page possible in $\mathcal{P}_{p,q}$. This way, as `InverseSuccessor` is only called on the previous value of p or q, `PartialFlush` does not need to scan a page high in $\mathcal{P}_{p,q}$: no relevant element has been inserted here. For instance, on Figure 4, when `PartialFlush` will be called on L_x, it will not check the root node.

This process avoids a high I/O complexity, but still needs a high RAM complexity: for each page scanned, all the elements are checked to see if they can be inserted deeper in $\mathcal{P}_{p,q}$. This means that an element at a page can be scanned $\sqrt{\frac{M}{B}}$ times, once per child page, where we want a constant cost. Thus, `PartialFlush` actually scans only the list P^* of each page, and move the elements to the appropriate list P_Q. Then, it moves all elements from the relevant list P_Q to the next level. This process follows the red arrows in Figure 3.

A.2 Analysis

We first begin by proving the correctness of the implementation, then its complexity. The optimization of the RAM complexity will be discussed after.

Correctness of the algorithm. We need to prove that the algorithms `Insert` and `InverseSuccessor` are correct. *Correct* means that if the input, the data, and the invariant requirements are verified when a function is called, then the output and invariant requirements are verified when the function terminates, and the function does not violate Lemma 2. In addition, no insertion of a new element in the tree not mentioned in the result requirement is performed.

Lemma 2. *An element is never moved to an upper level. No duplicates are possible. If an element is removed from the tree, it is added to $\mathcal{T}$.*

Proof. This lemma will be proved by Theorem 8, which states that the algorithms are correct.

Lemma 3. *For any k, x, P, the procedures* `Insertion(x, k)` *and* `Flush(P, k)` *are correct.*

Proof. We prove this result by induction on $d - k$.

If $k = d$, the call to `Insertion(x, k)` adds x either to the array $\mathcal{T}$ or to the appropriate page leaf so is correct. `Flush` cannot be called on such input, so the property is verified.

Suppose the property true for $k - 1$, and we now prove it for k.

Consider the procedure `Flush`. For each element of the page P, it is removed from P, then, by induction, inserted to a deeper level or in $\mathcal{T}$. So `Flush` is correct as it does not violate Lemma 2 and empty P.

Consider the procedure `Insertion`.

If the test Line 2 occurs, x is inserted in $\mathcal{T}$ so the call is correct.

Algorithm `InverseSuccessor(`x`)`
Input: A number $x \notin \mathcal{C}$
Data: `LCA-ABOVE`(L_x)
Output: The smallest number y greater than x and not present in $\mathcal{C}$
Result: `LCA-ABOVE`(L_y)
1 $y \leftarrow x$;
`// scan` $\mathcal{T}$ `to find` y
2 **repeat**
3 $tmp \leftarrow y$;
4 $y \leftarrow$ `NextIn`$\mathcal{T}$`(`y`);`
5 **if** $L_y \neq L_{tmp}$ **then**
`// flush the next elements from the buffer tree to` $\mathcal{T}$
6 `PartialFlush(`d, L_y, L_{tmp}`);`
7 **until** $\mathcal{T}[y] = 0$;
8 **return** y;

Procedure `PartialFlush(`k, L, L_{tmp}`)`
Input: A level $k \leq d$ and two leaves L and L_{tmp}
Data: `LCA-ABOVE`(L_{tmp})
Result: `LCA-ABOVE`$(L,\ k+1)$
1 $P \leftarrow$ `GetPage(`L, k`);`
`// Flush partially the ancestor pages starting from` $\mathbf{LCA_P}(L,\ L_{tmp})$
2 **if** $k > 0$ *and* `GetPage(`L_{tmp}, k`)` $\neq P$ **then**
3 `PartialFlush(`$k-1$, L, L_{tmp}`)`
4 **if** $k = d$ **then** `// in this case, we have` $P = L$
`// move the leaf page to` $\mathcal{T}$
5 **foreach** $z \in L^*$ **do**
6 remove z from L^*;
7 `InsertIn`$\mathcal{T}$`(`z`);`
8 **else**
`// move each unlabeled element to its corresponding list`
9 **foreach** $z \in P^*$ **do**
10 move z to $P_{\texttt{GetPage}(z,\ k+1)}$;
`// Move deeper the elements that are in the same page as` L
11 **foreach** $z \in P_{\texttt{GetPage}(L,\ k+1)}$ **do**
12 Remove z from P;
13 `Insertion(`z, $k+1$`);`

Algorithm 5: Insertion algorithm and sub-functions

Now suppose that $\max(\mathbf{LCA}(L_x,\ L_p),\ \mathbf{LCA}(L_x,\ L_q)) > k$. Then, the recursive call Line 7 occurs. By induction, x is inserted at a level larger than $\max(\mathbf{LCA}(L_x,\ L_p),\ \mathbf{LCA}(L_x,\ L_q)\ , k)$, so the current call is correct.

Otherwise, `Flush` is called line 10, then x is inserted at level k. So the function is correct as it does not violate Lemma 2 and respects the result requirements.

Lemma 4. *For any k, L, L_x, the procedure* `PartialFlush(`*k, L,* L_{tmp}`)` *is correct.*

Proof. We prove this result by induction on k.

If $k = 0$, for any L', if $\mathbf{LCA}(L,\ L') > 0$ then elements of L' in the root are moved deeper or to $\mathcal{T}$ in Line 13, as the `Insertion` procedure is correct by Lemma 3, so the call of `PartialFlush` is correct.

Suppose the property true for $k-1$, and we now prove it for $k < d$.

First, suppose that $\mathbf{LCA}(L,\ L_{tmp}) < k$. Then, the recursive call Line 3 occurs, and by induction, we have $\texttt{LCA-ABOVE}(L,\ k)$. For any L' such that $\mathbf{LCA}(L,\ L') > k$, if elements of L' are in P, they are moved deeper or to $\mathcal{T}$ at Line 13, as the `Insertion` procedure is correct. If they were in P^*, they are moved to the appropriate page on Line 10. So we have $\texttt{LCA-ABOVE}(L,\ k+1)$.

Therefore, as the procedure respects Lemma 2, it is correct.

Then, suppose that $\mathbf{LCA}(L,\ L_{tmp}) \geq k$. We have $\texttt{LCA-ABOVE}(L_{tmp})$, let's prove that we then have $\texttt{LCA-ABOVE}(L,\ \mathbf{LCA}(L,\ L_{tmp}))$. We will illustrate the cases by Figure 5. Let L' be a leaf page.

If $\mathbf{LCA}(L,\ L_{tmp}) \geq \mathbf{LCA}(L,\ L')$, then $\mathbf{LCA}(L,\ L') \leq \mathbf{LCA}(L_{tmp},\ L')$. As we have $\texttt{LCA-ABOVE}(L_{tmp})$, no element of L' is above $\mathbf{LCA}(L_{tmp},\ L')$ so no element of L' is above $\mathbf{LCA}(L,\ L')$. This corresponds to the case where L' is at the position of Node B, C or D in Figure 5. Note that only the position B implies $\mathbf{LCA}(L,\ L') < \mathbf{LCA}(L_{tmp},\ L')$.

Otherwise, we have $\mathbf{LCA}(L,\ L_{tmp}) < \mathbf{LCA}(L,\ L')$. Then, we have $\mathbf{LCA}(L_{tmp},\ L') = \mathbf{LCA}(L,\ L_{tmp})$ and by definition of $\texttt{LCA-ABOVE}(L_{tmp})$, no element of L' is above $\mathbf{LCA}(L_{tmp},\ L')$ so $\mathbf{LCA}(L,\ L_{tmp})$. This corresponds to the page A of Figure 5.

In both cases, the property $\texttt{LCA-ABOVE}(L,\ \mathbf{LCA}(L,\ L_{tmp}))$ is then respected.

Therefore, we have $\texttt{LCA-ABOVE}(L,\ \mathbf{LCA}(L,\ L_{tmp}))$ so we have the weaker property $\texttt{LCA-ABOVE}(L,\ k)$. Then, as previously, the moves of Line 13 ensure $\texttt{LCA-ABOVE}(L,\ k+1)$ and the correctness of the procedure.

Then, if both cases, the procedure is correct.

Now, we prove the result for $k = d$. By induction, we have $\texttt{LCA-ABOVE}(L,\ k)$ after Line 3, if $L = L_{tmp}$ or not. In Line 6, all the elements of L_y are moved to $\mathcal{T}$, so we get $\texttt{LCA-ABOVE}(L,\ k+1)$.

Therefore, the procedure is correct for any k.

Theorem 8. *The algorithms* `Insert` *and* `InverseSuccessor` *are correct.*

Proof. First, we study the `Insert` algorithm.

As the procedure `Insertion` is correct and $x \notin \mathcal{C}$, the call to `Insertion` has a valid input and x is inserted in the tree at a level not smaller than $\max(\mathbf{LCA}(L_x,\ L_p),\ \mathbf{LCA}(L_x,\ L_q))$. If L_x equals L_p or L_q, x is inserted directly in

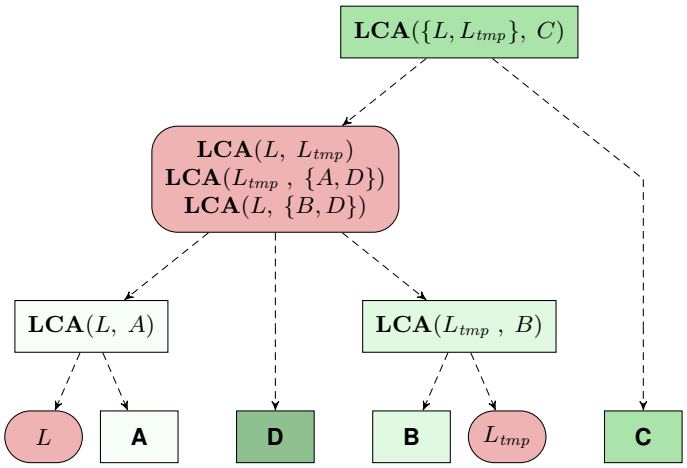


Fig. 5. Abstract tree representing the position of the Least Common Ancestors between two leaf pages L, L_{tmp} and four leaf pages $A,\ B,\ C,\ D$ representing all the possible cases.

$\mathcal{T}$. So the invariant is respected as no element has been moved to a smaller level and x does not violate it.

Now, we study the `InverseSuccessor` algorithm. After each call to `PartialFlush` Line 5, we have `LCA-ABOVE`$(L_y,\ k+1)$, which is `LCA-ABOVE`(L_y). In particular, all elements of L_y are in $\mathcal{T}$. This property is ensured at the beginning by the requirement `LCA-ABOVE`(L_x). The algorithms maintain this property each time y is in a new leaf page, so when the tests Line 7 occur, they are equivalent to testing $y \notin \mathcal{C}$. Therefore, the output of `InverseSuccessor` is correct. Furthermore, there exists a list of leaf pages $\mathcal{L} = \{l_1 \dots l_t\}$ (which is the list of successive L_{tmp}) such that $l_1 = L_x$, $l_t = l_y$ and for each i, a call to `PartialFlush(`d, l_i, l_{i+1}`)` has been triggered. When such a call is triggered, if we had `LCA-ABOVE`(l_i), we get `LCA-ABOVE`(l_{i+1}).

So, as we have `LCA-ABOVE`(L_x) at the beginning of the `InverseSuccessor` call, we have `LCA-ABOVE`(L_y) at the end. So the procedure `InverseSuccessor` is correct.

Complexity of the algorithm. Now, we analyze the total complexity of the algorithm.

We assume Assumption 9 concerning the size of B and M. Apart from the tall cache assumption, this assumes that the order of B is larger than a logarithmic function of N. We recall that $\bar{N}$ represents the number of elements inserted. $\bar{N}$ is equal to N in the original algorithm and to $N/\log\log N$ with the pre-sieving step.

Assumption 9 *We suppose that* $\sqrt{M/B} > \log_{M/B}(N/M)$ *and* $\sqrt{M/B} > \log^2 \log N / \log_{M/B} \frac{N}{B}$.

We analyze the cost of the inserts and the inverse successor queries separately. Note that the height of the tree is $d = \lceil \log_{\sqrt{M/B}} N/M \rceil = O(\log_{M/B} N/M)$.

Lemma 5. *It is possible to always maintain simultaneously in memory all the pages of* $\mathcal{P}_{p,q}$.

Proof. These pages represent one node of size M and $\Theta\left(\log_{M/B} \frac{N}{M}\right)$ pages of size $\sqrt{MB}$. By Assumption 9, this sums to a number of elements m equal to:

$$
\begin{aligned}
m &= O\left(M + \sqrt{MB} \log_{M/B} \frac{N}{M}\right) \\
m &= O\left(M + \sqrt{MB} \sqrt{\frac{M}{B}}\right) \\
m &= O\left(M\right)
\end{aligned}
$$

Then, we assume, to compute the I/O complexity, that the set $\mathcal{P}_{p,q}$ is in memory at the beginning of the functions, and when p or q is changed, the new set $\mathcal{P}_{p,q}$ must be brought in memory at the end of the function. In other words, during the execution of `InverseSuccessor`, $\mathcal{P}_{p,q}$ is always in memory, and at the end, $\mathcal{P}_{p,q}$ plus all the pages related to y are in memory.

Lemma 6. *The complexity of performing all* `Insert` *and* `Insertion` *calls is* $\langle O\left(\frac{\bar{N}}{B} \log_{M/B} \frac{N}{B}\right),\ O\left(\bar{N} \log_{M/B} \frac{N}{B}\right)\rangle$*, assuming that when* `PartialFlush` *calls* `Insertion` *on Line 13, the corresponding page is already in memory.*

Proof. The cost of performing one flush, ignoring the cost of the recursive flushes, is $\langle O\left(\sqrt{M/B}\right),\ O\left(\sqrt{MB}\right)\rangle$. We must move $\sqrt{MB}$ elements to the next level; each requires $O(1)$ computation to find the page it occurs in. The cost to write out $\sqrt{MB}$ elements consecutively requires $O(\sqrt{M/B})$ I/Os. Then we get the above time, plus we may need to do an extra I/O per list when the block is initially brought in. Since there are $O(\sqrt{M/B})$ lists, this gives a total of $O(\sqrt{M/B})$ I/Os per flush. We move $\sqrt{MB}$ elements during this flush, so the per-element flush cost is $O(1/B)$.

Each element can be involved in at most $d = \Theta\left(\log_{M/B} \frac{N}{B}\right)$ flushes and insertions, the depth of the tree. Therefore, the amortized cost of the total number of flushes per element is $\langle O\left(\frac{1}{B} \log_{M/B} \frac{N}{B}\right),\ O\left(\log_{M/B} \frac{N}{B}\right)\rangle$.

When a call to `Insertion` occurs, only the actions listed below can have a non-null I/O cost and a non-constant RAM cost.

On Line 7, this extra insertion has a null I/O cost, because the deeper page is already in memory by Lemma 5.

On Line 10, the cost of this flush is already counted above.

On Line 11, we need to separate the cases. If the call of `Insertion` comes from `Insert`, then by Lemma 5, the root is already in memory so the I/O cost is null. If the call comes from `Flush`, then the cost is already counted above. If the call comes from `PartialFlush`, by hypothesis of the Lemma, the page P is already in memory so the I/O cost is null. If the call comes from the recursive call Line 7, the I/O cost is

null because by Lemma 5, this page is already in memory. In all cases, the RAM cost is constant.

On Line 2, by Lemma 5, the corresponding slot of $\mathcal{T}$ is already in memory.

Therefore, as there are less than N elements inserted, the complexity of performing the total number of insertions is $\langle O\left(\frac{\bar{N}}{B}\log_{M/B}\frac{N}{B}\right),\ O\left(\bar{N}\log_{M/B}\frac{N}{B}\right)\rangle$.

We now need to prove Lemma 7, which analyzes the complexity of a toy algorithm, Algorithm 6, before proving Lemma 8 on the I/O complexity of the `InverseSuccessor` calls.

Lemma 7. *The I/O complexity, amortized against the calls to* `Insertion`*, of Algorithm 6 is* $O\left(\frac{b}{\sqrt{MB}}+\frac{b}{B\log N}\right)$ *and the total number of recursive calls to* `PartialFlush` *is* $O\left(\frac{b}{\sqrt{MB}}\right)$.

Proof. Let's compute the cost of Algorithm 6, assuming it is launched in Algorithm 2, so that all the pages related to a are in memory.

Algorithm `InverseSuccessorLoop(`*a, b*`)`
Input: Two numbers such that $a < b$, and $a = p$ in Algorithm 2
1 $x \leftarrow a$;
2 **while** $x < b$ **do**
3 $x \leftarrow$ `InverseSuccessor(`*x*`)`;

Algorithm 6: Theoretical study algorithm

First, we have to show that when `PartialFlush(`*d, L_y, L_{tmp}*`)` occurs, the I/O complexity, without the calls to `Insertion`, is $O\left(\sqrt{\frac{M}{B}}\left(d-\mathbf{LCA}(L_y,\ L_{tmp})\right)\right)$.

Indeed, during one call, only the page P has to be brought into memory, plus what the recursive call requires. The recursive call cannot be called on a level higher than $\mathbf{LCA}(L_y,\ L_{tmp})$, so there are at most $d-\mathbf{LCA}(L_y,\ L_{tmp})$ recursive calls. Note that for the last call, the page is also associated to L_y, so is already in memory. The cost to bring the page P into memory is $\Theta(1+k/B)=O(\sqrt{MB})$ where k is the number of elements contained in P at the time when the page is brought.

We now compute the I/O complexity of Algorithm 6. The subarray of $\mathcal{T}$ between a and b has to be brought into memory for the tests on $\mathcal{T}$ Line 7 in `InverseSuccessor`, which has a cost of $O\left(\frac{b-a}{B\log N}\right)$. Indeed, each machine word contains $\log N$ bits.

Then, we define the list $\mathcal{L}$ as in the proof of Theorem 8. $\mathcal{L}$ is the list of leaves $\mathcal{L}=\{l_1\dots l_t\}$ (which is the list of successive L_{tmp}) such that $l_1=L_a$, $l_t=L_b$ and for each i, a call to `PartialFlush(`*d, l_i, l_{i+1}*`)` has been triggered inside a call to `InverseSuccessor`. The number of pages contained in $\mathcal{L}$ is exactly

$$\sum_i \left(d-\mathbf{LCA}(l_i,\ l_{i+1})\right)$$

This term is bounded by the number of pages of the tree surrounding L_a and L_b. See Figure 6 for an illustration.

We split this set of pages into a set of $O\left(\frac{b-a}{M}\right)$ internal pages and $O\left(\frac{b-a}{\sqrt{MB}}\right)$ leaf pages. We count a cost of $O\left(\sqrt{\frac{M}{B}}\right)$ I/Os to bring into memory an internal page and

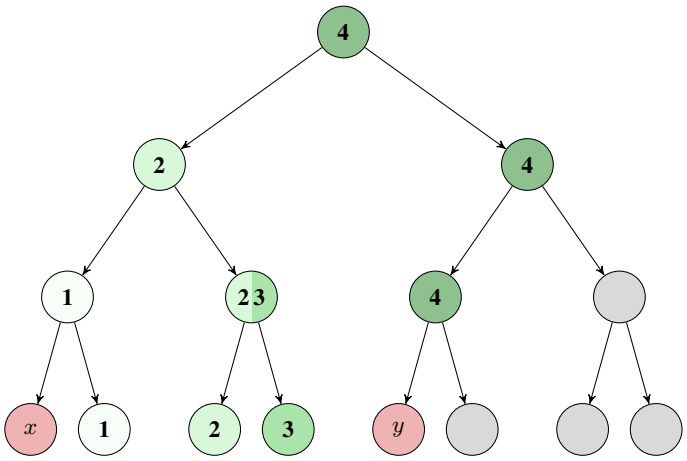


Fig. 6. Illustration of the successive calls to `PartialFlush` in a call of `InverseSuccessor`. The ancestors of x are assumed to be in memory. The calls are done on a leaf, then on its ancestors. The ith call to `PartialFlush` scans the nodes labeled by i.

$O\left(1+\frac{k_P}{B}\right)$ I/Os to bring into memory a leaf page, where k_p is the number of elements in this page.

Therefore, the I/O complexity is

$$O\left(\frac{b-a}{M}\sqrt{\frac{M}{B}}+\frac{b-a}{\sqrt{MB}}+\sum_{P\in\mathcal{L}}\frac{k_P}{B}\right)=O\left(\frac{b-a}{\sqrt{MB}}+\frac{1}{B}(\#\textit{ insertions in }\mathcal{T})\right)$$

Indeed, each element present in a leaf page will be added to $\mathcal{T}$. As each element can only be inserted once in $\mathcal{T}$, we can amortize this cost against the insertion cost, so the additional I/O cost of Algorithm 6 is

$$O\left(\frac{b-a}{\sqrt{MB}}+\frac{b-a}{B\log N}\right)$$

Now, we need to compute the total number of recursive calls of `PartialFlush`. During a call to `InverseSuccessorLoop(`*a, b*`)`, each page of the tree surrounding L_a and L_b makes one recursive call. So the number of recursive calls is $O\left(\frac{b}{\sqrt{MB}}\right)$, by the same argument as above.

Lemma 8. *The complexity of performing all* `InverseSuccessor` *calls, in addition to the total cost of the* `Insertion` *calls, is*

$$\langle O\left(\frac{N\log\log N}{B\log N}\right),\ O\left(\frac{N\log\log N}{\log N}\right)\rangle$$

Furthermore, when a call to `Insertion` *is done, the concerned page is already in memory.*

Proof. We can group the calls to `InverseSuccessor` in Algorithm 2 in one call to `InverseSuccessorLoop(`1, $\sqrt{N}$`)` for the ps and for each p, one call to `InverseSuccessorLoop(`p, $\frac{n}{p}$`)` for the qs associated. Indeed, as by Lemma 5, the appropriate pages are kept in memory, the I/O complexity is not changed by this modification.

By Lemma 7, the total I/O complexity of all the calls to `InverseSuccessor` is then:

$$C_{I/O} = O\left(\frac{\sqrt{N}}{B} + \sum_{p\in\mathbb{P},\ p<\sqrt{N}} \left(\frac{N}{p}\left(\frac{1}{\sqrt{MB}} + \frac{1}{B\log N}\right)\right)\right)$$

$$= O\left(\frac{\sqrt{N}}{B} + \frac{N\log\log N}{\sqrt{MB}} + \frac{N}{B}\frac{\log\log N}{\log N}\right)$$

$$= O\left(\frac{N\log\log N}{B\log N} + \frac{N\log\log N}{\sqrt{MB}}\right)$$

Concerning the RAM complexity, in the total calls to `PartialFlush`, the cost of moving the elements is bounded by the complexity of the total calls to `Insertion`. Indeed, only a constant RAM complexity per element per level is needed: to move it from a list P^* to its list P_Q, which can happen only once per element per page. For the leaves level, again, only a constant cost per element is required. Therefore, we amortize this cost against the insertion cost, and do not count it here.

We have to add a constant cost per recursive call, to take into account the calls where no element is moved, which, by Lemma 7, sums to:

$$C^1_{\mathrm{RAM}} = O\left(\frac{N}{\sqrt{MB}}\log\log N\right)$$

The only term remaining to compute the total RAM complexity of `InverseSuccessor` is the term without counting the calls to `PartialFlush`. We know that there are $O(\bar{N} + \sqrt{N}) = O(\bar{N})$ calls to `InverseSuccessor`. Indeed, there is one call per modification of the value of p or q.

After each call to `NextIn`$\mathcal{T}$ Line 4, we have $\mathcal{T}[y] = 0$. Therefore, either a call to `PartialFlush` is triggered Line 5, or the call terminates. There are $O\left(\frac{N}{\sqrt{MB}}\log\log N\right)$ calls to `PartialFlush`, so $O\left(\frac{N}{\sqrt{MB}}\log\log N + \bar{N}\right)$ calls to `NextIn`$\mathcal{T}$.

Now, note that the RAM cost of the function `NextIn`$\mathcal{T}$ called on x and returning y is $O\left(1 + \frac{y-x}{\log N}\right)$, as each line executes in constant time and a machine word contains $\log N$ bits. Therefore, the remaining RAM term is equal to :

$$
\begin{aligned}
C^2_{\mathrm{RAM}} &= O\left(\frac{N}{\sqrt{MB}}\log\log N + \bar{N} + \sum_{p\in\mathbb{P},\ p<\sqrt{N}}\left(\frac{N}{p\log N}\right)\right)\\
&= O\left(\bar{N} + N\log\log N\left(\frac{1}{\sqrt{MB}} + \frac{1}{\log N}\right)\right)\\
&= O\left(\bar{N} + \frac{N\log\log N}{\log N} + \frac{N\log\log N}{\sqrt{MB}}\right)
\end{aligned}
$$

So the additional RAM complexity, with regards to the cost of the insertions, is:

$$
C_{\mathrm{RAM}} = O\left(\frac{N\log\log N}{\log N} + \frac{N\log\log N}{\sqrt{MB}}\right)
$$

Now, note that by Assumption 9, we have $\sqrt{M/B} = \Omega(\log^2\log N/\log_{M/B}\frac{N}{B})$, so

$$
\frac{N\log\log N}{\sqrt{MB}} = \frac{N}{B}\frac{\log\log N}{\sqrt{M/B}} = O\left(\frac{N\log_{M/B}\frac{N}{B}}{B\log\log N}\right)
$$

Therefore, the total additional cost of the `InverseSuccessor` calls is:

$$
\langle O\left(\frac{N\log\log N}{B\log N}\right),\ O\left(\frac{N\log\log N}{\log N}\right)\rangle
$$

Therefore, by combining the above results, we get

Theorem 10. *The linear sieve of Eratosthenes implemented with buffer trees, assuming that $\sqrt{M/B} > \log_{M/B} N$ and $\sqrt{M/B} > \log^2_{M/B}(N/B)/\log\log N$, has a complexity of*

$$
\langle O\left(\frac{N}{B}\frac{\log_{M/B}\frac{N}{B}}{\log\log N}\right),\ O\left(N\frac{\log_{M/B}\frac{N}{B}}{\log\log N}\right)\rangle
$$

and a space requirement of

$$
O\left(N\left(\sqrt{\frac{B}{M}} + \frac{1}{\log N}\right)\right)
$$

Proof. Indeed, the insertions and flushes, including the last call to `GetSet` that empties the tree, have a complexity of

$$
\begin{aligned}
&\langle O\left(\frac{\bar{N}}{B}\log_{M/B}\frac{N}{B}\right),\ O\left(\bar{N}\log_{M/B}\frac{N}{B}\right)\rangle\\
=&\langle O\left(\frac{N}{B}\frac{\log_{M/B}\frac{N}{B}}{\log\log N}\right),\ O\left(N\frac{\log_{M/B}\frac{N}{B}}{\log\log N}\right)\rangle
\end{aligned}
$$

and the additional complexity of the pre-sieve step and the `InverseSuccessor` calls is

$$\langle O\left(\frac{N\log\log N}{B\log N}\right), O\left(\frac{N\log\log N}{\log N}\right)\rangle$$

Now, as we have $M/B = O(N)$ and $\log N = \Omega(\log^2 \log N)$, we have

$$\frac{\log\log N}{\log N} = O\left(\frac{1}{\log\log N}\right) = O\left(\frac{\log_{M/B}\frac{N}{B}}{\log\log N}\right)$$

So the global complexity for the entire algorithm is

$$\langle O\left(\frac{N}{B}\frac{\log_{M/B}\frac{N}{B}}{\log\log N}\right), O\left(N\frac{\log_{M/B}\frac{N}{B}}{\log\log N}\right)\rangle$$

Concerning the space complexity:

For the pre-sieve, we need a space $O(N/\log N)$. For the whole tree but the leaves, we need a space of $O(N\sqrt{B/M})$. For the bitarray, we need a space $O(N/\log N)$.

For the leaf pages, we can actually shrink their size by a factor $\log N$, so that they can contain at most $\sqrt{MB}/\log N$ elements. Indeed, as these elements will be flushed in a portion of $\mathcal{T}$ that fits in $\sqrt{MB}/\log N$ machine words, the amortized cost of such a flush per element is $O(1/B)$, which is the desired bound. The space used per each page is then $O(1+\sqrt{MB}/\log N)$. So the space used to store all the $O(N/\sqrt{MB})$ leaves is $O\left(N/\sqrt{MB} + N/\log N\right)$. Note that this space optimization has not been depicted in the pseudo-code for clarity.

Therefore, the space requirement is

$$O\left(N\left(\sqrt{\frac{B}{M}} + \frac{1}{\log N}\right)\right)$$

B Sieve of Atkin

We present here the pseudo-code depicting the different versions of the sieve of Atkins [6]. The two first sections only reformulate the original algorithms, and Appendix B.3 depicts our contribution: an I/O-efficient version of the sublinear sieve of Atkins.

B.1 Level Curve Tracing

We first present the non-optimized version of the sieve of Atkins, which has a linear time complexity, and does not optimize I/Os.

If $M = N^{1/2+o(1)}$, then the sieve can be performed in memory. Let $f(x,y)$ be a binary quadratic form and let $L = \{(x,y) \in \mathbb{N}^2 | f(x,y) \leq N\}$. Suppose that M can hold an array of Δ values. Then we can subdivide $L = L_0 \cup L_1 \cup \cdots \cup L_{\lceil n/m \rceil - 1}$, where $L_i = \{(x,y) \in L | i\Delta < f(x,y) \leq (i+1)\Delta\}$, and use our array to count the values of f over each L_i. For conciseness, we only describe the algorithm for the first quadratic form, but the other cases are similar.

generate a list of primes P up to $\sqrt{N}$ by any reasonable means;
for $i \leftarrow 0$ *to* $\lceil N/\Delta \rceil - 1$ **do**
  $A[1] \leftarrow 0, A[2] \leftarrow 0, \ldots, A[\Delta] \leftarrow 0$;
  $x \leftarrow 1$;
  while $x^2 + 4 \leq (i+1)\Delta$ **do**
    $y \leftarrow \left\lceil 1/2\sqrt{(i\Delta) - x^2} \right\rceil, k \leftarrow x^2 + 4y^2$;
    while $k \leq (i+1)\Delta$ **do**
      if $k \equiv 1 \pmod 4$ **then**
        $A[k - i\Delta] \leftarrow A[k - i\Delta] + 1$;
      $y \leftarrow y + 2, k \leftarrow x^2 + 4y^2$;
    $x \leftarrow x + 2$;
  foreach $p \in P$ **do**
    $j \leftarrow \lceil i\Delta/p^2 \rceil$;
    while $p^2 j \leq (i+1)\Delta$ **do**
      $A[p^2 j - i\Delta] \leftarrow 0, j \leftarrow j + 1$;
  for $j \leftarrow 1$ *to* Δ **do**
    if `Odd(`$A[j + i\Delta]$`)` **then**
      `Print(`$j + i\Delta$`);`

Algorithm 7: The "linear" sieve of Atkin for primes congruent to 1 mod 4

Each L_i is the region between two level curves, and the algorithm operates on them individually. For each x the algorithm calculates the smallest viable y within the region and then keeps incrementing it until it escapes the region. Because of the size of M and the choice of Δ, each x with $f(x, 1) \leq (i+1)\Delta$ has at least one y such that $(x, y) \in L_i$. Thus the overhead is at most linear.

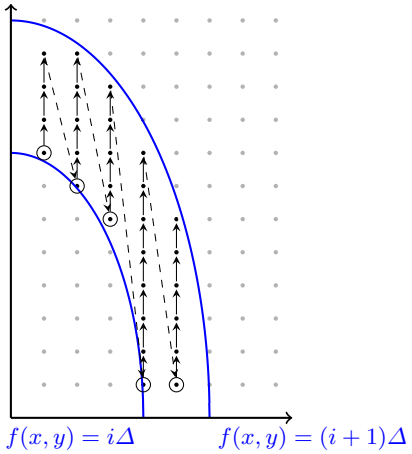


Fig. 7. This figure depicts the algorithm "tracing" the points between level curves of f. The y values of the encircled points must be calculated.

B.2 Pre-sieving with a wheel

Here we show the algorithm with a wheel sieve on L. For brevity we describe the strategy for a general binary form $f(x, y)$. This can then be implemented for each of the binary forms from Theorem 4 in Section 4. This algorithm has a time complexity of $N/\log\log N$, and is the main contribution of [6].

Let $W = 12 * \prod_{i=1}^{\sqrt{\log N}} p_i = N^{o(1)}$, and let $U \subseteq [W]^2$ be the set of points (x, y) such that $f(x, y)$ is a unit mod W. Because $f(x + aW, y + bW) \equiv f(x, y) \pmod{W}$ for any $a, b \in \mathbb{Z}$, we can reduce the domain on which we work to the W-translates of U. This is because the value of f on each of the remaining points must be of the form $kW + c$ where c shares a factor with W. Thus we loop through U, and for each point $d = (x, y) \in U$, we count the occurrences of the values of f on each of the W-translates within L. This is illustrated for a particular d in Appendix B.2. Then those values which occur an odd number of times will be primes or squareful. Those squareful numbers must be sieved, which can be done in a manner analogous to the sieve of Eratosthenes.

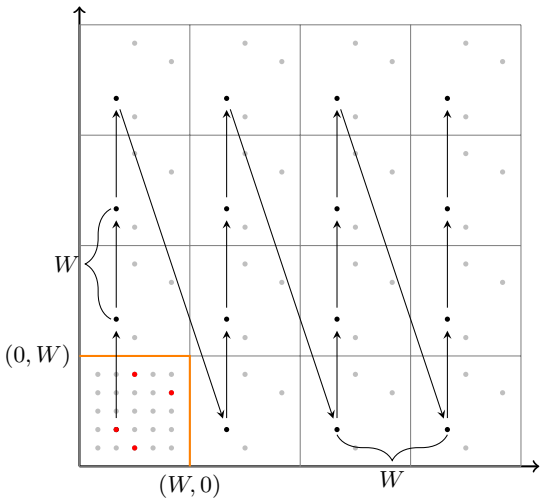


Fig. 8. A visualization of the wheel pre-sieve. Here the red points are the unit-valued points on $[W]^2$, and the grey points in $[W]^2$ have been eliminated. The black points are W-translates of $d = (x, y)$, and the grey points outside of $[W]^2$ are W-translates of other points in U.

Because of the choice of W, it follows from Merten's Theorem that $|U| = O(|W|^2/\log\log N)$. Thus the counting phase of the algorithm will take $O(N/\log\log N)$ time. Since we have already sieved the first $\sqrt{\log N}$ primes, it can be shown that the squarefree sieve can also be completed in $O(N/\log\log N)$.

B.3 Using a priority queue

The pseudo-code functions below describe our variant of the sieve of Atkins. Priority queues are used instead of arrays to improve the I/O efficiency. This version requires

$O\left(N^{1+o(N)}\right)$ space, but it can be segmented among level curves as in Appendix B.1 to use $O\left(N^{1/2+o(N)}\right)$. Note that in what follows objects are passed to functions as references. The code for some functions has been omitted.

```
W ← ComputeWheelModulus(N);
U ← ComputeUnitsMod(W);
create three empty lists of pairs L1, L2 and L3;
L1 ← ConstructPrincipalDomain(W,U,1,4,1,4);
L2 ← ConstructPrincipalDomain(W,U,3,1,7,12);
L3 ← ConstructPrincipalDomain(W,U,3,-1,11,12);
create an empty min priority queue Q that only stores values;
InsertValuesFromDomain(Q, W, L1, 1, 4) ;
InsertValuesFromDomain(Q, W, L2, 3, 1) ;
InsertValuesFromDomain(Q, W, L3, 3, −1) ;
Q.Insert(∞);
create an empty queue S;
S ← EliminateEven(Q);
Print all the primes dividing W;
EliminateSquaresAndPrint(S);
```

Algorithm 8: The main process of the Sieve of Atkin in external memory

```
ConstructPrincipalDomain(W, U, a, b, c, d)
    create an empty list L;
    foreach (x, y) ∈ [W]^2 do
        if ax^2 + by^2 ∈ U + WZ and ax^2 + by^2 ≡ c (mod d) then
            L.Add((x, y));
    return L;
```

Algorithm 9: ConstructPrincipalDomain: Relative to $f(x, y) = ax^2 + by^2$, returns a list of all the unit-valued (mod W) points (x, y) in $[W]^2$ with $f(x, y) \equiv c \pmod{d}$

```
InsertValuesFromDomain(Q, W, L, a, b)
    foreach (x, y) ∈ L do
        i ← 1;
        while a(x + iW)^2 + by^2 ≤ N do
            j ← 1;
            while a(x + iW)^2 + b(y + jW)^2 ≤ N do
                Q.Insert((x + iW)^2 + 4(y + jW)^2);
                j ← j + 1;
            i ← i + 1;
    return;
```

Algorithm 10: InsertValuesFromDomain: Relative to $f(x, y) = ax^2 + by^2$, inserts the value of f on every W-translate of every point in L into Q.

```
EliminateEven(Q)
    create an empty queue S;
    p' ← 0, c ← 0, k ← 0;
    while Q ≠ ∅ do
        p ← Q.Extract-Min();
        if p ≠ p' then
            if Odd(c) then
                k ← k + 1, S.Enqueue(p');
            c ← 1;
        else
            c ← c + 1;
        p' ← p;
    return S;
```

Algorithm 11: EliminateEven: Returns a queue with all the values in Q that occur an odd number of times.

```
EliminateSquaresAndPrint(S)
    create an empty key-sensitive min priority queue Q' that can store ⟨key, value⟩ pairs;
    c ← S.Dequeue();
    Q'.Insert(⟨c, c^2⟩);
    while S ≠ ∅ do
        ⟨p, v⟩ ← Q'.Find-Min();
        c ← S.Dequeue();
        while v < c do
            Q'.Extract-Min();
            Q'.Insert(⟨p, v + p^2⟩);
            ⟨p, v⟩ ← Q'.Find-Min();
        if v = c then
            while v = c do
                Q'.Extract-Min();
                Q'.Insert(⟨p, v + p^2⟩);
                ⟨p, v⟩ ← Q'.Find-Min();
        else
            Print(c);
            Q'.Insert(⟨c, c^2⟩);
    return;
```

Algorithm 12: EliminateSquaresAndPrint: Prints the squarefree numbers in S, which in this context are the primes (excluding those removed by the wheel, which are printed in the main procedure).

C Sieving the first $\sqrt{\log N}$ primes

We describe here a method to compute the numbers smaller than N that are co-prime to the first $\sqrt{\log N}$ primes. This method is used by the algorithm in Appendix A. First, we present the pseudo-code of the algorithm, before proving its correctness and its complexity.

Data: $S = \{2, 3, \cdots, N\}$
Result: A bit vector expliciting the co-primes to $\{p_1 \dots p_{\sqrt{\log N}}\}$ up to N
Compute the first $\sqrt{\log N}$ primes $p_1 \dots p_{\sqrt{\log N}}$;
Compute $P = \prod_{1<i<\sqrt{\log N}} p_i$;
Sieve $S_P = \{1 \dots P\}$ with the first primes in a bit vector s_P (value `COMPOSITE` or `COPRIME`);
Compute s'_P equal to s_P but with bits before $p_{\sqrt{\log N}}$ set to `COMPOSITE`;
Concatenate s_P with copies of s'_P to form a N-long bit vector s;
return s;

Algorithm 13: Low-primes sieving

Lemma 9. *This algorithm is correct: the returned bit vector explicits the co-primes to to $\{p_1 \dots p_{\sqrt{\log N}}\}$ up to N.*

Proof. We need to show that for all $x < N$, $s[x]$ is `COMPOSITE` if and only if there exists $k \leq \sqrt{\log N}$ such that p_k divides x.

First, suppose $x < P$. This property is ensured by the explicit sieving of S_P.

If x is greater than P, let $i = x \mod P$. Then, for any $k \leq \sqrt{\log N}$, p_k divides x if and only if p_k divides i. If $i \leq p_{\sqrt{\log N}}$, there exists $k \leq \sqrt{\log N}$ such that p_k divides i, so x. And $s[x] = s'_P[i] = 0$, so the property is true. If $i > p_{\sqrt{\log N}}$, there exists $k \leq \sqrt{\log N}$ such that p_k divides i if and only if $s_P[i] =$ `COMPOSITE`, and $s[x] = s'_P[i] = s_P[i]$.

Theorem 11. *The complexity of this algorithm is $\langle O\left(N/(B \log N)\right),\ O\left(N/\log N\right)\rangle$.*

Proof. First, note that P is equivalent to

$$P = \exp\left((1 + o(1))\sqrt{\log N} \log\log N\right) = O(N^{\frac{\log\log N}{\sqrt{\log N}}})$$

Computing the first primes, P, and s'_P from s_P do not exceed the bound.

Sieving S_P successively with $\sqrt{\log N}$ primes has a time complexity of $O(\sqrt{\log N}P)$ and an I/O complexity of $O(\sqrt{\log N}P/(B \log N))$, which does not exceed the bound.

Creating s from s'_P means achieving N/P copies of s'_P, which has a time complexity of $O(N/\log N)$ and an I/O complexity of $O(N/(B \log N))$.

C.1 Sieve of Eratosthenes using a RAM-efficient external-memory priority queue.

The RAM and I/O performance of sieve of Eratosthenes can be improved using the recently proposed RAM-efficient external-memory priority queue [5] in a folklore priority queue based implementation of the sieve.

The straightforward folklore sieve implementation is shown in Figure $1(a)$. The priority queue Q stores $\langle k, v\rangle$ pairs, where k is a prime (***key***) and v is its multiple (***value***). Initially, $\langle 2, 4\rangle$ is inserted into Q. When a pair $\langle k, v\rangle$ is deleted from Q, we check if v is two more than the last value v' deleted, and if so, $p = v - 1$ is not a multiple of any prime, and hence must be a prime itself. We then insert $\langle p, p^2\rangle$ into Q. We always insert the next multiple $v + k$ of k into Q.

The performance bounds of the sieve above with a RAM-efficient external-memory priority queue [5] Q is given by the theorem below. The bounds follow from the observation that the sieve performs $\Theta\left(\sum_{prime\ p\in[1,\sqrt{N}]} \frac{N}{p}\right) = \Theta\left(N \log\log N\right)$ operations on Q costing $\langle O\left(\frac{1}{B}\log_{\frac{M}{B}} N\right),\ O\left(\log_{\frac{M}{B}} N + \log\log M\right)\rangle$ each.

Theorem 12. *The sieve of Eratosthenes (shown in Figure $1(a)$) implemented using a RAM-efficient external-memory priority queue [5] has a complexity of* $\langle O\left(\text{SORT}\left(N\log\log N\right)\right),\ O\left(N\log\log N\left(\log_{\frac{M}{B}} N + \log\log M\right)\right)\rangle$ *and uses* $O\left(\sqrt{N}\right)$ *space for sieving primes in* $[1, N]$.

C.2 Sieve of Sorenson on a Segment

The sieve of Sorenson can be adapted to sieve for primes on the interval $[a, b]$ provided a sufficiently large pseudosquare table is available. We further assume that $M = \Omega(s) = \Omega(\pi(p)\log^2 b)$, where here and below p is determined as above but by b rather than N. In that case, we can determine the primes up to s in $O(s)$. We then perform the initial wheel sieve phase in memory on each segment, which takes $O((b-a)+s) = O((b-a)+\pi(p)\log^2 b)$ operations and $O((b-a)/B+s/B+1) = O((b-a)/B+\pi(p)\log^2 b/B+1)$ I/Os.

In the second phase we must exponentiate each number in the segment for (potentially) each pseudoprime up to p. It takes $\langle O(\frac{b-a}{B}+1), O((b-a)\pi(p))\rangle$.

In the third phase we can for each $k = 2, 3, \ldots, \lfloor\log b\rfloor$ compute $r = \lceil a^{1/k}\rceil$. Then we create the list of perfect powers in $[a, b]$ by taking $r^k, (r+1)^k, \ldots$ for each k until we reach b. This list will have $O((\sqrt{b}-\sqrt{a})\log b)$ elements and can be computed in $O((\sqrt{b}-\sqrt{a})\log^2 b + \log^2 b)$. Thus all the perfect powers can be sorted and removed from the candidate list in $\langle O((b-a)/B), O((b-a)+\log^2 b)\rangle$. We have shown:

Theorem 13. *On a segment from a to b, the sieve of Sorenson runs in* $\langle O((b-a+\pi(p)\log^2 b)/B+1), O((b-a)\pi(p)+\pi(p)\log^2 b)\rangle$

Appendix B

Anti-persistence on persistent storage: history-independent sparse tables and dictionaries [PODS 2016 conference]

Anti-Persistence on Persistent Storage: History-Independent Sparse Tables and Dictionaries

Michael A. Bender[*]
Stony Brook University

Jonathan W. Berry[†]
Sandia National Laboratories

Rob Johnson[*]
Stony Brook University

Thomas M. Kroeger[‡]
Sandia National Laboratories

Samuel McCauley[*]
Stony Brook University

Cynthia A. Phillips[†]
Sandia National Laboratories

Bertrand Simon[§]
Ecole Normale Supérieure de Lyon

Shikha Singh[*]
Stony Brook University

David Zage[¶]
Intel Corporation

ABSTRACT

We present history-independent alternatives to a B-tree, the primary indexing data structure used in databases. A data structure is history independent (HI) if it is impossible to deduce any information by examining the bit representation of the data structure that is not already available through the API.

We show how to build a history-independent cache-oblivious B-tree and a history-independent external-memory skip list. One of the main contributions is a data structure we build on the way—a history-independent packed-memory array (PMA). The PMA supports efficient range queries, one of the most important operations for answering database queries.

Our HI PMA matches the asymptotic bounds of prior non-HI packed-memory arrays and sparse tables. Specifically, a PMA maintains a dynamic set of elements in sorted order in a linear-sized array. Inserts and deletes take an amortized $O(\log^2 N)$ element moves with high probability. Simple experiments with our implementation of HI PMAs corroborate our theoretical analysis. Comparisons to regular PMAs give preliminary indications that the practical cost of adding history-independence is not too large.

Our HI cache-oblivious B-tree bounds match those of prior non-HI cache-oblivious B-trees. Searches take $O(\log_B N)$ I/Os; inserts and deletes take $O(\frac{\log^2 N}{B} + \log_B N)$ amortized I/Os with high probability; and range queries returning k elements take $O(\log_B N + k/B)$ I/Os.

Our HI external-memory skip list achieves optimal bounds with high probability, analogous to in-memory skip lists: $O(\log_B N)$ I/Os for point queries and amortized $O(\log_B N)$ I/Os for inserts/deletes. Range queries returning k elements run in $O(\log_B N + k/B)$ I/Os. In contrast, the best possible high-probability bounds for inserting into the folklore B-skip list, which promotes elements with probability $1/B$, is just $\Theta(\log N)$ I/Os. This is no better than the bounds one gets from running an in-memory skip list in external memory.

1. INTRODUCTION

A data structure is ***history independent*** (HI) if its internal representation reveals nothing about the sequence of operations that led to its current state [43, 47]. In this paper, we study history independence for persistent, disk-resident dictionary data structures.

We give two efficient history-independent alternatives to the B-tree, the primary indexing data structure used in databases. Specifically, we give an HI external-memory skip list and an HI cache-oblivious[1] B-tree.

One of the main contributions of the paper is a data structure we build on the way: a history-independent packed-memory array (PMA). As we explain, the PMA [14, 18] is an unlikely candidate data structure to be made history independent, since traditional PMAs rely on history so fundamentally. The HI PMA is one of the primary building blocks in the HI cache-oblivious B-tree. However, it can also be bolted onto any dictionary data structure, using the PMA to hold the actual elements and deliver fast range queries.

Notions of History Independence

Informally, history independence partially protects a disk-resident data structure when the disk is stolen by—or given to—a third party (the "observer"). This observer can access the data structure through the normal API but can also see

[1] A ***cache-oblivious*** [29, 30, 51] algorithm or data structure is memory-hierarchy universal, in that it has no memory-hierarchy-specific parameterization (see Section 1.1).

[*]Department of Computer Science, Stony Brook University, Stony Brook, NY 11794-2424 USA. Email: {bender, rob, smccauley,shiksingh}@cs.stonybrook.edu.

[†]MS 1326, PO Box 5800, Albuquerque, NM 87185 USA. Email: {jberry,caphill}@sandia.gov.

[‡]PO Box 969, MS 9011, Livermore, CA 94551 USA. Email: tmkroeg@sandia.gov.

[§]LIP, ENS de Lyon, 46 allee d'Italie, 69364 Lyon, France. Email: bertrand.simon@ens-lyon.fr.

[¶]Intel Corporation, 2200 Mission College Blvd, Santa Clara, CA 95054 USA. Email: zage@cerias.net.

PODS'16, June 26-July 01, 2016, San Francisco, CA, USA

DOI: http://dx.doi.org/10.1145/2902251.2902276

its bit and pointer representation on disk. An HI data structure's bit representation never leaks any information to the observer that he could not learn through the API.

There are two notions of history independence, ***weak history independence*** (WHI) and ***strong history independence*** (SHI) [47]. The notions are distinguished by how many times the observer can look at the data structure—that is by how many times an interloper can steal (or otherwise gain access to) the disk on which the data structure resides. A weakly history-independent data structure is history independent against an observer who can see the memory representation once. A strongly history-independent data structure is history independent against an observer who can see the memory representation multiple times.

In this paper, we focus on weak history independence. From a performance perspective, weak HI is a stronger notion because it allows provably stronger performance guarantees (see Section 2). Protecting against multiple observations reduces achievable performance.

Weak history independence is the appropriate notion of history independence in situations where only one observation is possible. For example, when the data is on a device that can be separated from the owner (say a portable or embedded device), the owner no longer interacts with the device. WHI provides the same level of protection in this case with significantly better performance compared to SHI.

History Independence and Data

History independence in a database can have major advantages, depending on the kind of data that is being stored and the security requirements.

HI data structures naturally support information-theoretically-secure delete. In contrast, with more standard secure delete (where the file system overwrites deleted data with zeros), information about deleted data can leak from the memory representation. For example, it reveals how much data was deleted and where in the keyspace it might have been. In fact, a long history of failed redactions is one of the original motivations for history independence [36]. History independence guarantees that the memory representation will leak no information about these previous (now secure) deletes. Encryption is not a panacea to protect history unconditionally, since determined attackers can recover the key used to encrypt on-disk data [35].

In a database, the source of the data can be more sensitive than the data itself. As a toy example, consider a database of known organized crime members maintained by the police. The police might want to share such a database with select individuals without revealing the order and times in which the data was added to the database. Revealing this order might leak information about how and when the data was collected, which could reveal sources that the police want to stay hidden. Journalists may desire the same property to ensure their sources' anonymity.

In such cases, we need to be particularly careful about avoiding information leaks by side channels. Since leakage can be subtle and hard to quantify, it is beneficial to have a guarantee that nothing extraneous is revealed, which is exactly what history independence guarantees. HI data structures naturally hide the order in which data was inserted.

History Independence in Persistent Storage

This paper focuses on history independence for external-memory dictionaries. This area of history independence was initiated by Golovin [32, 33]; see Section 1.4 for details.

The ubiquitous (non-history-independent) external-memory dictionary is the B-tree.

Our objective is to build history-independent, I/O efficient external-memory dictionaries.We support standard dictionary operations: insertions, deletions, searches, and range queries. Our two external-storage computational models (the external-memory model [3] and the cache-oblivious model [29, 30, 51]) apply to both rotating disks and SSDs.

We give weakly history-independent data structures for persistent storage; these are easy to implement and retain strong performance guarantees. We show that even range-query data structures that seem to be inherently history *dependent*—in particular, packed-memory array or sparse tables [14, 16–18, 38, 41, 66]—can be made weakly history independent. Overall, our results demonstrate that it is possible to build efficient external-memory (and even cache-oblivious) history-independent data structures for indexing.

History Independence in Persistent Storage vs. RAM

History independence has been vigorously explored in the context of data structures that reside in RAM [20, 21, 23, 36, 43, 46, 47, 61] (see Section 1.4), but significantly less so in external memory. Although there is some theoretical work on history-independent on-disk data structures [31–33] and experimental work on history-independent file systems [10–12, 56], the area is substantially less explored.

This lacuna may seem surprising, since many of the classical arguments in support of history independence especially apply to disks. Hard drives are more vulnerable than RAM because they are persistent and easier to steal, making it easier for an attacker to observe on-disk data.

1.1 I/O and Cache-Oblivious Models

We prove our results using the classic models for analyzing on-disk algorithms and data structures: the disk-access machine (DAM) model of Aggarwal and Vitter [3] and the cache-oblivious model of Frigo et al. [29, 30, 51]. The DAM has an internal memory (RAM) of size M and an arbitrarily large external memory (disk). Data is transferred between RAM and disk in blocks of size $B < M$. The performance measure is transfers (I/Os). Computation is free.

The cache-oblivious model extends the DAM model. Now parameters B and M are unknown to the algorithm designer or coder. They can only be used as parameters in analyses.

Thus, an optimal cache-oblivious data structure is not parameterized by any block, cache or RAM size, or memory- or disk-access times. Remarkably, many problems have optimal (and practical) cache-oblivious solutions, including B-trees [14, 15, 22]. Informally, a cache-oblivious B-tree has approximately optimal memory or I/O performance at every level of an unknown multilevel memory hierarchy.

1.2 Packed-Memory Arrays and External-Memory Dictionaries

In this paper, we give a history-independent PMA and two external-memory dictionaries: a history-independent external skip list and a history-independent cache-oblivious B-tree. This subsection puts our results in context.

We first define a PMA. Then we describe the technical issues involved in making a history-independent PMA. We (this paper's authors) were surprised when we first suspected that a history-independent PMA could indeed exist. Here, we try to articulate why the PMA might seem to be an unlikely candidate data structure to make history independent, but why it is nonetheless possible.

We next explain why a history independent PMA leads, almost directly, to an HI cache-oblivious B-tree, the first nontrivial history-independent cache-oblivious data structure.

Finally, we discuss the history-independent external-memory skip list. This HI data structure still offers high-probability guarantees, analogous to in-memory skip lists. See Section 2 for the definition of with high probability, which we also write as "whp".

Although the idea of a B-skip list, which promotes elements with probability $1/B$ rather than probability $1/2$ is folklore and has appeared in the literature repeatedly [1, 25, 26, 33], we prove that its high-probability I/O guarantees are asymptotically no better than those of an in-memory skip list run in external memory.

Packed-Memory Arrays

One of the classic data-structural problems is called ***sequential file maintenance***: maintain a dynamic set of elements in sorted order in a linear-sized array. If there are N elements, then the array has $\Theta(N)$ empty array positions or *gaps* interspersed among the elements to accommodate future insertions. The gaps allow some elements to shift left or right to open slots for new elements—like shifting books on a bookshelf to make room for new books.

Remarkably, there are data structures for these problems that are efficient even for *adversarial* inserts and deletes. Indeed, the number of element moves per update is only $O(\log^2 N)$ both amortized [38, 63] and in the worst case [64–66], which is optimal [24].

In external memory, this data structure is called a ***packed-memory array*** [14, 18]. It supports inserts, deletes, and range queries. Given the location where we want to insert or delete (which can be found using a separate indexing structure, e.g., [15, 22]), it takes only $O(1 + (\log^2 N)/B)$ amortized I/Os to shift the elements. Given the starting point, a range query returning k elements costs $\Theta(1 + k/B)$ I/Os.[2]

Prior PMAs operate as follows. To insert a new element after an existing element or to delete an element, find an enclosing subarray or ***range***, and ***rebalance***. This spreads out the elements (and gaps) within that range. The rebalance range is chosen based upon the density within the range, where ranges have minimum and maximum allowed densities. These thresholds depend upon the size of the range: small ranges have high thresholds for the maximum density and low thresholds for the minimum density. The larger a range is, the less variability is allowed in its density. The algorithmic subtlety has to do with choosing the right rebalance ranges and the right minimum and maximum density thresholds for each range size.

However, range densities are very history *dependent*. For example, if you repeatedly insert towards the front of an array or if you repeatedly delete from the back of the array, then the front of the array will be denser than the back. How could we possibly make a version of this data structure, that is history independent—that is, where newly inserted (or deleted) elements do not seem to increase (or decrease) some local density? We answer this question in this paper.

To use a more evocative image, picture a long trough where you are pouring sand in one location (corresponding to inserts) and letting out sand in another location (corresponding to deletes). As the sand piles up, the pile gradually flattens (corresponding to local rebalances). Although rebalances may flatten out the pile, we may still expect a bump for newly arrived sand, and a depression for recently departed sand. Perhaps surprisingly, we can avoid bumps and depressions with mostly local rebalances.

[2]This scanning bound is a further requirement on how the elements are distributed. Beyond the $O(N)$ overall space limitation, only $O(1)$ gaps can separate two consecutive elements.

Cache-Oblivious B-Trees

A B-tree is a dictionary supporting search, insert, delete, and range query operations. There are cache-oblivious versions [13, 15, 22, 40, 55].

A history-independent PMA can immediately yield a history-independent cache-oblivious B-tree. The idea is to take a PMA and "glue" it to a static cache-oblivious B-tree [51]. For details, see [15, 22]. We use a similar method to construct our history-independent cache-oblivious B-tree in Section 3.

External-Memory Skip Lists

The skip list is an elegant search-tree alternative introduced by Pugh [53]. Skip lists are randomized data structures having a weakly history independent pointer structure [31, 53].

Skip lists with N elements support searches, inserts, and deletes in $O(\log N)$ operations whp and range queries returning k elements in $O(\log N + k)$ operations whp [27, 42, 50]. They are heavily used in internal-memory algorithms [5, 8, 9, 28, 34, 37, 39, 49, 57].

This paper gives a simple and provably good external-memory history-independent skip list, which has high probability guarantees. Specifically, our skip list supports insert, deletes, and searches with $O(\log_B N)$ I/Os whp, and range queries returning k elements with $O(\log_B N + k/B)$ whp—which is just the search plus scan cost. Thus, our history independent, external memory skip list has high-probability bounds matching those of a B-tree.

Our challenge is to tweak the folklore B-skip list [1, 25, 26, 33] as little as possible (and in a history-independent way) so that we can achieve high-probability bounds for searches and inserts while maintaining optimal range queries.

1.3 Results

We begin by giving an efficient, history-independent packed memory array.

THEOREM 1. *There exists a weakly history-independent packed-memory array on N elements which can perform inserts and deletes in $O(\log^2 N)$ amortized element moves with high probability. This PMA requires $O(N)$ space, can perform inserts and deletes in amortized $O\big(\frac{\log^2 N}{B} + \log_B N\big)$ I/Os with high probability, and can perform a range query for k elements in $O(1 + k/B)$ I/Os.*

When $B = \Omega(\log N \log\log N)$ (reasonable on today's systems), then $\frac{\log^2 N}{B} = O(\log_B N)$, so inserts and deletes in our PMA have the same I/O complexity as in a B-tree.

Theorem 1 directly yields a history-independent cache-oblivious B-tree with these performance bounds:

THEOREM 2. *There exists a weakly history-independent, cache-oblivious B-tree on N elements which can perform inserts and deletes in $O\big(\frac{\log^2 N}{B} + \log_B N\big)$ amortized I/Os with high probability. This cache-oblivious B-tree requires $O(N)$ space and can answer a range query for k elements in $O(\log_B N + k/B)$ I/Os, i.e., the search plus the scan cost.*

We give a simple and provably good external-memory history-independent skip list, which has high probability guarantees analogous to in-memory skip lists.

THEOREM 3. *There exists a weakly history-independent, external-memory skip list on N elements which can perform look-ups in $O(\log_B N)$ I/Os with high probability. For a parameter $\varepsilon > 0$, the skip-list requires $O(\log_B N)$ amortized I/Os for inserts and deletes, with a worst case of $O(B^\varepsilon \log N)$ I/Os, all with high probability. This skip-list requires $O(N)$ space and can answer a range query for k elements in $O(\frac{1}{\varepsilon}\log_B N + k/B)$ I/Os with high probability.*

The parameter ε indicates a small trade-off between the worst-case rebuild cost after an update and the cost of medium-size range queries (see Section 6).

We contrast our data structure with the B-skip list, which promotes elements from one level to the next with probability $1/B$. We prove that with high probability, there exist at least $\Omega(\sqrt{NB})$ elements where the cost to search for any one of them is $O(\log \frac{N}{B})$. Thus, the high-probability I/O bounds for searching in a B-skip list are not asymptotically better than for searching in a regular (internal-memory) skip list that is implemented in external memory.

1.4 Related Work

History of History Independence

The history of history independence spans nearly four decades. The central notions of history independence predate its formalism.

One of the key ideas of history independence is unique representation, which was studied as far back as 1977 by Snyder [59]. Many uniquely represented data structures were published before the conception of history independence [4, 6, 7, 52–54, 59, 60]. Similar to unique representation, the idea of randomized structures that are *uniformly represented*, that is, have representations drawn from a distribution irrespective of the past history, emerged with Pugh's skip list [52, 53] and Aragon and Seidel's treap [7].

Nearly a decade later, Micciancio [43] defined ***oblivious data structures*** as those whose topology does not reveal the sequence of operations that led to the current state. In 2001, Naor and Teague [47] strengthened obliviousness to ***history independence*** ("anti-persistence"), generalizing it to include the entire *bit representation* of the structure, including memory addresses.

Hartline et al. [36] proved that a reversible data structure (i.e. one whose state graph is strongly connected) is SHI if and only if it fixes a ***canonical representation*** for each state depending only on initial (possibly random) choices made before any operations are performed.

Buchbinder and Petrank [23] further explored strong versus weak history independence, proving a separation between the two notions for heaps and queues in a comparison-based model.

History-independent data structures are well-studied when the objective is to minimize RAM computations. Examples include SHI hashing [20,46,47], dictionaries and order-maintenance [20], and history-independent data structures for computational geometry [21, 61].

Golovin [32, 33] began an algorithmic study of history-independent data structures in external-memory. First, Golovin proposed the B-treap [32], a strongly history-independent external-memory B-tree variant based on treaps [7].

Golovin notes that while the B-treap is a unique-representation data structure supporting B-tree operations with low overhead, from a practical point of view, it is complicated and difficult to implement [33]. Golovin thus proposes a strongly history-independent B-skip list [33] as a simpler alternative to the B-treap. This data structure achieves $O(\log_B N)$ I/Os in expectation for searches and updates and $O(k/B + \log_B N)$ I/Os in expectation for range queries returning k elements.

Golovin's B-skip list builds upon the folklore extension of skip lists (see e.g., [1, 25, 26, 33]): promote an element from one level to the next with probability $1/B$, rather than $1/2$. The folklore B-skip list's I/O bounds are only in expectation, and do not extend to good high probability bounds (see Lemma 15).

Other Applications of History Independence

The theoretical work on history independence in external memory complements the security and experimental work on history-independent data structures for persistent storage, such as file systems, cloud storage, voting systems and databases [10–12, 19, 44, 45, 56].

History independence has many applications in security and privacy. For example, history-independent data structures help to guarantee privacy in incremental signature schemes [43] and vote-storage mechanisms [19, 44, 45].

History-independent data structures often have nice properties. For example, skip lists [53] have weakly history-independent topologies, and are weight balanced [48] in a randomized sense. Canonical representations [36] find applications in other areas besides security, e.g. concurrent data structures [58], equality testing [60] and dynamic and incremental algorithms [2, 52].

2. PRELIMINARIES

An event E_n on a problem of size n occurs ***with high probability*** (***whp***) if $\Pr[E_n] \geq 1 - 1/n^c$ for some constant c. Often the event E_n is parametrized by some constant d, in particular, because the event is defined using Big Oh notation. (See, e.g., Theorem 11.) In this case, we can say more strongly that for every c, there is a d so that $\Pr[E_n] \geq 1 - 1/n^c$. We use high probability guarantees to bound a data structure's performance more tightly than can be done using expectation alone. Even if a data structure performs well in expectation, it may have a large number of poorly-performing operations (see, for example, Lemma 15).

Two instances I_1 and I_2 of a data structure are in the same ***state*** if they cannot be distinguished via any sequence of operations on the data structure. The ***memory representation*** of an instance I of a data structure is the bit representation of I, including data, pointers, unused buffer space, and all auxiliary parts of the structure, along with the physical addresses at which they are stored.

DEFINITION 4 (WEAK HISTORY INDEPENDENCE). *A data structure is weakly history independent (WHI) if, for any two sequences of operations X and Y that take the data structure from initialization to the same state, the distribution over memory representations after X is performed is identical to the distribution after Y.*

2.1 Building Blocks for History Independence

We use history-independent allocation [47] as a black box. We also use the weak history-independent dynamic arrays [36], rather than strongly independent dynamic arrays [36, 47].[3]

Weakly history-independent dynamic arrays take constant amortized time per update with high probability. The idea is to maintain the following invariants. For array A storing n elements: (1) the size $|A|$ is uniformly and randomly chosen from $\{n, \ldots, 2n-1\}$, and, (2) after each insert or delete resize with probability $\Theta(1/|A|)$.

2.2 Performance Advantages of WHI over SHI

We focus on weak history independence because of its performance benefits. In particular, weak history independence allows us to have high-probability guarantees in our data structures.

Strong history independence for reversible data structures requires a canonical representation [36].[4] While canonical representations are useful to have, maintaining them imposes strict limitations on the design and efficiency of the data structure, as argued by several authors [23, 36]. Moreover, amortization, strong history independence, and high-probability guarantees are largely incompatible.

In particular, SHI dynamic arrays cannot achieve the same with high probability guarantees as WHI dynamic arrays.

OBSERVATION 1. *No strongly-history-independent dynamic array can achieve $o(N)$ amortized resize cost per insert or delete with high probability.*

PROOF. Consider a strongly-history-independent dynamic array that needs to be strictly greater than 50% full. (The proof generalizes to arbitrary capacity constraints.) For integer k, the adversary chooses a random $\ell \in \{k, k+1, \ldots, 2k\}$. Then the adversary inserts up to ℓ elements into the array, and then alternates between adding and removing an element from the array, so that the array alternates between having ℓ and $\ell + 1$ elements.

Given the capacity constraints on the array, there must be at least two different canonical representations for the arrays of sizes $k, k+1, \ldots, 2k$. Thus, there is a probability of at least $1/k$ that the adversary forces an array resize (with cost $\Omega(N)$) on every insert and delete in the alternation phase.

Observe that k can be arbitrarily large. No matter how long the data structure runs, it cannot avoid an $\Omega(N)$ resize with probability greater than $1 - 1/k$. □

Most importantly, this observation applies to PMAs as well. A PMA with N elements at a given time is required to have size $\Theta(N)$, so it generalizes the dynamic array. That is, Observation 1 lets us conclude the following:

REMARK 1. *A strongly history-independent PMA cannot give any $o(N)$ amortized with high probability operation bounds. In contrast, our weakly history-independent PMA has a $O(\log^2 N)$ bound (Theorem 1).*

[3]Here it is assumed the contents of the dynamic array are stored (internally) in a history independent manner—thus the size of the array should not depend on the history of inserts and deletes.

[4]A data structure is reversible if the state-transition graph is strongly connected [36], which is true of all structures in this paper and most structures that support deletes.

Observation 1 similarly generalizes to other amortized data structures with large worst-case costs. Thus, while strong history independence provides stronger security guarantees, it can come at a high performance cost.

2.3 Oblivious Adversary and Oblivious Observer

Our performance analyses assume an ***oblivious adversary***, which determines the sequence of operations presented to the data structure. The oblivious adversary cannot see the outcomes of the data structure's coin flips nor the current state of memory. Said differently, the adversary is required to choose the entire sequence of operations before the data structure even starts running. The oblivious adversary is used for analyzing randomized structures such as skip lists [53] or treaps [7].

Our (weak) history-independence analyses assume an ***oblivious observer***. The observer cannot control the input sequence and does not see the data structure's coin flips. The observer gets to observe the data structure's memory representation once. We prove history independence by showing that for every state of the data structure, the distribution of memory representations is the same, regardless of how the data structure got to that state.

3. HISTORY-INDEPENDENT PACKED MEMORY ARRAY

This section gives a history-independent packed memory array (PMA). A PMA is an $\Theta(N)$-sized array that stores a sequence of elements in a user-specified order. There are up to $O(1)$ gaps between consecutive elements to support efficient insertions and deletions.

A PMA with N elements supports the following:

- `Query`(i, j)—return the ith through jth elements of the PMA, inclusive, where $0 \le i \le j < N$.
- `Insert`(i, x)—insert x as the ith element of the PMA, where $0 \le i \le N$. Elements with rank i through $N-1$ before the insert become the elements with $i+1$ though N after the insert.
- `Delete`(i)—delete the ith element of the PMA, where $0 \le i < N$.

The PMA's performance is given in Theorem 1.

3.1 High-Level Structure of HI PMA

Packed-memory arrays and sparse tables in the literature [14, 16–18, 38, 41, 66] are not history independent; the size of the array, densities of the subarrays, and rebalances depend strongly on the history.

We guarantee history independence for our PMA as follows. First, we ensure the size of the PMA is history independent. We resize using the HI dynamic array allocation strategy [36], as summarized in Section 2.1.

Next, we ensure that the N elements in the array of size $N_S = \Theta(N)$ are spread throughout the array according to a distribution (given below) that is independent of past operations.

We maintain this history-independent layout recursively. At the topmost level of recursion, we (implicitly) maintain a set of size $\Theta(N_S/\log N_S)$, which we call the ***candidate set***. The candidate set consists of elements that have rank $N/2 \pm \Theta(N_S/\log N_S)$. We pick a random element from the candidate set, which we call the ***balance element***. If the

balance element has rank r, then we recursively store the first $r-1$ elements in the first half of the array and the remaining $N-r-1$ elements in the second half of the array.

In general, when we are spreading elements out within a subarray A of the PMA, the candidate set has size $\Theta(|A|/\log N_S)$, and as before, the balance element is randomly chosen from this set. The base case is when $|A| = \Theta(\log N_S)$, at which point the elements are spread evenly throughout A.

Thus, how the elements are spread throughout the PMA depends only on the size N_S (which is randomly chosen as described in Section 2.1), the number of elements in the PMA N, and the random choices of all the balance elements.

We maintain the balance elements in each candidate set using a simple generalization of reservoir sampling [62] in which there are deletes, described below. In this particular instance of reservoir sampling, the size of a candidate set at any given level of recursion stays the same, unless the size of the entire PMA changes.

See Figure 1 for an illustration of our PMA. The top part represents how each level of recursion partions the elements into ranges. We show repeated elements across levels to aid visualization; they are only stored once in the data structure (at the bottom level). The division of elements into ranges at each level helps in maintaining the balance elements, which are stored in a separate structure.

3.2 Reservoir Sampling with Deletes

We first review a small tweak on standard reservoir sampling [62], ***reservoir sampling with deletes***, which we use to help build the PMA.

Game: We have a dynamic set of elements. The objective is to maintain a uniformly and randomly chosen ***leader*** of the set, where each element in the set has equal probability of being selected as leader. In other words, we are interested in reservoir sampling with a reservoir of size 1.

Since the set is dynamic, at each step t, an element may be added to or deleted from the set. The adversary is oblivious, which means that the input sequence cannot depend on the particular element chosen as leader.

We can maintain the leader using the following technique. Let n_t denote the number of elements in the set at time t (including any newly-arrived element). Initially, if the set is nonempty, we choose the leader uniformly at random. When a new element y arrives, y becomes the leader with probability $1/n_t$; otherwise the old leader remains. When an element is deleted, there are two cases. If that element was the leader, then we choose a new leader uniformly at random from the remaining elements in the set. If the deleted element was not the leader, the old leader remains.

LEMMA 5 ([62]). *At any time step t, if there are n_t elements in the pool, then each element has a probability $1/n_t$ to be the leader.*

3.3 Detailed Structure of the HI PMA

The size N_S of our history-independent PMA is a random variable that depends on the number of elements N stored in the PMA (similar to dynamic arrays in Section 2.1). We select parameter $\hat{N}$ randomly from $\{N, \ldots, 2N-1\}$, and N_S is a function of $\hat{N}$ (as described below).

We view the PMA as a complete binary tree of ranges, where a ***range*** is a contiguous sequence of array slots. This tree has height $h = \lceil \log \hat{N} - \log\log \hat{N} \rceil$. The root is the entire PMA and has depth 0. The leaves in the binary tree are ranges comprising $\lceil C_L \log \hat{N} \rceil$ slots, and C_L is a constant to be determined later. Thus, the PMA has a total of $N_S = 2^h \lceil C_L \log \hat{N} \rceil \le (2C_L+1)\hat{N} = \Theta(N)$ slots.

Consider a range R in the binary tree with left child R_1 and right child R_2. Recall from Section 3.1 that the ***balance element*** b_R of R is the first element of R_2; all the elements in R of smaller rank than b_R are stored in R_1. The values of b_R for each range/node are stored in a separate tree.

For each non-leaf range R at depth d, define the ***candidate set*** M_R to be the $\lceil c_1 \hat{N} 2^{-d} / \log \hat{N} \rceil$ middle elements of R. More precisely, if R holds ℓ elements, then we fix the size of M_R and set the first element of M_R to be the $1 + \lceil \ell/2 \rceil - \lceil |M_R|/2 \rceil$th element of R.

Our PMA is parameterized by a constant $0 < c_1 < 1 - 6/\log \hat{N}$.[5] A larger c_1 reduces the amortized update time and increases the space. We require $C_L \ge 1 + c_1 + 6/\log \hat{N}$. The value of C_L and c_1 need not change as $\hat{N}$ changes—values such as $c_1 = 1/2$ and $C_L = 2$ will work for sufficiently large $\hat{N}$ (over 4096 in this case).

3.4 Dynamically Maintaining Balance Elements

As elements are inserted into or deleted from the PMA, the candidate set M_R for some range R could change. This change might be caused by a newly inserted or deleted element that belongs to M_R or just because insertions at one end of R cause the median element of R to change.

As the candidate set M_R changes, we maintain the invariant that the balance element b_R is selected uniformly and randomly from M_R. (In particular, this means that b_R is selected history-independently.) We use reservoir sampling with deletes as the basis for maintaining this invariant.

INVARIANT 6. *After each operation, for each range R, balance element b_R is uniformly distributed over the candidate set M_R.*

Whenever one element leaves their candidate set, another one joins, since the candidate set size of each range is fixed between rebuilds of the entire PMA. Thus, we describe how to maintain the candidate set when exactly one element is added, and one leaves.

If the balance element is the element leaving the candidate set, we select the new balance element uniformly at random. Otherwise, when a new element enters the candidate set, it has a $1/|M_R|$ chance of becoming the new balance element (as in reservoir sampling).

When the balance element of a range changes, we rebuild the entire range and all of its subranges. Rebuilding a range of $|R|$ slots can be done in $\Theta(|R|)$ time; see Lemma 10.

3.5 Detecting Changes to the Candidate Set

In order to determine how inserts and deletes affect the candidate set of a range R, we need to know the rank of the element being inserted or deleted, the current candidate set of R, and the rank of the current balance element of R.

The rank of the element being inserted and deleted is specified as part of the insert or delete operation.

To compute the other information, our PMA maintains an auxiliary data structure containing the number of elements

[5] When $\hat{N} \le 64$, no such c_1 exists. For such small $\hat{N}$, we use a dynamic array instead.

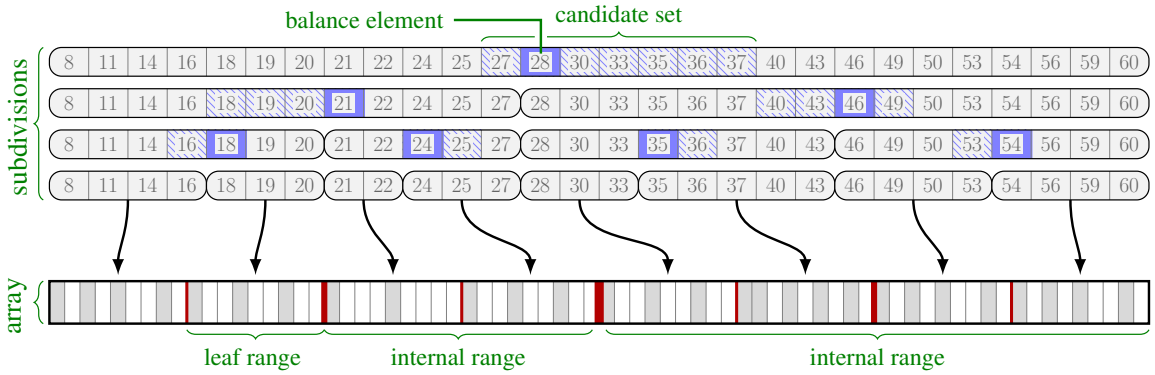


Figure 1: Illustration of our PMA showing the subdivisions of elements into ranges. In each range, the balance element is framed and the candidate set is hatched. The actual array is represented in the bottom, where occupied slots are shaded. The candidate-set size at any given level does not depend on the range. The rank of each balance element is stored in a separate tree.

ℓ_R in each range R. During an insert or delete, as we descend the tree of ranges, we keep track of the ranks of the first and last element in each range as follows. Suppose we are at range R whose first element has rank x. If we descend to R_1, then we know the rank of the first element of R_1 is also x. If we descend to R_2, then the rank of its first element is $x + \ell_{R_1}$. Given the rank of the first element of a range and the number of elements in that range, it is easy to compute its candidate set. Note also that the rank of the balance element of a range R whose first element has rank x is $x + \ell_{R_1}$.

We need to store ℓ_R for each range so that it can be accessed efficiently, both in terms of operations and I/Os. Since the ranges form a complete binary tree, we store the numbers ℓ_R in a binary tree organized in a Van Emde Boas layout (see [14, 51]). We call this auxiliary data structure the ***rank tree***.

The Van Emde Boas layout is a deterministic, static, cache-oblivious—and hence history-independent—layout of a complete binary tree. It supports traversing a root-to-leaf path in $O(\log N)$ operations and $O(\log_B N)$ I/Os. Thus, the rank tree is history independent.

Whenever the size of a range changes due to an insert or delete, we update the corresponding entry in the rank tree. Whenever we rebuild a range R in the PMA, we update all entries of the rank tree corresponding to descendants of R. Whenever we rebuild the entire PMA, we rebuild the entire rank tree.

4. CORRECTNESS AND PERFORMANCE

4.1 Balance-Element Structural Lemmas

First, we show correctness—the data structure always finds a slot for any element it needs to store.

LEMMA 7. *At all times, the size of a range is larger than the number of elements it contains.*

PROOF. Consider a range at depth d. This range has $N_S/2^d = 2^{h-d}\lceil C_L \log \hat{N} \rceil \geq C_L \hat{N}/2^d$ slots. We will show that the maximum number of elements it can contain is smaller than this number of slots.

The number of elements in the range is at most half of the elements in its parents range, plus half of the size of the parent's candidate set (rounding up). In other words, if $S(d)$ is the maximum number of elements in a range at depth d, $S(0) \leq \hat{N}$ and

$$\begin{aligned} S(d) &\leq \lceil S(d-1)/2 \rceil + \lceil M_R/2 \rceil \\ &\leq \lceil S(d-1)/2 \rceil + \frac{1}{2}\left\lceil \frac{c_1 \hat{N}}{2^{d-1} \log \hat{N}} \right\rceil + \frac{1}{2} \\ &\leq \frac{S(d-1)}{2} + \frac{c_1 \hat{N}}{2^d \log \hat{N}} + \frac{3}{2}. \end{aligned}$$

By induction,

$$S(d) \leq \frac{\hat{N}}{2^d} + \frac{c_1 d \hat{N}}{2^d \log \hat{N}} + 3.$$

Since $d \leq \log \hat{N}$, and $\hat{N}/2^d \geq \hat{N}/2^h \geq (\log \hat{N})/2$,

$$S(d) \leq \frac{\hat{N}}{2^d}(1 + c_1) + 3 \leq \frac{\hat{N}}{2^d}\left(1 + c_1 + \frac{6}{\log \hat{N}}\right).$$

Since we choose $C_L \geq 1 + c_1 + 6/\log \hat{N}$, we have $S(d) \leq C_L \hat{N}/2^d$. □

As Lemma 7 establishes, each leaf range in the PMA (which has $\Theta(\log N)$ slots) never fills up completely. Using a similar argument, it can be shown that each leaf also contains $\Omega(\log N)$ elements if $c_1 < 1 - 6/\log \hat{N}$. Because the elements are spread out evenly in the leaves, this implies there are $O(1)$ gaps between two consecutive elements.

LEMMA 8. *If $c_1 < 1 - 6/\log \hat{N}$, the leaves are always constant-factor full. There is $O(1)$ space between two consecutive elements in the array.*

Next, we prove weak history independence. As mentioned in Section 3.1, when we are spreading elements out within a subarray A of the PMA, the balance element is randomly chosen from the candidate set. The elements of A are recursively split between its children according to this balance element. The base case is when $|A| = \Theta(\log \hat{N})$, at which point the elements are spread evenly throughout A.

Thus, how the elements are spread throughout the PMA depends only on $\hat{N}$ (and the related N_S), the number of elements in the PMA N, and the random choices of all the balance elements. This immediately gives weak history-independence, as formalized in the following lemma.

LEMMA 9. *This PMA is weakly history independent.*

PROOF. We show that the memory representation of the PMA is based only on N and some randomness (in particular, the random choices made during balance element selection and the random choice of $\hat{N}$).

Let the PMA contain a set of elements S of size N, with a set of balance elements P. Then P partitions the elements of S into leaf ranges.

Since the elements are evenly spaced in each leaf range, the position of each element within the leaf is determined by the number of elements in that leaf. Since S is partitioned into leaf ranges by P, P determines the position of each element in the PMA. Thus P, $\hat{N}$, and N determine the memory representation of the data structure. By Invariant 6, P is selected from a distribution based only on N and $\hat{N}$.

Thus, any two sequences of operations X and Y that insert S into the PMA result in the same distribution on P, and the same distribution on memory representations. □

4.2 Proving the Performance Bounds

We begin by bounding the cost of a rebalance. Then we bound the total number of rebalances.

LEMMA 10. *Rebuilding a range R containing $|R|$ slots takes $O(|R|)$ RAM operations and $O(|R|/B+1)$ I/Os.*

PROOF. The algorithm first recursively chooses the balance elements for all ranges contained in this range R and updates them in the rank tree; this takes $O(|R|)$ time and $O(|R|/B + \log_B |R| + 1) = O(|R|/B + 1)$ I/Os. Then, all elements in R are gathered, and inserted into the appropriate leaf range using a sequence of linear scans. □

Our goal is to bound the cost of our PMA operations. Specifically, we want to show Theorem 11.

THEOREM 11. *Consider k (not necessarily consecutive) operations on a PMA during which its maximum size is N_M, its minimum size is $\Omega(N_M)$, and $k = \Omega(N_M)$. The amortized cost of these operations is $O(\log^2 N_M)$ with high probability with respect to k: in other words, these k operations require $O(k \log^2 N_M)$ total RAM operations with high probability.*

Before proving this theorem, we need some supporting lemmas and definitions.

DEFINITION 12. *A rebuild that is not charged to a range R is called a* **free rebuild**. *Free rebuilds come from two sources: (1) they are rebuilds of an ancestor range (whose cost is charged to the ancestor), or (2) they are triggered by the interstitial operations between the nonconsecutive operations of Theorem 11.*

We further categorize non-free rebuilds by their causes. An **out-of-bounds rebuild** *is a rebuild caused by the pivot leaving the candidate set. A* **lottery rebuild** *is a rebuild caused by deleting the pivot or by inserting into the candidate set an element that becomes a new pivot.*

Gearing up to Lemma 13, we concentrate on the cost of all rebalances at a single depth d. Let M_d be the size of the candidate set at depth d.

We give a lower bound on the probability that two out-of-bounds rebuilds happen in quick succession. This lemma holds regardless of the number of free and lottery rebuilds that happen in between.

LEMMA 13. *After any rebuild of a range R at depth d, consider a sequence of t operations on R, for any $t \in \{1, \ldots, \lfloor M_d/2 \rfloor\}$, with arbitrary free and lottery rebuilds occurring during these operations. The probability $p(t)$ that no out-of-bounds rebuild happens during these t operations is at least $1 - 2t/M_d$.*

PROOF. Let $p_i(t)$ be the probability that no out-of-bounds rebuild happens in the first t time steps, given that exactly i free and lottery rebuilds happen during the first t time steps. We prove the lemma by induction on i.

Define the ***guard number*** as the number of elements between the pivot and the closest endpoint of the candidate set.

First, the base case: for any t, $p_0(t)$ is at least the probability that the guard number after a rebuild is larger than t. Indeed, the balance element cannot be moved closer to an endpoint of the candidate set by more than one element per operation. As the pivot is sampled uniformly after any type of rebuild, we have

$$p_0(t) \geq \Pr[\text{guard number} > t] \geq 1 - 2t/M_d.$$

Now, assume by induction that $p_i(t) \geq 1 - 2t/M_d$ and we want to show $p_{i+1}(t) \geq 1 - t/M_d$. Let t' be the last time step before the $(i+1)$st non-out-of-bounds rebuild.

The following conditions ensure that there are no out-of-bounds rebuilds in the first t operations:

1. there is no out-of-bounds rebuild in the first t' operations, and
2. there is no out-of-bounds rebuild in the subsequent $t - t'$ operations.

These two events are independent since there is a fixed (free or lottery) rebuild between them. The first occurs with probability $p_i(t')$ and the second with probability $p_0(t - t')$. Thus, we have

$$\begin{aligned} p_{i+1}(t) &\geq p_i(t')\, p_0(t - t') \\ &\geq (1 - 2t'/M_d)\,(1 - 2(t - t')/M_d) \geq 1 - 2t/M_d, \end{aligned}$$

and the induction is complete. □

DEFINITION 14. *Consider an out-of-bounds rebuild of a range R at depth d.*

We call this rebuild a **good out-of-bounds rebuild** *if R has only free and lottery rebuilds for the next $M_d/4$ operations.*

By Lemma 13, an out-of-bounds rebuild is good with probability at least $1/2$.

We are now ready to prove Theorem 11.

PROOF OF THEOREM 11. Consider the sequence of rebuilds of ranges at a given depth d. We analyze lottery rebuilds and out-of-bounds rebuilds separately. Since variations in $\hat{N}$ slightly change the candidate set size, let M denote the smallest candidate-set size at depth d over the k operations. However, since $N = \Theta(N_M)$ at all times, $M = \Theta(|M_d|)$.

We give a (weak) bound on k/M which helps show that the high-probability bounds hold with respect to k.

In particular,

$$k/M \geq \frac{k \log N_M}{N_M} \geq \frac{k \log k \log N_M}{N_M \log k} = \Omega(\log k),$$

since $k/\log k = \Omega(N_M/\log N_M)$ if $k = \Omega(N_M)$.

Each operation has probability at most $1/M$ of causing a lottery rebuild. Thus, there are k/M lottery rebuilds in expectation. Then using Chernoff bounds, the probability that we have more than $(1+\delta)k/M$ rebuilds is less than $e^{-\delta k/3M}$. Recall that $k/M = \Omega(\log k)$. Substituting, there are $O(k/M)$ lottery rebuilds with high probability.

Now, we bound the out-of-bounds rebuilds. By the pigeonhole principle, there cannot be more than $k/(M/4) + 2^d = O(k/M)$ good out-of-bounds rebuilds (the second term comes from the number of ranges at depth d).

We bound how many bad out-of-bounds rebuilds can happen before reaching this limit on good out-of-bounds rebuilds. Any out-of-bounds rebuild is good with probability at least $1/2$. Then after $k/M = \Omega(\log k)$ out-of-bounds rebuilds, we obtain $\Theta(k/M)$ good out-of-bounds rebuilds with high probability, again by Chernoff bounds. As we can only get $O(k/M)$ good rebuilds, we have $O(k/M)$ out-of-bounds rebuilds in total.

Therefore, every k operations, there are $O(k/M)$ out-of-bounds and lottery rebuilds of ranges at depth d with high probability. Each rebuild costs $O(M \log N_M)$ RAM operations by Lemma 10 (because that is the number of slots in a range at depth d). Thus, the total rebuild cost for depth d is $O(k \log N_M)$.

Having determined the cost of rebalancing at each depth, we account for the total cost. The amortized rebalance cost, summing over all $h = O(\log N_M)$ levels, is $O(\log^2 N_M)$.

Each resize costs $O(N_M)$ and occurs with probability $O(1/N_M)$ after every insertion. Using Chernoff bounds, there are $O((k \log N_M)/N_M) = \Omega(\log k)$ resizes after k operations with high probability, leading to an additional amortized cost of $O(\log N_M)$.

Finally, each insert or delete requires extra bookkeeping: we must find the appropriate leaf range to insert the element by traversing the rank tree. This traversal takes $O(\log N_M)$ RAM operations, and rebuilding the leaf so that the elements are still evenly spaced takes $O(\log N_M)$ RAM operations.

This gives a total of $O(k \log^2 N_M)$ total RAM operations with high probability.

Using similar analysis, we can also bound the I/O performance. Recall that rebalancing a range of size R takes $O(R/B + 1)$ I/Os, and traversing a tree in the Van Emde Boas layout requires $O(\log_B N_M)$ I/Os. Carrying these terms through the above proof gives the desired bounds. □

To complete the proof of Theorem 1, we need to extend this analysis to handle the PMA changing size significantly.

PROOF OF THEOREM 1. We partition the $k = \Omega(N)$ operations on the PMA into types based on the size of the PMA. In particular, let $\hat{N}_t$ be the value of $\hat{N}$ during the tth operation. Then operation t is of type 0 if $N \geq \hat{N}_t > N/2$, type 1 if $N/2 \geq \hat{N}_t > N/4$, and type i if $N/2^i \geq \hat{N}_t > N/2^{i+1}$ for $0 \leq i \leq \lceil \log N \rceil$.

We analyze each type of operations as a whole, and bound its total cost, summing to $O(k \log^2 N)$ RAM operations in total. Each type is analyzed using two cases.

First, consider a type i which has at least $\sqrt{N}$ total operations. Then by Theorem 11, these operations take amortized $O(\log^2 N)$ RAM operations with high probability.

Second, consider a type i which has less than $\sqrt{N}$ operations. We call the operations of these types ***small-type operations***. We show that the total cost of all small-type operations is a lower-order term.

Since the PMA begins as empty, and each operation can only insert one element, there are at least $N/2^{i+1}$ operations of type i. Thus, each small-type operation t has $\hat{N}_t \leq \sqrt{N}$.

Then overall, a type which has less than $\sqrt{N}$ operations must operate on a PMA of size $\leq \sqrt{N}$. Thus, each type has total cost $O(N)$. Summing over the $O(\log N)$ such types, we get a worst-case total cost of $O(N \log N)$ for small-type operations; amortizing gives a cost of $O(\log N)$ RAM operations.

Thus the total amortized cost is $O(\log^2 N)$ with high probability with respect to N. The I/O cost to rebuild range R is $|R|/B$ by Lemma 10; carrying this term through the above analysis, we obtain the desired I/O bounds. □

4.3 Experimental Results

We implemented a normal PMA and our history-independent PMA. We found that while there was approximately a factor of 7 overhead in the run time, the asymptotic performance matched our analysis.

We also examined the number of element moves required during an insert. Figure 2 shows the number of moves required divided by $N \log^2 N$ vs. the number of elements inserted. The linear nature of this data supports our theoretical analysis.

These tests were run on a Dell server with an Intel Xeon processor (E5-2450 @ 2.10GHz). In these tests, inserting 100 million random numbers took approximately 23 minutes. Additionally the space overhead ranged from 1.8 to 5 times the number of elements.

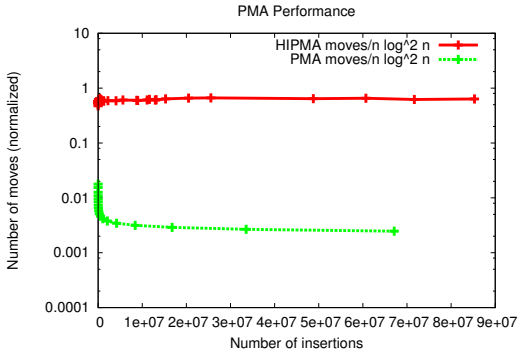


Figure 2: Experimental results of runtime for random inserts on normal and history-independent PMAs.

The history-independence of our PMA depends on the balance elements being uniformly distributed across the candidate set; see Lemma 5. To test this, we inserted values of 1-100,000 sequentially into a history-independent PMA and recorded the position of the balance for each range within this PMA where the candidate set size was eight or greater. We did this test 10,000 times and used the χ^2 goodness-of-fit test to compare these balance elements with a uniform

distribution. To ensure enough samples we only looked at ranges where the expected count for each bucket was ten or greater. This resulted in 148 p-values. If our null hypothesis, that these balances are uniformly distributed, holds, these p-values themselves should be uniformly distributed. Running the χ^2 goodness-of-fit test across these p-values showed a result consistent with data coming from a uniform distribution (p=0.47, n=148). As a result we can say there is no statistically significant evidence of that our balance elements have a deviation from a uniform distribution.

5. HISTORY-INDEPENDENT CACHE-OBLIVIOUS B-TREE

In this section we prove Theorem 2, which establishes the performance of our history-independent cache-oblivious B-tree. This N-element data structure, must, without knowledge of the block size B,

- insert or delete items with $O((\log^2 N)/B + \log_B N)$ amortized I/Os with high probability,
- use $O(N)$ space, and
- answer range queries containing k elements in $O(\log_B N + k/B)$ I/Os.

When $B = \Omega(\log N \log\log N)$, which is reasonable on today's systems, $\frac{\log^2 N}{B} + \log_B N = O(\log_B N)$, so that our history-independent cache-oblivious B-tree matches the I/O complexity of a standard B-tree.

The requirements above are similar to those achieved by our PMA. The key difference is that PMA items are searched by rank (before being inserted, deleted, or completing a range search) rather than value.

By slightly augmenting our PMA, we obtain a cache-oblivious B-tree achieving the above bounds. We call our data structure the ***augmented PMA***.

The augmented PMA has an additional, static-topology tree associated with it. Recall that our PMA has the sizes of each range stored in a complete binary tree, in a Van Emde Boas layout. We store an additional tree storing the *values* of each balance element. The two trees are identically structured, and identically maintained. Thus, this leads to only a constant factor increase in both space and running time. To search (by value) in the augmented PMA, we traverse a path in the new tree of balance-element values. This costs $O(\log_B N)$ I/Os and $O(\log N)$ operations. Once we have found the element, we can determine its rank by traversing the rank tree, summing the sizes of any left children each time we go to a right child.

Once the rank of the element is known, we insert, delete, or perform range queries as in the normal PMA, establishing the desired bounds.

6. HISTORY-INDEPENDENT EXTERNAL-MEMORY SKIP LIST

In this section we prove Theorem 3. We give a history-independent external-memory skip list.

In-memory skip lists support updates and queries in $O(\log N)$ time whp. The natural extension to external memory [1, 25, 26, 33] promotes elements with probability $1/B$ rather than probability $1/2$. We prove that for this extension, the high-probability I/O bounds are asymptotically no better than those of an in-memory skip list run in external memory (Lemma 15).

We build an external-memory skip list with good (i.e., B-tree-like) high-probability bounds for searches, updates, and range queries, while retaining the structure of the folklore B-skip list as much as possible.

6.1 High-Level Structure of the HI External-Memory Skip List

First, we describe why the folklore B-skip list fails to achieve high-probability I/O bounds. Then, we present the approach used in our skip list.

Golovin and others [1, 25, 26, 33] promote elements with probability $1/B$ (rather than $1/2$, as in the in-memory skip list). Consider an ***array***, a sequence of contiguous elements at any level that have not been promoted to the next level (i.e., lie between two ***promoted elements***). These arrays can have size $O(B \log N)$ whp, which is $O(\log N)$ times larger than the expected length. When this list is embedded in an array (analogous to the nodes in a B-tree), then a scan to search for these elements costs $O(\log N)$ I/Os whp.

We change the promotion probability to $1/B^\gamma$, where $1/2 < \gamma < 1 - \log\log B/\log B$. Now, all arrays have size $O(B^\gamma \log N)$ whp, so searches and updates take $O(B^\gamma \log N/B) = O(\log_B N)$ I/Os whp.

While a promotion probability of $1/B^\gamma$ results in fast searches and updates, it slows down range queries. If we pack arrays at the leaf level (the ***leaf arrays***) of the skip list into disk blocks [33], then most disk blocks will be underutilized, containing only B^γ elements on average. Thus, a range query returning k elements may require $O(\log_B N + k/B^\gamma)$ I/Os, which is worse than the target of $O(\log_B N + k/B)$ I/Os.

For efficient range queries, we pack multiple arrays into disk blocks at the leaf level as follows. Contiguous arrays, delimited by elements promoted twice, are packed together into a ***leaf node***. A leaf node is stored consecutively on disk; see Figure 3.

The packing strategy described above permits some leaf nodes to get too large, achieving size $\Theta(B^{2\gamma} \log N)$. If we pack elements as densely as possible, then every new insert would require rewriting the entire node at a cost of $O(B^{2\gamma-1} \log N)$ I/Os, which is worse than $\Theta(\log_B N)$ I/Os.

Similar to PMAs and HI dynamic arrays [36], we leave empty spaces between the elements in the leaves to support efficient inserts. We do so in a way that maintains history independence; see Invariant 16.

6.2 Detailed Structure of the HI External-Memory Skip List

A skip list S is a series of linked lists $\{S_0, S_1, \ldots, S_h\}$, where $S_0 \supseteq S_1 \supseteq \ldots \supseteq S_h$. Let $H = \{0, 1, \ldots, h\}$ be the ***levels*** of the skip list, and h the ***height*** of the skip list. The base list S_0 contains all the elements in the skip list; we call level 0 the ***leaf level***.

Each S_i, for $0 < i \leq h$, stores a sorted subset of the elements from S_{i-1}, along with the special element `front` to mark the beginning of the list; see Figure 3. The function $\mathit{level} : S_0 \to H$ determines the highest list that contains an element x, that is, if $\mathit{level}(x) = i$ then $x \in S_j$ for all $j \leq i$ and $x \notin S_k$ for $i < k \leq h$.

The height of each element is determined randomly, according to ***promotion probability*** p. Specifically, for $i \in H - \{0\}$, if an element is in S_{i-1}, it is also in S_i with probability p. Thus, $\mathit{level}(x)$ of an element x is the number of coin flips before we see a tail, when using a biased coin with probability p of flipping a head.

Next, we show that the folklore B-skip list, which has a

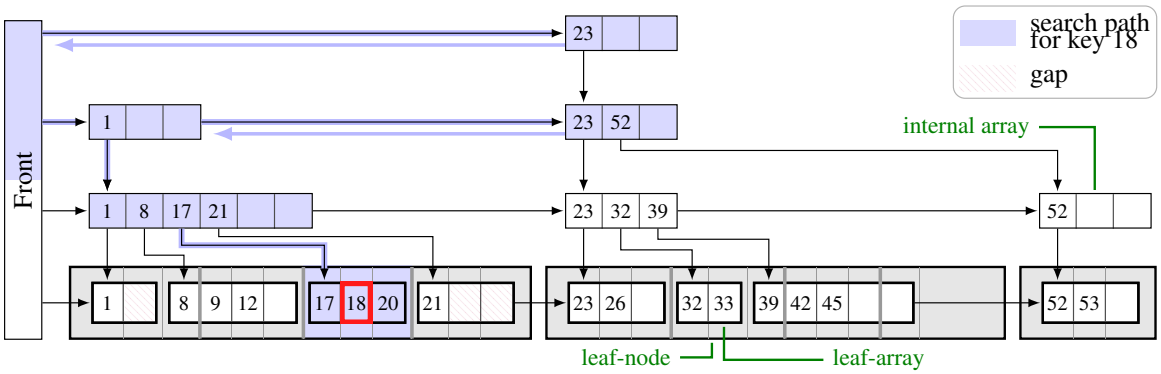


Figure 3: Illustration of a search path for element key 18 in the external skip list ($B = 3$ and $p = 1/2$).

promotion probability of $1/B$, performs poorly for some elements with high probability.

LEMMA 15. *In a B-skip list with promotion probability $p = 1/B$, there exist $\Omega(\sqrt{NB})$ elements with search cost $\Omega(\log(N/B))$ with high probability with respect to N/B.*

For our history-independent external-memory skip list, we choose promotion probability $p = 1/B^\gamma$, where $1/2 < \gamma \leq 1 - \log\log B/\log B$ is a constant. Let $1/p = B^\gamma$ be integral to simplify analysis. We parameterize our running times by $\varepsilon > 0$, with $\gamma = (\varepsilon + 1)/2$.

The parameter γ can be tuned—there is a trade-off between the cost of a range query and the worst-case cost of insertion. Specifically, a range query returning k elements has a cost of $O(\frac{1}{\varepsilon}\log_B N + k/B)$, while the worst-case insert cost is $O(B^\varepsilon \log N)$ I/Os. The expected insert cost remains $O(\log_B N)$ I/Os for all allowed values of γ.

We now describe how to partition the elements into arrays at nonleaf levels. We also describe how to pack the leaf arrays into leaf nodes.

Partitioning Non-Leaf Levels. We partition the list S_i at each level i for $1 \leq i \leq h$ into ***arrays***. The array at level i starts with a ***promoted element***, i.e., element x with *level* $(x) \geq i + 1$. It contains all elements up to (and not including) the next promoted element. The ***size*** of an array is the number of elements stored in it plus any empty slots. We maintain these sorted arrays history independently [36].

Partitioning the Leaf Level. We store the leaf level compactly to support I/O-efficient range queries. The ***leaf arrays*** at the leaf level are packed into a ***leaf node***. Formally, a leaf node $\mathcal{B}$ is a set of contiguous leaf arrays starting at some element x that has been promoted twice, that is, *level* $(x) \geq 2$.

We store the leaf arrays history independently (see Section 2.1), with the following modification: even when a leaf array has n elements with $n \leq B^\gamma$ elements, we maintain its size $n_s \geq B^\gamma$. We call the extra $n_s - n$ array slots ***gaps***. This modification retains history independence.

INVARIANT 16. *Let n be the number of elements in a leaf array of total size n_s then:*
- *If $n \leq B^\gamma$, then n_s is uniform in $[B^\gamma, 2B^\gamma - 1]$.*
- *If $n \geq B^\gamma$, then n_s is uniform in $[n, 2n - 1]$.*

Searches, Insertions, and Deletions. Search is implemented exactly as in a standard skip list: for an element y, start at the top list S_h, and scan right till a value $x > y$ is reached, in which case, descend a level down and continue till y is found or shown not to exist.

To insert or delete an element y, search for the leaf array where y belongs and insert or delete it. This involves shifting elements in the leaf array and causes an array resize with probability $O(1/B^\gamma)$ by Invariant 16. If a resize occurs, rebuild the entire leaf node containing the array.

When inserting y, determine *level* $(y) = \ell$ by tossing a biased coin with probability of heads p. At levels $1 \leq i < \ell$, y starts an array, splitting the existing array into two. If $\ell \geq 2$, y starts a leaf node, splitting the existing leaf node into two.

When deleting y, at levels $1 \leq i < \ell$, merge the leaf array that y started with its predecessor. If $\ell \geq 2$ then merge the leaf node that y started with its predecessor.

6.3 History Independence of External-Memory Skip List

The history independence of our external-memory skip list follows immediately from the following facts:
- *level* (x) for each element x is generated randomly,
- the elements within an array appear in sorted order,
- the size of each array is chosen history-independently (by Invariant 16 for leaf arrays and [36] for non-leaf arrays),
- within a leaf node, the leaf arrays are packed contiguously in sorted order, and,
- each array is allocated in blocks of size $\Theta(B)$ history-independently [47].

6.4 Performance Analysis

We bound the height of the external-memory skip list. The proof is similar to standard skip lists.

LEMMA 17. *An external-memory skip list with promotion probability p has height $h = O(\log_{1/p} N)$ whp.*

PROOF. For any element x, the probability that its level is more than $O(\log_{1/p} N)$ is given by: $\Pr[\textit{level}\,(x) \geq c\log_{1/p} N] \leq p^{c\log_{1/p} N} = 1/N^c$.

Applying the union bound we get, $\Pr[\forall x, \textit{level}\,(x) \geq c\log_{1/p} N] \leq N(1/N^c) = 1/N^{c-1}$. □

Thus, the height of our external-memory skip list with $p = 1/B^\gamma$ is $O(\log_B N)$.

Search. To analyze searches, we bound the size of the arrays and leaf nodes.

An array contains elements between two consecutive promoted elements. Thus, the size of an array is bounded by the length of the longest sequence of tails, when flipping a biased coin with $\Pr[head] = 1/B^\gamma$, which is $O(B^\gamma \log N)$.

LEMMA 18. *The number of I/Os required to perform a search is $O(\log_B N)$ with high probability.*

PROOF. Similar to the *backward analysis* in standard skip lists [53], examine the search path from bottom up starting at the leaf level. Each element visited by the path was either promoted and the search path came from the top, or was not promoted and the search path came from the left. The number of down moves is bounded by the height $h = c \log_B N$ (Lemma 17). At each level, the search path traverses at most two arrays.

The total length of the arrays touched by the search path is bounded by the number of coin flips required to obtain $c \log_B N$ heads, where $\Pr[head] = 1/B^\gamma$. We need $O(B^\gamma \log N)$ coin flips whp. Thus, the number of I/Os during the search at nonleaf levels is $O(\log N/B^{1-\gamma} + \log_B N) = O(\log_B N)$ whp, since $\gamma \leq 1 - \log\log B/\log B$. At the leaf level, the search scans one leaf array, which costs $O(B^\gamma \log N/B) = O(\log_B N)$. □

Insert. To bound the insert cost, we bound the cost of rebuilding a leaf node. The size of a leaf node is the number of elements stored in it plus the number of gaps.

The number of elements in a leaf node is bounded by the length of the longest sequence of tails in N coin flips with $\Pr[head] = p^2 = 1/B^{2\gamma}$, which is $O(B^{2\gamma} \log N)$ whp.

The number of gaps between any k consecutive leaf elements can be shown by a Chernoff bound argument to be $O(k + B^\gamma \log N)$. Thus, the size of a leaf node is $O(B^{2\gamma} \log N)$ whp. Rebuilding it costs $O(B^{2\gamma} \log N/B) = O(B^\varepsilon \log N)$ I/Os.

LEMMA 19. *The cost of performing an insert or delete operation is amortized $O(\log_B N)$ I/Os whp, with a worst case cost of $O(B^\varepsilon \log N)$ I/Os whp.*

PROOF. When an element y is inserted or deleted, the splits and merges at levels $1 \leq i < \mathit{level}(y)$ are dominated by the search cost of $O(\log_B N)$.

The cost of inserting in a leaf array is dominated by the cost of rebuilding the leaf node: $O(B^\varepsilon \log N)$ I/Os whp. The rebuild occurs with probability $O(1/B^\gamma)$. The amortized I/O cost is then $O(B^{2\gamma-1} \log N/B^\gamma) = O(\log N/B^{1-\gamma}) = O(\log_B N)$ in expectation and whp with respect to the number of operations, since $\gamma \leq 1 - \log\log B/\log B$. □

Range Query. To analyze a range query on a range of size k, we bound the number of leaf nodes across which the k elements are spread. That is, we bound the number of heads obtained on k biased coin flips with $\Pr[head] = 1/B^{2\gamma}$.

LEMMA 20. *The number of leaf nodes across which k consecutive leaf elements are stored is $O(\frac{1}{\varepsilon}\log_B N + k/B)$ with high probability.*

LEMMA 21. *A range query returning k elements costs $O(\frac{1}{\varepsilon}\log_B N + k/B)$ I/Os with high probability.*

PROOF. We break the analysis into several cases depending on the source of the cost.

- If the k consecutive elements fit in a single leaf node and there are no gaps, then the size of each such leaf array is $O(B^\gamma \log N)$ whp. Thus, $O(B^\gamma \log N/B + k/B) = O(\log_B N + k/B)$ I/Os suffice.
- If the k consecutive elements span across arrays with gaps. The sum of sizes of the gaps between these elements is $O(k + B^\gamma \log N)$. Thus, a range query takes $O(k/B + B^\gamma \log N/B) = O(\log_B N + k/B)$ I/Os.
- If the k consecutive elements span several leaf nodes, by Lemma 20 they span $O(k/B + \frac{1}{\varepsilon}\log_B N)$ leaf nodes. Every time the range query scan crosses a leaf node boundary, we incur an I/O to bring in the next leaf node, which requires $O(k/B + \frac{1}{\varepsilon}\log_B N)$ I/Os.

Thus, overall a range query scanning k consecutive elements takes $O(\frac{1}{\varepsilon}\log_B N + k/B)$. □

Space. Finally, we bound the space used.

LEMMA 22. *The external skip list on N elements requires $\Theta(N)$ space with high probability.*

PROOF. The non leaf levels of the external skip list store $\Theta(N)$ elements with only constant factor space. At the leaf level, the number of gaps between N elements is $O(N + B^\gamma \log N) = \Theta(N)$ whp. □

Lemmas 18, 19, 21, and 22 prove Theorem 3.

7. CONCLUSION

We show that adding history independence to some external-memory data structures can come at low cost. We focus on history-independent indexing structures, alternatives to the traditional B-tree, the primary indexing structure used in databases. We give HI PMAs and cache-oblivious B-trees with the same asymptotic time and space bounds as their non-HI counterparts. We give a HI skip list that performs even better than the non-HI B-skip list because the bounds are given with high probability rather than in expectation.

Acknowledgments

This research was supported in part by NSF grants CCF 1114809, CCF 1217708, IIS 1247726, IIS 1251137, CNS 1408695, and CCF 1439084, and by the Laboratory Directed Research and Development program at Sandia National Laboratories, a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

8. REFERENCES

[1] I. Abraham, J. Aspnes, and J. Yuan. Skip B-trees. In *Proc. of the 9th Annual International Conference on Principles of Distributed Systems (OPODIS)*, page 366, 2006.

[2] U. A. Acar, G. E. Blelloch, R. Harper, J. L. Vittes, and S. L. M. Woo. Dynamizing static algorithms, with applications to dynamic trees and history independence. In *Proc. of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 531–540, 2004.

[3] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, Sept. 1988.

[4] O. Amble and D. E. Knuth. Ordered hash tables. *The Computer Journal*, 17(2):135–142, 1974.

[5] A. Anagnostopoulos, M. Goodrich, and R. Tamassia. Persistent authenticated dictionaries and their applications. *Information Security*, pages 379–393, 2001.

[6] A. Andersson and T. Ottmann. Faster uniquely represented dictionaries. In *Proc. of the 32nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 642–649, 1991.

[7] C. R. Aragon and R. G. Seidel. Randomized search trees. In *Proc. of the 30th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 540–545, 1989.

[8] L. Arge, D. Eppstein, and M. T. Goodrich. Skip-webs: efficient distributed data structures for multi-dimensional data sets. In *Proc. of the 24th Annual ACM Symposium on Principles of Distributed Computing (PODS)*, pages 69–76, 2005.

[9] J. Aspnes and G. Shah. Skip graphs. *ACM Transactions on Algorithms*, 3(4):37, 2007.

[10] S. Bajaj, A. Chakraborti, and R. Sion. The foundations of history independence. *arXiv preprint arXiv:1501.06508*, 2015.

[11] S. Bajaj and R. Sion. Ficklebase: Looking into the future to erase the past. In *Proc. of the 29th IEEE International Conference on Data Engineering (ICDE)*, pages 86–97, 2013.

[12] S. Bajaj and R. Sion. HIFS: History independence for file systems. In *Proc. of the ACM SIGSAC Conference on Computer & Communications Security (CCS)*, pages 1285–1296, 2013.

[13] M. A. Bender, R. Cole, and R. Raman. Exponential structures for efficient cache-oblivious algorithms. In *Proc. of the 29th Annual International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 195–207, 2002.

[14] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. *SIAM Journal on Computing*, 35(2):341–358, 2005.

[15] M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. *Journal of Algorithms*, 3(2):115–136, 2004.

[16] M. A. Bender, M. Farach-Colton, and B. C. Kuszmaul. Cache-oblivious string B-trees. In *Proc. of the 25th Annual ACM Symposium on Principles of Database Systems (PODS)*, pages 233–242, 2006.

[17] M. A. Bender, J. T. Fineman, S. Gilbert, and B. C. Kuszmaul. Concurrent cache-oblivious B-trees. In *Proc. of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 228–237, 2005.

[18] M. A. Bender and H. Hu. An adaptive packed-memory array. *ACM Transactions on Database Systems*, 32(4):26, 2007.

[19] J. Bethencourt, D. Boneh, and B. Waters. Cryptographic methods for storing ballots on a voting machine. In *Proc. of the 14th Network and Distributed System Security Symposium (NDSS)*, 2007.

[20] G. E. Blelloch and D. Golovin. Strongly history-independent hashing with applications. In *Proc. of the 48th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 272–282, 2007.

[21] G. E. Blelloch, D. Golovin, and V. Vassilevska. Uniquely represented data structures for computational geometry. In *Proc. of the 11th Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 17–28, 2008.

[22] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height. In *Proc. of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 39–48, 2002.

[23] N. Buchbinder and E. Petrank. Lower and upper bounds on obtaining history independence. In *Advances in Cryptology*, pages 445–462, 2003.

[24] J. Bulánek, M. Koucký, and M. Saks. Tight lower bounds for the online labeling problem. In *Proc. of the 44th Annual ACM Symposium on Theory of Computing (STOC)*, pages 1185–1198, 2012.

[25] P. Callahan, M. T. Goodrich, and K. Ramaiyer. Topology B-trees and their applications. In *Proc. of the 4th International Workshop on Algorithms and Data Structures (WADS)*, pages 381–392, 1995.

[26] V. Ciriani, P. Ferragina, F. Luccio, and S. Muthukrishnan. Static optimality theorem for external memory string access. In *Proc. of the 43rd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 219–227, 2002.

[27] L. Devroye. A limit theory for random skip lists. *The Annals of Applied Probability*, pages 597–609, 1992.

[28] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *Proc. of the 23rd Annual ACM Symposium on Principles of Distributed Computing (PODS)*, pages 50–59, 2004.

[29] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. of the 40th Annual IEEE Symposium on the Foundations of Computer Science (FOCS)*, pages 285–298, 1999.

[30] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms*, 8(1):4, 2012.

[31] D. Golovin. *Uniquely Represented Data Structures with Applications to Privacy*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 2008, 2008.

[32] D. Golovin. B-treaps: A uniquely represented alternative to B-trees. In *Proc. of the 36th Annual International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 487–499. 2009.

[33] D. Golovin. The B-skip-list: A simpler uniquely represented alternative to B-trees. *arXiv preprint arXiv:1005.0662*, 2010.

[34] M. T. Goodrich and R. Tamassia. Efficient authenticated dictionaries with skip lists and commutative hashing. *US Patent App*, 10(416,015), 2000.

[35] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, 2009.

[36] J. D. Hartline, E. S. Hong, A. E. Mohr, W. R. Pentney, and E. C. Rocke. Characterizing history independent data structures. *Algorithmica*, 42(1):57–74, 2005.

[37] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A simple optimistic skiplist algorithm. *Proc. of the 14th Annual Colloquium on Structural Information and Communication Complexity (SIROCCO)*, page 124, 2007.

[38] A. Itai, A. Konheim, and M. Rodeh. A sparse table implementation of priority queues. *Proc. of the 8th Annual International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 417–431, 1981.

[39] R. Jacob, A. Richa, C. Scheideler, S. Schmid, and H. Täubig. A distributed polylogarithmic time algorithm for self-stabilizing skip graphs. In *Proc. of the 28th ACM Symposium on Principles of Distributed Computing (PODS)*, pages 131–140, 2009.

[40] Z. Kasheff. Cache-oblivious dynamic search trees. M.eng., Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 2004.

[41] I. Katriel. Implicit data structures based on local reorganizations. Master's thesis, Technion – Israel Inst. of Tech., Haifa, May 2002.

[42] P. Kirschenhofer and H. Prodinger. The path length of random skip lists. *Acta Informatica*, 31(8):775–792, 1994.

[43] D. Micciancio. Oblivious data structures: applications to cryptography. In *Proc. of the 29th Annual ACM Symposium on Theory of Computing (STOC)*, pages 456–464, 1997.

[44] D. Molnar, T. Kohno, N. Sastry, and D. Wagner. Tamper-evident, history-independent, subliminal-free data structures on prom storage-or-how to store ballots on a voting machine. In *Proc. of the 27th Annual IEEE Symposium on Security and Privacy (S&P)*, 2006.

[45] T. Moran, M. Naor, and G. Segev. Deterministic history-independent strategies for storing information on write-once memories. In *Proc. of the 34th International Colloquium on Automata, Languages and Programming (ICALP)*, 2007.

[46] M. Naor, G. Segev, and U. Wieder. History-independent cuckoo hashing. In *Proc. of the 35th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 631–642. Springer, 2008.

[47] M. Naor and V. Teague. Anti-persistence: history independent data structures. In *Proc. of the 33rd Annual ACM Symposium on Theory of Computing (STOC)*, pages 492–501, 2001.

[48] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIAM Journal on Computing*, 2(1):33–43, 1973.

[49] R. Oshman and N. Shavit. The SkipTrie: low-depth concurrent search without rebalancing. In *Proc. of the 32nd Annual ACM Symposium on Principles of Distributed Computing (PODS)*, pages 23–32, 2013.

[50] T. Papadakis, J. I. Munro, and P. V. Poblete. Analysis of the expected search cost in skip lists. In *Proc. of the 2nd Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 160–172, 1990.

[51] H. Prokop. Cache oblivious algorithms. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1999.

[52] W. Pugh. *Incremental computation and the incremental evaluation of functional programs*. PhD thesis, Cornell University, 1988.

[53] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[54] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Proc. of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 315–328, 1989.

[55] N. Rahman, R. Cole, and R. Raman. Optimised predecessor data structures for internal memory. In *Proc. of the 5th International Workshop on Algorithm Engineering (WAE)*, pages 67–78, 2001.

[56] D. S. Roche, A. J. Aviv, and S. G. Choi. Oblivious secure deletion with bounded history independence. *arXiv preprint arXiv:1505.07391*, 2015.

[57] N. Shavit and I. Lotan. Skiplist-based concurrent priority queues. In *Proc. of the 14th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 263–268, 2000.

[58] J. Shun and G. E. Blelloch. Phase-concurrent hash tables for determinism. In *Proc. of the 26th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 96–107, 2014.

[59] L. Snyder. On uniquely represented data strauctures. In *Proc. of the 18th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 142–146, 1977.

[60] R. Sundar and R. E. Tarjan. Unique binary search tree representations and equality-testing of sets and sequences. In *Proc. of the 22nd Annual ACM Symposium on Theory of Computing (STOC)*, pages 18–25, 1990.

[61] T. Tzouramanis. History-independence: a fresh look at the case of R-trees. In *Proc. of the 27th Annual ACM Symposium on Applied Computing (SAC)*, pages 7–12, 2012.

[62] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software*, 11(1):37–57, 1985.

[63] D. E. Willard. Inserting and deleting records in blocked sequential files. Technical Report TM81-45193-5, Bell Labs Tech Reports, 1981. (Cited in [66]).

[64] D. E. Willard. Maintaining dense sequential files in a dynamic environment. In *Proc. of the 14th Annual ACM Symposium on Theory of Computing (STOC)*, pages 114–121, 1982.

[65] D. E. Willard. Good worst-case algorithms for inserting and deleting records in dense sequential files. In *ACM SIGMOD Record*, volume 15:2, pages 251–260, 1986.

[66] D. E. Willard. A density control algorithm for doing insertions and deletions in a sequentially ordered file in a good worst-case time. *Information and Computation*, 97(2):150–204, 1992.

Bibliography

[1] A. Aggarwal and J. S. Vitter. “The input/output complexity of sorting and related problems.” In: *Communications of the ACM* 31.9 (Sept. 1988), pp. 1116–1127.

[2] M. Agrawal, N. Kayal, and N. Saxena. “PRIMES is in P.” In: *Annals of Mathematics* (2004), pp. 781–793.

[3] E. Agullo. “On the Out-Of-Core Factorization of Large Sparse Matrices.” PhD thesis. École normale supérieure de Lyon, France, 2008.

[4] E. Agullo, P. R. Amestoy, A. Buttari, A. Guermouche, J. L’Excellent, and F. Rouet. “Robust Memory-Aware Mappings for Parallel Multifrontal Factorizations.” In: *SIAM J. Scientific Computing* 38.3 (2016).

[5] E. Agullo, O. Beaumont, L. Eyraud-Dubois, and S. Kumar. “Are Static Schedules so Bad? A Case Study on Cholesky Factorization.” In: *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE. 2016, pp. 1021–1030.

[6] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez. “Implementing Multifrontal Sparse Solvers for Multicore Architectures with Sequential Task Flow Runtime Systems.” In: *ACM Trans. Math. Softw.* 43.2 (2016), p. 13.

[7] M. Amaris, G. Lucarelli, C. Mommessin, and D. Trystram. *Generic algorithms for scheduling applications on heterogeneous multi-core platforms*. Tech. rep. CoRR, 2017.

[8] M. Amaris, G. Lucarelli, C. Mommessin, and D. Trystram. “Generic Algorithms for Scheduling Applications on Hybrid Multi-core Machines.” In: *Euro-Par 2017: Parallel Processing*. 2017, pp. 220–231.

[9] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L’Excellent. “A fully asynchronous multifrontal solver using distributed dynamic scheduling.” In: *SIAM Journal on Matrix Analysis and Applications* 23.1 (2001), pp. 15–41.

[10] P. R. Amestoy, A. Guermouche, J.-Y. L’Excellent, and S. Pralet. “Hybrid scheduling for the parallel solution of linear systems.” In: *Parallel Computing* 32.2 (2006), pp. 136–156.

[11] P. R. Amestoy, A. Buttari, I. S. Duff, A. Guermouche, J. L’Excellent, and B. Uçar. “Mumps.” In: *Encyclopedia of Parallel Computing*. Ed. by D. A. Padua. Springer, 2011, pp. 1232–1238.

[12] L. Arge. “The buffer tree: A technique for designing batched external data structures.” In: *Algorithmica* 37.1 (2003), pp. 1–24.

[13] L. Arge and M. Thorup. “RAM-Efficient External Memory Sorting.” In: *Algorithms and Computation*. Vol. 8283. 2013, pp. 491–501.

[14] C. Ashcraft, R. G. Grimes, J. G. Lewis, B. W. Peyton, and H. D. Simon. “Progress in sparse matrix methods for large linear systems on vector computers.” In: *Int. Journal of Supercomputer Applications* 1(4) (1987), pp. 10–30.

[15] A. Atkin and D. Bernstein. “Prime sieves using binary quadratic forms.” In: *Mathematics of Computation* 73.246 (2004), pp. 1023–1030.

[16] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst. “Data-Aware Task Scheduling on Multi-accelerator Based Platforms.” In: *2010 IEEE 16th International Conference on Parallel and Distributed Systems*. Dec. 2010, pp. 291–298.

[17] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. “StarPU: a unified platform for task scheduling on heterogeneous multicore architectures.” In: *Concurrency and Computation: Practice and Experience* 23.2 (2011), pp. 187–198.

[18] W. G. Aulbur. “Parallel implementations of quasiparticle calculations of semiconductors and insulators.” PhD thesis. The Ohio State University, 1996.

[19] G. Aupy, C. Brasseur, and L. Marchal. “Dynamic memory-aware task-tree scheduling.” In: *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 758–767.

[20] O. Beaumont, N. Bonichon, L. Eyraud-Dubois, and L. Marchal. “Minimizing Weighted Mean Completion Time for Malleable Tasks Scheduling.” In: *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. May 2012, pp. 273–284.

[21] O. Beaumont, L. Eyraud-Dubois, and S. Kumar. “Approximation Proofs of a Fast and Efficient List Scheduling Algorithm for Task-Based Runtime Systems on Multicores and GPUs.” In: *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2017, pp. 768–777.

[22] O. Beaumont, T. Cojean, L. Eyraud-Dubois, A. Guermouche, and S. Kumar. “Scheduling of Linear Algebra Kernels on Multiple Heterogeneous Resources.” In: *International Conference on High Performance Computing, Data, and Analytics (HiPC)*. 2016.

[23] O. Beaumont and A. Guermouche. “Task Scheduling for Parallel Multifrontal Methods.” In: *Parallel Processing International Conference (Euro-Par)*. 2007, pp. 758–766.

[24] L. A. Belady. “A study of replacement algorithms for a virtual-storage computer.” In: *IBM Journal of Research and Development* 5.2 (June 1966), pp. 78–101.

[25] M. Benazouz, O. Marchetti, A. Munier-Kordon, and T. Michel. “A new method for minimizing buffer sizes for cyclo-static dataflow graphs.” In: *Embedded Systems for Real-Time Multimedia (ESTIMedia), 2010 8th IEEE Workshop on*. IEEE. 2010, pp. 11–20.

[26] M. Benazouz, O. Marchetti, A. Munier-Kordon, and P. Urard. “A new approach for minimizing buffer capacities with throughput constraint for embedded system design.” In: *Computer Systems and Applications (AICCSA), 2010 IEEE/ACS International Conference on*. IEEE. 2010, pp. 1–8.

[27] M. A. Bender, E. D. Demaine, and M. Farach-Colton. “Cache-Oblivious B-Trees.” In: *SIAM Journal on Computing* 35.2 (2005), pp. 341–358.

[28] M. A. Bender, Z. Duan, J. Iacono, and J. Wu. “A Locality-Preserving Cache-Oblivious Dynamic Dictionary.” In: *Journal of Algorithms* 3.2 (2004), pp. 115–136.

[29] M. A. Bender and H. Hu. “An adaptive packed-memory array.” In: *ACM Transactions on Database Systems* 32.4 (2007), p. 26.

[30] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi. “Characterization of scientific workflows.” In: *Workflows in Support of Large-Scale Science, 2008. WORKS 2008. Third Workshop on*. Nov. 2008, pp. 1–10.

[31] S. Bharathi and A. Chervenak. “Scheduling data-intensive workflows on storage constrained resources.” In: *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science (WORKS'09)*. ACM, 2009.

[32] R. Bleuse, T. Gautier, J. V. Lima, G. Mounié, and D. Trystram. “Scheduling data flow program in XKaapi: A new affinity based Algorithm for Heterogeneous Architectures.” In: *Euro-Par 2014: Parallel Processing*. 2014, pp. 560–571.

[33] R. Bleuse, S. Hunold, S. Kedad-Sidhoum, F. Monna, G. Mounié, and D. Trystram. “Scheduling Independent Moldable Tasks on Multi-Cores with GPUs.” In: *IEEE Transactions on Parallel and Distributed Systems* 28.9 (2017), pp. 2689–2702.

[34] R. Bleuse, S. Kedad-Sidhoum, F. Monna, G. Mounié, and D. Trystram. “Scheduling independent tasks on multi-cores with GPU accelerators.” In: *Concurrency and Computation: Practice and Experience* 27.6 (2015), pp. 1625–1638.

[35] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. “Cilk: An Efficient Multithreaded Runtime System.” In: *SIGPLAN Not.* 30.8 (Aug. 1995), pp. 207–216. ISSN: 0362-1340.

[36] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra. “PaRSEC: Exploiting heterogeneity for enhancing scalability.” In: *Computing in Science & Engineering* 15.6 (2013), pp. 36–45.

[37] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, et al. “Concurrent collections.” In: *Scientific Programming* 18.3-4 (2010), pp. 203–217.

[38] J. Bulánek, M. Koucky, and M. Saks. “Tight lower bounds for the online labeling problem.” In: *Proc. of the 44th Annual ACM Symposium on Theory of Computing (STOC)*. 2012, pp. 1185–1198.

[39] A. Buttari. “Fine Granularity Sparse QR Factorization for Multicore Based Systems.” In: *International Conference on Applied Parallel and Scientific Computing*. 2012, pp. 226–236.

[40] L.-C. Canon, L. Marchal, and F. Vivien. “Low-Cost Approximation Algorithms for Scheduling Independent Tasks on Hybrid Platforms.” In: *Euro-Par 2017: Parallel Processing*. 2017, pp. 232–244.

[41] *Chameleon, a dense linear algebra software for heterogeneous architectures*. https://project.inria.fr/chameleon.

[42] C.-Y. Chen and C.-P. Chu. “A 3.42-approximation algorithm for scheduling malleable tasks under precedence constraints.” In: *IEEE Transactions on Parallel and Distributed Systems* 24.8 (2013), pp. 1479–1488.

[43] L. Chen, D. Ye, and G. Zhang. “Online Scheduling of mixed CPU-GPU jobs.” In: *International Journal of Foundations of Computer Science* 25.06 (2014), pp. 745–761.

[44] F. A. Chudak and D. B. Shmoys. “Approximation algorithms for precedence-constrained scheduling problems on parallel machines that run at different speeds.” In: *Journal of Algorithms* 30.2 (1999), pp. 323–343.

[45] E. Ciurea and L. Ciupalâ. “Sequential and parallel algorithms for minimum flows.” In: *Journal of Applied Mathematics and Computing* 15.1 (2004), pp. 53–75.

[46] G. Cordasco, R. D. Chiara, and A. L. Rosenberg. "Assessing the Computational Benefits of AREA-Oriented DAG-Scheduling." In: *Euro-Par 2011 Parallel Processing - 17th International Conference, Euro-Par 2011, Bordeaux, France, August 29 - September 2, 2011, Proceedings, Part I*. 2011, pp. 180–192.

[47] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844, 9780262033848.

[48] J. R. Correa and A. S. Schulz. "Single-machine scheduling with precedence constraints." In: *Mathematics of Operations Research* 30.4 (2005), pp. 1005–1021.

[49] R. F. Da Silva, W. Chen, G. Juve, K. Vahi, and E. Deelman. "Community resources for enabling research in distributed scientific workflows." In: *e-Science (e-Science), 2014 IEEE 10th International Conference on*. Vol. 1. IEEE. 2014, pp. 177–184.

[50] T. A. Davis. *Direct Methods for Sparse Linear Systems*. Fundamentals of Algorithms. Philadelphia: Society for Industrial and Applied Mathematics, 2006.

[51] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng. "Algorithm 836: COLAMD, a column approximate minimum degree ordering algorithm." In: *ACM Trans. Math. Softw.* 30.3 (2004), pp. 377–380.

[52] T. A. Davis and Y. Hu. "The University of Florida Sparse Matrix Collection." In: *ACM Trans. Math. Softw.* 38.1 (Dec. 2011), 1:1–1:25. ISSN: 0098-3500.

[53] F. Desprez and F. Suter. "A bi-criteria algorithm for scheduling parallel task graphs on clusters." In: *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*. IEEE. 2010, pp. 243–252.

[54] D. E. Dolan and J. J. Moré. "Benchmarking optimization software with performance profiles." In: *Mathematical Programming* 91.2 (2002), pp. 201–213. ISSN: 1436-4646.

[55] M. Drozdowski and W. Kubiak. "Scheduling parallel tasks with sequential heads and tails." English. In: *Annals of Operations Research* 90.0 (1999), pp. 221–246. ISSN: 0254-5330.

[56] M. Drozdowski. "Scheduling multiprocessor tasks — An overview." In: *European Journal of Operational Research* 94.2 (1996), pp. 215–230. ISSN: 0377-2217.

[57] M. Drozdowski. "Scheduling Parallel Tasks – Algorithms and Complexity." In: *Handbook of Scheduling*. Ed. by J. Leung. Chapman and Hall/CRC, 2004. ISBN: 1584883979.

[58] J. Du and J. Y.-T. Leung. "Complexity of Scheduling Parallel Task Systems." In: *SIAM Journal on Discrete Mathematics* 2.4 (1989), pp. 473–487.

[59] I. S. Duff and J. K. Reid. "The multifrontal solution of indefinite sparse symmetric linear systems." In: *ACM Transactions on Mathematical Software* 9 (1983), pp. 302–325.

[60] B. Dunten, J. Jones, and J. Sorenson. "A space-efficient fast prime number sieve." In: *IPL* 59.2 (1996), pp. 79–84.

[61] C. Eisenbeis, F. Gasperoni, and U. Schwiegelshohn. "Allocating registers in multiple instruction-issuing processors." In: *Proceedings of the IFIP WG10. 3 working conference on Parallel architectures and compilation techniques*. IFIP Working Group on Algol. 1995, pp. 290–293.

[62] L. Eyraud-Dubois, L. Marchal, O. Sinnen, and F. Vivien. "Parallel Scheduling of Task Trees with Limited Memory." In: *TOPC* 2.2 (2015), p. 13.

[63] L. Fan, F. Zhang, G. Wang, and Z. Liu. "An effective approximation algorithm for the Malleable Parallel Task Scheduling problem." In: *Journal of Parallel and Distributed Computing* 72.5 (2012), pp. 693–704. ISSN: 0743-7315.

[64] D. Feitelson. "Workload modeling for computer systems performance evaluation." In: *Book Draft, Version 1.0.1* (2014), pp. 1–601.

[65] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. "Cache-oblivious algorithms." In: *Proceesdings of the 40th Annual Symposium on Foundations of Computer Science*. IEEE. 1999, pp. 285–297.

[66] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[67] M. Garey, D. Johnson, and L. Stockmeyer. "Some simplified NP-complete graph problems." In: *Theoretical Computer Science* 1.3 (1976), pp. 237–267. ISSN: 0304-3975.

[68] T. Gautier, X. Besseron, and L. Pigeon. "KAAPI: A Thread Scheduling Runtime System for Data Flow Computations on Cluster of Multi-processors." In: *International Workshop on Parallel Symbolic Computation*. London, Ontario, Canada, 2007, pp. 15–23. ISBN: 978-1-59593-741-4.

[69] A. V. Goldberg and R. E. Tarjan. "A New Approach to the Maximum Flow Problem." In: *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*. STOC '86. Berkeley, California, USA: ACM, 1986, pp. 136–146. ISBN: 0-89791-193-8.

[70] D. Golovin. "The B-skip-list: A simpler uniquely represented alternative to B-trees." In: *arXiv preprint arXiv:1005.0662* (2010).

[71] A. González-Escribano, A. J. C. van Gemund, and V. Cardeñoso-Payo. "Mapping Unstructured Applications into Nested Parallelism." In: *High Performance Computing for Computational Science - VECPAR 2002, 5th International Conference, Porto, Portugal, June 26-28, 2002, Selected Papers and Invited Talks*. 2002, pp. 407–420.

[72] R. L. Graham. "Bounds for certain multiprocessing anomalies." In: *Bell System Technical Journal* 45.9 (1966), pp. 1563–1581.

[73] R. L. Graham. "Bounds on multiprocessing timing anomalies." In: *SIAM journal on Applied Mathematics* 17.2 (1969), pp. 416–429.

[74] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. R. Kan. "Optimization and approximation in deterministic sequencing and scheduling: a survey." In: *Annals of discrete mathematics*. Vol. 5. Elsevier, 1979, pp. 287–326.

[75] S. L. Graham, M. Snir, C. A. Patterson, et al. *Getting up to speed: The future of supercomputing*. National Academies Press, 2005.

[76] D. Gries and J. Misra. "A linear sieve algorithm for finding prime numbers." In: *Communications of the ACM* 21.12 (1978), pp. 999–1003.

[77] E. Günther, F. König, and N. Megow. "Scheduling and packing malleable and parallel tasks with precedence constraints of bounded width." English. In: *Journal of Combinatorial Optimization* 27.1 (2014), pp. 164–181. ISSN: 1382-6905.

[78] G. Hardy, J. Littlewood, and G. Pólya. "Inequalities." In: Cambridge Mathematical Library. Cambridge University Press, 1952. Chap. 6.14. ISBN: 9780521358804.

[79] J. D. Hartline, E. S. Hong, A. E. Mohr, W. R. Pentney, and E. C. Rocke. "Characterizing history independent data structures." In: *Algorithmica* 42.1 (2005), pp. 57–74.

[80] P. Hénon, P. Ramet, and J. Roman. "PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems." In: *Parallel Computing* 28.2 (Jan. 2002), pp. 301–321.

[81] M. D. Hill and A. J. Smith. "Evaluating associativity in CPU caches." In: *IEEE Transactions on Computers* 38.12 (1989), pp. 1612–1630.

[82] S. Horsley. "ΚΟΣΚΙΝΟΝ ΕΡΑΤΟΣΘΕΝΟΥΣ. or, The Sieve of Eratosthenes. Being an Account of His Method of Finding All the Prime Numbers, by the Rev. Samuel Horsley, FRS." In: *Philosophical Transactions* (1772), pp. 327–347.

[83] A. Hugo, A. Guermouche, P. Wacrenier, and R. Namyst. "A Runtime Approach to Dynamic Resource Allocation for Sparse Direct Solvers." In: *43rd International Conference on Parallel Processing, ICPP 2014, Minneapolis, MN, USA, September 9-12, 2014*. 2014, pp. 481–490.

[84] A. Hugo, A. Guermouche, P. Wacrenier, and R. Namyst. "Composing multiple StarPU applications over heterogeneous machines: A supervised approach." In: *IJHPCA* 28.3 (2014), pp. 285–300.

[85] S. Hunold. "One step toward bridging the gap between theory and practice in moldable task scheduling with precedence constraints." In: *Concurrency and Computation: Practice and Experience* 27.4 (2015), pp. 1010–1026. ISSN: 1532-0634.

[86] M. S. Hybertsen and S. G. Louie. "Electron correlation in semiconductors and insulators: Band gaps and quasiparticle energies." In: *Physical Review B* 34.8 (1986), p. 5390.

[87] C. Imreh. "Scheduling problems on two sets of identical machines." In: *Computing* 70.4 (2003), pp. 277–294.

[88] M. Jacquelin, L. Marchal, Y. Robert, and B. Uçar. "On optimal tree traversals for sparse matrix factorization." In: *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE. 2011, pp. 556–567.

[89] A. Jain and C. Lin. "Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement." In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. June 2016, pp. 78–89.

[90] K. Jansen and H. Zhang. "An Approximation Algorithm for Scheduling Malleable Tasks Under General Precedence Constraints." English. In: *Algorithms and Computation*. Ed. by X. Deng and D.-Z. Du. Vol. 3827. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 236–245. ISBN: 978-3-540-30935-2.

[91] K. Jansen and H. Zhang. "Scheduling malleable tasks with precedence constraints." In: *Journal of Computer and System Sciences* 78.1 (2012), pp. 245–259. ISSN: 0022-0000.

[92] R. M. Karp. "Reducibility among combinatorial problems." In: *Complexity of computer computations*. Springer, 1972, pp. 85–103.

[93] S. Kedad-Sidhoum, F. Monna, and D. Trystram. "Scheduling Tasks with Precedence Constraints on Hybrid Multi-core Machines." In: *IEEE International Parallel and Distributed Processing Symposium Workshop*. 2015, pp. 27–33.

[94] N. Kell and J. Havill. "Improved upper bounds for online malleable job scheduling." English. In: *Journal of Scheduling* 18.4 (2015), pp. 393–410. ISSN: 1094-6136.

[95] H. Kellerer, R. Mansini, U. Pferschy, and M. G. Speranza. "An efficient fully polynomial approximation scheme for the subset-sum problem." In: *Journal of Computer and System Sciences* 66.2 (2003), pp. 349–370.

[96] C.-C. Lam, T. Rauber, G. Baumgartner, D. Cociorva, and P. Sadayappan. “Memory-optimal evaluation of expression trees involving large objects.” In: *Computer Languages, Systems & Structures* 37.2 (2011), pp. 63–75.

[97] M. Lampis, G. Kaouri, and V. Mitsou. “On the algorithmic effectiveness of digraph decompositions and complexity measures.” In: *Discrete Optimization* 8.1 (2011), pp. 129–138.

[98] E. L. Lawler. *Combinatorial optimization: networks and matroids*. Courier Corporation, 2001.

[99] E. A. Lee and D. G. Messerschmitt. “Synchronous data flow.” In: *Proceedings of the IEEE* 75.9 (1987), pp. 1235–1245.

[100] T. J. Lee and G. E. Scuseria. “Achieving chemical accuracy with coupled-cluster theory.” In: *Quantum Mechanical Electronic Structure Calculations with Chemical Accuracy*. Springer, 1995, pp. 47–108.

[101] R. Lepère, D. Trystram, and G. J. Woeginger. “Approximation algorithms for scheduling malleable tasks under precedence constraints.” In: *International Journal of Foundations of Computer Science* 13.04 (2002), pp. 613–627.

[102] J. Y. Leung. *Handbook of scheduling: algorithms, models, and performance analysis*. CRC Press, 2004.

[103] X. S. Li. “An Overview of SuperLU: Algorithms, Implementation, and User Interface.” In: *ACM Transactions on Mathematical Software* 31.3 (Sept. 2005), pp. 302–325.

[104] J. W. H. Liu. “The Role of Elimination Trees in Sparse Factorization.” In: *SIAM Journal on Matrix Analysis and Applications* 11.1 (1990), pp. 134–172.

[105] J. W. H. Liu. “An application of generalized tree pebbling to sparse matrix factorization.” In: *SIAM J. Algebraic Discrete Methods* 8.3 (1987), pp. 375–395.

[106] J. W. H. Liu. “On the storage requirement in the out-of-core multifrontal method for sparse factorization.” In: *ACM Transaction on Mathematical Software* (1986).

[107] K. Makarychev and D. Panigrahi. “Precedence-Constrained Scheduling of Malleable Jobs with Preemption.” In: *ICALP 2014*. 2014, pp. 823–834.

[108] E. Mäkinen. “Generating random binary trees—a survey.” In: *Information Sciences* 115.1-4 (1999), pp. 123–136.

[109] J. M. Martin. “Benchmark studies on small molecules.” In: *Encyclopedia of Computational Chemistry* (1998).

[110] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. “Evaluation techniques for storage hierarchies.” In: *IBM Systems Journal* 9.2 (1970), pp. 78–117. ISSN: 0018-8670.

[111] R. McNaughton. “Scheduling with Deadlines and Loss Functions.” In: *Management Science* 6.1 (1959), pp. 1–12.

[112] D. Micciancio. “Oblivious data structures: applications to cryptography.” In: *Proc. of the 29th Annual ACM Symposium on Theory of Computing (STOC)*. 1997, pp. 456–464.

[113] P. Michaud. “Some mathematical facts about optimal cache replacement.” In: *ACM Transactions on Architecture and Code Optimization (TACO)* 13.4 (2016), p. 50.

[114] V. Nagarajan, J. Wolf, A. Balmin, and K. Hildrum. “Flowflex: Malleable scheduling for flows of mapreduce jobs.” In: *Middleware 2013*. Springer, 2013, pp. 103–122.

[115] M. Naor and V. Teague. "Anti-persistence: history independent data structures." In: *Proc. of the 33rd Annual ACM Symposium on Theory of Computing (STOC)*. 2001, pp. 492–501.

[116] OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 4.0*. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf. July 2013.

[117] F. Pellegrini and J. Roman. "Sparse matrix ordering with scotch." In: *International Conference on High-Performance Computing and Networking*. Springer. 1997, pp. 370–378.

[118] Peter Shor (http://cs.stackexchange.com/users/198/peter-shor). *Minimum s-t cut in weighted directed acyclic graphs with possibly negative weights*. Computer Science Stack Exchange.

[119] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. "Hierarchical Task-Based Programming With StarSs." In: *IJHPCA* 23.3 (2009), pp. 284–299.

[120] A. Pothen and C. Sun. "A mapping algorithm for parallel sparse Cholesky factorization." In: *SIAM Journal on Scientific Computing* 14.5 (1993), pp. 1253–1257.

[121] G. N. S. Prasanna and B. R. Musicus. "Generalized Multiprocessor Scheduling and Applications to Matrix Computations." In: *IEEE TPDS* 7.6 (1996), pp. 650–664.

[122] G. N. S. Prasanna and B. R. Musicus. "The Optimal Control Approach to Generalized Multiprocessor Scheduling." In: *Algorithmica* 15.1 (1996), pp. 17–49.

[123] W. Pugh. "Skip lists: a probabilistic alternative to balanced trees." In: *Communications of the ACM* 33.6 (1990), pp. 668–676.

[124] A. Radulescu and A. J. Van Gemund. "A low-cost approach towards mixed task and data parallel scheduling." In: *Parallel Processing, 2001. International Conference on*. IEEE. 2001, pp. 69–76.

[125] 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: *Proceedings of the IEEE Symposium on Cluster Computing and the Grid (CCGrid'07)*. Los Alamitos, CA, USA: IEEE Computer Society, 2007, pp. 401–409.

[126] F. Sainz, S. Mateo, V. Beltran, J. L. Bosque, X. Martorell, and E. Ayguadé. "Leveraging OmpSs to Exploit Hardware Accelerators." In: *IEEE International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 2014, pp. 112–119.

[127] E. Saule, H. M. Aktulga, C. Yang, E. G. Ng, and Ü. V. Çatalyürek. "An Out-of-Core Task-based Middleware for Data-Intensive Scientific Computing." In: *Handbook on Data Centers*. Ed. by S. U. Khan and A. Y. Zomaya. Springer, 2015, pp. 647–667.

[128] D. Sbırlea, Z. Budimlić, and V. Sarkar. "Bounded memory scheduling of dynamic task graphs." In: *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM. 2014, pp. 343–356.

[129] M. Sergent, D. Goudin, S. Thibault, and O. Aumage. "Controlling the Memory Subscription of Distributed Applications with a Task-Based Runtime System." In: *Proceedings of the International Parallel and Distributed Processing Symposium Workshops*. IEEE. 2016, pp. 318–327.

[130] R. Sethi. "Complete register allocation problems." In: *SIAM journal on Computing* 4.3 (1975), pp. 226–248.

[131] R. Sethi and J. Ullman. "The Generation of Optimal Code for Arithmetic Expressions." In: *J. ACM* 17.4 (1970), pp. 715–728.

[132] J. P. Sorenson. "The pseudosquares prime sieve." In: *Algorithmic number theory*. 2006, pp. 193–207.

[133] F. Suter. *DAGGEN: A synthetic task graph generator*. https://github.com/frs69wq/daggen.

[134] O. Svensson. "Hardness of precedence constrained scheduling on identical machines." In: *SIAM Journal on Computing* 40.5 (2011), pp. 1258–1274.

[135] T. Tobita and H. Kasahara. "A standard task graph set for fair evaluation of multiprocessor scheduling algorithms." In: *Journal of Scheduling* 5.5 (2002), pp. 379–394.

[136] S. Toledo. "A survey of out-of-core algorithms in numerical linear algebra." In: *External Memory Algorithms, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, May 20-22, 1998*. 1998, pp. 161–180.

[137] S. Tomov, J. Dongarra, and M. Baboulin. "Towards dense linear algebra for hybrid GPU accelerated manycore systems." In: *Parallel Computing* 36.5-6 (2010), pp. 232–240.

[138] *TOP500 Supercomputer Site*. http://www.top500.org, List of November 2017.

[139] H. Topcuoglu, S. Hariri, and M. Y. Wu. "Performance-effective and low-complexity task scheduling for heterogeneous computing." In: *IEEE Trans. Parallel Distributed Systems* 13.3 (2002), pp. 260–274.

[140] S. Touati. "Register Pressure in Instruction Level Parallelism." Theses. Université de Versailles-Saint Quentin en Yvelines, June 2002.

[141] J. Valdes, R. E. Tarjan, and E. L. Lawler. "The Recognition of Series Parallel Digraphs." In: *SIAM J. Comput.* 11.2 (1982), pp. 298–313.

[142] V. Vizing. "Minimization of the maximum delay in servicing systems with interruption." In: *USSR Computational Mathematics and Mathematical Physics* 22.3 (1982), pp. 227–233. ISSN: 0041-5553.

[143] Q. Wang and K.-H. Cheng. "A Heuristic of Scheduling Parallel Tasks and Its Analysis." In: *SIAM Journal on Computing* 21.2 (1992), pp. 281–294.

[144] M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit. "Efficient Computation of Buffer Capacities for Cyclo-Static Dataflow Graphs." In: *2007 44th ACM/IEEE Design Automation Conference*. June 2007, pp. 658–663.

[145] M. H. Wiggers, M. J. Bekooij, P. G. Jansen, and G. J. Smit. "Efficient computation of buffer capacities for cyclo-static real-time systems with back-pressure." In: *Real Time and Embedded Technology and Applications Symposium, 2007. RTAS'07. 13th IEEE*. IEEE. 2007, pp. 281–292.

[146] D. E. Willard. "A density control algorithm for doing insertions and deletions in a sequentially ordered file in a good worst-case time." In: *Information and Computation* 97.2 (1992), pp. 150–204.

[147] A. YarKhan, J. Kurzak, and J. Dongarra. "Quark users' guide: Queueing and runtime for kernels." In: *University of Tennessee Innovative Computing Laboratory Technical Report ICL-UT-11-02* (2011).

[148] Y. Zinder and S. Walker. "Scheduling flexible multiprocessor tasks on parallel machines." In: *The 9th Workshop on Models and Algorithms for Planning and Scheduling Problems*. 2009.

List of publications[1]

Articles in International Refereed Journals

[J1] B. Simon, B. Jaumard, and T. H. Le. "Deadlock Avoidance and Detection In Railway Simulation Systems." In: *Transportation Research Record: Journal of the Transportation Research Board* 2448 (2014), pp. 45–52. DOI: 10.3141/2448-06.

[J2] L. Marchal, B. Simon, O. Sinnen, and F. Vivien. "Malleable task-graph scheduling with a practical speed-up model." In: *IEEE Transactions on Parallel and Distributed Systems* (2018). DOI: 10.1109/TPDS.2018.2793886.

Articles in International Refereed Conferences

[C1] B. Simon, B. Jaumard, and T. H. Le. "Deadlock Avoidance and Detection in Railway Simulation Systems." In: *Joint Rail Conference*. American Society of Mechanical Engineers. 2014. DOI: 10.1115/JRC2014-3864.

[C2] A. Guermouche, L. Marchal, B. Simon, and F. Vivien. "Scheduling Trees of Malleable Tasks for Sparse Linear Algebra." In: *European Conference on Parallel Processing (Euro-Par)*. 2015, pp. 479–490. DOI: 10.1007/978-3-662-48096-0_37.

[C3] M. A. Bender, J. Berry, R. Johnson, T. M. Kroeger, S. McCauley, C. A. Phillips, B. Simon, S. Singh, and D. Zage. "Anti-Persistence on Persistent Storage: History-Independent Sparse Tables and Dictionaries." In: *Proceedings of the Thirty-Fifth Symposium on Principles of Database Systems (PODS)*. 2016. DOI: 10.1145/2902251.2902276.

[C4] M. A. Bender, S. McCauley, B. Simon, S. Singh, and F. Vivien. "Resource Optimization for Program Committee Members: A Subreview Article." In: *Fun with Algorithms (FUN)*. 2016. DOI: 10.4230/LIPIcs.FUN.2016.7.

[C5] M. A. Bender, R. Chowdhury, A. Conway, M. Farach-Colton, P. Ganapathi, R. Johnson, S. McCauley, B. Simon, and S. Singh. "The I/O Complexity of Computing Prime Tables." In: *12th Latin American Theoretical Informatics Symposium (LATIN)*. 2016. DOI: 10.1007/978-3-662-49529-2_15.

[C6] L.-C. Canon, L. Marchal, B. Simon, and F. Vivien. "Online Scheduling of Sequential Task Graphs on Hybrid Platforms." In: *European Conference on Parallel Processing (Euro-Par)*. 2018.

[1] Authors are listed in alphabetical order except for [R1, J1, C1].

[C7] L. Marchal, H. Nagy, B. Simon, and F. Vivien. "Parallel scheduling of DAGs under memory constraints." In: *IPDPS 2018-32st IEEE International Parallel & Distributed Processing Symposium*. 2018.

Articles in International Refereed Workshops

[W1] L. Marchal, S. McCauley, B. Simon, and F. Vivien. "Minimizing I/Os in Out-of-Core Task Tree Scheduling." In: *19th Workshop on Advances in Parallel and Distributed Computational Models*. 2017. DOI: 10.1109/IPDPSW.2017.58.

Research Reports

[R1] B. Simon, B. Jaumard, and T. H. Le. *Deadlock Avoidance and Detection In Railway Simulation Systems*. Les Cahiers du GERAD: G-2013-43. 2013. URL: https://www.gerad.ca/fr/papers/G-2013-43.

[R2] L. Marchal, B. Simon, and F. Vivien. *Scheduling Malleable Task Trees*. INRIA Research Report 8587. 2014. URL: https://hal.inria.fr/hal-01059704.

[R3] A. Guermouche, L. Marchal, B. Simon, and F. Vivien. *Scheduling Trees of Malleable Tasks for Sparse Linear Algebra*. INRIA Research Report 8616. 2014. URL: https://hal.inria.fr/hal-01077413.

[R4] L. Marchal, B. Simon, O. Sinnen, and F. Vivien. *Malleable task-graph scheduling with a practical speed-up model*. INRIA Research Report 8856. 2016. URL: https://hal.inria.fr/hal-01274099.

[R5] L. Marchal, S. McCauley, B. Simon, and F. Vivien. *Minimizing I/Os in Out-of-Core Task Tree Scheduling*. INRIA Research Report 9025. 2017. URL: https://hal.inria.fr/hal-01462213.

[R6] L. Marchal, H. Nagy, B. Simon, and F. Vivien. *Parallel scheduling of DAGs under memory constraints*. INRIA Research Report 9108. 2017. URL: https://hal.inria.fr/hal-01620255.

[R7] L.-C. Canon, L. Marchal, B. Simon, and F. Vivien. *Online Scheduling of Sequential Task Graphs on Hybrid Platforms*. INRIA Research Report 9150. 2018. URL: https://hal.inria.fr/hal-01720064.