Scheduling Trees of Malleable Tasks for Sparse Linear Algebra

Abdou GUERMOUCHE, Loris MARCHAL, Bertrand SIMON, Frédéric VIVIEN

RESEARCH REPORT
N° 8616
October 2014
Project-Team ROMA

ISSN 0249-6399   ISRN INRIA/RR--8616--FR+ENG

Scheduling Trees of Malleable Tasks for Sparse Linear Algebra

Abdou GUERMOUCHE*, Loris MARCHAL†, Bertrand SIMON‡, Frédéric VIVIEN§

Project-Team ROMA

Research Report n° 8616 — October 2014 — 36 pages

* Abdou GUERMOUCHE is with Univ. of Bordeaux, France. E-mail: abdou.guermouche@inria.fr

† Loris MARCHAL is with CNRS, France. E-mail: loris.marchal@ens-lyon.fr

‡ Bertrand SIMON is with ENS de Lyon, France. E-mail: bertrand.simon@ens-lyon.fr

§ Frédéric VIVIEN is with Inria, France. E-mail: frederic.vivien@inria.fr

RESEARCH CENTRE GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Abstract: Scientific workloads are often described as directed acyclic task graphs. In this paper, we focus on the multifrontal factorization of sparse matrices, whose task graph is structured as a tree of parallel tasks. Among the existing models for parallel tasks, the concept of *malleable* tasks is especially powerful as it allows each task to be processed on a time-varying number of processors. Following the model advocated by Prasanna and Musicus [1, 2] for matrix computations, we consider malleable tasks whose speedup is p^α, where p is the fractional share of processors on which a task executes, and α ($0 < \alpha \leq 1$) is a parameter which does not depend on the task. We first motivate the relevance of this model for our application with actual experiments on multicore platforms. Then, we study the optimal allocation proposed by Prasanna and Musicus for makespan minimization using optimal control theory. We largely simplify their proofs by resorting only to pure scheduling arguments. Building on the insight gained thanks to these new proofs, we extend the study to distributed multicore platforms. There, a task cannot be distributed among several distributed nodes. In such a distributed setting (homogeneous or heterogeneous), we prove the NP-completeness of the corresponding scheduling problem, and propose some approximation algorithms. We finally assess the relevance of our approach by simulations on realistic trees. We show that the average performance gain of our allocations with respect to existing solutions (that are thus unaware of the actual speedup functions) is up to 16% for $\alpha = 0.9$ (the value observed in the real experiments).

Key-words: Scheduling, Task graph, Malleable task, Sparse linear algebra

Ordonnancement d'Arbres de Tâches Malléables pour l'Algèbre Linéaire

Résumé : Les applications de calcul scientifique sont souvent décrites comme des graphes de tâches dirigés et acycliques. Dans ce papier, on se concentre sur la factorisation multifrontale de matrices creuses, dont la structure du graphe de tâches est un arbre de tâches parallèles. Parmi les modèles de tâches parallèles existants, le concept de tâches *malléable* est particulièrement puissant comme il autorise chaque tâche à être executée sur un nombre de processeurs variable. Suivant le modèle préconisé par Prasanna et Musicus [1, 2] pour le calcul matriciel, on considère des tâches malléables dont l'accélération est p^α, où p est la fraction rationnelle de processeurs sur laquelle une tâche s'exécute, et α ($0 < \alpha \leq 1$) est un paramètre qui ne dépend pas de la tâche. Nous commençons par motiver la pertinence de ce modèle dans l'application qui nous intéresse avec des expériences réelles sur des plates-formes multi-cœurs Ensuite, nous étudions l'allocation optimale proposée par Prasanna et Musicus pour la minimisation du makespan en utilisant la théorie du contrôle optimal. Nous simplifions grandement leurs preuves en ayant recours uniquement à des purs arguments d'ordonnancement. Nous étendons ensuite l'étude à des plates-formes multi-cœurs distribuées. Ici, une tâche ne peut être distribuée à plusieurs nœuds. Dans une telle distribution (homogène ou hétérogène), nous prouvons la NP-complétude du problème d'ordonnancement correspondant, et proposons quelques schémas d'approximation. Nous certifions ensuite la pertinence de notre approche par des simulations sur des arbres réalistes. Nous montrons que le gain de performance moyen de nos allocations par rapport à des solutions existantes (et donc ignorant la fonction d'accélération réelle) va jusqu'à 16% pour $\alpha = 0.9$ (la valeur observée lors des expériences).

Mots-clés : Ordonnancement, Graphe de tâches, Tâche malléable, Algèbre linéaire creuse

Contents

1 Introduction **1**

2 Related work **2**

3 Validation of the malleable task model **3**

4 Model and notations **8**

5 Optimal solution for shared-memory platforms **10**

6 Extensions to distributed memory **15**
6.1 Two homogeneous multicore nodes 16
6.2 Two heterogeneous multicore nodes 24

7 Simulations **29**

8 Conclusion **32**

1 Introduction

Parallel workloads are often modeled as directed acyclic task graphs, or DAGs, where nodes represent tasks and edges represent the dependencies between tasks. Task graphs arise from many scientific domains, such as image processing, genomics, and geophysical simulations. In this paper, we focus on task graphs coming from sparse linear algebra, and especially on the factorization of sparse matrices using the multifrontal method. Liu [3] explains that the computational dependencies and requirements in Cholesky and LU factorization of sparse matrices using the multifrontal method can be modeled as a task tree, called the *elimination tree*. In the present paper, we thus focus on dependencies that can be modeled as a tree.

In the abundant existing literature, several variants of the task graph scheduling problem are addressed, depending on the ability to process a task in parallel: tasks are either *sequential* (not amenable to parallel processing), *rigid* (requesting a given number of processors), *moldable* (able to cope with any fixed number of processor) or even *malleable* (processed on a variable number of processors) in the terminology of Drozdowski [4, chapter 25]. When considering moldable and malleable tasks, one has to define how the processing time of a task depends on the number of processors allocated to this task (over time). In this study, we consider a special case of malleable tasks, where the speedup function of each task is p^α, where p is the number of processors allocated to the task, and $0 < \alpha \leq 1$ is a global parameter. In particular, when the share of processors p_i allocated to a task T_i is constant, its processing time is given by L_i/p_i^α, where L_i is the sequential duration of T_i. The case $\alpha = 1$ represents the unrealistic case of a perfect linear speed-up, 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. This model has been advocated by Prasanna and Musicus [2] for matrix operations, and we present some new motivation for this model in our context. As in [2], 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 is allocated 3.4 processors, one processor will dedicate 60% of its processing 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 [5], KAAPI [6], or PaRSEC [7], and it allows to simplify the scheduling problem, and thus, to derive optimal allocation algorithms.

In this context, our objective is to minimize the total processing time of a task graph described as a tree of malleable tasks, on a homogeneous platform, composed of p identical processors. To achieve this goal, we take

advantage of two sources of parallelism of our application: the *tree parallelism* which allows tasks independent from each others (such as siblings) to be processed concurrently, and the *task parallelism* which allows a task to be processed on several processors. A solution to this problem describes both in which order the tasks are processed, but also which share of computing resources is allocated to each task.

In [1, 2], 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 [1, 2] 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 equal to p^α presented above, they prove characteristics 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.

In this paper, our contributions are the following:

- In Section 3, we show that the model of malleable tasks using the p^α speed-up function is justified in the context of sparse matrix factorization.
- In Section 5, we propose a new and simpler proof for the results of [1, 2] on series-parallel graphs, using pure scheduling arguments.
- In Section 6, we extend the previous study on distributed memory machines, where tasks cannot be distributed across several distributed nodes. We provide NP-completeness results and approximation algorithms.
- In Section 7, we evaluate the algorithm of Section 5 on a set of realistic trees and estimate its improvement compared to more straightforward solutions which are unaware of the p^α speed-up function.

2 Related work

Malleable task scheduling is one of the classical formalism to deal with parallel tasks, as presented in the survey [4, chapter 26]. The problem of minimizing the completion time of a graph made of malleable tasks has been studied in many papers, including [8, 9]. In these studies, it is usually assumed that the speedup function is unknown, but complies with some reasonable assumptions, such as: (i) the processing time of a task is non-increasing with the number of allocated processors and (ii) the work (processing time × number of allocated processors) is non-decreasing with the

number of processors. Under these assumptions, Jansen and Zhang [9] derive a 3.29 approximation algorithm for arbitrary precedence constraints. In the particular case of a series-parallel precedence graph, Lepere et al. [8] obtain a 2.62 approximation. However, although polynomial, these algorithms relies on complex optimization techniques, which makes them difficult to implement in a practical setting.

As presented in the previous section, the closest work to ours is the one of Prasanna and Musicus [1, 2]. Some of us have already implemented the solution of Prasanna and Musicus for sparse matrix factorization in a real multifrontal solver [10]. Due to special constraints in the task parallelization, they 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 [11]. As in [1, 2], they assumed non-integer processor allocation, which was achieved at runtime by using time-sharing among tasks.

3 Validation of the malleable task model

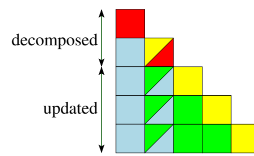
In this section, we concentrate on evaluating the model proposed by Prasanna and Musicus in [1, 2] for our target application. This model states that the speedup of a task processed on p processors is p^α. Thus, the processing time of a task T_i of size L_i which is allocated a share of processors $p_i(t)$ at time t is equal to the smallest value C_i such that

$$\int_0^{C_i} (p_i(t))^\alpha \, dt \;\geq\; L_i,$$

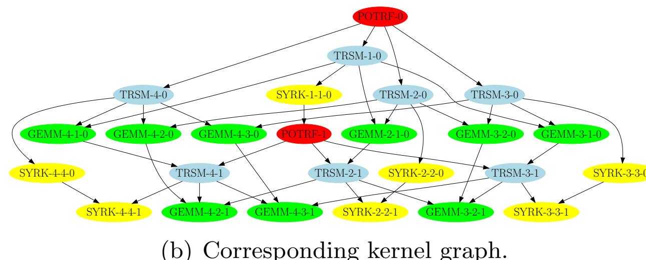
where α is a constant independent of the task. When dealing with a constant share of processors p_i, we have $C_i = L_i/p_i^\alpha$. Our goal is (i) to find whether this formula well describes the evolution of the processing time of a task for various shares of processors and (ii) to check that different tasks of the same application exhibit the same α parameter. The platform we are targeting is a modern multicore platform composed of a set of nodes each including a number of multicore processors. For the purpose of this study we will 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, where each task may be processed in parallel. 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 [12] as implemented in `MUMPS` [13] or `qr_mumps` [14] or the supernodal approach as implemented in `SuperLU` [15] or in `PaStiX` [16] are based on tree-shaped

task graphs (namely the assembly tree [17, 18]). 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 for an illustration.



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



(b) Corresponding kernel graph.

Figure 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 (in particularly because of the increasing use of accelerators such as GPU or Xeon Phi), it becomes interesting to rely on an optimized dynamic runtime system to allocate and schedule the kernels on the computing resources. These runtime systems (such as StarPU [5], KAAPI [6], or PaRSEC [7]) are able to process the kernels of a given task on a prescribed subset of the computing cores, and this subset may evolve with time. This motivates the use of a malleable task model, where the share of processors allocated to a task vary with time. This approach has been recently used and evaluated [19] in the context of the `qr_mumps` solver using the StarPU runtime system.

In order to assess the fact that dense kernels used within sparse direct solvers fit the model introduced by Prasanna and Musicus in [2] we conducted an experimental study on several dense linear algebra kernels. 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 2, 3 and 4 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. All these values are very close to one, which means that the parallelization is almost perfect.

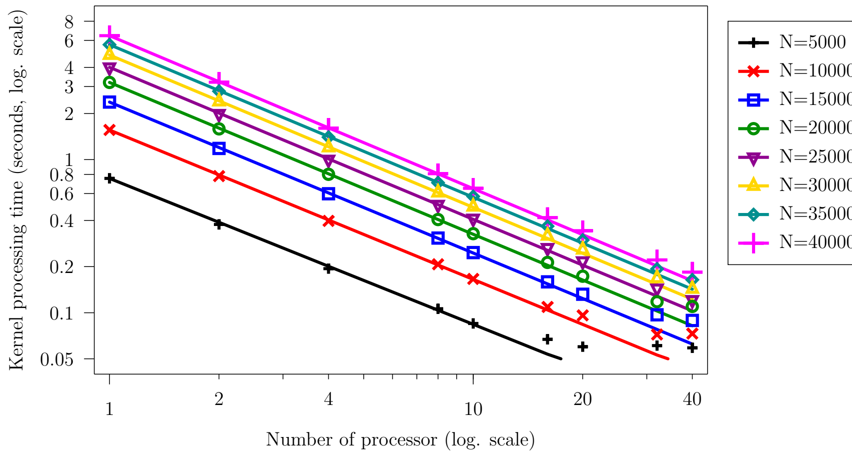


Figure 2: Timings (points) and model (lines) of QR kernel for $M = 1024$

Figures 5 and 6 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 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

[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).

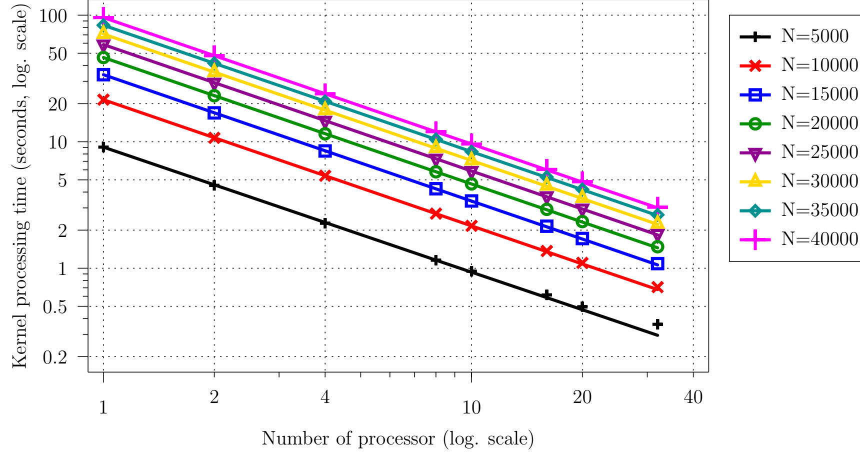


Figure 3: Timings (points) and model (lines) for QR with $M = 4096$

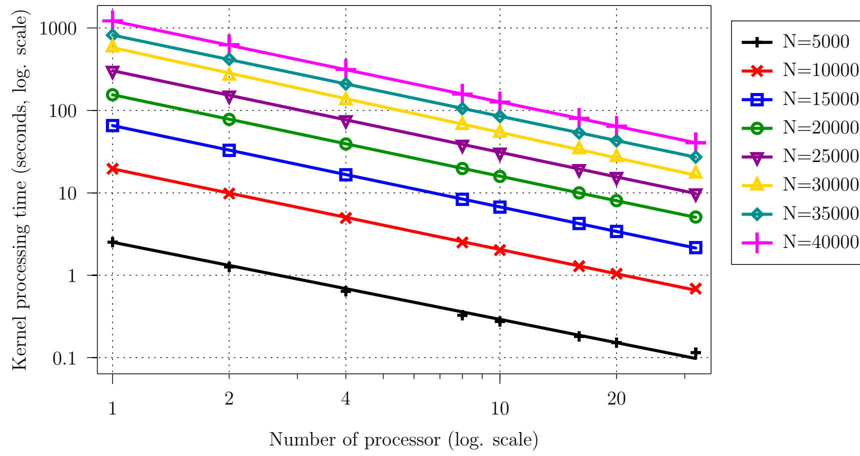


Figure 4: Timings (points) and model (lines) of Cholesky kernel

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: Values of α measured for dense kernels

(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 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.

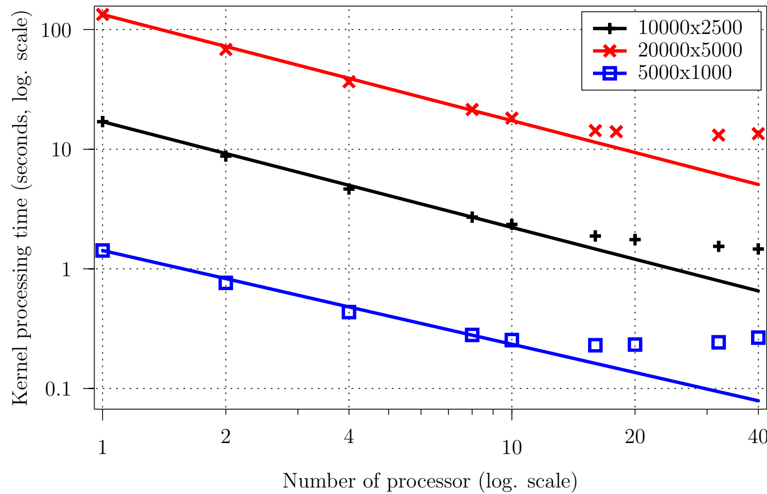


Figure 5: Timings (points) and model (lines) of `qr_mumps` frontal matrix factorization kernel with 1D partitioning.

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 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.

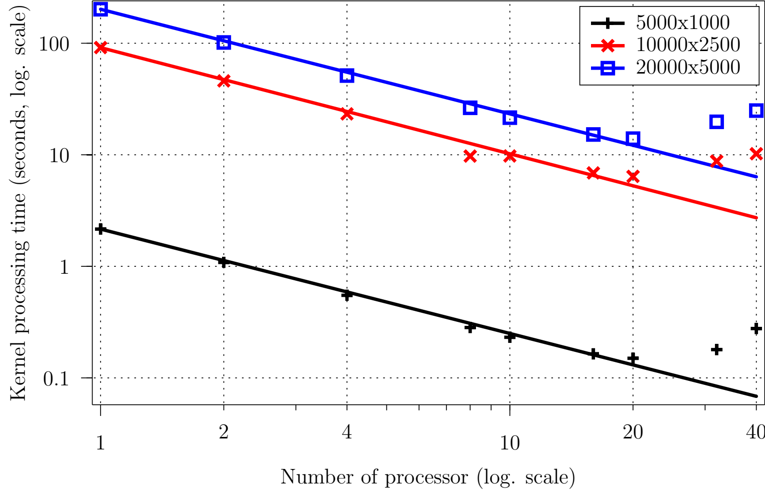


Figure 6: Timings (points) and model (lines) of `qr_mumps` frontal matrix factorization kernel with 2D partitioning.

4 Model and notations

We now present the application and platform models and introduce the notations used throughout the paper.

We consider an in-tree G of n malleable tasks $T_1, \ldots, T_n$. L_i denotes the length, that is the sequential processing time, of task T_i. The set of children of T_i in the tree is denoted by $\mathit{Children}\,(T_i)$ and its parent by $\mathit{parent}\,(T_i)$. Due to precedence constraints, T_i cannot be started until all tasks in $\mathit{Children}\,(T_i)$ are completed.

We consider that the number of available computing resources may vary with time: $p(t)$ gives the (possibly rational) 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. Although our study is motivated by an application running on a single multicore node (as outlined in the previous section), we use the term *processor* instead of *computing core* in the following sections for readability and consistency with the scheduling literature.

As motivated in the previous section, we assume that the speedup function for a task allocated p processors is p^α, with $0 < \alpha \leq 1$ is a fixed parameter. A schedule $\mathcal{S}$ is a set of nonnegative piecewise continuous functions $\left\{p_i(t) \;\middle|\; i \in I\right\}$ 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 $\int_\Delta p_i(t)^\alpha \, \mathrm{d}t$. Then, T_i is completed when the total work performed is equal to its length L_i. The completion time of task T_i is thus the smallest value C_i such that $\int_0^{C_i} p_i(t)^\alpha dt \geq L_i$. We define $w_i(t)$ as the ratio of work of the task T_i that is done during the time interval $[0, t]$: $w_i(t) = \int_0^t p_i(x)^\alpha \, \mathrm{d}x / L_i$. A schedule is a valid solution if and only if:

- it does not use more processors than available: $\forall t, \sum_{i \in I} p_i(t) \leq p(t))$;
- it completes all the tasks: $\exists \tau,\ \forall i \in I\ w_i(\tau) = 1$;
- and it respects precedence constraints: $\forall i \in I, \forall t$, if $p_i(t) > 0$ then, $\forall j \in \mathit{Children}\,(T_i), w_j(t) = 1$.

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

Note that because of the speedup function p^α, the computations in the following sections will make a heavy use of the functions $f : x \mapsto x^\alpha$ and $g : x \mapsto x^{(1/\alpha)}$. We assume that we have at our disposal a polynomial time algorithm to compute both f and g. We are aware that this assumption is very likely to be wrong, as soon as $\alpha < 1$, since f and g 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 f and g 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). A SP graph is recursively defined as a single task, a series composition of two SP graphs, or a parallel composition of two SP graphs. The two subgraphs in a parallel composition are called branches. Series compositions are ordered so that it is clear which subgraph should be executed first. The resulting Directed Acyclic Graph expresses the precedence constraints. Two nodes having the same set of predecessors, i.e., the same in-neighbors are called *siblings*.

A tree can easily be transformed into an SP graph by joining the leaves according to its structure (see Figure 7), the resulting graph is then called a *pseudo-tree*. 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 7 can be represented as $\left(\left(\left(((4 \parallel 5) \parallel 6)\ ;\ 2\right) \parallel 3\right)\ ;\ 1\right)$. Thanks to the construction of pseudo-trees, an algorithm which solves the previous scheduling problem on SP-graphs also gives an optimal solution for trees.

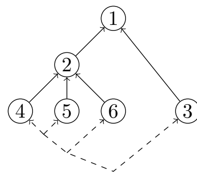


Figure 7: Example of a tree transformed into an SP graph.

5 Optimal solution for shared-memory platforms

The purpose of this section is to give a simpler proof of the results of [1, 2] using only scheduling arguments. We consider a SP-graph G as presented in the previous section to be scheduled on a shared-memory platform, which allows each task to be distributed across the whole platform. We consider here the case $\alpha < 1$ to prove the uniqueness of the optimal schedule.

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. Moreover, we prove that in an optimal schedule, all siblings terminate at the same time. In addition, the ratio of processors allocated to each task T_i, defined by $r_i(t) = p_i(t)/p(t)$, is constant and equal to r_i from the moment at which T_i is initiated to the moment at which it is terminated. Similarly, for each branch of a parallel composition, the total ratio of processors allocated to the set of tasks composing this branch is constant.

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, it means that the whole schedule can be defined by the ratios of processors allocated to the leaves at the beginning of the schedule. Then, all the children of a node T_i terminate at the same time, and its ratio r_i becomes the sum of its children ratios.

We first need to define the length $\mathcal{L}_G$ 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. *We recursively define the length $\mathcal{L}_G$ associated to a SP graph G:*

- $\mathcal{L}_{T_i} = L_i$
- $\mathcal{L}_{G_1 \,;\, G_2} = \mathcal{L}_{G_1} + \mathcal{L}_{G_2}$
- $\mathcal{L}_{G_1 \,\|\, G_2} = \left(\mathcal{L}_{G_1}^{1/\alpha} + \mathcal{L}_{G_2}^{1/\alpha}\right)^{\alpha}$

Lemma 2. *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, and we therefore obtain a schedule with a shorter makespan. □

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

Lemma 3. *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 $\mathcal{P}$ of makespan M, 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 is the constant number of available processors.

We want to build a valid schedule $\mathcal{R}$ with a makespan smaller than M. To achieve this, we define three intermediate and not necessarily valid schedules $\mathcal{Q}_1$, $\mathcal{Q}_2$, and $\mathcal{Q}_3$, which nevertheless respect the resource constraint (no more than p processors are used at time t). These schedules will be equal to $\mathcal{P}$ except on Δ where successive modifications will eventually allow to complete strictly before the time t_2 all the work done under $\mathcal{P}$ by the time t_2.

The constant share of processors allocated to task T_i on Δ in $\mathcal{Q}_1$ 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 2. We get $\sum_{i \in I} q_i = p$. So $\mathcal{Q}_1$ respects the resource constraint. Let $W_i^\Delta(\mathcal{P})$ (resp. $W_i^\Delta(\mathcal{Q}_1)$) denotes the work done on T_i during Δ under schedule $\mathcal{P}$ (resp. $\mathcal{Q}_1$). We have

$$W_i^\Delta(\mathcal{P}) = \int_\Delta p_i(t)^\alpha dt = |\Delta| \int_{[0,1]} p_i(t_1 + t|\Delta|)^\alpha dt$$

$$W_i^\Delta(\mathcal{Q}_1) = \int_\Delta \left(\frac{1}{|\Delta|}\int_\Delta p_i(t)dt\right)^\alpha dx$$

$$= |\Delta| \left(\int_{[0,1]} p_i(t_1 + t|\Delta|)dt\right)^\alpha$$

As $\alpha < 1$, the function $x \mapsto x^\alpha$ is concave and then, by Jensen inequality, $W_i^\Delta(\mathcal{P}) \le W_i^\Delta(\mathcal{Q}_1)$. 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 [20]. Then, as p_j is not constant on Δ, and cannot be made constant by modifications on a set of null measure, we have $W_j^\Delta(\mathcal{P}) < W_j^\Delta(\mathcal{Q}_1)$.

Because $W_j^\Delta(\mathcal{Q}_1) > W_j^\Delta(\mathcal{P})$, j is allocated too many processors under $\mathcal{Q}_1$. We partition this surplus of processors among the tasks other than j executed during Δ. There exists $\varepsilon > 0$ such that, if T_j is allocated the constant share of processors $q_j - (n-1)\varepsilon$ during Δ (where $n = |I|$), the work on T_j will still be strictly larger than under $\mathcal{P}$. We create from $\mathcal{Q}_1$ a

[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.

new schedule $\mathcal{Q}_2$. $\mathcal{Q}_1$ is identical to $\mathcal{Q}_2$ except that on Δ, task j is allocated a share $q_j - (n-1)\varepsilon$, and any task $i \in I$, $i \neq j$, is allocated a share $q_i + \varepsilon$. $\mathcal{Q}_2$ still respects the resource constraint.

Then, the work performed during Δ on each task of I is strictly larger under $\mathcal{Q}_2$ than under $\mathcal{P}$. Therefore, there exists $t'_2 < t_2$ such that for all $i \in I$, $W_i^{[t_1,t_2]}(\mathcal{P}) < W_i^{[t_1,t'_2]}(\mathcal{Q}_2)$. Let $d = t_2 - t'_2$. We create a schedule $\mathcal{Q}_3$ from $\mathcal{Q}_2$. The only modification is that the share of processors allocated to task $i \in I$ drops to 0 at the time $t^{(i)} < t'_2$ where $W_i^{[t_1,t_2]}(\mathcal{P}) = W_i^{[t_1,t^{(i)}]}(\mathcal{Q}_2)$. We construct a last schedule $\mathcal{R}$ defined on $[0, M-d]$. $\mathcal{R}$ is equal to $\mathcal{P}$ on $[0, t_1]$ and to $\mathcal{Q}_3$ on $[t_1, t'_2]$. On $[t'_2, M-d]$, $\mathcal{R}(t)$ is equal to $\mathcal{P}(t+d)$. Then, $\mathcal{R}$ is a valid schedule whose makespan is strictly shorter than that of an optimal schedule. 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 4. *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 $r_1(t)$ is constant and equal to $\pi_1 = 1 \Big/ \left(1 + (L_2/L_1)^{1/\alpha}\right) = L_1^{1/\alpha} \Big/ \mathcal{L}_{1\,\|\,2}^{1/\alpha}$ up to the completion of G.*

Proof. First, we prove that $r_1(t)$ is constant on any optimal schedule. Therefore, as by Lemma 2 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 $\mathcal{S}$, 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 $\mathcal{S}$ does not complete before the end of B. Let $|A|$ and $|B|$ be the durations of intervals A and B. By Lemma 3, the ratio of processors $r_1(t)$ allocated to T_1 in $\mathcal{S}$ has a constant value r_1^A on A and a constant value r_1^B on B (these values can potentially be 0 or 1). We want to prove that $r_1^A = r_1^B$. Suppose by contradiction that $r_1^A \neq r_1^B$. We assume without loss of generality that $r_1^A < r_1^B$.

We want to prove that $\mathcal{S}$ is not optimal, and so that we can do the same work than $\mathcal{S}$ does on $A \cup B$ in a smaller makespan. We assume without loss of generality that $|A|p^\alpha = |B|q^\alpha$ (otherwise, we shorten one interval to decrease $|A|$ or $|B|$). We set $r_1 = \left(r_1^A + r_1^B\right)/\,2$. We define the schedule $\mathcal{S}'$ as equal to $\mathcal{S}$ except on $A \cup B$ where the ratio allocated to T_1 is r_1 (see Figure 8). The work W_1 on task T_1 under $\mathcal{S}$ and W'_1 under $\mathcal{S}'$ during $A \cup B$ are equal to

$$W_1 = |A|p^\alpha \left(r_1^A\right)^\alpha + |B|q^\alpha \left(r_1^B\right)^\alpha$$
$$W'_1 = r_1^\alpha \left(|A|p^\alpha + |B|q^\alpha\right)$$

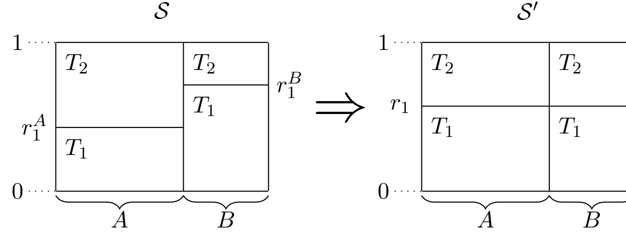


Figure 8: Schedules $\mathcal{S}$ and $\mathcal{S}'$ on $A \cup B$. The abscissae represent the time and the ordinates the ratio of processing power.

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

$$\begin{aligned} & \frac{\left(r_1^B\right)^\alpha - (r_1)^\alpha}{r_1^B - r_1} < \frac{(r_1)^\alpha - \left(r_1^A\right)^\alpha}{r_1 - r_1^A} \\ \Rightarrow \quad & |B|q^\alpha \left(\left(r_1^B\right)^\alpha - (r_1)^\alpha\right) < |A|p^\alpha \left((r_1)^\alpha - \left(r_1^A\right)^\alpha\right) \\ \Rightarrow \quad & |A|p^\alpha \left(r_1^A\right)^\alpha + |B|q^\alpha \left(r_1^B\right)^\alpha < r_1^\alpha \left(|A|p^\alpha + |B|q^\alpha\right) \\ \Rightarrow \quad & W_1 < W_1' \end{aligned}$$

because $r_1^B - r_1 = r_1 - r_1^A$ and $|B|q^\alpha = |A|p^\alpha$. Symmetrically, we also have $W_2 < W_2'$. Therefore, $\mathcal{S}'$ performs strictly more work for each task during $A \cup B$ than $\mathcal{S}$. Thus $\mathcal{S}'$ can be modified as in the proof of Lemma 3 to do the same work in a smaller makespan. Therefore, $\mathcal{S}$ is not optimal.

There remains to prove that in an optimal schedule $\mathcal{S}$, $r_1(t) = \pi_1$; 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. The sum of the durations of the A_k's is the makespan of $\mathcal{S}$. Then, as $\mathcal{S}$ 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:

$$L_1 = \sum_k |A_k| r_1^\alpha p_k^\alpha = r_1^\alpha V$$

$$\text{and} \qquad L_2 = \sum_k |A_k| (1 - r_1)^\alpha p_k^\alpha = (1 - r_1)^\alpha V$$

$$\text{Then, } L_2 = (1 - r_1)^\alpha \frac{L_1}{r_1^\alpha} \text{ and } r_1 = \frac{1}{1 + \left(\frac{L_2}{L_1}\right)^{1/\alpha}} = \pi_1.$$

□

Lemma 5. *Let G be the parallel composition of tasks T_1 and T_2, with $p(t)$ a step function, and $\mathcal{S}$ an optimal schedule. Then, the makespan of G under $\mathcal{S}$ is equal to the makespan of the task T_G of length $\mathcal{L}_G = \mathcal{L}_{1 \parallel 2}$.*

Proof. We characterize $p(t)$ by the sequences (A_k) and (p_k) as in the proof of Lemma 4. Let Δ be the domain of definition of $\mathcal{S}$, so that $\Delta = [0, \tau]$. We define, for each task T_i, the function $w_i(t)$ representing the ratio of its work done during $[0, t]$: $w_i(t) = \int_0^t p_i(x)^\alpha \, dx / L_i$; hence, $w_1(0) = 0$ and $w_1(\tau) = 1$. For $t \in \Delta$, 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 4, the ratio of processors allocated to T_1 is constant over Δ and equal to:

$$r_1 = \frac{L_1^{1/\alpha}}{L_1^{1/\alpha} + L_2^{1/\alpha}} = \left(L_1 / \mathcal{L}_{1\,\|\,2}\right)^{1/\alpha}.$$

Then, we have:

$$\begin{aligned} w_1(t) &= \frac{\bar{t}\left(p_{k(t)} r_1\right)^\alpha + \sum_{k<k(t)} |A_k| \left(p_k r_1\right)^\alpha}{L_1} \\ &= \frac{\bar{t} p_{k(t)}^\alpha + \sum_{k<k(t)} |A_k| p_k^\alpha}{\mathcal{L}_{1\,\|\,2}} \end{aligned}$$

Similarly, for T_2, we have:

$$\begin{aligned} w_2(t) &= \frac{\bar{t}\left(p_{k(t)}(1-r_1)\right)^\alpha + \sum_{k<k(t)} |A_k| \left(p_k(1-r_1)\right)^\alpha}{L_2} \\ &= \frac{\bar{t} p_{k(t)}^\alpha + \sum_{k<k(t)} |A_k| p_k^\alpha}{\mathcal{L}_{1\,\|\,2}} \end{aligned}$$

We define $w(t)$ as the ratio of work that is done for the equivalent task T_G of length $\mathcal{L}_{1\,\|\,2}$ under the processor profile $p(t)$, until the task is terminated. We have:

$$w(t) = \frac{t p_{k(t)}^\alpha + \sum_{k<k(t)} A_k p_k^\alpha}{\mathcal{L}_{1\,\|\,2}} = w_1(t) = w_2(t).$$

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

Theorem 6. *For every graph G, if $p(t)$ is a step function, G has the same optimal makespan than the equivalent task T_G of length $\mathcal{L}_G$. 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, this is implicitly its optimal makespan. We first remark that in any optimal schedule, as $p(t)$ is a step function and because of Lemma 3, only step functions are used to allocate processors to tasks, and so Lemma 5 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, as by Lemma 2, all processors have to be used.
- G is the series composition of G_1 and G_2. By induction, G_1 (resp. G_2) has the same makespan than task T_1 (resp. T_2) of length $\mathcal{L}_1$ (resp. $\mathcal{L}_2$) under any processor profile. Therefore, the makespan of the series composition of T_1 and T_2 is equal to the makespan of G. Then, it is equal to the makespan of the task of length $\mathcal{L}_G = \mathcal{L}_{1\,;\,2} = \mathcal{L}_1 + \mathcal{L}_2$.

  By induction, the unique optimal schedules of G_1 and G_2 under $p(t)$ processors can be computed, so there is a unique optimal schedule of G under $p(t)$ processors: the concatenation of these two schedules.
- G is the parallel composition of G_1 and G_2. By induction, G_1 (resp. G_2) has the same makespan than task T_1 (resp. T_2) of length $\mathcal{L}_1$ (resp. $\mathcal{L}_2$) under any processor profile. Consider an optimal schedule $\mathcal{S}$ of G and let $p_1(t)$ be the processor profile allocated to G_1. Let $\tilde{S}$ be the schedule of $(T_1 \,\|\, T_2)$ that allocates $p_1(t)$ processors to T_1. $\tilde{S}$ is optimal and gives the same makespan as $\mathcal{S}$ for G because T_1 and G_1 (resp. T_2 and G_2) have the same makespan under any processor profile. Then, by Lemma 5, $\tilde{S}$ (so $\mathcal{S}$) gives the makespan equal to the optimal makespan of $\mathcal{L}_{1\,\|\,2} = \mathcal{L}_G$. Moreover, by Lemma 4 applied on $(T_1 \,\|\, T_2)$, 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, 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 ratios π_1 and π_2 for each parallel composition can also be computed in polynomial time. Finally, these ratios imply the computation in linear time of the ratios of the processor profile that should be allocated to each task after its children are completed, which describes the optimal schedule.

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 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 $\mathcal{R}$, renders the problem much more difficult. We

concentrate first on platforms with two homogeneous nodes and then with two heterogeneous nodes.

In this part, we refer to the shared-memory makespan-minimizing schedule induced by Theorem 6, which is denoted by the PM schedule (that stands for Prasanna and Musicus, who first depicted it).

6.1 Two homogeneous multicore nodes

In this section, we consider a multicore processor 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 7. *Given two homogenous nodes of p processors, n independent tasks of sizes $L_1, ..., L_n$ and a bound T, the problem of finding a schedule of the n tasks on the two nodes that respects $\mathcal{R}$, 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 [21], and uses tasks of length $L_i = a_i^\alpha$, 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 if there exists a partition of A in two sets that sum to the same value. Let $s = \sum_i a_i$. We reduce this problem to the homogeneous scheduling problem. Let $p = s/2$ and let L_i for $1 \leq i \leq n$ be equal to $L_i = a_i^\alpha$. We recall that the computation of the L_is is assumed polynomial. Let $\mathcal{J}$ be the instance of the homogeneous scheduling problem composed of p, the L_is and the bound $T = 1$. We show that there is a solution to the partition problem A if and only if $\mathcal{I}$ has a solution.

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

$$M = \left(\frac{\sum_i L_i^{1/\alpha}}{2p}\right)^\alpha = 1 = T$$

Then, by Theorem 6, 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 $p_i = 2p \cdot L_i^{1/\alpha}/s = a_i$ (such schedules are not differentiated in the shared memory model of previous section). Therefore, only such a schedule can be a solution to $\mathcal{I}$.

Such a schedule respects the $\mathcal{R}$ 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 $p = s/2$, 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. We recall that a ρ-approximation provides on each instance a solution whose objective z is such that $z \leq \rho z*$, where $z*$ is the optimal value of the objective on this instance.

Theorem 8. *There exists a polynomial time $\left(\frac{4}{3}\right)^\alpha$-approximation algorithm for the makespan minimization problem of a tree of malleable tasks on two homogenous nodes.*

The proof of this theorem is done by induction on the structure of the tree and relies on some lemmas. The approximation algorithm is summarized in Algorithm 11. Lemmas 10 and 15 are directly used in the proof and require Definitions 11 and 12. The proof of Lemma 15 relies on Lemmas 13 and 14. Lemma 9 allows the restriction to a slightly simpler class of graphs.

We denote by G the tree to be scheduled, and p is the number of processors of each node. Let $\mathcal{S}_{\text{OPT}}$ be a makespan-optimal schedule of G of makespan M_{OPT}. We also consider the optimal schedule $\mathcal{S}_{\text{PM}}$ of G on a single node made of $2p$ processors (without the constraint $\mathcal{R}$). $\mathcal{S}_{\text{PM}}$ can be computed as described in the previous section. Its makespan is $M_{2p}^{\text{PM}} = \mathcal{L}_G/(2p)^\alpha$, which is a lower bound on M_{OPT}.

One can observe that a 2^α approximation is immediate: a solution is the PM schedule of G under only p processors, whose makespan is $M_p^{\text{PM}} = \mathcal{L}_G/p^\alpha$. As the optimal makespan is not smaller than M_{2p}^{PM}, M_p^{PM} is indeed a 2^α-approximation.

Let $\{c_i\}$, for $i \in [1, n_c]$, be the set of children of the root of G, and let C_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 $\mathcal{L}_{C_i}$'s are in non-increasing order. We denote $\sigma_c = \sum_{i=1}^{n_c} \mathcal{L}_{C_i}^{1/\alpha}$, and $x = \frac{2\mathcal{L}_{C_1}^{1/\alpha}}{\sigma_c}$, which means that xp processors are dedicated to C_1 in $\mathcal{S}_{\text{PM}}$.

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

Lemma 9. *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. 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 $\tilde{G}$. It is then immediate to adapt it to the original graph, by allocating p processors to each task of this chain.*

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

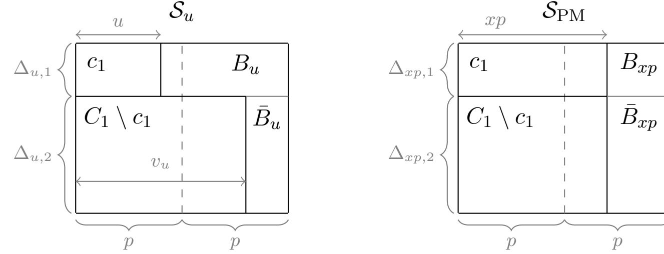


Figure 9: A schedule $\mathcal{S}_u$, for $u < p$ on the left and the schedule $\mathcal{S}_{\text{PM}} = \mathcal{S}_{xp}$ on the right.

Lemma 10. *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 share allocated to C_i in $\mathcal{S}_{\text{PM}}$. Each p_i is constant because $\mathcal{S}_{\text{PM}}$ is PFC by definition. By hypothesis, we have $p_i \leq p$ for all i, as $\mathcal{L}_{C_1}$ is the largest $\mathcal{L}_{C_i}$ and its share is equal to $xp \leq p$.

If $n_c = 2$, both p_1 and p_2 are not larger than p so equal p. Therefore, the schedule $\mathcal{S}_{\text{PM}}$ respects restriction $\mathcal{R}$, 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 greater 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 greater 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 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. We now compute the PM schedule of S_1 with p processors and $S_2 \cup S_3$ with p processors. The makespan is then $M = \max\left(\mathcal{L}_{S_1}, \mathcal{L}_{S_2 \parallel S_3}\right)/p^{\alpha} = \mathcal{L}_{S_2 \parallel S_3}/p^{\alpha}$. Indeed, we have $\Sigma_1 \leq p \leq \Sigma_2 + \Sigma_3$ and $\mathcal{L}_{S_1}/\Sigma_1^{\alpha} = \mathcal{L}_{S_2 \parallel S_3}/(\Sigma_2 + \Sigma_3)^{\alpha}$, as these quantities represent the makespan of each subpart of the tree in $\mathcal{S}_{\text{PM}}$, and all subtrees C_i terminate simultaneously in $\mathcal{S}_{\text{PM}}$. So $\mathcal{L}_{S_1} < \mathcal{L}_{S_2 \parallel S_3}$.

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 $\mathcal{S}_{\text{PM}}$, $\Sigma_2 + \Sigma_3 \leq \frac{4}{3}p$ processors are allocated to $S_2 \cup S_3$. Then, the makespan of $\mathcal{S}_{\text{PM}}$ verifies $M_{2p}^{\text{PM}} \geq \mathcal{L}_{S_2 \parallel S_3}/\left(\frac{4}{3}p\right)^{\alpha}$, and so $M/M_{2p}^{\text{PM}} \leq \left(\frac{4}{3}\right)^{\alpha}$. Therefore, the schedule is indeed a $\left(\frac{4}{3}\right)^{\alpha}$approximation. □

Definition 11. *For any $0 < q \leq p$, let $\mathcal{R}_q$ be the constraint that forces q processors to be allocated to c_1.*

We denote by B the subgraph $G \setminus \textit{root} \setminus C_1$.

Definition 12. *We define the schedule $\mathcal{S}_u$ parametrized by $u \in]0,p] \cup \{xp\}$, which respects $\mathcal{R}_u$ but not $\mathcal{R}$. 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 $C_1 \setminus c_1$ and of the potential remaining part $\bar{B}_u$ of B, is scheduled on $2p$ processors by a PM schedule, regardless of the $\mathcal{R}$ constraint. We denote by v_u the share allocated to $C_1 \setminus c_1$ and by M_u the makespan of the schedule.*

See Figure 9 for an illustration of this definition. Let $G_{u,1}$ be the graph $c_1 \parallel B_u$ and $G_{u,2}$ be the graph $(C_1 \setminus c_1) \parallel \bar{B}_u$. We denote by $\Delta_{u,1}$ (resp. $\Delta_{u,2}$) the execution time of $G_{u,1}$ (resp. $G_{u,2}$) in $\mathcal{S}_u$. Then, $M_u = \Delta_{u,1} + \Delta_{u,2}$.

One can note that the PM schedule $\mathcal{S}_{\text{PM}}$ is equal to $\mathcal{S}_{xp}$, where $u = v_u = xp$.

Lemma 13. *For any $u \in]0,p]$, under the constraint $\mathcal{R}_u$, the makespan-optimal schedule is M_u.*

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

First, suppose that c_1 terminates before B in $\mathcal{S}$. This means that a time range is dedicated to schedule B at the end of the schedule. We can modify slightly $\mathcal{S}$ by moving this time range to the beginning of the schedule. This is possible as there is no heredity constraint between B and C_1, and the same tasks of B can be performed in the new processor profile, by a PM schedule on B. So we now assume that the schedule terminates at the execution of c_1.

Because of $\mathcal{R}_u$, $\mathcal{S}$ 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 $C_1 \setminus c_1$ can be executed.

Suppose that $\mathcal{S}$ differs from $\mathcal{S}_u$ in the execution of B in parallel to c_1. Consider the schedule of C_1 fixed, and let $p_b(t)$ be the number of processors allocated to B at the time t in $\mathcal{S}$. As B_u is scheduled according to PM ratios in $\mathcal{S}_u$, and $\mathcal{S}$ differs from this schedule, in $\mathcal{S}$, B is not scheduled according to PM ratios under the processor profile $p_b(t)$. Then, this schedule can be modified to schedule B in a smaller makespan, and then to schedule the whole graph G in a smaller makespan, which contradicts the makespan-optimality of $\mathcal{S}$.

So $\mathcal{S}$ and $\mathcal{S}_u$ are equal during the time interval $\Delta_{u,1}$. Then, it remains to schedule the graph $G_{u,2} = (C_1 \setminus c_1) \parallel \bar{B}_u$, which has a unique optimal schedule, the PM schedule, that is followed by $\mathcal{S}_u$. Therefore, $\mathcal{S} = \mathcal{S}_u$. □

Lemma 14. *$\mathcal{S}_p$ is the makespan-optimal schedule among the $\mathcal{S}_w$ for $w \in]0,p]$, i.e., we have $p = \arg\min_{w \in]0,p]} (M_w)$.*

Proof. Let $u_{\text{OPT}} = \underset{w\in]0,p]}{\arg\min}\,(M_w)$. 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 $\mathcal{S}_u$, which is makespan-optimal among the $\mathcal{S}_w$, for $w \in]0,p]$.

Suppose by contradiction that $u < p$. We will build an other schedule $\bar{\mathcal{S}}$ following the constraint $\mathcal{R}_{\bar{u}}$ for a certain $\bar{u}$ respecting $u < \bar{u} < p$, that will give a contradiction with the optimality of $\mathcal{S}_u$.

The following paragraphs prove the inequality $v > p$, which can be intuitively deduced by an observation of the schedules.

As we have $x > 1$, we know that $\mathcal{L}_{C_1\setminus c_1} > \mathcal{L}_{\bar{B}_{xp}} = \mathcal{L}_B - \mathcal{L}_{B_{xp}} > \mathcal{L}_B - c_1$. The first inequality holds because in $\mathcal{S}_{xp}$, the subgraphs $C_1 \setminus c_1$ 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 $C_1 \setminus c_1$ (resp. $\bar{B}_{xp}$). Therefore, we get $\Delta_{2,xp} = \mathcal{L}_{C_1\setminus c_1}/(xp)^\alpha = \mathcal{L}_{\bar{B}_{xp}}/((2-x)p)^\alpha$. As $x > 1$, more processors are allocated to $C_1 \setminus c_1$, so $\mathcal{L}_{C_1\setminus c_1} > \mathcal{L}_{\bar{B}_{xp}}$. By the same reasoning between c_1 and B_{xp} in $\mathcal{S}_{xp}$, we get $\mathcal{L}_{B_{xp}} < c_1$ and the second inequality holds. See Figure 9 for an illustration.

With similar arguments between the subgraphs c_1 and B_u in the schedule $\mathcal{S}_u$, and using the hypothesis $u < p$, we get $\mathcal{L}_{B_u} > c_1$. 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: $\Delta_{1,u} = \mathcal{L}_{B_u}/(2p - u)^\alpha = c_1/u^\alpha$.

Combining these two inequalities, we have $\mathcal{L}_{\bar{B}_u} < \mathcal{L}_B - c_1 < \mathcal{L}_{C_1\setminus c_1}$, and by using the same reasoning in the other way with the parallel execution of $C_1 \setminus c_1$ and $\bar{B}_u$ in $\mathcal{S}_u$, we finally prove $v > p$.

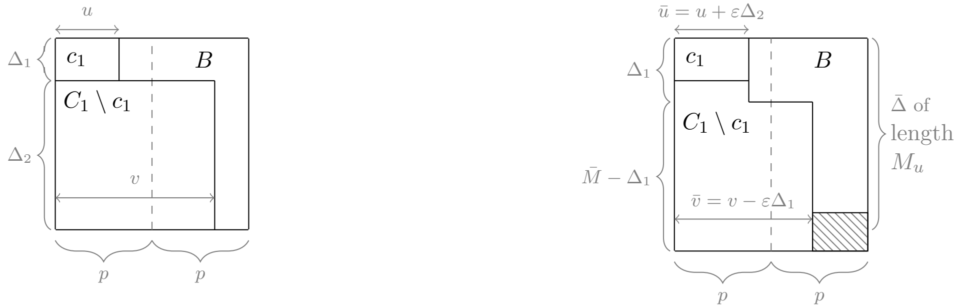


Figure 10: Schedules $\mathcal{S}_u$ (left) and $\bar{\mathcal{S}}$(right), assuming that B begins after C_1 in $\bar{\mathcal{S}}$.

Let $\varepsilon > 0$ small enough such that $u + \varepsilon\Delta_2 < p$ and $v - \varepsilon\Delta_1 > p$. Let $\bar{u} = u + \varepsilon\Delta_2$ and $\bar{v} = v - \varepsilon\Delta_1$. One can note that $0 < u < \bar{u} < p < \bar{v} < v$.

Let $\bar{\mathcal{S}}$ be the schedule allocating $\bar{u}$ processors to C_1 during a time Δ_1 at the end of the schedule, and $\bar{v}$ processors to C_1 before. The subgraph B is scheduled following PM ratios in parallel to C_1, in such a way that it terminates at the same time than c_1 and there is no idle time after the beginning of its execution. The subgraph C_1 is scheduled in the same way than B, following PM ratios as soon as its execution begins. One can note that B and C_1 do not necessarily begin simultaneously. See Figure 10 for an illustration of the case where B begins after C_1. Let $\bar{M}$ be the makespan of $\bar{\mathcal{S}}$.

As $\bar{u} > u$, c_1 is completed in a time smaller than Δ_1 in $\bar{\mathcal{S}}$, so $\bar{\mathcal{S}}$ respects the constraint $\mathcal{R}_{\bar{u}}$. Then, by Lemma 13, as $\bar{\mathcal{S}} \neq \mathcal{S}_{\bar{u}}$, we know that $\bar{M} > M_{\bar{u}}$. In addition, by the definition of M_u, we get $\bar{M} > M_{\bar{u}} \geq M_u$.

We can assume without loss of generality that C_1 and B are both a unique task. This can be reached by replacing them by their equivalent task, which does not change their execution time. We do not lose generality by this transformation here because both subgraphs are scheduled under the PM rules. For example, we could not consider the equivalent task of the total graph G because constraints of the form $\mathcal{R}_w$, with $w \neq xp$, are followed, and so the PM rules are not respected.

Let $\bar{\Delta}$ be the interval of time ending when $\mathcal{S}_u$ terminates and having a length of M_u. See Figure 10 for an illustration.

Let $\bar{W}_C$ (resp. $\bar{W}_B$) be the total length of the task C_1 (resp. B) that is executed in $\bar{\mathcal{S}}$ during $\bar{\Delta}$. Similarly, we define W_C (resp. W_B) the total length of the task C_1 (resp. B) that is executed in $\mathcal{S}_u$. These two last quantities are equal to:

$$W_C = \Delta_1 u^\alpha + \Delta_2 v^\alpha \, (= \mathcal{L}_{C_1})$$
$$W_B = \Delta_1 (2p - u)^\alpha + \Delta_2 (2p - v)^\alpha \, (= \mathcal{L}_B)$$

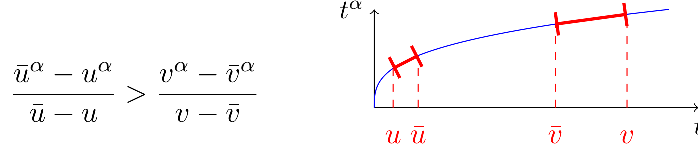
As $\bar{M} > M_u$, we must not have simultaneously $\bar{W}_C \geq W_C$ and $\bar{W}_B \geq W_B$. Indeed, if it was the case, then all the tasks of G would be completed by $\bar{\mathcal{S}}$ in a makespan smaller than M_u, which is a contradiction. For each task, we separate two cases.

If C_1 begins in $\bar{\mathcal{S}}$ during $\bar{\Delta}$, then $\bar{W}_C = \mathcal{L}_C = W_C$ because the execution of C_1 would hold entirely in $\bar{\Delta}$.

Otherwise, we have:

$$\bar{W}_C = \Delta_1 \bar{u}^\alpha + \Delta_2 \bar{v}^\alpha$$

We know that $0 < u < \bar{u} < \bar{v} < v$. Therefore, by the concavity of the function $t \mapsto t^\alpha$, we have the inequality below. The plot on the right illustrates the inequality. Indeed, the slope of the red segment corresponding to u and $\bar{u}$ is larger than the other one.



Then, we derive from this inequality the fact that $\bar{W}_C$ is larger than W_C:

$$\begin{aligned}
\frac{\bar{u}^\alpha - u^\alpha}{\bar{u} - u} &> \frac{v^\alpha - \bar{v}^\alpha}{v - \bar{v}} \\
\frac{\bar{u}^\alpha - u^\alpha}{\varepsilon \Delta_2} &> \frac{v^\alpha - \bar{v}^\alpha}{\varepsilon \Delta_1} \\
\Delta_1 \left(\bar{u}^\alpha - u^\alpha\right) &> \Delta_2 \left(v^\alpha - \bar{v}^\alpha\right) \\
\Delta_1 \bar{u}^\alpha + \Delta_2 \bar{v}^\alpha &> \Delta_1 u^\alpha + \Delta_2 v^\alpha \\
\bar{W}_C &> W_C
\end{aligned}$$

Therefore, in any case, we have $\bar{W}_C \geq W_C$

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

Otherwise, we have:

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

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

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

Then, in any case, we have both $\bar{W}_C \geq W_C$ and $\bar{W}_B \geq W_B$, so we get the contradiction.

Therefore, we have $u \geq p$ and so $u = p$. □

Lemma 15. *The makespan of $\mathcal{S}_p$ is a lower bound of M_{OPT}.*

Proof. One can note that in $\mathcal{S}_{\mathrm{OPT}}$, a constant share $u_* \leq p$ processors must be allocated to c_1 due to $\mathcal{R}$, as in $\mathcal{S}_{u_*}$. Indeed, if this share is not constant, because of the concavity of the function $t \mapsto t^\alpha$, 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 4.

Let $\mathcal{R}'$ be the constraint that enforces a schedule to respect one of the constraint $\mathcal{R}_w$, for any $w \in]0, p]$. Otherwise stated, $\mathcal{R}'$ enforces a schedule to allocate a constant share w not larger than p to c_1.

Let $\mathcal{S}'_{\text{OPT}}$ be the makespan-optimal schedule respecting $\mathcal{R}'$, and let M'_{OPT} be its makespan. As $\mathcal{S}_{\text{OPT}}$ respects $\mathcal{R}'$, we have $M_{\text{OPT}} \geq M'_{\text{OPT}}$.

Furthermore, there exists $u' \leq p$ such that $\mathcal{S}'_{\text{OPT}} = \mathcal{S}_{u'}$. Therefore, $M'_{\text{OPT}} \geq \min_{w \in]0,p]} M_w$, and, by Lemma 14, we get $M'_{\text{OPT}} \geq M_p$.

Finally, we have $M_{\text{OPT}} \geq M_p$. □

We are now able to prove Theorem 8, by induction on the tree structure. The corresponding approximation algorithm is described in Algorithm 11.

Proof of Theorem 8. We prove here by induction on the tree structure of G that Algorithm 11 launched on G returns a $\left(\frac{4}{3}\right)^\alpha$-approximation.

First, we treat the initialization case, where $n = 1$. This case is included in the test $x \geq 1$ *and* x *is a leaf*. An optimal schedule is indeed the one that allocates p processors to the unique task.

As stated in Lemma 9, 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, no schedule can have a makespan smaller than $x^\alpha M_{2p}^{\text{PM}}$, as no more than p processors can be allocated to c_1. Then, the schedule that only differs from $\mathcal{S}_{\text{PM}}$ by reducing the share allocated to c_1 to p is makespan-optimal.
- if $x \leq 1$, the result is given by Lemma 10.

Now, we suppose the result true for $m < n$. 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 $\mathcal{S}_p$, whose makespan M_p is a lower bound of M_{OPT} as stated in Lemma 15.

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

More formally, we have $G_{p,2}$, which is the parallel composition $(C_1 \setminus c_1) \parallel \bar{B}_p$, composed of at most $n-1$ nodes. So, by induction, a schedule $\mathcal{S}^r$ achieving a $\left(\frac{4}{3}\right)^\alpha$-approximation can be computed for $G_{p,2}$. This means that its makespan M^r is at most $\left(\frac{4}{3}\right)^\alpha \Delta_{p,2}$, as $\mathcal{S}_p$ completes $G_{p,2}$ with PM ratios in a time $\Delta_{p,2}$, which is then the optimal time.

Require: A graph G, the number p of processor of each node of the platform $\mathcal{P}$
Ensure: A schedule $\mathcal{S}$ of G on $\mathcal{P}$ that is a $\left(\frac{4}{3}\right)^\alpha$-approximation of the makespan

1: **function** HomogeneousApp(G, p)
2: $\tilde{G} \leftarrow G$
3: Modify G as in Lemma 9
4: Compute the PM schedule $\mathcal{S}_{\text{PM}}$ of G on $2p$ processors
5: Compute the c_i's, the C_i's, B, and x
6: **if** $x \geq 1$ and c_1 is a leaf **then**
7: Build $\mathcal{S}$: shrink from $\mathcal{S}_{\text{PM}}$ the share of processors allocated to c_1 to p processors
8: **else if** $x \leq 1$ **then**
9: Build $\mathcal{S}$: map the C_i's as in Lemma 10, and compute the PM schedule on each part
10: **else** ▷ we have $x > 1$ and c_1 is not a leaf
11: Compute the schedule $\mathcal{S}_p$ and partition G in $G_{p,1}$ and $G_{p,2}$ as in Definition 12
12: $\mathcal{S}^r \leftarrow$ HomogeneousApp($G_{p,2}, p$)
13: Build $\mathcal{S}$: schedule $G_{p,2}$ as in $\mathcal{S}^r$ then $G_{p,1}$ as in $\mathcal{S}_p$
14: **end if**
15: Adapt the schedule $\mathcal{S}$ to the original graph $\tilde{G}$ if $G \neq \tilde{G}$ by scheduling the additional tasks on p processors
16: **return** $\mathcal{S}$
17: **end function**

Figure 11: Approximation algorithm for the homogeneous problem.

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

$$\begin{aligned} M = \Delta_{p,1} + M^r &\leq \Delta_{p,1} + \left(\frac{4}{3}\right)^\alpha \Delta_{p,2} \\ &\leq \left(\frac{4}{3}\right)^\alpha (\Delta_{p,1} + \Delta_{p,2}) \leq \left(\frac{4}{3}\right)^\alpha M_p \leq \left(\frac{4}{3}\right)^\alpha M_{\text{OPT}} \end{aligned}$$

Then, $\mathcal{S}$ is a $\left(\frac{4}{3}\right)^\alpha$-approximation.
Therefore, Algorithm 11 is a polynomial time $\left(\frac{4}{3}\right)^\alpha$-approximation. □

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 and will

be referred to as the p-part of the platform, while the second one includes q cores, and will be referred to as the q-part. 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 $L_1, ..., L_n$. The problem of finding a schedule of these independent tasks on two nodes of size p and q will be noted the (p,q)-SCHEDULING problem. Thanks to the homogenous case presented above, we already know that its decision version is NP-complete.

The (p,q)-SCHEDULING problem is close to the SUBSET SUM problem. Given n integers and a target s, the optimization version of SUBSET SUM is the problem of finding a subset of these integers that sum to the largest possible number not larger than s.

We first need a few definitions before stating Theorem 18 which implies the construction of an FPTAS for a restricted version of the (p,q)-SCHEDULING problem in Corollary 19.

Consider an instance $\mathcal{I}$ of (p,q)-SCHEDULING. Let S be $\sum_i x_i$, where $x_i = L_i^{1/\alpha}$ and X be the set $\{x_i, i \in [1,n]\}$. We note $r = \max\left(\frac{q}{p}, \frac{p}{q}\right)$.

We denote by A the subset of the indices of the tasks allocated to the p-part, and by $\bar{A}$ 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 $\mathcal{S}_A$.

For $0 < \kappa < 1$, a κ-approximation of SUBSET SUM returns a subset A of X such that the sum of its elements ranges between κOPT and OPT, where

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

For $\lambda > 1$, a λ-approximation of (p,q)-SCHEDULING returns a schedule such that its makespan is not larger than λ times the optimal makespan. We note $\varepsilon_\lambda = \lambda^{1/\alpha} - 1$ and $\varepsilon_\kappa = 1 - \kappa$.

An AS (approximation scheme) $\mathcal{A}$ resolving SUBSET SUM is defined as follows. Given an instance $\mathcal{J}$ of SUBSET SUM and a parameter $0 < \kappa < 1$, it computes a solution to $\mathcal{J}$ achieving a κ-approximation in a time complexity $f_{\mathcal{A}}(n, \varepsilon_\kappa)$.

An AS $\mathcal{B}$ resolving (p,q)-SCHEDULING is defined as follows. Given an instance $\mathcal{I}$ of the (p,q)-scheduling problem and a parameter $\lambda > 1$, it computes a solution to $\mathcal{I}$ achieving a λ-approximation in a time complexity $f_{\mathcal{B}}(\mathcal{J}, \varepsilon_\lambda)$.

Remark 16. *There exist a FPTAS for* SUBSET SUM*: Kellerer et al. [22] have designed a 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 achieve to design a FPTAS for a restricted version of (p,q)-SCHEDULING, where the x_is are integer. This problem is

defined in Definition 17 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 12 and its complexity when using the FPTAS of [22] is given in Corollary 19 of Theorem 18.

Definition 17. *The* (p,q)-SCHEDULING RESTRICTED *problem is defined from the* (p,q)-SCHEDULING *problem by replacing the entries* L_i *by* $x_i = L_i^{1/\alpha}$, *and is restricted to the case where the* x_i *are integers, and* p *and* q *are given in unary.*

Theorem 18. *Given a AS* $\mathcal{A}$ *of* SUBSET SUM *of time complexity* $(n,\varepsilon_\kappa) \mapsto f_{\mathcal{A}}(n,\varepsilon_\kappa)$, *Algorithm 12 performs a AS to* (p,q)-SCHEDULING RESTRICTED *with time complexity* $(n,p,q,\alpha,\lambda) \mapsto O\left(n + f_{\mathcal{A}}\left(n, \frac{\varepsilon_\lambda}{r}\right)\right)$, *assuming that raising a number to the power* α *or* $1/\alpha$ *can be done is constant time.*

Corollary 19. *The* (p,q)-SCHEDULING RESTRICTED *problem admits a FPTAS of time complexity*

$$O\left(\min\left(\frac{nr}{\lambda^{1/\alpha}-1}, n + \left(\frac{r}{\lambda^{1/\alpha}-1}\right)^2 \log\left(\frac{r}{\lambda^{1/\alpha}-1}\right)\right)\right)$$

and space complexity $0\left(n + \frac{r}{\lambda^{1/\alpha}-1}\right)$.

Indeed, parametrized by the FPTAS of [22], Algorithm 12 is an AS of the (p,q)-SCHEDULING RESTRICTED *problem of such a complexity.*

Proof of Theorem 18. Let $\mathcal{I}$ be an instance of (p,q)-SCHEDULING RESTRICTED, $\lambda > 1$ and $\mathcal{A}$ be an AS 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 part 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 $\varepsilon_\kappa = \frac{\varepsilon_\lambda}{r}$. One can check that $0 < \kappa < 1$.

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

Let $\mathcal{S}_{\text{OPT}}$ be an optimal schedule of $\mathcal{I}$. If we denote by A_{O} the subset of the tasks allocated to the p-part of the platform, we have either:

$$\begin{aligned} &\sum_{i\in A_{\text{O}}} x_i \leq \Sigma_{\text{ideal}} \leq \frac{pS}{p+q} = pM_{\text{ideal}}^{1/\alpha} \\ \text{or} \quad &\sum_{i\in \bar{A_{\text{O}}}} x_i = S - \sum_{i\in A_{\text{O}}} x_i \leq \frac{qS}{p+q} = qM_{\text{ideal}}^{1/\alpha} \end{aligned} \qquad (1)$$

We first suppose that the left inequality holds. The other case is treated at the end of the proof.

Then,

$$\left(\frac{\sum_{i\in A_{\text{O}}} x_i}{p}\right)^{\alpha} \leq M_{\text{ideal}}$$

So the p-part of the schedule terminates before the ideal schedule. Therefore, the q-part of the schedule terminates after the p-part as $M_{\text{OPT}} \geq M_{\text{ideal}}$. We denote $\Sigma_{\text{OPT}} = \sum_{i\in A_{\text{O}}} x_i$.

Then, the makespan M_{OPT} is equal to the time needed to complete the tasks of $\bar{A_{\text{O}}}$. Then we have

$$M_{\text{OPT}} = \left(\frac{\sum_{i\in \bar{A_{\text{O}}}} x_i}{q}\right)^{\alpha} = \left(\frac{S - \sum_{i\in A_{\text{O}}} x_i}{q}\right)^{\alpha}$$

Let Λ be the set of subsets of X:

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

We now prove in the following paragraph that a subset $A \in \Lambda$ is computed by the algorithm $\mathcal{A}$ launched on X, $s = \Sigma_{\text{ideal}} = \frac{pS}{p+q}$, and ε_κ. First, we recall that the x_i are assumed to be integers in the formulation of (p,q)-Scheduling Restricted.

We know that $\sum_{i\in A_{\text{O}}} x_i = \Sigma_{\text{OPT}}$, so $A_O \in \Lambda$. Then, there does not exist any subset A of X such that $\Sigma_{\text{OPT}} < \sum_{i\in A} x_i \leq \Sigma_{\text{ideal}}$, because the associated schedule $\mathcal{S}_A$ would have a makespan smaller than M_{OPT}, which contradicts the optimality of $\mathcal{S}_{\text{OPT}}$. So A_O is an optimal solution to the instance submitted to $\mathcal{A}$. Therefore, $\mathcal{A}$ launched on this instance with the parameter γ will return a set $A \in \Lambda$ in time $f_{\mathcal{A}}(n, \varepsilon_\gamma)$.

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

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

We have

$$\sum_{i\in \bar{A}} x_i = S - \sum_{i\in A} x_i$$

and

$$\sum_{i\in A} x_i \geq \kappa \Sigma_{\text{OPT}}$$

So

$$\frac{\sum_{i\in\bar{A}} x_i}{q} \leq \frac{S - \kappa\Sigma_{\mathrm{OPT}}}{q}$$

and

$$\sum_{i\in A} x_i \leq \Sigma_{\mathrm{OPT}} \leq \Sigma_{\mathrm{ideal}}$$

This last inequality implies that the tasks allocated to the p-part of the platform are terminated before M_{ideal}, and so necessarily before the tasks allocated to q-part. Therefore, we have

$$M_A = \left(\frac{\sum_{i\in\bar{A}} x_i}{q}\right)^{\alpha}$$

.

and so

$$\left(\frac{M_A}{M_{\mathrm{OPT}}}\right)^{1/\alpha} \leq \frac{S - \kappa\Sigma_{\mathrm{OPT}}}{S - \Sigma_{\mathrm{OPT}}}$$

Then, as $0 < \kappa < 1$ and $\Sigma_{\mathrm{OPT}} \leq \Sigma_{\mathrm{ideal}}$, we get

$$\begin{aligned}
\left(\frac{M_A}{M_{\mathrm{OPT}}}\right)^{1/\alpha} &\leq \frac{S - \kappa\Sigma_{\mathrm{ideal}}}{S - \Sigma_{\mathrm{ideal}}} \\
&\leq \frac{1 - \frac{\kappa p}{p+q}}{1 - \frac{p}{p+q}} \\
&\leq \frac{p + q - \kappa p}{q} \\
&\leq 1 + \frac{p}{q}(1 - \kappa)
\end{aligned}$$

Then, r is an upper bound of $\frac{p}{q}$, so

$$\frac{M_A}{M_{\mathrm{OPT}}} \leq (1 + r\,(1 - \kappa))^{\alpha}$$

Finally, by the definition of κ, we get

$$\frac{M_A}{M_{\mathrm{OPT}}} \leq \lambda$$

We have supposed so far that the left inequality of (1) holds. Otherwise, the second one holds. Note that both hypotheses only differ by an exchange of the roles of p and q. Then, as the problem is strictly symmetric in p and q, by an analogue reasoning, one can prove that $\mathcal{A}$ launched on X, $\frac{qS}{p+q}$, ε_κ returns a set B in

$$\Lambda' = \left\{ B \subset X \;\middle|\; (1-\varepsilon_\kappa) \sum_{i \in \bar{A_\text{O}}} x_i \le \sum_{i \in B} x_i \le \sum_{i \in \bar{A_\text{O}}} x_i \right\}$$

and that the schedule that associates B to the q-part of the processors has a makespan smaller than λM_OPT. Indeed, we needed to obtain this conclusion that $r \ge \frac{p}{q}$, and as we also have $r \ge \frac{q}{p}$, the same method works in this case.

To conclude, Algorithm 12 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 $\mathcal{S}_A$ and $\mathcal{S}_{\bar{B}}$. Therefore, regardless of which inequality of (1) holds, the returned schedule has a makespan smaller than λM_OPT, and so Algorithm 12 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. Then, the additional complexity of the algorithm is the one of launching two times the AS $\mathcal{A}$. The overall complexity is then $O\left(n + f_\mathcal{A}\left(n, \frac{\varepsilon_\lambda}{r}\right)\right)$.

□

Require: A graph G composed of n independent tasks T_i of length L_i, the parameters p and q of the processor platform $\mathcal{P}$, and the requested approximation ratio λ

Ensure: A schedule $\mathcal{S}$ of G on $\mathcal{P}$ that is a λ-approximation of the makespan

1: **function** HETEROGENEOUSAPP(G,p,q,λ)
2:     **if** $\lambda > (1+r)^\alpha$ **then**
3:         **return** the PM schedule on the largest part
4:     **end if**
5:     $A \leftarrow \mathcal{A}\left(X, \frac{pS}{p+q}, \frac{\varepsilon_\lambda}{r}\right) \quad ; \quad B \leftarrow \mathcal{A}\left(X, \frac{qS}{p+q}, \frac{\varepsilon_\lambda}{r}\right)$
6:     **return** the schedule with the minimum makespan between $\mathcal{S}_A$ and $\mathcal{S}_{\bar{B}}$
7: **end function**

Figure 12: Approximation scheme for the (p, q)-scheduling problem.

7 Simulations

In this section, we present the results of simulations which compare the optimal allocation presented in Section 5 (referred to as the PM strategy, for Prasanna-Musicus) to allocations that are unaware of the speed-up function p^α. Our objective is to show the potential gain in makespan obtained by taking this speed-up 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 [11], as already mentioned in Section 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.

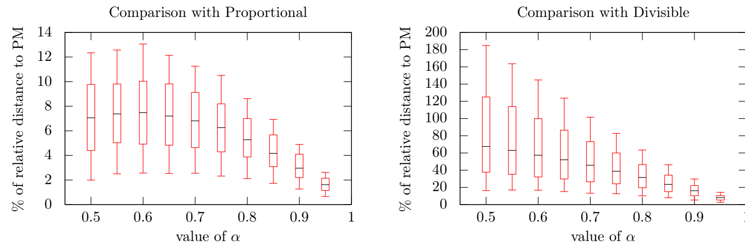


Figure 13: Comparison to the PM schedule with $p(t) = 40$.

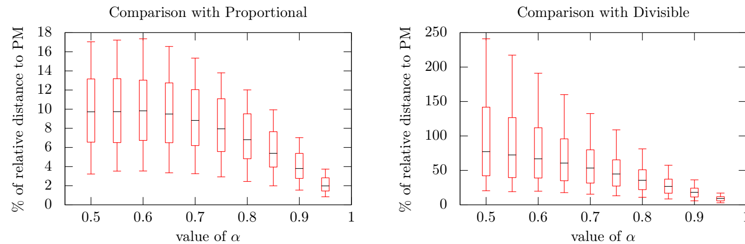


Figure 14: Comparison to the PM schedule with $p(t) = 100$.

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 (`http://www.cise.ufl.edu/research/sparse/matrices/`). The details concerning the computation of the data set can be found in [23]. 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. Figure 15 presents an example of this iterative aggregation, which is described in details in the following paragraphs. Note that this process transforms the tree into a SP-graph.

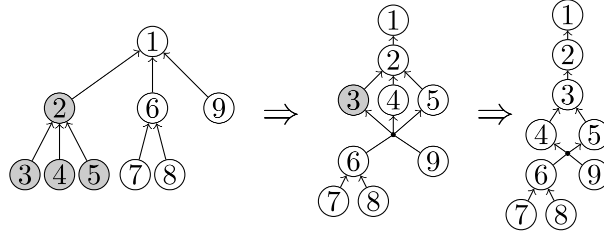


Figure 15: Example of the iterative aggregation of the left tree where the tasks that are allocated less than one processor in the PM schedule are shaded. The subtree rooted at task 2 is moved then modified.

More precisely, we convert each tree into a 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 below.

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 $\mathcal{L}$ (defined in Section 5) of each of these subgraphs. It 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 $(1 \parallel 2 \parallel 3 \parallel 4 \parallel 5)$ can return the graph $((3' \parallel 4' \parallel 5') \,; 2' \,; 1')$ where i' is the graph returned by the corresponding call of `Agreg` on i. A more complex example is illustrated on Figure 15.

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 6). Nevertheless, Proportional builds a different allocation and may use less than one processor for some tasks. Thus, we

evaluate its schedule 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 13 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 Proportional. We have also performed these tests with 100 processors, see Figure 14, which results on average in a 25% (resp. 10%) increase in the relative distance with Proportional and (resp. with Divisible).

8 Conclusion

In this paper, 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 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 [1, 2] 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% is interesting when comparing

real software implementations, which is also why the PM allocation has already been considered for sparse solvers [10], (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 [24], which makes smaller values of α more relevant.

The perspectives to extend this work follow two main directions. First, it would be interesting to extend the approximations proposed for the heterogeneous case to a number of nodes larger than two, and to more heterogeneous nodes, for which the value of α differs from one node to another. This is a promising model for the use of accelerators (such as GPU or Xeon Phi). The second direction concerns an actual implementation of the PM allocation scheme in a sparse solver. Our last results show that the large implementation effort necessary for such an experimentation is worth it, especially when considering multicore platforms with many cores and limited memory bandwidth.

References

[1] G. N. S. Prasanna and B. R. Musicus, "Generalized multiprocessor scheduling and applications to matrix computations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, no. 6, pp. 650–664, 1996.

[2] ——, "The optimal control approach to generalized multiprocessor scheduling." *Algorithmica*, vol. 15, no. 1, pp. 17–49, 1996.

[3] J. W. H. Liu, "The role of elimination trees in sparse factorization," *SIAM Journal on Matrix Analysis and Applications*, vol. 11, no. 1, pp. 134–172, 1990.

[4] M. Drozdowski, "Scheduling parallel tasks – algorithms and complexity," in *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, J. Leung, Ed., 2004.

[5] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[6] T. Gautier, X. Besseron, and L. Pigeon, "Kaapi: A thread scheduling runtime system for data flow computations on cluster of multiprocessors," in *International Workshop on Parallel Symbolic Computation*, 2007, pp. 15–23.

[7] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, "PaRSEC: Exploiting heterogeneity for enhancing scalability," *IEEE Computing in Science and Engineering*, to appear.

[8] R. Lepère, D. Trystram, and G. J. Woeginger, "Approximation algorithms for scheduling malleable tasks under precedence constraints," *Int. J. Found. Comput. Sci.*, vol. 13, no. 4, pp. 613–627, 2002.

[9] K. Jansen and H. Zhang, "Scheduling malleable tasks with precedence constraints," in *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2005, pp. 86–95.

[10] O. Beaumont and A. Guermouche, "Task scheduling for parallel multifrontal methods," in *Parallel Processing International Conference (Euro-Par)*, 2007, pp. 758–766.

[11] A. Pothen and C. Sun, "A mapping algorithm for parallel sparse cholesky factorization," *SIAM Journal on Scientific Computing*, vol. 14, no. 5, pp. 1253–1257, 1993.

[12] I. S. Duff and J. K. Reid, "The multifrontal solution of indefinite sparse symmetric linear systems," *ACM Transactions on Mathematical Software*, vol. 9, pp. 302–325, 1983.

[13] P. Amestoy, A. Buttari, I. S. Duff, A. Guermouche, J. L'Excellent, and B. Uçar, "Mumps," in *Encyclopedia of Parallel Computing*, D. A. Padua, Ed. Springer, 2011, pp. 1232–1238.

[14] A. Buttari, "Fine granularity sparse QR factorization for multicore based systems," in *International Conference on Applied Parallel and Scientific Computing*, 2012, pp. 226–236.

[15] X. S. Li, "An overview of SuperLU: Algorithms, implementation, and user interface," *ACM Transactions on Mathematical Software*, vol. 31, no. 3, pp. 302–325, September 2005.

[16] P. Hénon, P. Ramet, and J. Roman, "PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems," *Parallel Computing*, vol. 28, no. 2, pp. 301–321, Jan. 2002.

[17] R. Schreiber, "A new implementation of sparse gaussian elimination," *ACM Trans. Math. Softw.*, vol. 8, no. 3, pp. 256–276, Sep. 1982.

[18] 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," *Int. Journal of Supercomputer Applications*, vol. 1(4), pp. 10–30, 1987.

[19] A. Hugo, A. Guermouche, P.-A. Wacrenier, and R. Namyst., "A runtime approach to dynamic resource allocation for sparse direct solvers," in *International Conference on Parallel Processing (ICPP)*, 2014, p. To appear.

[20] G. Hardy, J. Littlewood, and G. Pólya, *Inequalities*, ser. Cambridge Mathematical Library. Cambridge University Press, 1952, ch. 6.14.

[21] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979.

[22] H. Kellerer, R. Mansini, U. Pferschy, and M. G. Speranza, "An efficient fully polynomial approximation scheme for the subset-sum problem," *Journal of Computer and System Sciences*, vol. 66, no. 2, pp. 349–370, 2003.

[23] L. Eyraud-Dubois, L. Marchal, O. Sinnen, and F. Vivien, "Parallel scheduling of task trees with limited memory," INRIA, Research Report RR-8606, 2014.

[24] S. L. Graham, M. Snir, C. A. Patterson *et al.*, *Getting up to speed: The future of supercomputing.* National Academies Press, 2005.

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée

655 avenue de l'Europe Montbonnot

38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399