Malleable task-graph scheduling with a practical speed-up model

Loris Marchal, Bertrand Simon, Oliver Sinnen, and Frédéric Vivien

Abstract—Scientific workloads are often described by Directed Acyclic task Graphs. Indeed, DAGs represent both a theoretical model and the structure employed by dynamic runtime schedulers to handle HPC applications. A natural problem is then to compute a makespan-minimizing schedule of a given graph. In this paper, we are motivated by task graphs arising from multifrontal factorizations of sparse matrices and therefore work under the following practical model. Tasks are malleable (i.e., a single task can be allotted a time-varying number of processors) and their speedup behaves perfectly up to a first threshold, then speedup increases linearly, but not perfectly, up to a second threshold where the speedup levels off and remains constant. After proving the NP-hardness of minimizing the makespan of DAGs under this model, we study several heuristics. We propose model-optimized variants for PROPSCHEDULING, widely used in linear algebra application scheduling, and FLOWFLEX. GREEDYFILLING is proposed, a novel heuristic designed for our speedup model, and we demonstrate that PROPSCHEDULING and GREEDYFILLING are 2-approximation algorithms. In the evaluation, employing synthetic data sets and task graphs arising from multifrontal factorization, the proposed optimized variants and GREEDYFILLING significantly outperform the traditional algorithms, whereby GREEDYFILLING demonstrates a particular strength for balanced graphs.

Index Terms—Scheduling, task graph, malleable tasks, speedup model, approximation algorithms, proportional mapping

1 INTRODUCTION

COMPLEX computations are often described as Directed Acyclic Graphs (DAGs), where nodes represent computational tasks and edges represent dependences between these tasks. This formalism is both very common in the theoretical scheduling literature [1] and sees an increasing interest in High Performance Computing: to cope with the complexity and heterogeneity in modern computer design, many HPC applications are now expressed as task graphs and rely on dynamic runtime schedulers such as StarPU [2], KAAPI [3], StarSS [4], and PaRSEC [5] for their execution. Even the OpenMP standard now includes DAG scheduling constructs [6].

Task graphs are helpful to express the structure of applications and to take advantage of the potential parallelism they express, sometimes called *inter-task parallelism*. However, tasks are often coarse grain and each task can be executed in parallel on several processing cores. To achieve good performance, we also want to take into account this *intra-task parallelism*. There have been a number of studies to mix both sources of parallelism when scheduling task graphs, such as [7], [8]. The main difficulty in this context is to come up with an expressive and yet tractable model for tasks. Task characteristics are summarized through a *speed-up* function that relates the execution time of a task to the number of processors it is allotted. In the present paper, we focus on a simple model where the speedup function is a continuous piecewise linear function, defined by two thresholds on the processor allotment: before a first threshold, the speedup is perfect, that is, equal to the number of processors; between the two thresholds, it is linear, but not perfect anymore; after the second threshold, it stalls and stays constant. We later show that this models well the performance of linear algebra kernels which are our present concern. This model extends the well-studied simple single-threshold model, with a perfect speedup before the threshold, and constant speedup thereafter. This simplified model has been studied both in theoretical scheduling [9] and for practical schedulers [10]. Contrarily to most existing studies, we also assume that tasks are *preemptible* (a task may be interrupted and resumed later), *malleable* (the number of processors allocated to a task can vary over time) and we allow *fractional allocation* of processors. We claim that this model is reasonable based on the following two arguments. Firstly, changing the allocation of processors is easily achieved using the time sharing facilities of operating system schedulers or hypervisors: actual runtime schedulers are able to dynamically change the allocation of a task [11]. Secondly, given preemption and malleability, it is possible to transform any schedule with fractional allocation to a schedule with integral allocation using McNaughton's wrap-around rule [12] (as shown in Section 3). Hence, we can consider fractional allocations that are simple to design and analyze, and then transform them into integral ones when needed.

The here presented study is motivated and driven by task graphs coming from sparse linear algebra, and especially from the factorization of sparse matrices using the multifrontal method. Liu [13] explains that the computational dependences and requirements in Cholesky and LU factorization of sparse matrices using the multifrontal method can be modeled as a task tree, called the *assembly tree*. Our targets are therefore such trees and the experimental evaluation will focus on them. Having that said,

- *L. Marchal, B. Simon and F. Vivien are with CNRS, INRIA and University of Lyon, LIP, ENS Lyon, 46 allée d'Italie, Lyon, France. E-mail: {loris.marchal,bertrand.simon,frederic.vivien}@ens-lyon.fr*
- *O. Sinnen is with Dpt. of Electrical and Computer Engineering, University of Auckland, New Zealand E-mail: o.sinnen@auckland.ac.nz*

the proposed algorithms in this paper are not limited to trees, but apply to series-parallel graphs (SP-graphs) or sometimes to general DAGs. We will describe and analyze them correspondingly for the sake of generality.

The contributions of this paper are as follows. We propose a practical piecewise linear speedup model which is divided into three parts. Up to a first threshold the speedup is perfect, equalling the number of processors. Then it grows linearly, but with a slope < 1 until a second threshold is reached, after which the speedup remains constant. Using such a piecewise linear function to describe the speedup seems a straightforward approximation. Our objective is to demonstrate that such a model is useful to create better schedules, in particular for proving approximation results and for designing novel, more efficient, scheduling heuristics. Section 3 details the model and Section 4 experimentally validates it, demonstrating its ability to more closely follow speedup curves typical for linear algebra kernels. For this model, we show the NP-completeness of the decision problem associated with the minimization of the makespan for a given graph (Section 5). We study previously proposed algorithms PROPMAPPING (Proportional Mapping) which is commonly used by runtime schedulers, and FLOWFLEX and propose model-optimised variants of these algorithms. Further, the novel GREEDYFILLING is proposed, designed for the new speedup model (Section 6). Both, GREEDYFILLING and PROPMAPPING are shown to be 2-approximation algorithms. In Section 7 we perform simulations both on synthetic series-parallel graphs and on real task trees from linear algebra applications demonstrating the general superiority of the new GREEDYFILLING and the model-optimised variants of the traditional algorithms.

2 RELATED WORK

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

2.1 Models of parallel tasks

The literature contains numerous models for "parallel tasks"; names and notations vary and their usage is not always consistent. The simplest model for parallel tasks is the model of *rigid* tasks, sometimes simply called *parallel tasks* [14]. A rigid task must always be executed on the same number of processors (that must be simultaneously available). In the model of *moldable* tasks, the scheduler has the freedom to chose on which number of processors to run a task, but this number cannot change during the execution. This model is sometimes called *multiprocessor tasks* [15]. The most general model is that of *malleable* tasks: the number of processors executing a task can change in any way at any time throughout the task execution. However, numerous articles use the name malleable to denote moldable tasks like, for instance, [14], [16], [17]. Depending on the variants, moldable and malleable tasks can run on any number of processors, from 1 to p, or each task T_i may have a maximum parallelism which is often denoted by δ_i [15]. Furthermore, depending on the assumptions, tasks may be preempted to be restarted later on the same set of processors, or on a potentially different one (preemption+migration). It should be noted that the model of malleable tasks is a generalization of the model of moldable tasks with preemption and migration.

An important feature of the models for moldable and malleable tasks is the task speed-up functions that relate a task execution time to the number of processors it uses. Some authors, like Hunold [18], do not make any assumption on the speed-up functions. More commonly, it is assumed that the task execution time is a non-increasing function of the number of processors [16], [18], [19], [20]. Another classical assumption is that the work is a non-decreasing function [16], [18], [20] —the work is the product of the execution time and of the number of processors used— which defines the model sometimes called *monotonous penalty assumptions*. Some other works consider that the speed-up function is a concave function [19]. Several of the models considered in the literature satisfy all above assumptions: non-decreasing concave speed-up function and non-decreasing work. This is for instance the case with the model studied by Prasanna and Musicus [21] where the processing time p_i of task i is $p_i(k) = \frac{p_i}{k^\alpha}$ with α being a task-independent constant between 0 and 1 and k the number of allotted processors [21], [22]. Another instance is the simple single-threshold model, that is, the linear model [9], [10], [23], [24], [25]: $p_i(k) = \frac{p_i}{k}$. Havill and Mao [26] added to that model an overhead affine in the number of processors used: $p_i(k) = \frac{p_i}{k} + (k-1)c$. This model is also closely related to the Amdahl's law where $p_i(k) = \frac{p_i^{(p)}}{k} + p_i^{(s)}$. Amdahl's law is considered in the experimental evaluation of [20].

Finally, the number of processors alloted to a task can, depending on the assumptions, either only take integer values, or can also take fractional ones [19], [21], [22].

2.2 Results for moldable tasks

Du and Leung [27] have shown that the problem of scheduling moldable tasks with preemption and arbitrary speed-up functions is NP-hard.

In the scope of the monotonous penalty model, Lepère, Trystram, and Woeginger [16] presented a $3 + \sqrt{5} \approx 5.23606$ approximation algorithm for general DAGs, and a $\frac{3+\sqrt{5}}{2} + \epsilon \approx 2.61803 + \epsilon$ approximation algorithm for series-parallel graphs and DAGs of bounded width.

Wang and Cheng presented [24] a $3 - \frac{2}{p}$-approximation algorithm to minimize the makespan while scheduling moldable task graphs with linear speed-up and maximum parallelism δ_j (problem $P|prec, any, spdp\text{-}lin, \delta_j|C_{\max}$).

Havill and Mao [26] consider the problem of scheduling independent moldable jobs in an online setting, with arbitrary arrival times (note that they use the term *malleable*). In the model where the processing time is described by $p_i(k) = \frac{p_i}{k} + (k-1)c$, they propose a simple yet efficient algorithm which is in particular a 4-approximation for large k. An improved algorithm has then been proposed by Kell and Havill [28] for a small number of processors.

2.3 Results for malleable tasks

The problem of scheduling independent malleable tasks with linear speedups, maximum parallelism per task, and

with integer allotments, that is $P|var, spdp\text{-}lin, \delta_j|C_{\max}$, can be solved in polynomial time [9], [29] using a generalization of McNaughton's wrap-around rule [12]. Drozdowski and Kubiak showed in [9] that this problem becomes NP-hard when dependences are introduced: $P|prec, var, spdp\text{-}lin, \delta_j|C_{\max}$ is NP-hard. Balmin et al. [10] present a 2-approximation algorithm for this problem. Their algorithm builds integral allotments by first scheduling the DAG on an infinite number of processors and then using the optimal algorithm for independent tasks to build an integral-allotment schedule for each interval of the previous schedule during which a constant number of processors greater than p was used. In this paper we extend this result with Corollary 2, showing that this algorithm is also a $2 - \frac{\delta_{\min}}{p}$ approximation for makespan minimization with fractional allotments (where $\delta_{\min}$ is the minimum threshold over all tasks).

Makarychev and Panigrahi [19] consider the problem $P|prec, var|C_{\max}$ under the monotonous penalty assumption and when allotments are rational. They provide a $(2+\epsilon)$-approximation algorithm, of unspecified complexity (their algorithm relies on the resolution of a rational linear program; this linear program is not explicitly given). Furthermore, they prove that there is no "online algorithm with sub-polynomial competitive ratio" (an online algorithm is an algorithm that considers tasks in the order of their arrival).

Carroll and Grosu [30] study the problem of scheduling independent malleable tasks in an online setting, with arbitrary arrival times and deadlines. They use the same processing time as Havill and Mao [26], and each task comes with a value. They propose an incentive strategy to maximize the sum of the values of the tasks completed before their deadline, hence rational users truthfully declares his parameters.

2.4 Series-parallel graphs

Series-parallel graphs can be recognized and decomposed into a tree of series and parallel combinations in linear time [31]. It is well-known that series-parallel graphs capture the structure of many real-world scientific workflows [32]. A possible way to extend algorithms designed for series-parallel graphs to general graphs is to first transform a graph into a series-parallel graph, using a process sometimes called SPization [33] before applying a specialized algorithm for SP-graphs. This was for example done in [34]. However, note that no SPization algorithm guarantees that the length of the critical path is increased by only a constant ratio.

3 Application Model

We consider a workflow of tasks whose precedence constraints are represented by a task graph $G = (V, E)$ of n nodes, or tasks: a task can only be executed after the termination of all its predecessors. We assume that G is a *series-parallel* graph. Such graphs are built recursively as series or parallel composition of two or more smaller SP-graphs, and the base case is a single task. Trees can be seen as a special-case of series-parallel graphs. A tree can be turned into an SP-graph by simply adding one dummy task without computation cost, that has an edge with every leaf of the tree.

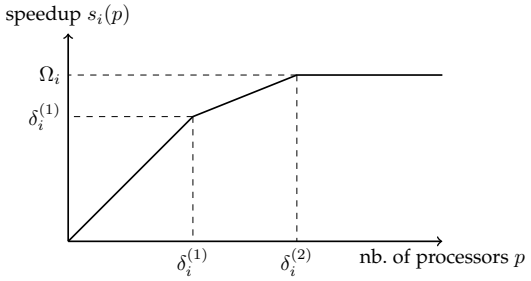


Figure 1: Illustration of the proposed speedup model and its notations.

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

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

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

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

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

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

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

Model variants. The objective of the proposed three-phase model with two thresholds is to accurately match the memory hierarchy: the closer the processors, the faster they can communicate. In the following, we also use a simpler variant of our model with only two phase: when both thresholds are equal ($\delta_i^{(1)} = \delta_i^{(2)}$) we necessarily get $\Omega_i = \delta_i^{(1)}$ and we get back to a capped perfect threshold. The problem of minimizing the makespan of a graph with

this model is noted $P|prec, var, frac, spdp\text{-}lin, \delta_j|C_{\max}$ and is studied in [9], [10].

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

Other notations. In the following, we will often use the length of the *critical path* of a task T_i, which is defined as the minimum time needed to complete all the tasks on any path from this task to any output task of the graph, provided that an unlimited number of processors is available. This corresponds to the classical notion of bottom-level [35], when the duration of each task is set to w_i/Ω_i. By extension, the critical path of the entire graph G is the longest critical path of all its tasks.

3.1 Extension of McNaughton wrap-around rule

When scheduling tasks with perfect speedup, it is possible to remove the assumption of fractional allocation without degrading the makespan thanks to malleability, using the so-called "Macnaughton wrap-around rule" [12]. We adapt here this result to our model with two integer thresholds. For the sake of simplicity, the proof is presented only for two tasks but easily extends to any allocation.

Lemma 1. *Consider two tasks A and B sharing an integer number of processors p on a given interval M: A (resp. B) is allocated a fractional number of processors p_A (resp. p_B). We can produce a schedule with preemption where A and B are allocated integer numbers of processors at all times, and in which both tasks perform the same amount of work.*

Proof. We build a schedule where $\lfloor p_A \rfloor$ (resp. $\lceil p_B \rceil$) processors are allocated to task A (resp. B) during t units of time, and $\lceil p_A \rceil$ (resp. $\lfloor p_B \rfloor$) during $M-t$ units of time. t is chosen such that the area dedicated to task A is the same as in the original allocation, which means:

$$M \lceil p_A \rceil - t = M p_A$$

so $t = M(\lceil p_A \rceil - p_A)$. The same holds correspondingly for task B. This ensures that we can apply this transformation to both tasks without exceeding the processor limit.

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

$$\begin{aligned} & t \times s(\lfloor p_A \rfloor) + (M-t) \times s(\lceil p_A \rceil) \\ & = M \times s(\lceil p_A \rceil) - t(s(\lceil p_A \rceil) - s(\lfloor p_A \rfloor)) \\ & = M \left(s(\lceil p_A \rceil) - \frac{\lceil p_A \rceil - p_A}{\lceil p_A \rceil - \lfloor p_A \rfloor} (s(\lceil p_A \rceil) - s(\lfloor p_A \rfloor)) \right) \end{aligned}$$

We know that

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

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

$$t \times s(\lfloor p_A \rfloor) + (M-t) \times s(\lceil p_A \rceil) = M \times s(p_A)$$

which completes the proof. □

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

4 Experimental validation of the model

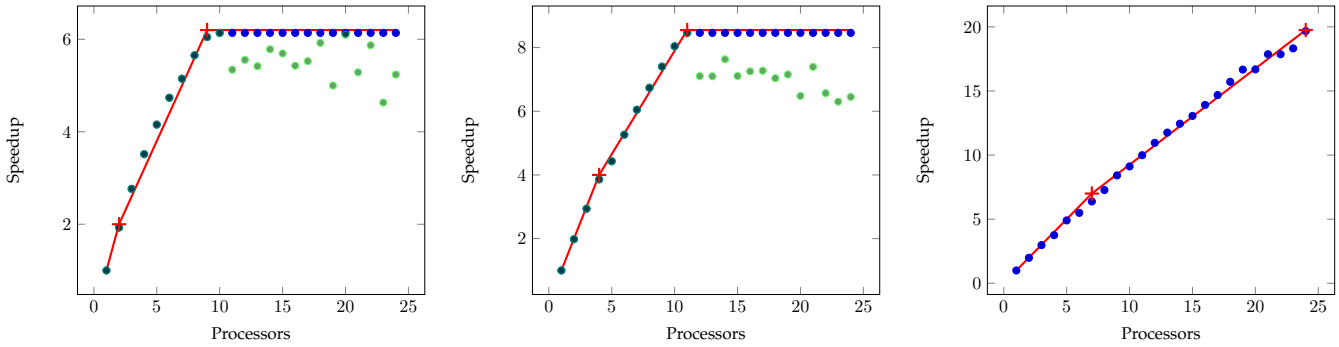
In this section, we show that the proposed model is realistic enough to model parallel tasks coming from an actual application. In order to do this, we ran such tasks on a parallel platform of 24 cores. It consists of two Haswell Intel Xeon E5-2680 processors, each one containing 12 cores running at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

To compute the speedup graph as shown in Figure 1, we ran each task with $p = 1, \ldots, 24$ cores. Note that on this platform, the "number of processors" has to be understood as "number of cores". We first tested a dense numerical algebra routine: the dense Cholesky factorization. We noticed that the speedup of such dense tasks was perfect, i.e., equal to the number of cores used, up to using the full platform ($p = 24$). However, usual parallel applications are not made (only) of tasks with perfect parallelism, and our model precisely aims at taking these speedup limitations into account. Thus, we focused on another linear algebra application, which is the multifrontal QR decomposition of sparse matrices as performed by QR_MUMPS [36]. Each task of this application is the QR decomposition of a dense rectangular matrix. For the set of matrices described in Section 7, we computed the size of all the dense QR decompositions associated to its multifrontal QR decomposition. For each of the ten thousand resulting sizes, we timed such a task on $p = 1, \ldots, 24$ cores. Each timing was performed 5 times using random data and we retain the average performance.

In the following, we present the speedup of three tasks which are representative of all possible sizes. The first case, presented in Figure 2a, corresponds to small matrix sizes. The green dots represent the actual speedup measured on the platform. Note that for up to 10 processors, the speedup increases with the number of processors and then the performance decreases and exhibits a larger variability for larger number of processors: when adding too many processors, more time is spent in communicating and synchronizing the processors, which hinders the performance. This behaviour is not unusual; a "smart" implementation of the task would be aware of this and would limit the number of processors to be used to 10, even if more processors are allocated to the task. Thus, we first transform the measured speedup so that it is never decreasing with the number of processors. Formally, this *corrected* speedup, plotted with blue dots on the figures, is given by:

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

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



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

Figure 2: Speedup and fitted model for different matrix sizes

the sum of residuals. The resulting speedup model is plotted in red.

Figures 2b and 2c plots the same measured speedup, corrected speedup and fitted model for larger matrices: a medium-size matrix on Figure 2b with one large dimension and one small, and a large matrix on Figure 2c. As expected, we notice that the thresholds increase with the matrix size. For the larger matrices, the second threshold is set to 24 although it would probably be larger on a larger platform. Overall, we notice that the fitting of the model is very accurate, the median coefficient of determination being larger than 0.98 (a value of 1 means a perfect fit).

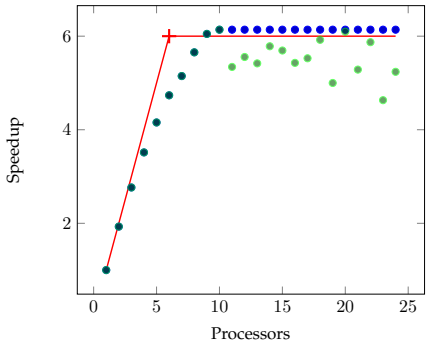


Figure 3: Speedup and single-threshold model for matrix size 1984x1834.

Some of the available algorithms for solving our problem consider the single-threshold variant presented in the previous section. Thus, we also fitted the previous (corrected) speedup measurements with this model, composed of a first perfect linear speedup and then a plateau where the speedup is equal to the threshold. Figure 3 plots the obtained speedup model using the same matrix size as Figure 2a. We see that this model is less accurate, the median coefficient of determination being 0.90: it is too optimistic before the threshold, as the task is not perfectly parallel, and it is too pessimistic after the threshold, as better performance can be reached with a number of cores larger than the threshold.

5 Problem Complexity

Task malleability and perfect speed-up make this problem much easier than most scheduling problems. However, quite surprisingly, limiting the possible parallelism with thresholds is sufficient to make it NP-hard. We restrict ourselves here to the model where both thresholds are equal ($\delta_i = \delta_i^{(1)} = \delta_i^{(2)} = \Omega_i$ for all tasks i) as it is already NP-hard. Note that a similar result already appeared in [9], however its proof is more complex and not totally specified, which makes it difficult to check; this is why we propose this new proof.

Theorem 1. *The decision version of the problem of minimizing the makespan is NP-complete.*

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

To prove completeness, we perform a reduction from the 3SAT problem which is known to be NP-complete [37]. An instance $\mathcal{I}$ of this problem consists of a boolean formula, namely a conjunction of m disjunctive clauses, $C_1, \ldots, C_m$, of 3 literals each. A literal may either be one of the n variables $x_1 \ldots x_n$ or the negation of a variable. We are looking for an assignment of the variables which leads to a TRUE evaluation of the formula.

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

start both at time $t_i + 1$ and (ii) at least one chain L_{x_i} or $L_{\overline{x_i}}$ corresponding to one of the three literals of any given clause starts at time $t_i + 1$.

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

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

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

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

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

where:

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

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

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

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

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

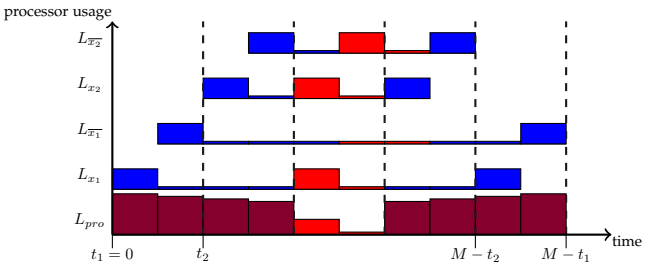


Figure 4: Example of a possible schedule for the instance $\mathcal{J}$ associated to the formula $x_1 \vee x_2 \vee \bar{x}_2$.

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

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

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

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

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

From a valid schedule to a truth assignment: We now assume that instance $\mathcal{J}$ has a valid schedule S and we aim at reconstructing a solution for $\mathcal{I}$. First we prove some properties on the starting times of chains through the following lemma. The proof of this lemma has been moved to the web supplementary material.

Lemma 2. *In any valid schedule S for $\mathcal{J}$,*

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

For each literal chain which starts at time $t_i + 1$, we associate the value TRUE in an assignment of the variables of $\mathcal{I}$, and we associate the value FALSE to all other literals. Thanks to the previous lemma, we know that exactly one literal in the pair $(x_i, \overline{x_i})$ is assigned to TRUE. Furthermore, not three tasks of size $1 - \frac{2}{3}n\varepsilon$ can be scheduled at time

Algorithm 1: PROPMAPPING ($G = (V, E, w)$, p)

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

$2n+2(k-1)$ because of the profile chain L_{pro}, as this would lead to a number of occupied processors of:

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

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

6 HEURISTICS DESCRIPTION AND COMPETITIVE ANALYSIS

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

6.1 Performance analysis of proportional mapping

A widely used algorithm for this problem is "proportional mapping" [38]. In this algorithm, a sub-graph is allocated a number of processors that is proportional to the ratio of its weight to the sum of the weights of all sub-graphs under consideration. Based on the structure of the considered SP-graph G, Algorithm 1 allocates a share of processors to each sub-graph and eventually each task. Any given graph G (with SP-graph characteristics) can be decomposed into its series and parallel components using an algorithm from [31], and thus be processed by Algorithm 1. Observe that thresholds are not considered in this proportional mapping.

The schedule corresponding to this proportional mapping simply starts each task as soon as possible (i.e., after all its predecessors have completed), as given in Algorithm 2. Indeed, given the proportional mapping of processors, there are always enough processors available to do that. It is worth noting that the created schedule is compatible with the moldable model: each task uses the same number of processors throughout its entire execution. As such, Algorithm 2 can also be used for the moldable model.

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

Algorithm 2: PROPSCHEDULING ($G = (V, E, w, p)$

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

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

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

The critical path cp of G, as defined in Section 3, is a longest path in G, where the length is defined as the sum of the work of each task on the path divided by its maximum speedup, $len(cp) = \sum_{i \in cp} \frac{w_i}{\Omega_i}$. Naturally, the critical path length is another lower bound on the optimal makespan, $len(cp) \leq M_{\text{OPT}}$.

Consider the schedule produced by PROPSCHEDULING. There is at least one path Φ in G from the entry task to the exit task, with no idle time between consecutive tasks. In other words, on Φ the execution of a task starts when the execution of the preceding task finishes. Such a path always exists because we start tasks as early as possible, so this property is always true between the tasks of a serial composition, and it is true for at least one task in each parallel composition. The execution length of Φ is the makespan M, because it includes no idle time and that it goes from entry to exit task. It is given by

$$M = \sum_{i \in \Phi} \frac{w_i}{s_i(\min\{p_i, \delta_i^{(2)}\})}$$

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

$$M = \sum_{i \in A} \frac{w_i}{s_i(\delta_i^{(2)})} + \sum_{i \in B} \frac{w_i}{s_i(p_i)} = \sum_{i \in A} \frac{w_i}{\Omega_i} + \sum_{i \in B} \frac{w_i}{s_i(p_i)}$$

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

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

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

$$rM_\infty \geq M_B$$

We then get the desired inequality:

$$M \leq len(cp) + rM_\infty \leq (1+r)M_{\text{OPT}}. \quad \square$$

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

6.2 Optimizations of proportional mapping

The main drawback of PROPMAPPING is that it assumes perfect speedup. When applied to actual tasks with imperfect speedup functions, some tasks may finish later than expected by the algorithm. In some cases, sibling tasks (tasks that share the same successor) may complete earlier, thus leaving some processors idle, which induces performance loss. In order to address this issue, a natural idea is to redistribute the processors left idle by the termination of some task T_i to T_i's siblings, that is, to the tasks that share the same successor T_j and are still running. This is for example what is done in [39]. We design such an algorithm, called PROPMAPREBALSIBLINGS, which redistributes the processing power of terminated tasks to their siblings, proportionally to the weight of the target tasks.

Note that both the original PROPMAPPING or this optimization are agnostic of both thresholds. Thus, we introduce a new variant of PROPMAPPING called PROPMAPREBALTHRESHOLD that takes advantage of the speedup model introduced above. It also consists of redistributing processors left idle when a task terminates while its successor is not ready yet. The main difference with the previous variant is that idle processors are not redistributed only to siblings, but to all currently running tasks for which $p_i < \delta_i^{(2)}$. Again, the redistribution is done according to the weight of the tasks. Both variants are detailed in Algorithm 3 and Algorithm 4, and have a complexity of $O(|V|^2)$.

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

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

6.3 A novel algorithm: Greedy-Filling

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

Algorithm 4: PROPMAPREBALTHRESHOLD ($G = (V, E, w, \delta^{(2)})$, p,)

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

Algorithm 5: GREEDYFILLING ($G = (V, E, w, \delta^{(1)}, \delta^{(2)})$, p)

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

We now detail this algorithm, presented in Algorithm 5. First, each task is given a priority. In practice, we use for each task T_i its critical path (i.e. bottom-level, see Section 3), as it is a lower bound on the overall completion time once task T_i has started. The algorithm builds the schedule in chronological order while maintaining the set of free tasks. The difference is that, instead of sharing the resources according to the weight of tasks, we consider them in the order defined by priorities. We allocate each task T_i up to $\delta_i^{(1)}$ processors if possible, so as to stay in the perfect parallelism zone. If there are processors in excess, we reconsider the tasks in the same order, increasing their allocation up to $\delta_i^{(2)}$.

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

Theorem 3. GREEDYFILLING *is a* $1 + r - \frac{\delta_{\min}^{(2)}}{p}$ *approximation*

for makespan minimization, with $\delta^{(2)}_{\min} = \min_{T_i \in V} \delta^{(2)}_i$ *and* $r = \max_i \frac{\delta^{(2)}_i}{\Omega_i}$.

Proof. This proof is a transposition of the classical proof by Graham [40]. In any schedule produced by GREEDYFILLING, let T_1 be a task whose completion time is equal to the completion time of the whole task graph. We consider the last time t_1 prior to the start of the execution of T_1 at which not all processors were fully used. If the execution of T_1 did not start at time t_1 this is only because at least one ancestor T_2 of T_1 was executed at time t_1. Then, by induction we build a dependence path $\Phi = T_k \rightarrow \ldots \rightarrow T_2 \rightarrow T_1$ such that all processors are fully used during the execution of the entire schedule except, maybe, during the execution of the tasks of Φ.

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

$$\begin{aligned} Idle &\leq \sum_{i=1}^{k} \left(\frac{w_i}{\Omega_i} \times (p - \delta^{(2)}_i) \right) \leq (p - \delta^{(2)}_{\min}) \times \sum_{i=1}^{k} \frac{w_i}{\Omega_i} \\ &\leq (p - \delta^{(2)}_{\min}) len(cp) \leq (p - \delta^{(2)}_{\min}) M_{\mathrm{OPT}}. \end{aligned}$$

Let *Used* denote the sum of the busy areas in the schedule. As s_i is concave (cf. Equation (1), the busy area dedicated to schedule the task T_i is maximized when T_i is allocated to $\delta^{(2)}_i$ processors. Then, the area is equal to $\frac{\delta^{(2)}_i w_i}{\Omega_i}$ and

$$Used \leq \sum_i \frac{\delta^{(2)}_i w_i}{\Omega_i}.$$

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

$$Used \leq \sum_i w_i r \leq r p M_{\mathrm{OPT}}.$$

Let M be the makespan of the considered schedule. Then we have:

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

□

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

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

The proof of Theorem 3 can easily be adapted to the single threshold model, which gives the following result. This is particularly useful to prove that the FLOWFLEX algorithm, presented below, is an approximation algorithm.

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

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

6.4 The FlowFlex algorithm

We now introduce FLOWFLEX, a scheduling algorithm introduced in [10] and designed for a model similar to the single threshold variant described in Section 3, which considers that $\delta_i = \delta^{(1)}_i = \delta^{(2)}_i = \Omega_i$ for all tasks i. FLOWFLEX first allocates to each task its maximal number of processors δ_i, as if there was an infinite number of processors available. Then, in each time interval where the allocation is constant, if the total number of allocated processors exceeds p, the allocation is scaled down proportionally. This algorithm is detailed in Algorithm 6.

In its original version, FLOWFLEX assumes a perfect speedup before the threshold. Thus, scaling down the shares of the tasks proportionally preserves the simultaneous completion of the amount of work performed in the original interval. This is no longer true with imperfect speedup functions. This is why we introduce an optimized version FLOWFLEXREBALANCE that redistributes idle processors among running tasks once a task completes the amount of work it had to process in a given interval. The redistribution is done proportionally to the thresholds δ_i, which corresponds to the original allocation before scaling. This optimized variant is described in Algorithm 7.

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

7 EXPERIMENTAL COMPARISON

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

Algorithm 6: FLOWFLEX ($G = (V, E, w, \delta)$, p)

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

Algorithm 7: FLOWFLEXREBALANCE ($G_{DAG} = (V, E, w, \delta)$, p)

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

7.1 Datasets

First, we consider a set of 30 synthetic random SP-graphs composed each of 200 nodes. In order to compute a random SP-graph of $x > 1$ nodes, we follow the following recursive strategy: toss k uniformly in $[1, x-1]$; with a probability of $1/2$, build a series composition of two random SP-graphs of respectively k and $x - k$ nodes and, otherwise, build a parallel composition of these graphs. Then, in order to generate a random task (i.e., a random graph of $x = 1$ node), we choose its weight w uniformly in $[1; 1000]$. The first threshold, $\delta^{(1)}$, is defined by $\delta^{(1)} = \lceil w/100 \rceil$; hence, $\delta^{(1)} \in [1; 10]$. The second threshold, $\delta^{(2)}$, is uniformly drawn in $[\delta^{(1)}, 2\delta^{(1)}]$. The slope between the thresholds is uniformly drawn in $[0.5, 1]$. Therefore, in this dataset (called SYNTH), each task perfectly follows our speedup model.

Second, we consider a set of 24 trees whose size vary from 39 to 5900 nodes. These elimination trees have been generated (with either `colamd` [41] or `scotch` [42] ordering) using QR_MUMPS [36] on matrices from the University of Florida Sparse Matrix Collection [43], such that each task of a tree corresponds to the dense QR factorization of the associated matrix. The completion time of a task solely depends on the dimensions of the matrix. In order to determine the actual behavior of such a task, we benchmarked the time necessary to perform this task for a number of processors ranging from 1 to 24, as detailed in Section 4. Thus, in this dataset (called TREES), a task is characterized both by its parameters in our model ($\delta^{(1)}$, $\delta^{(2)}$ and Ω) and by the set of its completion times recorded through actual executions for up to 24 processors. The actual execution times are used in the experiments to determine the finish times of the scheduled trees (makespans).

7.2 Results

In order to compare the performance of these algorithms, we use a generic tool called *performance profile* [44]. For a given dataset, we compute the performance of each heuristic on each graph and for each considered value for the total number of available processors (namely 1, 2, 4, 6, 8, 10, 12, 16, 20 and 24). Then, instead of computing an average above all the cases, a performance profile reports a cumulative distribution function. Given a heuristic and a threshold τ expressed in percentage, we compute the fraction of test cases in which the performance of this heuristic is at most $\tau\%$ larger than the best observed performance, and plot these results. Therefore, the higher the curve, the better the method: for instance, for an overhead $\tau = 5\%$, the performance profile shows how often a given method lies within 5% of the smallest makespan obtained.

In Figure 5, we present the performance profiles for the SYNTH dataset on the left, and the makespan obtained by each heuristic on a sample graph on the right. On this latter plot, the y-axis has been normalized by the classical lower bound on makespan: the maximum of the critical path and of the total work divided by the number of processors.

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

The right-hand side of Figure 5 presents the typical results for a sample graph. The respective performance of heuristics is roughly independent of the number of available processors, and GREEDYFILLING presents the best results.

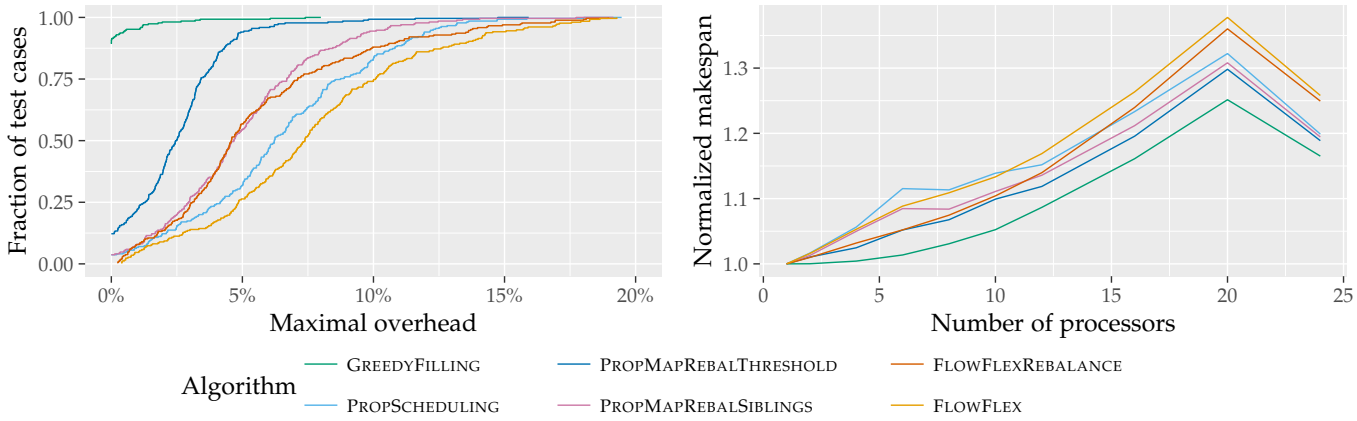


Figure 5: Performance profiles for up to 24 processors on Synth (left, where the best performance is top-left) and performance of the heuristics on a sample graph (right, where the best performance is bottom). Note that the following figures use the same legend.

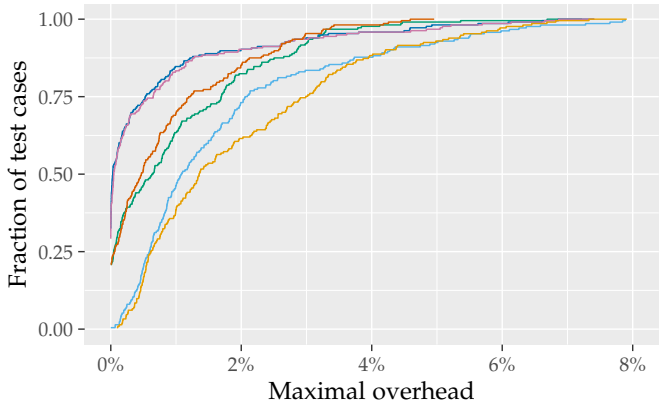


Figure 6: Performance profiles for up to 24 processors on Trees. Note the scale difference to Figure 5.

Overall, the shape of the curves were predictable: when there are very few available processors, there is little possibility of wasting computational resources and all heuristics achieve near-perfect performance; when the number of processors is very large all heuristics that are aware of the second threshold provide similar processor allocation and achieve similar near-perfect performance. The hardest part is in the intermediate zone when the most significant differences can be observed.

We present the performance profile for the Trees dataset in Figure 6, with additional representative samples in Figures 7 and 8. The legend of these graphs is the same as the one of Figure 5. The first observation to be made is that the difference between the graphs has significantly decreased. For each of the four heuristics GreedyFilling, PropMapRebalThreshold, PropMapRebalSiblings, and FlowFlexRebalance, in 80% of the cases the overhead is at most 2% and in 63% of the cases it is at most 1%.

An explanation for this is that the trees of this dataset often contain a task (near the root one) whose completion time is far beyond the rest of the graph, as illustrated on the right in Figures 7 and 8.

Within the small difference between the algorihtm, the results are similar to the previous data set (ignoring GreedyFilling for the moment): FlowFlex and PropMapping are the two worst heuristics; both heuristics are clearly outperformed by their variants; PropMapRebalThreshold achieves the best performance among these variants, but this time the performance of PropMapRebalSiblings is almost indistinguishable from that of PropMapRebalThreshold. GreedyFilling also performs better than the previously proposed algorithms FlowFlex and PropMapping, but its relative performance compared with the proposed variants has changed: its performance on actual trees (Figure 6) is now slightly behind these variants, when it was clearly the best solution on synthetic ones (Figure 5). The better performance of PropMapRebalSiblings compared to GreedyFilling may be surprising because PropMapRebalSiblings does not have any knowledge on the computed (estimated) thresholds. This performance is actually due to the structure of the graphs, as detailed below.

One should recall that the performance profiles gather results over the whole dataset. Varying performance of an algorithm can depend upon the structure of the tree and the processing power available. GreedyFilling achieves very good results when the structure of the graph is well-balanced, which is generally the case in the Synth dataset (Figure 5) as the graphs are generated recursively, as well as in the actual tree of Figure 7. This remark comes from the fact that GreedyFilling tries to maximize the efficiency of the allocation from the beginning of the schedule: if possible, it limits the allocation to every task to its first threshold, so that the overall speedup remains perfect. This explains why GreedyFilling is the best heuristic for medium numbers of processors in Figure 7: the tree is well-balanced, and for this range of processors, maintaining a perfect speedup is more efficient than balancing the allocation in the way PropMapRebalSiblings does. However, GreedyFilling performance degrades relatively when some branches in the tree are far from being critical and should have their execution delayed, even if this means exceeding the first threshold on other tasks and having a non-perfect speedup.

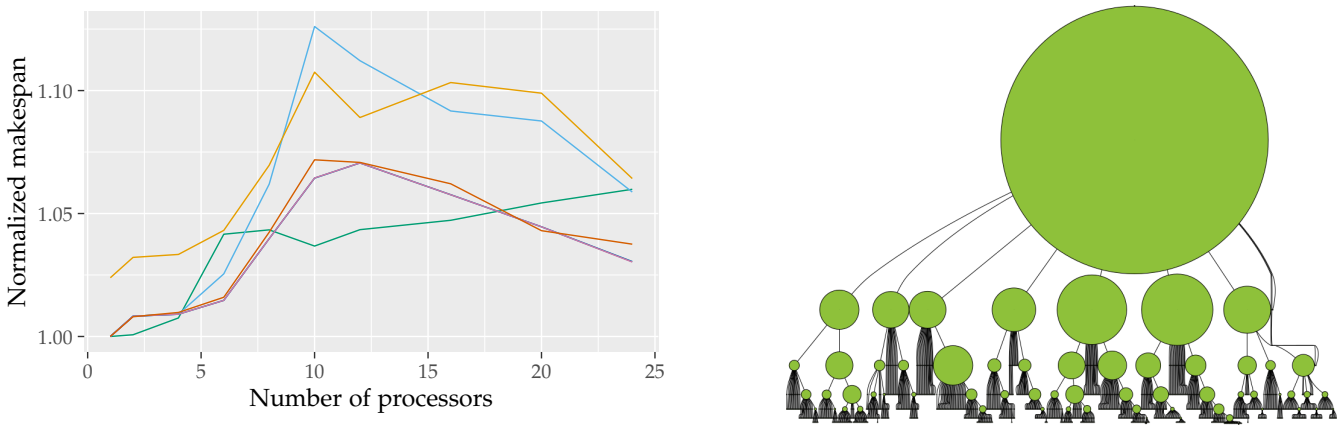


Figure 7: Performance of the heuristics and visual representation of the tree lp-nug30-colamd, where the area of a node is proportional to its sequential execution time.

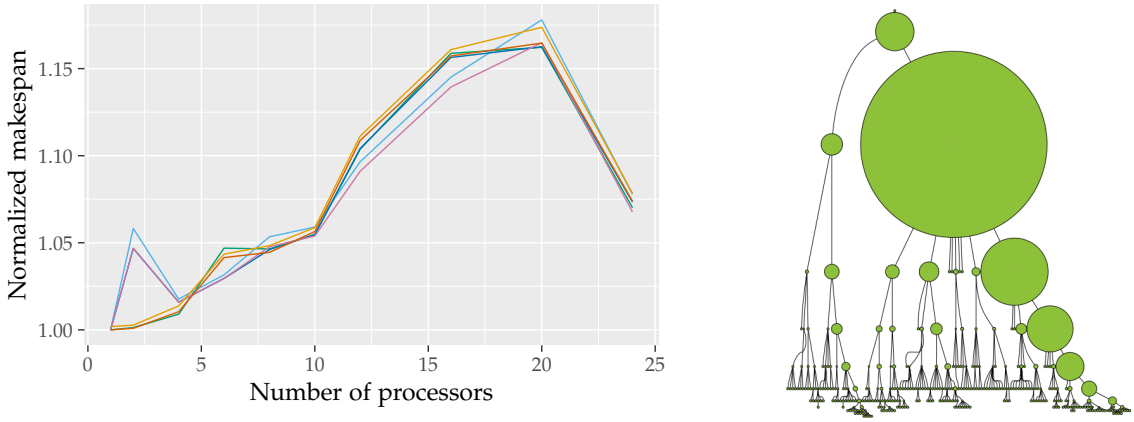


Figure 8: Performance of the heuristics and visual representation of the tree GL7-d24-scotch, where the area of a node is proportional to its sequential execution time.

Therefore it only achieves average performance on the TREES dataset (Figure 6), where other heuristics frequently have slightly better performance. For instance, the tree of Figure 8 has a highly critical branch on the right side, and GREEDYFILLING does not allocate enough processors to this branch at the beginning of the schedule, which leads to performance worse than that of the simple PROPSCHEDULING for average numbers of processors. With few processors, GREEDYFILLING fully prioritizes the critical branch as the first thresholds are not reached yet, and therefore achieves very good performance. In such a tree and with sufficient processing power, PROPMAPREBALSIBLINGS and PROPMAPREBALTHRESHOLD are the better choice as they progress quicker on the critical branches.

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

In order to measure the benefits of the double-threshold model, we adapted GREEDYFILLING, which first allocates each free task to their threshold, before distributing the processors in excess among the free tasks, to the single threshold model. In order to simulate this variant on both datasets, we have computed for each task the value of the single δ_j threshold that fits the best the measured speedup, see Section 4. We then compare in Figure 9 the performance of GREEDYFILLING in the single-threshold model and the double-threshold model. The single-threshold model leads, as expected, to a lower performance. On the SYNTH dataset, the double-threshold model obtains better results than the single threshold model for more than 80% of the test cases. This proportion is equal to around 70% for the TREES dataset. The improvement is more important on the SYNTH dataset, because, as discussed previously, the structure of the trees in the dataset TREES impacts the performance of GREEDYFILLING. On both settings, GREEDYFILLING significantly benefits from the better accuracy of the double-threshold model. Note that in the case of the SYNTH dataset,

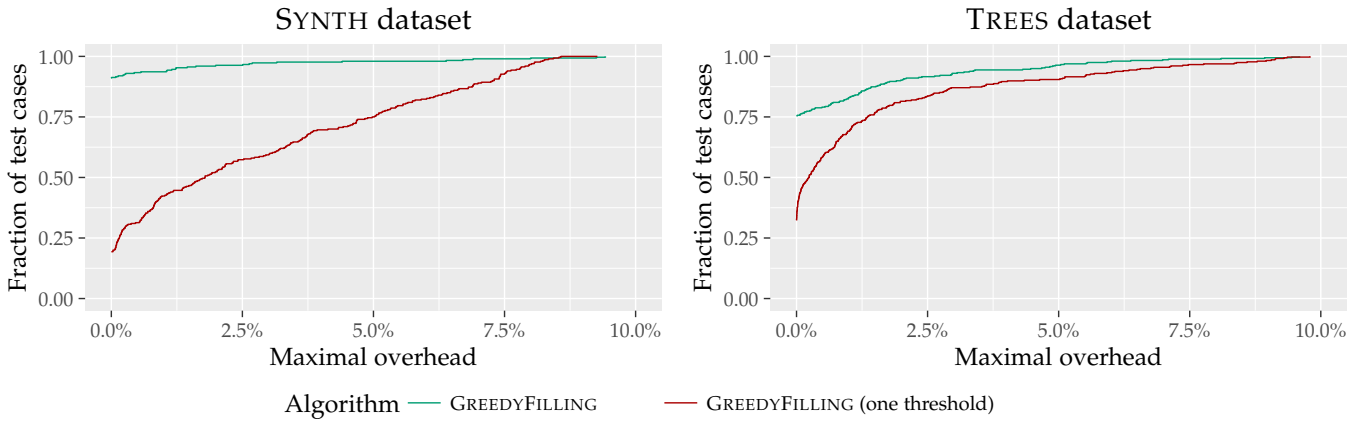


Figure 9: Performance profiles comparing the performance of GREEDYFILLING with the double-threshold model and the single-threshold model, on both datasets.

the improvement of GREEDYFILLING over the other heuristics (as illustrated of Figure 5) is mainly due to the use of the double-threshold model: with the single-threshold model, GREEDYFILLING would perform much worse, comparabily to the other heuristics.

8 CONCLUSION

In this paper, we have proposed a simple, but practical speedup model for graphs of malleable tasks, which is an interesting trade-off between tractability and accuracy. We have first provided an NP-hardness proof of the makespan minimization problem under this model. This was followed by a study of heuristic solutions, where we proposed model-optimized variants of the existing algorithms PROPMAPPING and FLOWFLEX. Designed for the new speedup model, we also proposed the novel GREEDYFILLING algorithm and demonstrated that GREEDYFILLING and PROPMAPPING are 2-approximation algorithms. To evaluate the algorithms, we performed simulations both on synthetic series-parallel graphs and on real task trees from linear algebra applications. They demonstrated the general superiority of the new GREEDYFILLING and the model-optimised variants of the traditional algorithms. In general, employing the new speedup model helps to improve the scheduling results. In a future work, a similar study could be conducted for computing systems that do not allow to reconfigure task mapping, i.e., for moldable tasks.

9 ACKNOWLEDGMENTS

The authors would like to thank Abdou Guermouche for providing the QR-MUMPS datasets as well as technical support for the experimental work. This work was partially supported by the SOLHAR project and by the LABEX MILYON (ANR-10-LABX-0070), both being operated by the French National Research Agency (ANR). Experiments presented in this paper were carried out using the PLAFRIM experimental testbed (https://www.plafrim.fr/), being developed under the Inria PlaFRIM development action with support from Bordeaux INP, LABRI and IMB and other entities: Conseil Régional d'Aquitaine, Université de Bordeaux and CNRS (and ANR in accordance to the programme d'investissements d'Avenir).

REFERENCES

[1] M. Drozdowski, "Scheduling parallel tasks – algorithms and complexity," in *Handbook of Scheduling*, J. Leung, Ed. Chapman and Hall/CRC, 2004.

[2] 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.

[3] T. Gautier, X. Besseron, and L. Pigeon, "KAAPI: A thread scheduling runtime system for data flow computations on cluster of multiprocessors," in *PASCO'07*, 2007, pp. 15–23.

[4] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta, "Hierarchical task-based programming with StarSs," *IJHPCA*, vol. 23, no. 3, pp. 284–299, 2009.

[5] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, "PaRSEC: Exploiting heterogeneity for enhancing scalability," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 36–45, 2013.

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

[7] A. Radulescu, C. Nicolescu, A. J. C. van Gemund, and P. Jonker, "CPR: mixed task and data parallel scheduling for distributed systems," in *IPDPS'01*, 2001, p. 39.

[8] P. Dutot, T. N'Takpé, F. Suter, and H. Casanova, "Scheduling parallel task graphs on (almost) homogeneous multicluster platforms," *IEEE TPDS*, vol. 20, no. 7, pp. 940–952, 2009.

[9] M. Drozdowski and W. Kubiak, "Scheduling parallel tasks with sequential heads and tails," *Annals of Operations Research*, vol. 90, no. 0, pp. 221–246, 1999.

[10] V. Nagarajan, J. Wolf, A. Balmin, and K. Hildrum, "Flowflex: Malleable scheduling for flows of mapreduce jobs," in *Middleware 2013*. Springer, 2013, pp. 103–122.

[11] A. Hugo, A. Guermouche, P. Wacrenier, and R. Namyst, "Composing multiple StarPU applications over heterogeneous machines: A supervised approach," *IJHPCA*, vol. 28, no. 3, pp. 285–300, 2014.

[12] R. McNaughton, "Scheduling with deadlines and loss functions," *Management Science*, vol. 6, no. 1, pp. 1–12, 1959.

[13] J. W. H. Liu, "The role of elimination trees in sparse factorization," *SIAM SIMAX journal*, vol. 11, no. 1, pp. 134–172, 1990.

[14] E. Günther, F. König, and N. Megow, "Scheduling and packing malleable and parallel tasks with precedence constraints of bounded width," *Journal of Combinatorial Optimization*, vol. 27, no. 1, pp. 164–181, 2014.

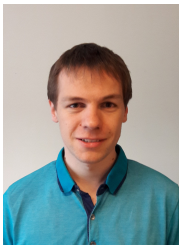
[15] M. Drozdowski, "Scheduling multiprocessor tasks — an overview," *EJOR*, vol. 94, no. 2, pp. 215 – 230, 1996.

[16] R. Lepère, D. Trystram, and G. J. Woeginger, "Approximation algorithms for scheduling malleable tasks under precedence constraints," *IJFCS*, vol. 13, no. 04, pp. 613–627, 2002.

[17] K. Jansen and H. Zhang, "An approximation algorithm for scheduling malleable tasks under general precedence constraints," in *ISAAC 2005*, 2005, pp. 236–245.

[18] S. Hunold, "One step toward bridging the gap between theory and practice in moldable task scheduling with precedence constraints," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 4, pp. 1010–1026, 2015.
[19] K. Makarychev and D. Panigrahi, "Precedence-constrained scheduling of malleable jobs with preemption," in *ICALP 2014*, 2014, pp. 823–834.
[20] L. Fan, F. Zhang, G. Wang, and Z. Liu, "An effective approximation algorithm for the malleable parallel task scheduling problem," *J. of Parallel and Distributed Computing*, vol. 72, no. 5, pp. 693–704, 2012.
[21] G. N. S. Prasanna and B. R. Musicus, "Generalized multiprocessor scheduling and applications to matrix computations," *IEEE TPDS*, vol. 7, no. 6, pp. 650–664, 1996.
[22] A. Guermouche, L. Marchal, B. Simon, and F. Vivien, "Scheduling trees of malleable tasks for sparse linear algebra," in *Proceedings of Euro-Par: Parallel Processing*, 2015, pp. 479–490.
[23] O. Beaumont, N. Bonichon, L. Eyraud-Dubois, and L. Marchal, "Minimizing weighted mean completion time for malleable tasks scheduling," in *IPDPS 2012*, 2012, pp. 273–284.
[24] Q. Wang and K.-H. Cheng, "A heuristic of scheduling parallel tasks and its analysis," *SIAM Journal on Computing*, vol. 21, no. 2, pp. 281–294, 1992.
[25] Y. Zinder and S. Walker, "Scheduling flexible multiprocessor tasks on parallel machines," in *The 9th Workshop on Models and Algorithms for Planning and Scheduling Problems*, 2009.
[26] J. T. Havill and W. Mao, "Competitive online scheduling of perfectly malleable jobs with setup times," *European Journal of Operational Research*, vol. 187, no. 3, pp. 1126–1142, 2008.
[27] J. Du and J. Y.-T. Leung, "Complexity of scheduling parallel task systems," *SIAM J. on Discrete Math.*, vol. 2, no. 4, pp. 473–487, 1989.
[28] N. Kell and J. Havill, "Improved upper bounds for online malleable job scheduling," *J. of Sched.*, vol. 18, no. 4, pp. 393–410, 2015.
[29] V. Vizing, "Minimization of the maximum delay in servicing systems with interruption," *USSR Computational Mathematics and Mathematical Physics*, vol. 22, no. 3, pp. 227 – 233, 1982.
[30] T. E. Carroll and D. Grosu, "Incentive compatible online scheduling of malleable parallel jobs with individual deadlines," in *Parallel Processing (ICPP), 2010 39th International Conference on*. IEEE, 2010, pp. 516–524.
[31] J. Valdes, R. E. Tarjan, and E. L. Lawler, "The recognition of series parallel digraphs," *SIAM J. Comput.*, vol. 11, no. 2, pp. 298–313, 1982.
[32] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M.-H. Su, and K. Vahi, "Characterization of scientific workflows," in *Workflows in Support of Large-Scale Science, 2008. WORKS 2008. Third Workshop on*, nov. 2008, pp. 1 –10.
[33] A. González-Escribano, A. J. C. van Gemund, and V. Cardeñoso-Payo, "Mapping unstructured applications into nested parallelism," in *VECPAR 2002*, 2002, pp. 407–420.
[34] G. Cordasco, R. D. Chiara, and A. L. Rosenberg, "Assessing the Computational Benefits of AREA-Oriented DAG-Scheduling," in *Proceedings of Euro-Par: Parallel Processing*, 2011, pp. 180–192.
[35] O. Sinnen, *Task Scheduling for Parallel Systems*, ser. Wiley series on parallel and distributed computing. Wiley, 2007.
[36] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez, "Implementing multifrontal sparse solvers for multicore architectures with sequential task flow runtime systems," *ACM Trans. Math. Softw.*, vol. 43, no. 2, p. 13, 2016.
[37] M. R. Garey and D. S. Johnson, *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W. H. Freeman and Co, 1979.
[38] 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.
[39] A. Hugo, A. Guermouche, P. Wacrenier, and R. Namyst, "A runtime approach to dynamic resource allocation for sparse direct solvers," in *ICPP 2014*, 2014, pp. 481–490.
[40] R. L. Graham, "Bounds for certain multiprocessing anomalies," *Bell System Technical Journal*, vol. 45, no. 9, pp. 1563–1581, 1966.
[41] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng, "A column approximate minimum degree ordering algorithm," *ACM Trans. Math. Softw.*, vol. 30, no. 3, pp. 377–380, 2004.
[42] F. Pellegrini and J. Roman, "Sparse matrix ordering with Scotch," in *HPCN-Europe 1997*. Springer, 1997, pp. 370–378.
[43] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, 2011.
[44] D. E. Dolan and J. J. Moré, "Benchmarking optimization software with performance profiles," *Mathematical Programming*, vol. 91, no. 2, pp. 201–213, 2002.

Loris Marchal graduated in Computer Sciences and received his PhD from École Normale Supérieure de Lyon (ENS Lyon, France), in 2006. He is now a CNRS researcher at the LIP laboratory of ENS Lyon. His research interests include parallel computing and scheduling.



Bertrand Simon graduated in Computer Sciences at École Normale Supérieure de Lyon (ENS Lyon, France). He is now pursuing his PhD at the LIP laboratory of ENS Lyon. His research interests include data structures, parallel computing and scheduling.

Oliver Sinnen graduated in Electrical and Computer Engineering at RWTH Aachen University, Germany and received his PhD from Instituto Superior Técnico (IST), University of Lisbon, Portugal. He is a Senior Lecturer in the Department of Electrical and Computer Engineering at the University of Auckland, New Zealand, where he leads the Parallel and Reconfigurable Computing Lab. Oliver authored the book "Task Scheduling for Parallel Systems", Wiley.

Frédéric Vivien graduated in Computer Sciences and received his PhD from École Normale Supérieure de Lyon in 1997. From 1998 to 2002, he was an associate professor at the Louis Pasteur University in Strasbourg, France. He is currently an INRIA senior researcher at ENS Lyon, France, where he leads the INRIA project-team Roma. His main research interests include parallel computing, scheduling, and resilience techniques. He is the author of two books.

Web supplementary material for the article: Malleable task-graph scheduling with a practical speed-up model

Loris Marchal, Bertrand Simon, Oliver Sinnen, and Frédéric Vivien

We provide here the proof of Lemma 2, which is used in the proof of Theorem 1. We first recall its statement.

Lemma 2. *In any valid schedule S for $\mathcal{J}$,*

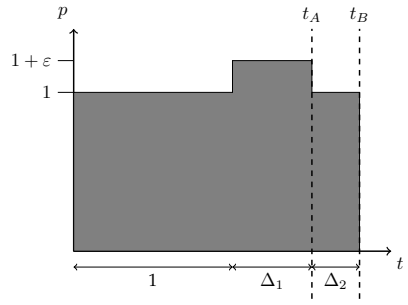
i) each pair of chains $L_{x_i}, L_{\overline{x_i}}$ is completely processed during time interval $[t_i, M - t_i]$,
ii) one of them is started at time t_i and the other one at time $t_i + 1$,
iii) all tasks of both chains are allocated their threshold,
iv) there is no idle time between any two consecutive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

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

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

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

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



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

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

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

We symmetrically apply the same reasoning to the last tasks C and D of these two chains, and their *starting* times t_C and t_D, assuming that C is started before D. By setting $t_D = M - t_i - 1 - \Delta'_1$, we get $t_C \leq M - t_i - 2 + \Delta'_1\varepsilon$. We distinguish between two cases, depending on the chains to which A, B, C, and D belong to:

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

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

  which gives

  $$\Delta'_1\varepsilon \geq 1 - \Delta_1\varepsilon$$

  We have $\Delta_1 \leq 1$ and similarly, $\Delta'_1 \leq 1$. Together with the previous inequality, this gives $\varepsilon \geq 1/2$ which is not possible since $\varepsilon = 1/4n$. Hence B and C cannot belong to the same chain.
- In the second case, we consider that A and C belong to the same chain. Because exactly $4(n-i)+2m+1$ tasks need to be processed between A and C (and between B and D), we have

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

  which gives

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

- *L. Marchal, B. Simon and F. Vivien are with CNRS, INRIA and University of Lyon, LIP, ENS Lyon, 46 allée d'Italie, Lyon, France. E-mail: {loris.marchal,bertrand.simon,frederic.vivien}@ens-lyon.fr*
- *O. Sinnen is with Dpt. of Electrical and Computer Engineering, University of Auckland, New Zealand E-mail: o.sinnen@auckland.ac.nz*

and

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

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

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

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

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

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