Scheduling Independent Moldable Tasks on Multi-Cores with GPUs

Raphaël Bleuse, Sascha Hunold, Safia Kedad-Sidhoum, Florence Monna, Grégory Mounié, Denis Trystram

Scheduling Independent Moldable Tasks on Multi-Cores with GPUs

Raphaël Bleuse, Sascha Hunold, Safia Kedad-Sidhoum, Florence Monna, Grégory Mounié, Denis Trystram

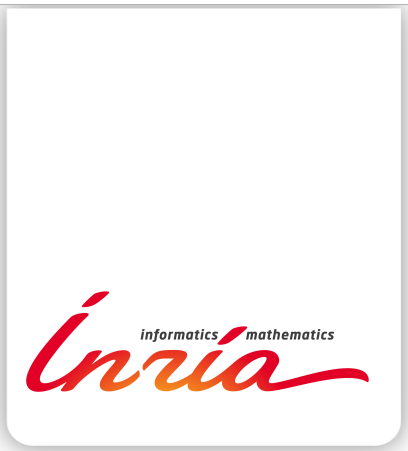
RESEARCH REPORT

N° 8850

January 2016

Project-Teams Datamove

ISSN 0249-6399 ISRN INRIA/RR--8850--FR+ENG



Scheduling Independent Moldable Tasks on Multi-Cores with GPUs

Raphaël Bleuse*, Sascha Hunold†, Safia Kedad-Sidhoum‡,
Florence Monna‡, Grégory Mounié*, Denis Trystram*

Project-Teams Datamove

Research Report n° 8850 — January 2016 — 14 pages

Abstract: The number of parallel systems using accelerators is growing up. The technology is now mature enough to allow sustained petaflop/s. However, reaching this performance scale requires efficient scheduling algorithms to manage the heterogeneous computing resources.
We present a new approach for scheduling independent tasks on multiple CPUs and multiple GPUs. The tasks are assumed to be parallelizable on CPUs using the moldable model: the final number of cores allotted to a task can be decided and set by the scheduler. More precisely, we design an algorithm aiming at minimizing the makespan—the maximum completion time of all tasks—for this scheduling problem. The proposed algorithm combines a dual approximation scheme with a fast integer linear program (ILP). It determines both the partitioning of the tasks, i.e. whether a task should be mapped to CPUs or a GPU, and the number of CPUs allotted to a moldable task if mapped to the CPUs. A worst case analysis shows that the algorithm has an approximation ratio of $\frac{3}{2} + \epsilon$. However, since the complexity of the ILP-based algorithm could be non-polynomial, we also present a proved polynomial-time algorithm with an approximation ratio of $2 + \epsilon$.
We complement the theoretical analysis of our two novel algorithms with an experimental study. In these experiments, we compare our algorithms to a modified version of the classical HEFT algorithm, adapted to handle moldable tasks. The experimental results show that our algorithm with the $\frac{3}{2} + \epsilon$ approximation ratio produces significantly shorter schedules than the modified HEFT for most of the instances. In addition, the experiments provide evidence that this ILP-based algorithm is also practically able to solve larger problem instances in a reasonable amount of time.

Key-words: scheduling, heterogeneous computing, moldable tasks, dual approximation scheme, integer linear programming

* *Univ. Grenoble Alpes – LIG, France*
Inria team/project Datamove
E-mail: {raphael.bleuse, gregory.mounie, denis.trystram}@imag.fr
† *Vienna University of Technology, Faculty of Informatics, Institute of Information Systems, Favoritenstrasse 16/184-5, 1040 Vienna, Austria.*
E-mail: hunold@par.tuwien.ac.at
‡ *Sorbonne Universités, UPMC Univ. Paris 06, UMR 7606, LIP6, France*

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

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

Ordonnancement de tâches indépendantes modelables sur des processeurs multi-cœurs et GPU

Résumé : La technologie des accélérateurs est maintenant suffisamment mure pour permettre des performances soutenues de l'ordre du petaflop/s. Néanmoins, atteindre ce niveau de performance demande des algorithmes d'ordonnancement efficaces capable de gérer l'hétérogénéité des ressources de calculs.

Nous présentons une nouvelle approche pour les algorithmes d'ordonnancement pour multiprocesseurs (CPU) et multi-GPU. Chaque tâche peut être exécutée en parallèles sur les CPU. Nous utilisons le modèle des tâches modelables: l'ordonnanceur peut choisir le nombre de cœurs alloués à chaque tâche sur CPU. Notre but est de minimiser la date de terminaison de la dernière tâche (makespan). Notre algorithme combine un schéma d'approximation duale avec la résolution, rapide, d'un programme linéaire en nombre entier (PLNI). Le programme linéaire décide à la fois de l'équilibrage entre CPU et GPU, mais aussi du nombre de cœurs pour les tâches allouées sur CPU. Une analyse en pire cas de notre algorithme donne un ratio d'approximation de $\frac{3}{2} + \epsilon$. Puisque, la compléxité du PLNI pourrait ne pas être polynômial, nous proposons aussi un algorithme pleinement polynomial avec un ratio d'approximation de $2 + \epsilon$.

Une étude expérimentale vient compléter l'analyse théorique. Nous comparons notre algorithme avec une version modifiée de l'algorithme HEFT, adapté aux tâches modelables. L'étude montre que notre algorithme obtient des ordonnancements significativement plus courts sur la plupart des instances. De plus, ces expérimentations montrent que le PLNI est en pratique capable de résoudre des instances de grande taille en un temps raisonnable.

Mots-clés : Ordonnancement, calcul sur machines hétérogènes, tâches modelables, schéma d'approximation duale, programmation linéaire en nombre entier

1 Introduction

Today's available parallel computing systems often consist of compute nodes, which contain multi-core CPUs and additional hardware accelerators [1]. Such accelerators (General Purpose Graphic Processor Units, denoted by GPU for short) have a simpler architecture than traditional CPUs. They offer a high degree of parallelism as they possess a large number of compute cores but only provide a limited amount of memory. As a consequence, the heterogeneity of the compute nodes architectures is increasing. Still, these hybrid systems are becoming more and more popular, and it is foreseeable that the trend of using such hybrid systems will grow as GPUs consume significantly less power per flop than standard CPUs [2].

Recent works have addressed the issue of efficiently utilizing such hybrid platforms, e.g., to improve the performance of numerical kernels [3], [4]; biological sequence alignments [5]; or molecular dynamics codes [6]. The scheduling algorithms employed in these approaches were tailor-made for the targeted applications. However, these scheduling algorithms do not exploit a high-level view of all tasks in order to provide an efficient and transparent solution for any type of parallel application. The challenge today is to develop effective and automatic management of the resources for executing generic applications on new parallel hybrid platforms.

We have already done first steps to devise such a generic scheduling algorithm for heterogeneous compute nodes. In particular, we have developed an approximation algorithm with a constant worst-case performance guarantee that provides solutions for the scheduling problem of independent, sequential tasks onto CPUs or GPUs for the makespan objective [7], [8]. However, the algorithm had two main drawbacks. First, even though the proposed algorithm has a polynomial-time complexity, it relies on dynamic programming, in which a vast state space has to be explored. For that reason, the practical applicability of the algorithm is limited due to its large run-time. Second, tasks could potentially benefit from internal (data-)parallelism on CPUs, as our previous algorithm worked for sequential tasks only. Thus, in the present work we assume that tasks are *moldable*, i.e., they are computational units that may be executed by several (more than one) processors. Then, the run-time of such a moldable task depends in the number of processors allotted to it. Such a model allows to exploit the two types of parallelism offered by hybrid parallel computing platforms: the inherent parallelism induced by GPU's architecture, and the parallelization of tasks on several CPUs. Our objective within this work is to propose a generic method to leverage these two different kind of parallelism.

Compared to the state of the art, we make the following contributions in the present article:

- We present a novel algorithm—combining dual approximation and integer linear programming—that can solve the scheduling problem of independent, moldable tasks on hybrid parallel compute platforms consisting of m CPUs and k GPUs.
- We prove that the algorithm has an approximation ratio of $\frac{3}{2} + \epsilon$.
- We show through a sequence of experiments that even though our algorithm is based on integer linear programming, which may be theoretically non-polynomial, it is still practically relevant, as it provides competitive schedules but has a relatively short run-time.
- We also present a fully polynomial-time algorithm for the same scheduling problem, for which we prove an approximation ratio of $2 + \epsilon$.

The paper is organized as follows: in Section 2, we define the scheduling problem targeted in this work. We examine existing related works on scheduling with GPUs and moldable tasks in Section 3. We present a novel scheduling algorithm, which is based on integer linear programming (ILP), in Section 4 and provide the theoretical analysis of the algorithm in Section 5. Section 6 presents a fully polynomial approximation algorithm, which we introduce to compare with our ILP-based algorithm. In Section 7, we present an experimental study that compares the solution quality (makespan) of various scheduling algorithms for a variety of test instances. We conclude the paper in Section 8.

2 Problem Definition

We consider a multi-core parallel platform composed of m identical CPUs and k identical GPUs. An instance of the problem is described as a set $\{T_1, \ldots, T_n\}$ of n independent tasks considered as *moldable* when assigned to the CPUs and sequential when assigned to a GPU. The processing time of any task T_j is represented by a function $\overline{p_j} : l \mapsto \overline{p_{j,l}}$ that represent the processing time when executed on l CPUs and by $\underline{p_j}$ that is the processing time when executed on a GPU. We assume that these processing times are known in advance. It is a common assumption when dealing with classical numerical codes like those considered in the experiments.

The *scheduling* problem consists in finding a function σ that associates for each task T_j its starting time and the computing resources assigned for its execution. A task is either assigned to a single GPU or to a subset of the available CPUs, under the constraints that the task starts its execution simultaneously on all the allocated resources and occupies them without interruption until its completion time.

We define the CPU work function w_j of a task T_j—corresponding to its computational area on the CPUs in the Gantt chart representation of a schedule—as $w_j : l \mapsto w_{j,l} = l \times \overline{p_{j,l}}$ for $l \leqslant m$. According to the usual executions of parallel programs, we assume that the tasks assigned to the CPUs are monotonic: assigning more CPUs to a task usually decreases its execution time at a price of an increased work. This is due to some internal communications and synchronizations. There are two types of monotonies: namely the *time monotony* which is achieved when $\overline{p_j}$ is a non-increasing function for any task and the *work monotony* that is achieved when w_j is an increasing function for any task. A set of tasks is said to be *monotonic* when it achieves both monotonies. This assumption may be interpreted by the well-known Brent's lemma [9], which states that the parallel execution of a task achieves some speedup if it is large enough, but does not lead to super-linear speedups. Notice that an instance of the problem can always be transformed to fulfill the time monotony property by replacing function $\overline{p_j}$ by $\overline{p_j}' : l \mapsto \min\{\overline{p_{j,q}} \mid q = 1, \ldots, l\}$. Such a transformation does not affect the optimal solution of the scheduling. In the sequel, we always assume that the set of tasks of the considered instance is monotonic. There is no need of such

an hypothesis on the GPUs as the tasks are considered sequential on this architecture.

The makespan is defined as the maximum completion time of the last finishing task. For the problem considered here, the objective is to minimize the makespan of the whole schedule, which is the maximum of the makespan on the CPUs and the makespan on the GPUs. The problem is denoted by $(Pm, Pk) \mid mold \mid C_{\max}$.

Observe that if all the tasks are sequential and the processing times are the same on both devices ($p_j = \overline{p_{j,1}}$) for $j = 1, \ldots, n$, the problem $(Pm, Pk) \mid mold \mid C_{\max}$ is equivalent to the classical $P \mid\mid C_{\max}$ problem, which is NP-hard. Thus, the problem of scheduling moldable tasks with GPUs is also NP-hard, and we are looking for efficient algorithms with guaranteed approximation ratio. Recall that for a given problem the approximation ratio ρ_A of an algorithm A is defined as the maximum over all the instances I of the ratio $\frac{f(I)}{f^*(I)}$ where f is any minimization objective and f^* is its optimal value.

This study considers algorithms that provide non-preemptive schedules with contiguous processor assignment. It is clear that the optimal assignment could use CPUs that are not consecutive ones. However, this restriction does not hinder the achieved results [10].

3 Related Work

In this section we will first discuss the most relevant works related to heterogeneous scheduling. Then we will present the existing results dealing with moldable tasks.

From a scheduling perspective, $(Pm, Pk) \mid\mid C_{\max}$ is a special case of $R \mid\mid C_{\max}$. Lenstra *et al.* [11] propose a PTAS for the problem $R \mid\mid C_{\max}$ with running time bounded by the product of $(n+1)^{m/\epsilon}$ and a polynomial of the input size. Let us notice that if parameter m is not fixed, then the algorithm is not fully polynomial. The authors also prove that unless $P = NP$, there is no polynomial-time approximation algorithm for $R \mid\mid C_{\max}$ with an approximation factor less than $\frac{3}{2}$ and present a 2-approximation algorithm. This algorithm is based on rounding the optimal solution of the preemptive version of the problem. Shmoys and Tardos generalize the rounding technique for any fractional solutions [12]. Recently, Shchepin and Vakhania [13] introduce a new rounding technique which yields an improved approximation factor of $2 - \frac{1}{m}$. This is so far the best approximation result for $R \mid\mid C_{\max}$. If we look at the more specific problem of scheduling unrelated machines of few different types, Bonifaci and Wiese [14] present a PTAS to solve this problem. However, the time complexity of the polynomial algorithm is not provided so that the algorithm does not seem to be potentially useful from a practical perspective. Finally, it is worth noticing that if all the tasks of the addressed problem have the same acceleration on the GPUs, the problem reduces to a $Q \mid\mid C_{\max}$ problem with two machines speeds.

A family of scheduling algorithms based on the dual approximation scheme for the problem $(Pm, Pk) \mid\mid C_{\max}$ with sequential tasks has been developed in a previous paper [7]. These algorithms provide a $1 + \epsilon + \mathcal{O}(\frac{1}{q})$ approximation for any $\epsilon > 0$ with a cost of $\mathcal{O}(n^2 k^q m^q)$.

The problem of scheduling independent moldable tasks on homogeneous parallel systems has been extensively studied in the last decade. Among other reasons, the interest in studying this problem was motivated by scheduling jobs in batch processing in HPC clusters. For instance, the documentation of TORQUE mentions a basic moldable submission mechanism. A noteworthy work is the implementation and evaluation of a moldable scheduler for OAR by Lionel Eyraud [15].

Jansen and Porkolab [16] proposed a polynomial time approximation scheme based on a linear programming formulation for scheduling independent moldable tasks. The complexity of their scheme, although linear in the number of tasks, is highly dependent on the accuracy of the approximation due to an exponential factor in the number of processors. Thus, although the result is of significant theoretical interest, this algorithm cannot be considered for a practical use.

Most existing previous works are based on a two-phase approach, initially proposed by Turek, Wolf and Yu [17]. The basic idea here was to select first an assignment (the number of processors assigned to each task) and in a second step to solve the resulting rigid (non-moldable) scheduling problem, which is a classical scheduling problem with multiprocessor tasks. As far as the makespan objective is concerned, this problem is related to a 2-dimensional strip-packing problem for independent tasks [18], [19].

It is clear that applying an approximation of guarantee ρ for the rigid problem on the assignment of an optimal solution provides the same guarantee ρ for the moldable problem if ever an optimal assignment can be found. Two complementary ways have been proposed for solving the problem, either focusing on the first phase of assignment or on the scheduling (second phase). Ludwig [20] improved the complexity of the assignment selection in the special case of monotonic tasks leading to a 2-approximation. The other way corresponds to choosing an assignment such that the resulting non-moldable problem is not a general instance of strip-packing, and hence better specific approximation algorithms can be applied. Using the knapsack problem as an auxiliary problem for the selection of the assignment, this technique leads to a $(\frac{3}{2} + \epsilon)$-approximation algorithm for any $\epsilon > 0$ by Mounié *et al.* [10].

Furthermore, an extensive both theoretical and experimental comparison of low-cost scheduling algorithms for moldable tasks is carried out in [21].

4 Algorithm APPROX-3/2

The principle of the algorithm is based on a dual approximation [22]. Recall that a ρ-dual approximation algorithm for a minimization problem takes a real number λ (called the guess) as an input, and either delivers a solution whose objective function's value is at most $\rho\lambda$, or answers correctly that there exists no solution whose objective function's value is at most λ.

Our aim here is to optimize the makespan, and we target $\rho = \frac{3}{2}$. Let λ be the current real number input for the dual approximation. In the whole section, we suppose there exists a schedule of length lower than λ, and we show how to build a schedule of length at most $\frac{3\lambda}{2}$.

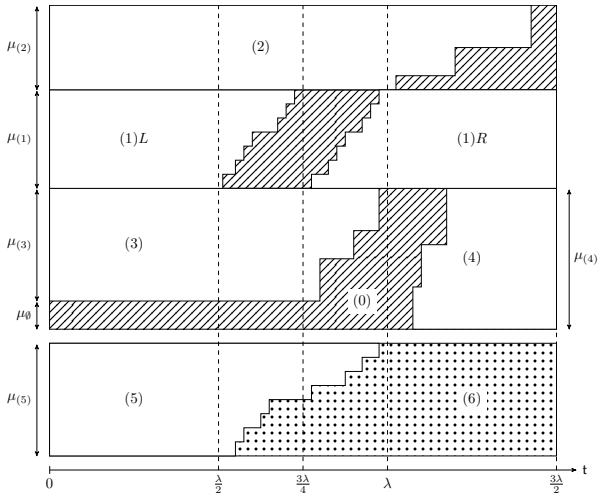


Fig. 1: Structure of the schedule. The number of processors used by set (i) is denoted by $\mu_{(i)}$. The number of CPUs below set (3) is denoted by $\mu_\emptyset$.

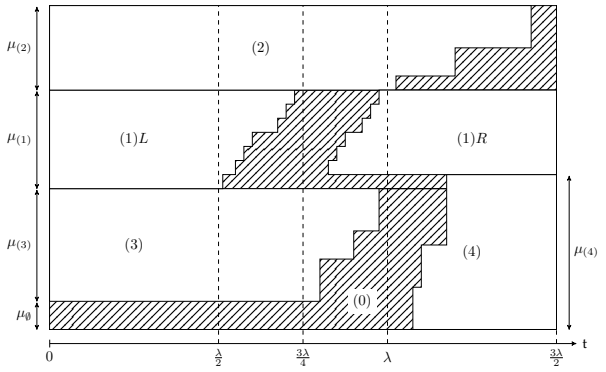


Fig. 2: Structure of the schedule on CPUs with an odd number of tasks in set (1).

Given a positive number h, we define—as in [10]—for each task T_j its *canonical number* of CPUs $\gamma(j,h)$. It is the minimal number of CPUs needed to execute task T_j in time at most h. If T_j cannot be executed in time less than h on m CPUs, we set by convention $\gamma(j,h) = +\infty$. Observe that $w_{j,\gamma(j,h)}$ is the minimal work area needed to execute T_j on CPUs in time at most h. Also note that if the set of tasks is monotonic, the canonical number of CPUs can be found in time $\mathcal{O}(\log m)$ by binary search.

4.1 Partitioning Tasks

The idea of the algorithm is to partition the set of tasks on the CPUs into five sets, and on the GPUs into two sets, as depicted in Figure 1. This choice of the task assignement to CPUs is detailed below:

- (0): the set containing the sequential tasks assigned to the CPUs with a processing time at most $\frac{\lambda}{2}$.
- (1): the set containing the sequential tasks assigned to the CPUs with a processing time greater than $\frac{\lambda}{2}$ and at most $\frac{3\lambda}{4}$. This set is partitioned in two shelves as depicted in Figure 1: namely, the left set $(1)L$ and the right set $(1)R$.
- (2): the set containing the tasks—either parallel or sequential—assigned to the CPUs with different canonical numbers of CPUs for the times λ and $\frac{3\lambda}{2}$. Task T_j is then assigned to $\gamma(j, \frac{3\lambda}{2})$ CPUs.
- (3): the set containing the tasks assigned to their canonical number of CPUs for time λ. If this number is 1, then the processing time of the corresponding task is strictly greater than $\frac{3\lambda}{4}$.
- (4): the set containing the parallel tasks assigned to their canonical number of CPUs for time $\frac{\lambda}{2}$. Note that $\gamma(j, \frac{\lambda}{2})$ is greater than 1.

Similarly, the tasks assigned to GPUs are partitioned in two sets:

- (5): the set containing the tasks assigned to a GPU with a processing time greater than $\frac{\lambda}{2}$ and at most λ.
- (6): the set containing the tasks assigned to a GPU with a processing time at most $\frac{\lambda}{2}$.

Such a partition ensures that both the makespans on the CPUs and on the GPUs are lower than $\frac{3\lambda}{2}$.

Note that if there is an even number of tasks assigned to set (1), both sets $(1)L$ and $(1)R$ occupy the same number of CPUs. On the contrary, if set (1) contains an odd number of tasks, the right set occupies one less processor (as shown in Figure 2).

4.2 Mathematical Formulation

Partitioning tasks into the seven above-mentioned sets using a list algorithm does not achieve the desired performance guarantee. Therefore, we propose an Integer Linear Program (ILP) for solving the assignment problem.

4.2.1 Objective Function and Constraints

We define W_C as the computational area of the CPUs on the Gantt chart representation of a schedule, i.e. the sum of all the works of the tasks assigned to some of the CPUs:

$$W_C = \sum_{T_j \in (0)\cup(1)} w_{j,1} + \sum_{T_j \in (2)} w_{j,\gamma(j,\frac{3\lambda}{2})} + \sum_{T_j \in (3)} w_{j,\gamma(j,\lambda)} + \sum_{T_j \in (4)} w_{j,\gamma(j,\frac{\lambda}{2})} \quad (1)$$

We want to obtain a specific 5-set schedule on the CPUs and a 2-set schedule on the GPUs, hence we look for an assignment minimizing:

- (C_1) The total computational area W_C on the CPUs. It is at most $m\lambda$.

The assignment must satisfy the following constraints:

- (C_2) Sets $(1)L$, (2) and (3) use a total of at most m processors.
- (C_3) Sets $(1)R$, (2) and (4) use a total of at most m processors.
- (C_4) The total computational area on the GPU is at most $k\lambda$.
- (C_5) Set (5) uses a total of at most k processors.
- (C_6) Each task is assigned to exactly one set.
- (C_7) The number of tasks assigned to set (1) is the total number of tasks processed in its two shelves.
- (C_8) The tasks of set (1) are evenly shared between its two sets $(1)L$ and $(1)R$, i.e. there is at most one task less in $(1)R$. The idle time induced by the difference is used to process a fraction of a task assigned to set (4).

Such an assignment clearly defines a schedule of length at most $\frac{3\lambda}{2}$ which allows us to build a solution.

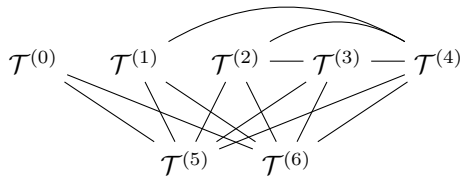


Fig. 3: Intersection graph of the eligible allocation sets, in its most generic shape. Actual instances may have fewer edges.

4.2.2 Filtering

The structure of the schedule allows tasks to belong only to a limited number of shelves. Hence we define for each task j the filtering function $F(j)$ computing the set of possible containing shelves. For each set (i) we also define the set $\mathcal{T}^{(i)}$ of tasks eligible for an allocation in (i). The eligible allocation sets are explicitly defined in Equation (2). We furthermore define for each task T_j several binary variables $x_j^{(i)}$, where $i \in F(j)$. If T_j is assigned to set (i), we define $x_j^{(i)}$ to be 1. Otherwise we set $x_j^{(i)}$ to be 0. We also define for set (1) the variable $left^{(1)}$ (resp. $right^{(1)}$), that corresponds to the number of tasks assigned to the $(1)L$ (resp. $(1)R$) shelf of set (1) (see Figure 1).

$$
\begin{aligned}
\mathcal{T}^{(0)} &= \left\{ j \mid \overline{p_{j,1}} \leq \frac{\lambda}{2} \right\} \\
\mathcal{T}^{(1)} &= \left\{ j \mid \frac{\lambda}{2} < \overline{p_{j,1}} \leq \frac{3\lambda}{4} \right\} \\
\mathcal{T}^{(2)} &= \left\{ j \mid \lambda < \overline{p_{j,\gamma(j,\frac{3\lambda}{2})}} \leq \frac{3\lambda}{2} \right\} \\
\mathcal{T}^{(3)} &= \left\{ j \mid \frac{\lambda}{2} < \overline{p_{j,\gamma(j,\lambda)}} \leq \lambda \right\} \setminus \mathcal{T}^{(1)} \\
\mathcal{T}^{(4)} &= \left\{ j \mid \overline{p_{j,\gamma(j,\frac{\lambda}{2})}} \leq \frac{\lambda}{2} \ \wedge \ \gamma(j,\frac{\lambda}{2}) > 1 \right\} \\
\mathcal{T}^{(5)} &= \left\{ j \mid \frac{\lambda}{2} < \underline{p_j} \leq \lambda \right\} \\
\mathcal{T}^{(6)} &= \left\{ j \mid \underline{p_j} \leq \frac{\lambda}{2} \right\}
\end{aligned}
\tag{2}
$$

This filtering step helps a lot reducing the search space. The intersection graph of the eligible allocation sets shown in Figure 3 explains this behavior. Each task can simultaneously belong to only a limited number of sets, since most sets are mutually exclusive. In most cases, a task belong to 2 or 3 sets.

4.2.3 Integer Linear Program

Determining if such an assignment exists reduces to solving an ILP that can be formulated as follows:

$$
\begin{aligned}
\min W_C^{(ILP)} = & \sum_{j\in\mathcal{T}^{(0)}} w_{j,1}x_j^{(0)} + \sum_{j\in\mathcal{T}^{(1)}} w_{j,1}x_j^{(1)} \\
& + \sum_{j\in\mathcal{T}^{(2)}} w_{j,\gamma(j,\frac{3\lambda}{2})}x_j^{(2)} + \sum_{j\in\mathcal{T}^{(3)}} w_{j,\gamma(j,\lambda)}x_j^{(3)} \\
& + \sum_{j\in\mathcal{T}^{(4)}} w_{j,\gamma(j,\frac{\lambda}{2})}x_j^{(4)}
\end{aligned}
$$

$$
\begin{aligned}
\text{s.t. } & W_C^{(ILP)} \leqslant m\lambda && (C_1)\\
& \sum_{j\in\mathcal{T}^{(2)}} \gamma(j,\frac{3\lambda}{2})x_j^{(2)} + \sum_{j\in\mathcal{T}^{(3)}} \gamma(j,\lambda)x_j^{(3)} + left^{(1)} \leqslant m && (C_2)\\
& \sum_{j\in\mathcal{T}^{(4)}} \gamma(j,\frac{\lambda}{2})x_j^{(4)} + \sum_{j\in\mathcal{T}^{(2)}} \gamma(j,\frac{3\lambda}{2})x_j^{(2)} + right^{(1)} \leqslant m && (C_3)\\
& \sum_{j\in\mathcal{T}^{(5)}} \underline{p_j}x_j^{(5)} + \sum_{j\in\mathcal{T}^{(6)}} \underline{p_j}x_j^{(6)} \leqslant k\lambda && (C_4)\\
& \sum_{j\in\mathcal{T}^{(5)}} x_j^{(5)} \leqslant k && (C_5)\\
& \sum_{i\in F(j)} x_j^{(i)} = 1 \quad \forall j \in \{1,\dots,n\} && (C_6)\\
& \sum_{j\in\mathcal{T}^{(1)}} x_j^{(1)} = left^{(1)} + right^{(1)} && (C_7)\\
& 0 \leqslant left^{(1)} - right^{(1)} \leqslant 1 && (C_8)\\
& x_j^{(i)} \in \{0,1\} \quad \forall j \in \{1,\dots,n\}, \forall i \in F(j) && (C_9)\\
& left^{(1)}, right^{(1)} \in \{0,\dots,m\} && (C_{10})
\end{aligned}
$$

The first eight equations of this integer linear program correspond to the constraints listed above in order to obtain a 5-set schedule on the CPUs and a 2-set schedule on the GPUs. The last two equations (C_9), (C_{10}) are integrity constraints for the variables of the integer linear program.

5 Analysis of the Algorithm APPROX-3/2

The integer linear program presented above derives from the structural properties of the schedule we aim to construct. The analysis—rather technical—is structured in three steps. First we explain how the estimation of the instance's makespan λ helps us to sort and allocate tasks. We then give some insight on the structure of the proposed partitioning. We finally prove the correctness of the dual approximation, i.e. we prove the reject condition is actually matched by the algorithm.

5.1 Structure of a Schedule of Makespan λ

To take advantage of the dual approximation paradigm, we have to make explicit the consequences of the assumption that there exists a schedule of length at most λ. We state below some straightforward properties of such a schedule. They should give the insight for the construction of the solution.

Proposition 1. *In a solution of makespan at most λ, the execution time of each task is at most λ. The computational area on the CPUs (resp. GPUs) is at most $m\lambda$ (resp. $k\lambda$).*

Remark that for the problem of scheduling moldable tasks on identical processors [10], it is enough to look at the $2m$ tasks with the longest processing times. If they have a computational area larger than $m\lambda$, then a schedule of length λ cannot exist. In the case of heterogeneous processors some of these tasks can be assigned to a GPU, therefore the n tasks have to be considered in our case.

Proposition 2. *In a solution of makespan at most λ, if there exist two consecutive tasks on the same processor such that one of the task has an execution time greater than $\frac{\lambda}{2}$, then the other task has an execution time lower than $\frac{\lambda}{2}$.*

Proposition 3. *Two tasks with sequential processing times on CPU greater than $\frac{\lambda}{2}$ and lower than $\frac{3\lambda}{4}$ can be executed successively on the same CPU within a time at most $\frac{3\lambda}{2}$.*

We now look at exploiting the properties of a schedule of makespan at most λ, in order to construct the seven sets. The constraints of the integer linear program derive from these properties.

From Proposition 3, as we aim at a makespan of $\frac{3\lambda}{2}$, two tasks from set (1) can be executed successively on the same CPU. The whole set occupies $\mu_{(1)}$ CPUs. The number of tasks in set $(1)R$ is given by $\mu_{(1)} - \mathbf{1}_{(1)\text{odd}}$ where $\mathbf{1}_{(1)\text{odd}}$ is an indicator function equals to 1 when the number of tasks in set (1) is odd.

From Proposition 2, the tasks whose execution times on CPUs are greater than $\frac{\lambda}{2}$ do not use more than $m - \mu_{(1)}$ CPUs, and hence can be executed concurrently on the CPUs in set (3). They occupy $\mu_{(3)}$ CPUs.

Set (2) does not exist in a solution of makespan λ, since the processing times of all the tasks in (2) are greater than λ with the number of CPUs they are assigned to. However, with this assignment and the monotony of the tasks on CPUs, the work of the tasks in (2) is lower than their corresponding work in the optimal schedule. Therefore, every task assigned to (2) in the constructed schedule is a gain on the total work on the CPUs. The tasks of (2) occupy $\mu_{(2)}$ CPUs and the inequality $\mu_{(1)} + \mu_{(2)} + \mu_{(3)} \leqslant m$ must be satisfied.

The remaining tasks on CPUs have execution times lower than $\frac{\lambda}{2}$ on CPU, and those that are not sequential can be executed within a time at most $\frac{\lambda}{2}$ in set (4). These tasks cannot be executed on the CPUs occupied by tasks from set (2) but can be processed after the tasks from set (3). They cannot be on the CPUs that already process two tasks of (1), but if the number of tasks in (1) is odd, there is a CPU that only processes one task from $(1)L$ and a task from (4) can be executed on this CPU. Therefore, if we denote by $\mu_{(4)}$ the number of CPUs occupied by tasks of (4), the inequality $\mu_{(1)} - \mathbf{1}_{(1)\text{odd}} + \mu_{(2)} + \mu_{(4)} \leqslant m$ must be satisfied.

The remaining sequential tasks on CPUs have execution times lower than $\frac{\lambda}{2}$ on CPU and are assigned to set (0).

With the same reasoning, the tasks on GPUs whose execution times are greater than $\frac{\lambda}{2}$ do not use more than k GPUs, and hence can be executed concurrently in set (5).

The remaining tasks on GPUs have execution times lower than $\frac{\lambda}{2}$ on GPU and can be executed within a time at most $\frac{\lambda}{2}$ in set (6) on the GPUs. It can be after a task from (5) or on the remaining free GPUs.

5.2 Structure of the Partitioning

We now have to prove that under the assumption that the dual approximation do not reject the current guess λ, i.e. $W_C^{(ILP)} \leqslant m\lambda$, the ILP solution leads to a feasible seven-shelf schedule.

The structure of the partitioning verifies some properties exposed hereinafter.

Lemma 4. *With the assumption that $W_C^{(ILP)} \leqslant m\lambda$, the tasks assigned to sets (1), (2), (3) and (4) occupy at most m CPUs, in a time at most $\frac{3\lambda}{2}$.*

Proof. From Constraints (C_2) and (C_3), the assignment of the tasks of the four sets is such that they occupy at most m CPUs. The tasks are scheduled two by two in (1). According to Constraint (C_8), set (1) may have an even number of tasks (see Figure 1) or an odd number of tasks (see Figure 2). Whenever set (1) is assigned an odd number of tasks, an extra processor is available to compute tasks from set (4). The tasks of (4) are scheduled after tasks of (3) or on remaining free CPUs. With this schedule, at most m CPUs are occupied and the makespan is at most $\frac{3\lambda}{2}$. □

Lemma 5. *If $W_C^{(ILP)} \leqslant m\lambda$, the tasks assigned to set (0) fit in the remaining free computational space, while keeping the makespan under $\frac{3\lambda}{2}$.*

Proof. The tasks of set (0) all have a sequential processing time on CPU lower than $\frac{\lambda}{2}$ by construction, and they necessarily fit into the remaining computational space in the allowed area of $\frac{3\lambda}{2}\, m$ (represented by the dashed area in the Figures 1 and 2). The schedule would otherwise contradict Proposition 1.

The following algorithm can be used to schedule these tasks:

1) Consider the remaining tasks $T_1, \ldots, T_f$ ordered by non-increasing sequential processing time on CPU, where f is the number of remaining tasks.
2) At each step s ($s = 1, \ldots, f$) assign task T_s to the least loaded CPU, at the latest possible date, or between set (3) and set (4) if relevant. Update the CPU's load.

At each step, the least loaded CPU has a load at most λ: it would otherwise contradict the fact that the total work area of the tasks is bounded by $m\lambda$ (according to Proposition 1). Hence, the idle time interval on the least loaded CPU has a length at least equal to $\frac{\lambda}{2}$, and can contain the task T_s. This proves the correctness of the algorithm above. □

Lemma 6. *If $W_C^{(ILP)} \leqslant m\lambda$, the tasks assigned to sets (5) and (6) occupy at most k GPUs, in a time at most $\frac{3\lambda}{2}$.*

Proof. When the tasks of set (5) are assigned to the GPUs, they take up to k GPUs from Constraint (C_5), and their processing time is lower than λ: the dual approximation would otherwise reject the solution. The tasks of (5) are scheduled first, one per GPU.

The tasks of set (6) all have a processing time on GPU lower than $\frac{\lambda}{2}$ by construction and they necessarily fit into the remaining computational space in the allowed area of $\frac{3\lambda}{2}\, k$. The schedule would otherwise contradict Proposition 1 and Constraint (C_4).

The following algorithm can be used to schedule these tasks:

1) Consider the remaining tasks $T_1, \ldots, T_f$ ordered by non-increasing processing time on GPU, where f is the number of remaining tasks .

2) At each step s ($i = 1, \ldots, f$) assign task T_s to the least loaded GPU, at the latest possible date. Update the GPU's load.

At each step, the least loaded GPU has a load at most λ: it would otherwise contradict the fact that the total work area of the tasks is bounded by $k\lambda$ (according to Proposition 1 and Constraint (C_4)). Hence, the idle time interval on the least loaded GPU has a length at least equal to $\frac{\lambda}{2}$ and can contain the task T_s. The correctness of the algorithm above is proved. □

These three lemmas allow us to derive the following theorem:

Theorem 7. *If $W_C^{(ILP)} \leqslant m\lambda$, then there exists a schedule of length at most $\frac{3\lambda}{2}$ built upon the assignment of the tasks given by the solution of ILP.*

Proof. The solution of ILP returns an assignment such that the computational area on the CPUs is minimized, therefore its value $W_C^{(ILP)}$ is lower than the computational area on the CPUs in the optimal schedule, W_C^*, which is lower than $m\lambda$ since we assumed that there exists a schedule of makespan at most λ. The three lemmas show that the schedule constructed with the assignment of the tasks given by the solution of ILP has a makespan lower than $\frac{3\lambda}{2}$. □

If the computational area on the CPUs, i.e. the objective of the integer linear program $W_C^{(ILP)}$, is greater than $m\lambda$, the dual approximation algorithm rejects λ. Indeed, this computational area is minimized in the resolution of (ILP): if we had $\lambda \leqslant C^*_{\max}$, we would get $W_C^{(ILP)} \leqslant W_C^*$, which is impossible since we have $W_C^* \leqslant m\lambda$. Therefore in that case there exists no solution with a makespan at most λ, and the algorithm rejects the current guess λ.

We have so far proved that—for a given guess λ of the dual approximation algorithm—if the solution of ILP has a computational area on the CPUs greater than $m\lambda$, then there is no solution of makespan λ, and the guess has to be rejected. If the solution of ILP has a computational area on the CPUs lower than $m\lambda$, then we can construct a solution with a makespan at most $\frac{3\lambda}{2}$, with the corresponding sets on the CPUs and GPUs.

5.3 Correctness of the Dual Approximation

It remains to be proved that the existence of a solution of makespan at most λ implies the existence of a solution with the seven-shelf structure. To do so, we first expose and prove two technical lemmas before stating the existence theorem (Theorem 10).

Lemma 8. *Suppose there exists a solution σ_{ref} of makespan at most λ. The assignment of tasks on the GPUs in σ_{ref} is compatible with the seven-shelf structure.*

Proof. All tasks assigned to the GPUs by σ_{ref} are sequential. Hence we can assign these tasks in two distinct sets: tasks with a processing time strictly greater than $\frac{\lambda}{2}$ and tasks with a processing time lower than $\frac{\lambda}{2}$. These two sets exactly match the sets (5) and (6) of the structure we seek. □

Lemma 8 allows us to only consider tasks assigned to the CPUs in the proof of the existence of the sought schedule.

Lemma 9. *If there exists a solution σ_{ref} of makespan at most λ, then there exists a solution σ_{struct} with the seven-shelf structure whose sub-solution $\overline{\sigma}_{struct}$ (considering only tasks assigned to CPUs) uses at most m CPUs with a lower CPU load than the CPU load of σ_{ref}.*

Proof. First, let us prove that the big tasks of σ_{ref}, namely tasks with a processing time greater than $\frac{\lambda}{2}$, fit in sets (1), (2) and (3) without using more than m CPUs and without increasing the CPU load:

- The tasks assigned to set (1) are sequential tasks of length greater than $\frac{\lambda}{2}$: their work is minimal. Since their processing time is at most $\frac{3\lambda}{4}$, only one of the tasks assigned to set (1) can fit on one CPU in σ_{ref}, whereas in σ_{struct}, these tasks are stacked by pair, one in shelf $(1)L$, the other in shelf $(1)R$. As a result, the tasks in set (1) in σ_{struct} use less processors than they would in σ_{ref}.
- The tasks assigned to sets (2) and (3) are using their canonical number of CPUs for a time limit at least λ, hence they generate a lower or equal work than they would in σ_{ref}. As these tasks use their canonical number of processors for a time limit greater than λ, they use less processors than they would in σ_{ref}. Observe that the tasks assigned to set (2) use less processors than they do in σ_{ref} thanks to the relaxed time limit.

We now have to consider the tasks of σ_{ref} assigned to CPUs with a processing time lower than $\frac{\lambda}{2}$. All the tasks with a sequential time lower than $\frac{\lambda}{2}$ are assigned to set (0). The remaining tasks are the tasks that have been assigned to more than one CPU in σ_{ref}, with a processing time lower than $\frac{\lambda}{2}$. The monotony assumption ensures that they can fit in any set among sets (1), (2), (3) and (4) without increasing the computational load. In order to prove that there exists such an assignment of these remaining tasks, we consider the integer linear program introduced in Section 4.2 that we relax by removing Constraint (C_3). This allows set (4) to occupy as many CPUs as needed. The tasks already assigned to the GPUs as in σ_{ref} thanks to Lemma 8 have their corresponding variables in the integer linear program set according to their assignment. The same is done for the variables of the integer linear program corresponding to the tasks already assigned to sets (1), (2), (3) and (0) above in the proof. We let the integer linear program choose the remaining assignments. By doing so, since Constraint (C_3) was removed, set (4) could use too many CPUs. It remains to prove that the assignment returned by the revised integer linear program does not need to use more than m CPUs. Two cases are to be distinguished: either every CPU is busy or some CPUs remain idle after assigning tasks to sets (1), (2) and (3). The first case's proof is straightforward while the latter is done in three steps.

Let us first consider the case where every CPU is busy. By construction, at most one processor—in set (1)—is loaded less than λ but more than $\frac{3\lambda}{4}$. As all the tasks assigned to set (4) have a processing time larger than $\frac{\lambda}{4}$, we cannot use more than m CPUs without contradicting the facts that the integer linear program is minimizing the CPU load and that σ_{ref} exists.

Let us consider now the case where some CPUs remain idle. We denote by $\mu_\emptyset$ their number.

(i) We begin by proving that at most one task in set (4) does not fit. As $\mu_\emptyset > 0$, every task of set (4) has a work greater than $\mu_\emptyset\lambda$, otherwise it would have been assigned to set (3) by the integer linear program. The maximum amount of work by which a task of set (4) could be overreaching is bounded by the gap left between $m\lambda$ and the work of the tasks filling sets (1), (2) and (3). Because of the structure in five sets on the CPUs, such a gap is at most $\frac{3\lambda}{4}\mu_\emptyset + \frac{\lambda}{4}\mathbf{1}_{(1)\text{odd}}$, which is strictly smaller than the work of any task assigned to set (4). The existence of a task in set (4) executed only on processors not meant to do so by the five set structure would contradict the fact that sets (1), (2) and (3) were filled by the integer linear program while minimizing the CPU load. Therefore there is at most a fraction of a task assigned to (4) whose execution requires processors that do not exist.

In the next two steps, we consider an arbitrary assignment for the tasks assigned to set (4) and suppose exactly one task does not fit. We focus on this particular task, denoted by T_Δ. Proving its existence contradicts the fact that the work is minimized by the integer linear program. We denote by $(3)\Delta$ the subset of (3) that shares processors with the task T_Δ.

(ii) We show now that the inequality $\mu_{(3)\Delta} > 2\,\mu_\emptyset$ holds under the assumption that T_Δ exists. The integer linear program chose to assign task T_Δ to set (4). As set (4) is the one creating the most work amongst sets (1), (2), (3) and (4), this choice had to be made because otherwise constraints would have been violated. We know for sure that $\mu_{(3)\Delta} > 0$, otherwise this would contradict Step 1. Moreover, as T_Δ was not assigned on $\mu_\emptyset$ processors in set (2), its work is greater than $\frac{3\lambda}{2}\mu_\emptyset$. Such a case is only possible if we have enough space next to set (3), which is equivalent to the following inequality:

$$\frac{3\lambda}{4}\mu_{(3)\Delta} + \frac{3\lambda}{2}\mu_\emptyset < (\mu_{(3)\Delta} + \mu_\emptyset)\lambda \tag{4}$$

This inequality reduces to the one we are interested in, i.e. $\mu_{(3)\Delta} > 2\,\mu_\emptyset$.

(iii) To finish the proof, let us shows that the previous point leads to a contradiction, hence $\overline{\sigma}_{\text{struct}}$ fits into m CPUs. Inequality (4) can be rewritten in the following form:

$$\frac{3\lambda}{4}\mu_{(3)\Delta} + \frac{\lambda}{2}\left(\mu_\emptyset + \mu_{(3)\Delta}\right) > \lambda\left(\mu_\emptyset + \mu_{(3)\Delta}\right)$$

The left part of the sum is a lower-bound of the work of set $(3)\Delta$. The monotony ensures that the work of T_Δ is greater than $\frac{\lambda}{2}\left[\gamma(T_\Delta, \frac{\lambda}{2}) - 1\right]$, and we know that the number of processors needed by task T_Δ is at least $\mu_{(3)\Delta} + \mu_\emptyset + 1$. Hence the work of T_Δ is greater than $\frac{\lambda}{2}\left(\mu_\emptyset + \mu_{(3)\Delta}\right)$. This brings the contradiction as this would mean that the total work is greater than $m\lambda$. □

Theorem 10. *If there exists a solution of makespan at most λ, then there exists a solution with the seven-shelf structure we are looking for with a makespan at most $\frac{3\lambda}{2}$ and a lower CPU load.*

Proof. The theorem is a direct consequence of Lemmas 8 and 9. □

The reject condition of the dual approximation is the contrapositive of Theorem 10.

5.4 Building the Schedule

We have described the core step of the dual-approximation algorithm, with a fixed guess. A binary search is used with successive guesses to approach the optimal makespan. Using an initial lower (resp. upper) bound $B_{\min}$ (resp. $B_{\max}$) of the optimal makespan, the number of iterations of this binary search is bounded by $\log\left(B_{\max} - B_{\min}\right)$.

Each iteration of the dual approximation algorithm consists in solving an ILP. The complexity of this step is not bounded by a polynomial function. However, solving the ILP with a standard linear solver (e.g., CPLEX or Gurobi) shows a very good efficiency as described in Section 7.4. Indeed, the filtering functions allow to reduce the search space size of the ILP, a task can be assigned to at most four sets instead of seven. Moreover, as the number of tasks increases, every task's relative execution time shrinks. Thus, for large instances, most of the tasks will be assigned to sets (0) and (6) only.

This has to be compared to an algorithm using dynamic programming to solve the allocation problem. Even if this paradigm would lead to a proved polynomial complexity, the size of the search space makes it intractable to explore. Adapting the technique proposed in [7] would result in an algorithm whose complexity is $\mathcal{O}(n^2m^4k^2)$ in our case.

6 Algorithm APPROX-2

As stated in Section 5.4, APPROX-3/2 is not proved to be polynomial. To get more insight on dual approximation algorithms, we devise APPROX-2 that is a simpler polynomial-time algorithm providing an approximation ratio of $2+\epsilon$. APPROX-2 uses the same principle as APPROX-3/2: it partitions the computing resources, allocates the tasks to a partition, and then schedules them within their partition.

6.1 Sketch

We consider a guess λ of the optimal makespan value. The scheduling problem on the CPUs is simplified by forcing the number of CPUs a task can use to its canonical number of CPUs, with respect to λ. The algorithm then works as follows:

1) Allocate the tasks that fits in λ only on one type of architecture.
2) Sort the tasks by decreasing work ratio $\frac{w_{j,\gamma(j,\lambda)}}{p_j}$. The approximation ratio derives from this sort as explained in Lemma 12.
3) Allocate the tasks on the GPUs until each GPU has a load more than λ.
4) Schedule the remaining rigid tasks on the CPUs with a 2-approximation algorithm. List algorithms or strip-packing algorithms are viable options.

If all the tasks do not fit with a makespan at most 2λ, then the algorithm rejects this guess. Otherwise, we have founded a valid schedule.

6.2 Analysis

We now analyze some properties of APPROX-2. First we study the approximation ratio, then the complexity.

Lemma 11. *The makespan of the tasks allocated to the GPUs is smaller than* 2λ.

Proof. By construction, all the tasks considered for an allocation on a GPU are smaller than λ. As the algorithm stops loading a GPU when its load exceeds λ, the makespan bound is straightforward. □

Lemma 12. *If there exists a solution of makespan at most* λ*, then the makespan of the tasks allocated to the CPUs is smaller than* 2λ.

Proof. Using the canonical number of CPUs with respect to λ ensures that every task allocated to some CPUs generates a minimal amount of work (as stated in Section 4). In particular, this amount of work is at most the amount of work generated in the optimal schedule. The GPUs have been—by construction—allocated a greater share of work than the optimal solution. Moreover, the tasks are sorted by decreasing work ratio $\frac{w_{j,\gamma(j,\lambda)}}{p_j}$. This specific order implies that the work remaining on the CPUs is smaller than $m\lambda$ if there exists a solution of makespan at most λ. The makespan bound follows from the fact we schedule the remaining tasks with a list algorithm [23] or a strip-packing algorithm [24]. Using the strip-packing algorithm would provide a contiguous solution. □

The two previous lemmas prove APPROX-2 provides a solution whose makespan is at most 2λ.

APPROX-2 is an algorithm of low polynomial complexity. It indeed only relies on sorting the tasks, and on keeping track of the computing resources using priority queues. Moreover, each task is considered at most once when scheduled. Hence, and more precisely, the complexity of the algorithm belongs to $\mathcal{O}\left(n\left[\log(n) + \log(k) + m\log(m)\right]\right)$.

7 Experimental Evaluation

After providing the theoretical foundation for solving the given scheduling problem, we will now examine the applicability of our approach. For that reason, we will compare the makespans produced by APPROX-3/2 and APPROX-2 as well as the run-time required to compute the solutions. In our evaluation, we will also consider the scheduling solutions from heuristics, which are modifications of the classical Heterogeneous Earliest Finish Time algorithm (HEFT) [25].

We start by explaining the problem instances used in our analysis. After that, we give a description of the heuristics used to evaluate our proposed algorithms. We present implementation details for all algorithms, and last, we show and discuss the experimental results.

7.1 Problem Instances

Finding the right problem instances for evaluating scheduling algorithms is generally a hard problem. Real world instances are often considered to be essential when choosing test instances. However, testing an algorithm on a small set of real world instances will often not support the claim that an algorithm is generally well applicable. Another problem is that influencing factors, such as the number of tasks or the size of tasks, are fixed in real world instance. Therefore, we generated scheduling instances that allows us to study the general applicability of our algorithms and to investigate the influence of experimental factors.

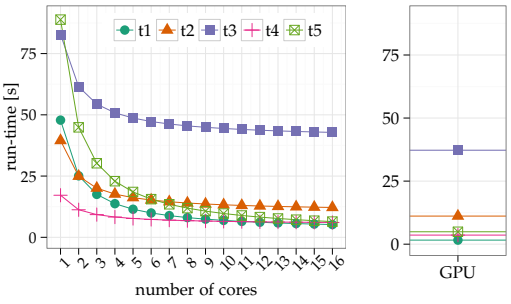


Fig. 4: Example of a problem instance: Each of the five tasks exhibits a different parallel scalability on the multi-core machine (left) and has a different run-time on the GPU (right).

To generate the instances, we first select the number of tasks (n), the number of CPUs (m), and the number of GPUs (k). Then, the instance generator decides on the run-time of all tasks as follows:

1) It randomly picks the sequential time on the CPU of one task.
2) It defines the speedup of one task on the CPU, by picking the sequential fraction of this task. The time for the sequential fraction defines the lower bound of a task's run-time, as only the run-time of the parallel fraction of a task can be reduced by adding more CPUs (Amdahl's law).
3) It picks a speedup factor that defines how much faster a particular task can run on a GPU compared to being executed on all m CPUs.
4) The generation process is repeated for all the n tasks.

We now provide a more detailed description of each step of the instance generation process.

Step 1: The sequential run-time $\overline{p_{j,1}}$ of task T_j is picked from a uniform distribution in the interval $[\overline{p}^{min}, \overline{p}^{max}]$.

Step 2: Next, the speedup model of each task is determined. To this end, we apply Amdahl's law to model the speedup of a moldable task. The law states that the parallel execution time is bounded by the sequential fraction of a program. Therefore, we select the sequential fraction β_j of each task, where β_j follows uniform distribution in $[\beta^{min}, \beta^{max}]$. The knowledge of the sequential run-time $\overline{p_{j,1}}$ and the sequential fraction β_j allows us to model and compute the parallel execution time on l CPUs of task T_j as: $\overline{p_{j,l}} = \beta_j \overline{p_{j,1}} + (1-\beta_j)\frac{\overline{p_{j,1}}}{l}$ (for all l in $2, \ldots, m$).

Step 3: We assume that GPUs can accelerate the execution of a task, i.e., a task will—most likely—be faster on a GPU than on all CPUs of the multi-core system. Thus, we model the time for task T_j on the GPU relative to the parallel time and all m CPUs $\overline{p_{j,m}}$. To obtain the time on the GPU ($\underline{p_j}$), we pick a speedup factor g and set $\underline{p_j} = g\,\overline{p_{j,m}}$. The value of g follows a normal distribution with mean $mean_g$ and standard deviation sd_g. In this way, we also allow tasks that are slower on the GPU than on the CPUs. We also set some bound on the maximum speedup or maximum slowdown for each task on the GPU. For that reason, we introduce

TABLE 1: Parameter settings used to generate scheduling instances.

description	variable	values
number of tasks	n	$\{10, 50, 100, 1000\}$
number of CPUs	m	$\{4, 16, 64, 128, 256, 384, 512\}$
number of GPUs	k	$\{1, 2, 4, 8, 16, 32\}$
minimum sequential run-time of tasks	$\overline{p}^{min}$	10
maximum sequential run-time of tasks	$\overline{p}^{max}$	100
minimum sequential fraction of a task	β^{min}	0
maximum sequential fraction of a task	β^{max}	0.5
mean speedup factor for tasks on GPUs	$mean_g$	0.2
standard deviation of speedup factor for tasks on GPUs	sd_g	0.5
minimum speedup factor for tasks on GPUs	min_g	0.1 (10× speedup on the GPU)
maximum speedup factor for tasks on GPUs	max_g	1.5 (50% slowdown on the GPU)

the variables min_g and max_g, which denote the minimum (maximum speedup) and maximum value of g (maximum slowdown) of a task on the GPU.

We generated 10 samples for each parameter combination of n, m, and k with the values shown in Table 1. Figure 4 shows the characteristics of one of these instances, which contains only five tasks for the sake of readability. Each task has a different scalability behavior (caused by a different sequential fraction), and all tasks in this example have a shorter run-time when executed on a GPU.

7.2 HEFT-like Heuristics

In the present paper, we have proposed two algorithms that provide approximate solutions to the scheduling problem stated in Section 2. In order to compare our approaches with practically relevant algorithms, we also include HEFT-like algorithms in our evaluation. We call them HEFT-like algorithms as they work similar to the original HEFT algorithm [25], but target a slightly different scheduling problem. HEFT-like algorithms are used in practice, for instance, the run-time system StarPU uses a very similar algorithm (called MCT for minimum completion time) to schedule tasks on CPUs and GPUs [26].

Now, we describe our variants and implementation of the HEFT-like algorithms for scheduling moldable tasks on a multi-core system with multiple GPUs. Our implementation resembles the general idea of the original algorithm proposed by Topcuoglu *et al.* [25], except that—since we have no precedence constraints—we change the priority function used to sort the tasks. Similar to HEFT, our algorithm places the highest priority task on the CPUs or one of the GPUs that minimize the *earliest finish time* (**EFT**). We expect that HEFT-like algorithms are sensitive to the type of prioritization function. To avoid a possible bias towards one prioritization function, we consider three different strategies, which are:

1) **LPT**: This strategy sorts the tasks in decreasing order of their execution times (**L**ongest **P**rocessing **T**ime).
2) **SPT**: This strategy sorts the tasks in increasing order of their execution times (**S**hortest **P**rocessing **T**ime).
3) **RATIO**: This strategy sorts the tasks in decreasing order of the following ratio: execution time on the CPUs over the run-time on a GPU, i.e., $\frac{\overline{p_{j,l}}}{\underline{p_j}}$, where l is either 1 for sequential tasks or m for parallel tasks.

For the strategies **LPT** and **SPT**, the execution time of task T_j is computed as $\max(\overline{p_{j,l}}, \underline{p_j})$, for $l \in \{1, m\}$.

The question is now: how many CPUs should be assigned to each task when computing the schedule? We use two simple schemes: the strategy **PAR** allots all CPUs to a task ($l = m$), whereas the strategy **SEQ** allots only one CPU to a task ($l = 1$). Considering the monotonic assumption for the run-time of moldable tasks, the strategy **PAR** is a greedy way of minimizing the execution time of a task, as the run-time function is non-increasing in the number of CPUs. The second strategy (**SEQ**) favors task parallelism and minimizes every task's work. It is certainly possible that these HEFT-like implementations can be improved, but such considerations are outside the scope of this paper. In total, we have created six different HEFT-like heuristics, called Heuristic 1–6, which are listed in Table 2.

7.3 Implementation Details

We implemented APPROX-3/2 using the programming languages Julia and Python. Logically, the algorithm APPROX-3/2 consists of two steps: (i) find the best λ by applying the bisection method to partition the tasks into sets, and (ii) build the schedule from the computed partitioning. The first step has been implemented in Julia, as it features the domain-specific modeling language JuMP, which provides an abstraction layer above different ILP solvers, such as Gurobi, CPLEX, or GLPK. Hence, we only need to write our ILP problem using the JuMP API[1] and can use different solvers to find a solution. The second step, the building of the schedule, has been implemented in Python.

As stated above, the lower and upper bound of the scheduling problem are adjusted during the iterative search for the best λ using the bisection method. The bisection method stops when the ratio between upper and lower bound is below a certain threshold (the *cutoff* value). For both algorithms, APPROX-3/2 and APPROX-2, we have used a *cutoff* value of 1.01 (~1%) in all experiments.

The algorithm APPROX-2 has been entirely implemented in Julia. Here, the algorithm also intents to find the best λ, but since it maps tasks to devices (CPUs or GPUs) greedily, the actual schedule is built on the fly.

The HEFT-like heuristics has been written in Python. Similarly to the implementation of APPROX-2, the actual schedule can be built directly, as there is no previous partitioning step.

We have used the following software versions: Julia 0.3.11, Python 2.7.10, JuMP 0.10.1, Gurobi binding for JuMP

1. Application Programming Interface

TABLE 2: HEFT-like heuristics used for comparison.

name	mapping	sorting	parallel tasks on CPUs
Heuristic 1	**EFT**	**LPT**	no (**SEQ**)
Heuristic 2	**EFT**	**SPT**	no (**SEQ**)
Heuristic 3	**EFT**	**RATIO**	no (**SEQ**)
Heuristic 4	**EFT**	**LPT**	yes (**PAR**)
Heuristic 5	**EFT**	**SPT**	yes (**PAR**)
Heuristic 6	**EFT**	**RATIO**	yes (**PAR**)

0.1.29, CPLEX binding for JuMP 0.0.13, Gurobi Optimizer for OS X 6.0.0, CPLEX Optimization Studio for OS X 12.6.1.0, and Mac OS X 10.10.5. For the experiments shown in the present paper, we have used the Gurobi Optimizer to solve the integer linear programs.

7.4 Experimental Results

First, we evaluate the produced makespan of each scheduling instance, which is the most important property of the scheduling algorithms described in the present paper.

Figure 5 compares the makespans of the schedules generated by APPROX-3/2, APPROX-2, and the six different HEFT-like heuristics. For a better comparability, we normalize the makespan for each scheduling instance by the makespan obtained from APPROX-3/2. Thus, the algorithm APPROX-3/2 will always have a relative makespan of 1.0 (red horizontal line). The relative makespan of the other algorithms, APPROX-2 and the six heuristics, will most likely differ from 1.0. If the computed relative makespan is smaller than 1.0, the produced schedule of one of the competing scheduling algorithms is shorter than the one of APPROX-3/2. Similarly, if the relative makespan is larger than 1.0 then APPROX-3/2 was able to find a shorter schedule. We can observe that the HEFT-like heuristics produce competitive results when the number CPUs and GPUs is small (cf. Figure 5a, case m=4 and k=4). If the number of tasks, CPUs, and GPUs is increased, the results in Figure 5b provide evidence that APPROX-3/2 produces significantly shorter schedules than its competitors. The results of the heuristics 4–6 using the **PAR** strategy (cf. Table 2) have been omitted, as they have been found to be largely inferior compared to the **SEQ** versions. Among the HEFT-like algorithms, the heuristics that use an **LPT** strategy produced the shortest schedules. Interestingly, the solutions obtained from the approximation algorithm APPROX-2 are most often not better than the much simpler HEFT-like heuristics, indicating that an approximation factor of 2 is simply too large for a practical applicability.

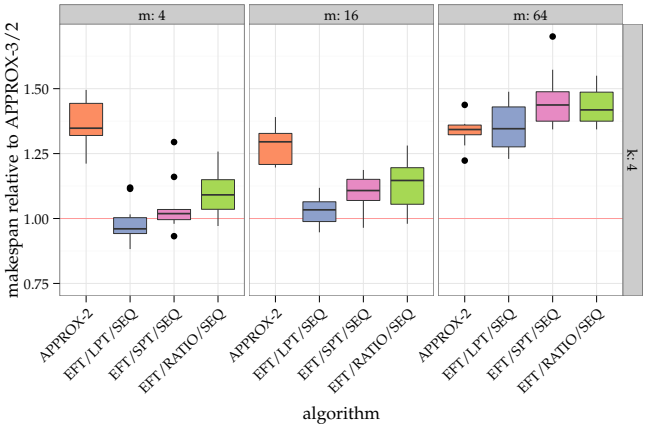
The solution quality (the makespan) is only one metric to assess scheduling algorithms. The algorithm APPROX-3/2 requires to solve an ILP for each value of λ. Therefore, an analysis of the run-time of the algorithms is of equal importance. The run-times measured do not include the time to read and parse the input files and the time to write the final schedules to disk. In addition, the results are only meant to show general trends of the run-time requirements of the different algorithms, as the algorithms have been implemented using different programming languages.

Figure 6 compares mean run-time of the different scheduling algorithms for various values of n, m, and k. In particular, the run-time of the algorithms APPROX-3/2 and APPROX-2 includes all iterations that were required to obtain the final value of λ. The experiments were conducted on a quad-core Intel i7-3615QM with a clock speed of 2.3 GHz. Since the run-times of the various HEFT-like heuristics were very similar, as only the prioritization function is changed, we only show the time for Heuristic 1 (**EFT**, **LPT**, **SEQ**). As expected, the run-time of Heuristic 1 grows linearly with the number of tasks, CPUs, or GPUs, and has been found to be the shortest among all scheduling algorithms tested. The run-time of the APPROX-2 algorithm is significantly longer than the run-time of the heuristics due to the iterative nature of the algorithm. It is also not surprising that the APPROX-3/2 algorithm has the longest mean run-time for all considered cases. We can also see that the run-time of APPROX-3/2 grows proportionally faster than the run-times of the other algorithms, which is a consequence of solving an ILP in each iteration. Nevertheless, APPROX-3/2 computes the solutions relatively quickly, as it takes about five seconds to compute the schedule for the largest instance in our test set ($n = 1000$, $m = 512$, $k = 16$, cf. Figure 6b). It goes without saying that this run-time is too large to schedule many small-grained tasks onto CPUs or GPUs, but it is a very promising alternative algorithm for scheduling longer running tasks (or even different parallel application).

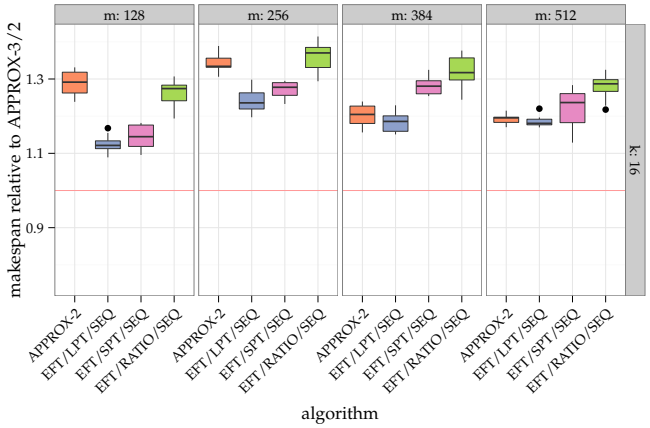
We have also studied the effectiveness of the filtering step that we introduced in Section 4.2.2, and Figure 7 shows these results. For different numbers of tasks (n), the graphs show the distributions of the mean number of possible partitions per task after the filtering has been applied. We recall that the internal ILP find a partitioning of all tasks into seven disjoint sets. That means, each of the n tasks can only be in one of the seven partitions. Thus, the ILP initially allocates a table of $n \times 7$ binary variables. In the filtering step, some of the variables are set to 0, i.e., the number of partitions that a task can be assigned will be reduced. Ideally, the number of available partitions per task reduces from seven to one when the filtering is applied, and the solution can be obtained immediately. Figure 7 shows the number of available partitions for increasing values of n. The "mean number of possible partitions" is computed over all the tasks of one iteration. For example, for one problem instance, the mean number of possible partitions (over all the tasks) is 2 in iteration 1 and 2.5 in iteration 2. In this case, the distributions shown in Figure 7 will contain the values 2 and 2.5. We observe that for the majority of the tasks, except in the case of $n = 10$, only two partitions are available (on average) after applying the filtering step, which supports our claim that the filtering step is effective.

Figure 8 complements the previous results by an analysis of the number of iterations required, such that the bisection method converges. In the experiments conducted as part of the present paper, the required number of iterations was ranging from 10 to 17.

In summary, we can state that APPROX-3/2 is able to find significantly shorter schedules than the APPROX-2 algorithm or the HEFT-like heuristics. On the contrary, APPROX-3/2 needs more time to compute the solutions. However, even for larger instances ($n = 1000$, $m = 512$, $k = 16$) APPROX-3/2 can be used to obtain the solution in a couple of seconds. If the average task duration lies in the range of seconds, applying APPROX-3/2 will definitely provide an advantage

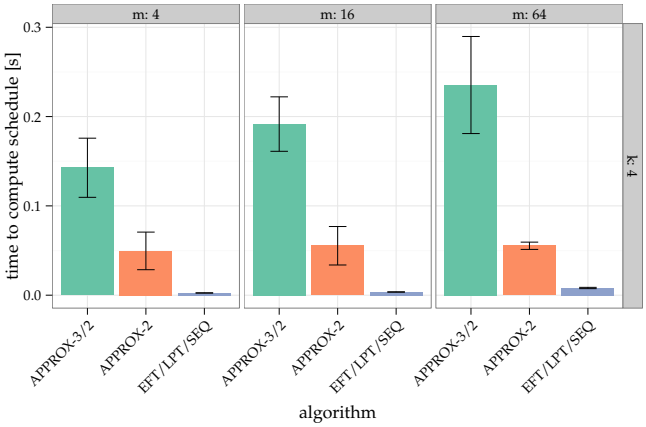


(a) $n = 100$

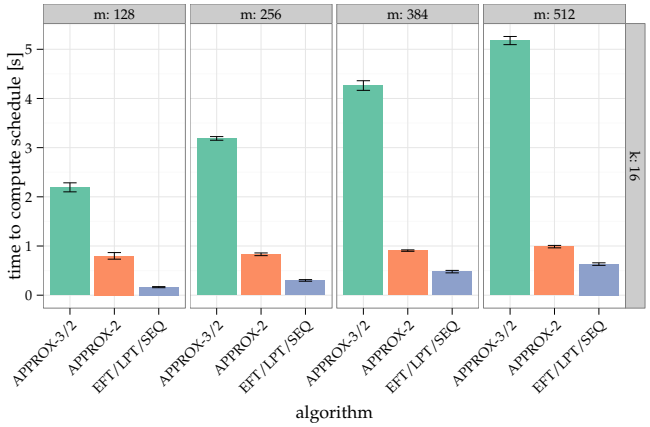


(b) $n = 1000$

Fig. 5: Comparison of the relative of makespan obtained with APPROX-2 and the HEFT-like algorithms with respect to the makespan produced by APPROX-3/2 (n tasks, m CPUs, k GPUs) .



(a) $n = 100$



(b) $n = 1000$

Fig. 6: Comparison of the mean run-time (incl. 95% confidence interval) of each scheduling algorithms to compute the solutions (n tasks, m CPUs, k GPUs).

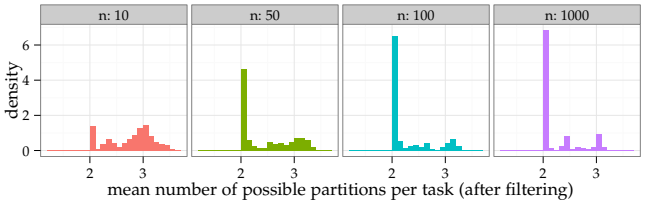


Fig. 7: Distribution of the (mean) number of possible partitions per task after the filtering has been applied for APPROX-3/2. The graphs show distributions for all values of m and k presented in Table 1.

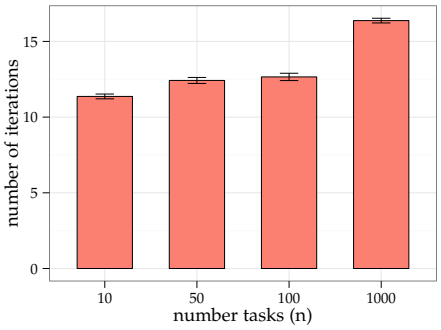


Fig. 8: Distribution of iterations (of the bisection method) performed by APPROX-3/2 to converge to a solution.

compared to the other scheduling algorithms.

8 Conclusions

In this paper, we presented a new scheduling algorithm using a generic methodology (in the opposite of specific ad hoc algorithms) for hybrid architectures (multi-core machine with GPUs) with the moldable task model on CPUs. We proposed an algorithm with a constant approximation ratio of $\frac{3}{2} + \epsilon$. The main idea of the approach is to determine an adequate partition of the set of tasks on the CPUs and the GPUs using a

dual approximation scheme and integer linear programming. Still, we do not provide any proof of the complexity of this ILP-based approach. Instead, we compared this approach with another algorithm with a proved polynomial-time complexity, at the cost of degrading the approximation ratio to $2 + \epsilon$. An experimental analysis on realistic instances has been provided to assess the computational efficiency and the schedule quality of the proposed method when compared to classical HEFT algorithms. The main conclusion is that the ILP-based algorithm is stable because of its approximation guaranty, with a reasonable running time. Moreover this proposed algorithm outperforms all HEFT algorithms when dealing with instances of large size, which is often the case on computing platforms.

Acknowledgments

This work has been partially supported by a DGA-MRIS scholarship and the French program GDR-RO.

References

[1] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100X GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU," in *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. ACM, 2010, pp. 451–460.

[2] Y. Abe, H. Sasaki, M. Peres, K. Inoue, K. Murakami, S. Kato *et al.*, "Power and performance analysis of GPU-accelerated systems," in *Proceedings of the 2012 USENIX conference on Power-Aware Computing and Systems (HotPower)*, vol. 12, 2012, pp. 10–10.

[3] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov, "QR factorization on a multicore node enhanced with multiple GPU accelerators," in *Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS'11)*. IEEE Computer Society, 2011, pp. 932–943.

[4] F. Song, S. Tomov, and J. Dongarra, "Enabling and scaling matrix computations on heterogeneous multi-core and multi-GPU systems," in *Proceedings of the 26th ACM International Conference on Supercomputing (ICS'12)*. ACM, 2012, pp. 365–376.

[5] A. Boukerche, J. M. Correa, A. Melo, and R. P. Jacobi, "A hardware accelerator for the fast retrieval of DIALIGN biological sequence alignments in linear space," *IEEE Transactions on Computers*, vol. 59, no. 6, pp. 808–821, 2010.

[6] J. C. Phillips, J. E. Stone, and K. Schulten, "Adapting a message-driven parallel application to GPU-accelerated clusters," in *Proceedings of the Supercomputing Conference (SC'08)*. IEEE Press, 2008, pp. 8:1–8:9.

[7] R. Bleuse, S. Kedad-Sidhoum, F. Monna, G. Mounié, and D. Trystram, "Scheduling independent tasks on multi-cores with GPU accelerators," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 6, pp. 1625—-1638, 2015.

[8] F. Monna, "Scheduling for new computing platforms with GPUs," Ph.D. dissertation, Université Pierre et Marie Curie - Paris VI, Nov. 2014.

[9] R. P. Brent, "The parallel evaluation of general arithmetic expressions," *Journal of the ACM (JACM)*, vol. 21, no. 2, pp. 201–206, 1974.

[10] G. Mounié, C. Rapine, and D. Trystram, "A 3/2-approximation algorithm for scheduling independent monotonic malleable tasks," *SIAM Journal on Computing*, vol. 37, no. 2, pp. 401–412, 2007.

[11] J. K. Lenstra, D. B. Shmoys, and E. Tardos, "Approximation algorithms for scheduling unrelated parallel machines," *Mathematical Programming*, vol. 46, no. 1, pp. 259–271, 1990.

[12] D. B. Shmoys and E. Tardos, "An approximation algorithm for the generalized assignment problem," *Mathematical Programming*, vol. 62, no. 1, pp. 461–474, 1993.

[13] E. V. Shchepin and N. Vakhania, "An optimal rounding gives a better approximation for scheduling unrelated machines," *Operations Research Letters*, vol. 33, no. 2, pp. 127–133, 2004.

[14] V. Bonifaci and A. Wiese, "Scheduling unrelated machines of few different types," *CoRR*, vol. abs/1205.0974, 2012.

[15] L. Eyraud, "Théorie et pratique de l'ordonnancement d'applications sur les systèmes distribués," Ph.D. dissertation, Institut National Polytechnique de Grenoble, 2006.

[16] K. Jansen and L. Porkolab, "Linear-time approximation schemes for scheduling malleable parallel tasks," in *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 99)*, 1999, pp. 490–498.

[17] J. Turek, J. Wolf, and P. Yu, "Approximate algorithms scheduling parallelizable tasks," in *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '92)*, 1992, pp. 323–332.

[18] E. G. Coffman, M. R. Garey, D. S. Johnson, and R. E. Tarjan, "Performance bounds for level-oriented two-dimensional packing algorithms," *SIAM Journal on Computing*, vol. 9, no. 4, pp. 808–826, 1980.

[19] M. Bougeret, P.-F. Dutot, K. Jansen, C. Otte, and D. Trystram, "A fast 5/2-approximation algorithm for hierarchical scheduling," in *Proceedings of the Euro-Par 2010*, ser. LNCS. Springer, 2010, vol. 6271, pp. 157–167.

[20] W. Ludwig and P. Tiwari, "Scheduling malleable and nonmalleable parallel tasks," in *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'94)*. Society for Industrial and Applied Mathematics, 1994, pp. 1670–176.

[21] L. Fan, F. Zhang, G. Wang, and Z. Liu, "An effective approximation algorithm for the malleable parallel task scheduling problem," *Journal of Parallel and Distributed Computing*, vol. 72, no. 5, pp. 693–704, 2012. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2012.01.011

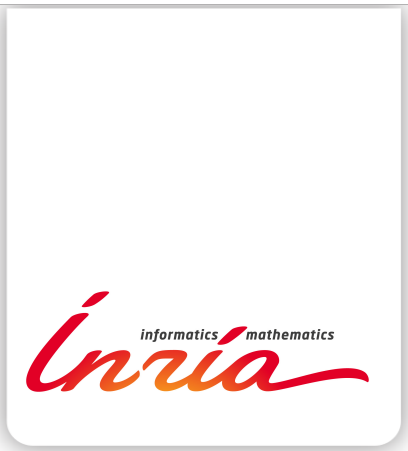
[22] D. S. Hochbaum and D. B. Shmoys, "Using dual approximation algorithms for scheduling problems: Theoretical and practical results," *Journal of the ACM (JACM)*, vol. 34, no. 1, pp. 144–162, 1987.

[23] M. R. Garey and R. L. Grahams, "Bounds for multiprocessor scheduling with resource constraints," *SIAM Journal on Computing*, vol. 4, no. 2, pp. 187–200, 1975.

[24] A. Steinberg, "A strip-packing algorithm with absolute performance bound 2," *SIAM Journal on Computing*, vol. 26, no. 2, pp. 401–409, 1997.

[25] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, 2002.

[26] 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.



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