Scheduling Independent Moldable Tasks on Multi-Cores with GPUs

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

Abstract—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 $\frac{3}{2}+\epsilon$. Since the time complexity of the ILP-based algorithm could be non-polynomial, we also present a polynomial-time algorithm with an approximation ratio of $2+\epsilon$. We complement the theoretical analysis of our two novel algorithms with a simulation study. In these simulations, we compare our algorithms to a modified version of the classical HEFT algorithm, which we adapted to handle moldable tasks. The simulation results show that our algorithm with the $\left(\frac{3}{2}+\epsilon\right)$-approximation ratio produces significantly shorter schedules than the modified HEFT for most of the instances. In addition, our results provide evidence that our ILP-based algorithm can solve larger problem instances in a reasonable amount of time.

Index Terms—scheduling, heterogeneous computing, moldable tasks, dual approximation scheme, integer linear programming

1 Introduction

Today's available parallel computing systems often consist of compute nodes that contain multi-core CPUs and additional hardware accelerators [1]. Such accelerators (General Purpose Graphic Processor Units, denoted by GPUs 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. These hybrid systems are becoming more popular, and it is foreseeable that the trend of using such hybrid systems will grow, especially since 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 works were tailor-made for the targeted applications. Therefore, these scheduling algorithms lack a high-level view of all tasks, to provide an efficient and transparent solution for any type of parallel application. The challenge is to develop an effective and automatic resource manager for executing generic applications on parallel and 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, which provides solutions for the problem of scheduling independent, sequential tasks on CPUs or GPUs with the makespan objective [7], [8]. However, the algorithm has two main drawbacks. First, although 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 can potentially benefit from internal (data-)parallelism on CPUs, while our previous algorithm works 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. The run-time of a moldable task depends on the number of processors allotted to it. This model allows exploiting the two types of parallelism offered by hybrid parallel computing platforms: the inherent parallelism induced by the GPU architecture, and the parallelization of tasks on several CPUs. The goal of this work is to propose a generic method to leverage these two different kinds 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 proposed algorithm has an approximation ratio of at most $\frac{3}{2}+\epsilon$.
- We show through a sequence of simulations that

- *R. Bleuse, G. Mounié and D. Trystram are with Univ. Grenoble Alpes – LIG, France*
  E-mail: {raphael.bleuse, gregory.mounie, denis.trystram}@imag.fr
- *S. Kedad-Sidhoum and F. Monna are with Sorbonne Universités, UPMC Univ. Paris 06, UMR 7606, LIP6, France*
- *S. Hunold is with TU Wien, Faculty of Informatics, Institute of Information Systems, Favoritenstraße 16/184-5, 1040 Vienna, Austria.*
  E-mail: hunold@par.tuwien.ac.at
- *D. Trystram is senior member of Institut Universitaire de France*

although our algorithm is based on integer linear programming—with a theoretical worst-case exponential-time complexity—it is still practically relevant, as it provides competitive schedules, and has a relatively short run-time.

- In addition, we present a fully polynomial-time algorithm for the same scheduling problem. We prove that this algorithm has an approximation ratio of at most $2 + \epsilon$.

The ϵ-term in the approximation ratios comes from the binary search of the dual approximation and the integer linear program solver's accuracy. One can remove the ϵ-term from the ratios under the assumptions that every task has an integer processing time and that solvers use rational numbers.

The paper is organized as follows: in Section 2, we define the scheduling problem targeted in this work. We examine related work 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 its theoretical analysis in Section 5. In Section 6, we devise a fully polynomial approximation algorithm, which is introduced to compare results with schedules obtained from our ILP-based algorithm. In Section 7, we assess the solution quality (makespan) of various scheduling algorithms for a variety of test instances through simulation. We conclude the paper in Section 8.

2 Problem Definition

We consider a parallel multi-core 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 to be *moldable* when assigned to the CPUs and to be sequential when assigned to a GPU. The processing time of any task T_j is represented by a function $\overline{p_j} : l \mapsto \overline{p_{j,l}}$, determining the processing time when executed on l CPUs, and by p_j denoting the processing time when executed on a GPU. We assume that these processing times are known in advance.

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 a task starts its execution simultaneously on all the allocated resources and occupies them without interruption until its completion time (i.e., no preemption).

We define the CPU work function w_j of a task T_j as $w_j : l \mapsto w_{j,l} = l \times \overline{p_{j,l}}$ for $l \leqslant m$. It corresponds to the computational area of T_j on the CPUs in the Gantt chart representation of a schedule. We assume that the assignment of CPUs to tasks has a monotonic behavior: assigning more CPUs to a task decreases its processing time, but comes at the cost of an increased work. The increased work represents the parallelization cost (internal communications and synchronizations). Such a hypothesis is equivalent to the 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. One can identify two types of monotonies: a task is *time monotonic* when $\overline{p_j}$ is a non-increasing function, and a task is *work monotonic* when w_j is a non-decreasing function. A task is said to be *monotonic* if it is both time monotonic and work monotonic. Throughout this work, we assume that all the tasks of the considered instance are monotonic. There is no need of such a hypothesis on the GPUs as the tasks are considered sequential on this architecture.

The makespan is defined as the maximum completion time of all tasks. 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 $(Pm, Pk) \mid mold \mid C_{\max}$.

Notice 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 a guaranteed approximation ratio. Recall that for a given problem, the approximation ratio ρ_A of an algorithm A is defined as the supremum of the ratio $\frac{f(I)}{f^*(I)}$ over all the instances I, where f is any minimization objective and f^* is its optimal value.

3 Related Work

From a scheduling perspective, $(Pm, Pk) \mid\mid C_{\max}$ is more general than $P \mid\mid C_{\max}$, and it is a special case of $R \mid\mid C_{\max}$. Lenstra et al. [10] proposed 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. If the parameter m is not fixed then the algorithm is not fully polynomial. The authors also proved that, unless $P = NP$, there is no polynomial-time approximation algorithm for $R \mid\mid C_{\max}$ with an approximation factor of less than $\frac{3}{2}$, and they presented a 2-approximation algorithm. This algorithm is based on rounding the optimal solution of the preemptive version of the problem. Shmoys and Tardos [11] generalized the rounding technique for any fractional solution. Another rounding technique, which yields an improved approximation factor of $2 - \frac{1}{m}$, was introduced by Shchepin and Vakhania [12]. This is, so far, the best-known 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 [13] presented a PTAS to solve it. However, the precise time complexity of this polynomial-time algorithm is not provided, and we expect the algorithm might be less relevant in practice.

Finally, it is worth noticing that if all tasks of the addressed problem have the same acceleration on GPUs, the problem reduces to a $Q \mid\mid C_{\max}$ problem with two machine 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 $\epsilon > 0$ and a running time 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. Some complex numerical simulations

can benefit from the moldable model to leverage different levels of parallelism in an efficient way [14]. 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 by 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 exponential in the number of processors. Thus, although the result is of significant theoretical interest, this algorithm has little practical applicability.

Many previous works are based on a two-phase approach, initially proposed by Turek, Wolf, and Yu [17]. The basic idea is to first select an assignment (the number of processors assigned to each task), and then 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 a ρ-approximation algorithm on the rigid problem also yields a ρ-approximation for the moldable problem, if an optimal assignment of processors to tasks can be found. Two complementary ways for solving the problem have been proposed, either by focusing on the assignment (first phase) 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 algorithm. The other way corresponds to choosing an assignment such that the resulting non-moldable problem is not a general instance of strip-packing: hence, better specific approximation algorithms can be applied. Using the knapsack problem as an auxiliary problem for the selection of the assignment, Mounié et al. [21] designed a $(\frac{3}{2}+\epsilon)$-approximation algorithm for any positive ϵ. This algorithm relies on a structure that only targets $P \mid\mid C_{\max}$, and the assignment selection process does not scale for $(Pm, Pk) \mid\mid C_{\max}$. In the present work, we adapt the structure of the algorithm to deal with the heterogeneous case by designing a completely new and direct assignment selection algorithm. Furthermore, Fan et al. [22] carried out—from a theoretical and experimental point of view—an extensive comparison of low-cost scheduling algorithms for moldable tasks.

Scheduling algorithms for moldable, parallel tasks with precedence constraints have also been the focus of previous works, where many approximation algorithms were developed under certain assumptions on the speedup or run-time behavior of moldable tasks. A common assumption—also used in the present paper—is that a moldable task's run-time is monotonically decreasing in the number of processors. However, Hunold [23] demonstrated that in practical situations several parallel applications may violate this assumption, i.e., assigning more processors will effectively increase the run-time. Nevertheless, our proposed algorithm can be used in practice: it requires that the run-time and work functions of real-world applications have to be adapted in order to fulfill the monotony assumption.

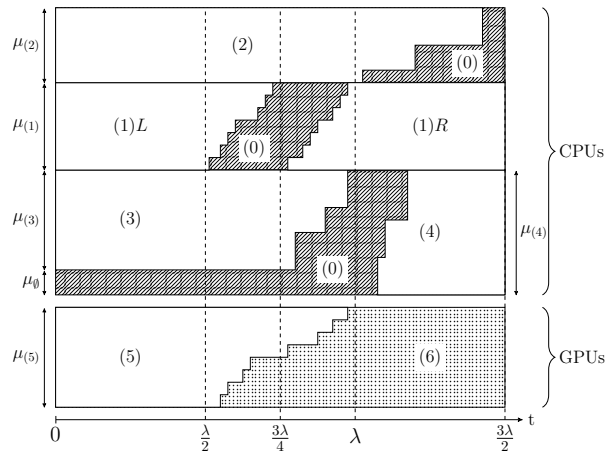


Fig. 1: Structure of the schedule. The number of processors used by set (i) is denoted by $\mu_{(i)}$. The number of CPUs below set (3) is denoted by $\mu_{\emptyset}$.

4 Algorithm APPROX-3/2

The principle of the algorithm is based on a dual approximation scheme [24]. 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 an approximate solution with objective function value at most $\rho\lambda$, or answers correctly that no solution with objective function value at most λ exists.

Our goal is to optimize the makespan, and we target an approximation ratio of $\rho = \frac{3}{2}$. Let λ be the current real number input for the dual approximation. In the whole section, we suppose there exists a schedule of length at most λ, and we show how to build a schedule of length at most $\frac{3\lambda}{2}$. The dual approximation technique helps to drastically reduce the complexity of the algorithm. Knowing an estimate of the optimal makespan allows for reducing the search space by looking for schedules with a given structure (see Section 4.1), and allows for an efficient sweep of the candidate solutions (see Section 4.2.2).

Given a positive number h, we define for each task T_j its *canonical number* of CPUs $\gamma(j,h)$ [21]. 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 at most 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 into seven sets, five for the CPUs and two for the GPUs, as depicted in Fig. 1. This choice of the task assignment to CPUs is detailed below:

(0) The set contains the sequential tasks that are assigned to CPUs with a processing time at most $\frac{\lambda}{2}$.

(1) The set contains the sequential tasks that are assigned to CPUs with a processing time strictly greater than $\frac{\lambda}{2}$ and at most $\frac{3\lambda}{4}$. The tasks of this set are partitioned and

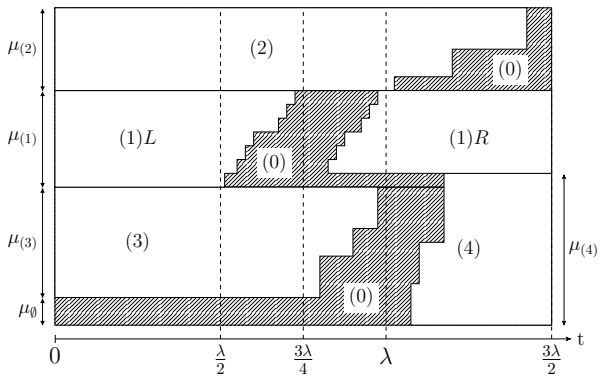


Fig. 2: Structure of the schedule on CPUs with an odd number of tasks in set (1).

assigned to one of two shelves as depicted in Fig. 1, namely, the left set $(1)L$ and the right set $(1)R$.

(2) The set contains either sequential or parallel tasks that possess a canonical number of CPUs for time $\frac{3\lambda}{2}$, but the processing time of these tasks with this canonical number of CPUs must be strictly greater than λ. As this set targets CPUs, all tasks of this set are assigned to $\gamma(j, \frac{3\lambda}{2})$ CPUs.

(3) The set contains the tasks that are 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 $\frac{3\lambda}{4}$ and at most λ. If a task of this set is assigned more than one processor, its processing time is strictly greater than $\frac{\lambda}{2}$ and at most λ.

(4) The set contains the parallel tasks that are 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 into two sets:

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

(6) The set containing the tasks that are assigned to a GPU with a processing time at most $\frac{\lambda}{2}$.

Such a partitioning ensures that the makespan on the CPUs and on the GPUs is at most $\frac{3\lambda}{2}$.

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 Fig. 2).

4.2 Mathematical Formulation

Partitioning tasks into the seven above-mentioned sets using a greedy list scheduling 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 work of the tasks assigned to CPUs:

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

We want to obtain a specific five-set schedule on the CPUs and a two-set schedule on the GPUs. Hence, we look for an assignment that minimizes the total computational area W_C on the CPUs.

The assignment must satisfy the following constraints:

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

(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 GPUs 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 defines a schedule of length at most $\frac{3\lambda}{2}$, which allows us to construct the desired solution. Notice that there are no constraints for sets (0) and (6). We show in Section 5.2 (see Lemmas 5 and 6) that this set of constraints is sufficient to ensure that we can build a feasible solution.

4.2.2 Filtering

Due the structure of the schedule, tasks belong only to a limited number of shelves. Hence, we define for each task T_j the filtering function $F(j)$ computing the set of possible containing shelves. For each set (i) we also define the set of tasks $\mathcal{T}^{(i)}$ that are eligible for an allocation in (i). The eligible allocation sets are explicitly defined as follows:

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

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. $right^{(1)}$),

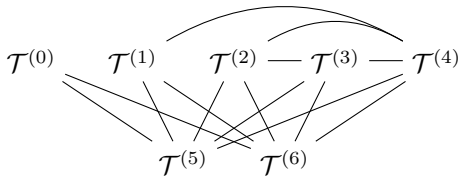


Fig. 3: Intersection graph of the eligible allocation sets in its most generic shape. Actual instances usually have fewer edges.

that corresponds to the number of tasks assigned to the $(1)L$ (resp. $(1)R$) shelf of set (1) (see Fig. 1).

This filtering step helps to reduce the search space. The intersection graph of the eligible allocation sets, shown in Fig. 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 belongs to 2 or 3 sets. For example, let us consider a task T_j with a sequential processing time of at most $\frac{\lambda}{2}$ on a CPU. Depending on its processing time on a GPU , the set of possible shelves for T_j is either $F(j) = \{0, 5\}$ or $F(j) = \{0, 6\}$.

4.2.3 Integer Linear Program

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

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

$$\text{s.t.} \quad W_C^{(ILP)} \leqslant m\lambda, \tag{C_1}$$

$$\sum_{j\in\mathcal{T}^{(2)}} \gamma(j,\frac{3\lambda}{2})x_j^{(2)} + \sum_{j\in\mathcal{T}^{(3)}} \gamma(j,\lambda)x_j^{(3)} + left^{(1)} \leqslant m, \tag{C_2}$$

$$\sum_{j\in\mathcal{T}^{(4)}} \gamma(j,\frac{\lambda}{2})x_j^{(4)} + \sum_{j\in\mathcal{T}^{(2)}} \gamma(j,\frac{3\lambda}{2})x_j^{(2)} + right^{(1)} \leqslant m, \tag{C_3}$$

$$\sum_{j\in\mathcal{T}^{(5)}} \underline{p_j}x_j^{(5)} + \sum_{j\in\mathcal{T}^{(6)}} \underline{p_j}x_j^{(6)} \leqslant k\lambda, \tag{C_4}$$

$$\sum_{j\in\mathcal{T}^{(5)}} x_j^{(5)} \leqslant k, \tag{C_5}$$

$$\sum_{i\in F(j)} x_j^{(i)} = 1 \quad \forall j \in \{1,\dots,n\}, \tag{C_6}$$

$$\sum_{j\in\mathcal{T}^{(1)}} x_j^{(1)} = left^{(1)} + right^{(1)}, \tag{C_7}$$

$$0 \leqslant left^{(1)} - right^{(1)} \leqslant 1, \tag{C_8}$$

$$x_j^{(i)} \in \{0,1\} \quad \forall j \in \{1,\dots,n\}, \forall i \in F(j), \tag{C_9}$$

$$left^{(1)}, right^{(1)} \in \{0,\dots,m\}. \tag{C_{10}}$$

The first eight constraints of this integer linear program correspond to the constraints defined in Section 4.2.1, which allow us to obtain a five-set schedule on the CPUs and a two-set schedule on the GPUs. The last two constraints (C_9) and (C_{10}) are integrity constraints for the variables of the integer linear program.

5 Analysis of APPROX-3/2

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

5.1 Structure of a Schedule with Makespan λ

Assuming that there exists a schedule with length at most λ allows us to efficiently leverage the dual approximation paradigm. We state some straightforward properties of such a schedule, which should give insight for constructing a feasible solution.

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

Notice that for the problem of scheduling moldable tasks on identical processors [21], 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 all n tasks have to be considered in our case.

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

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

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

As we aim at a makespan of $\frac{3\lambda}{2}$, we know from Proposition 3 that 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, which is equal to 1 if the number of tasks in set (1) is odd.

From Proposition 2 we know that the remaining tasks (i.e., not belonging to set (1)) with CPU processing times greater than $\frac{\lambda}{2}$ do not use more than the remaining CPUs, that is $m - \mu_{(1)}$ CPUs. Hence, these tasks can be executed concurrently on the CPUs in set (3), and they occupy $\mu_{(3)}$ CPUs.

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

The remaining tasks on the CPUs have processing times at most $\frac{\lambda}{2}$, and those that are not sequential can be executed in 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 set (1) is odd, there is a CPU that only processes one task from set $(1)L$ and a task from set (4) can be executed on this CPU. Therefore, if we denote by $\mu_{(4)}$ the number of CPUs occupied by tasks of set (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 processing times at most $\frac{\lambda}{2}$, and are assigned to set (0).

With the same reasoning, the tasks on GPUs with processing time 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 the GPUs have processing times at most $\frac{\lambda}{2}$, and can be executed in time at most $\frac{\lambda}{2}$ in set (6) on the GPUs. A task from set (6) can be scheduled after a task from set (5) or on the remaining free GPUs.

5.2 Structure of the Partitioning

We now prove that, under the assumption that the dual approximation does not reject the current guess λ, i.e., $W_C^{(ILP)} \leqslant m\lambda$, the ILP solution leads to a feasible seven-set 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, for time at most $\frac{3\lambda}{2}$.*

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

Proof. By construction, all tasks of set (0) have a sequential processing time on a CPU lower than $\frac{\lambda}{2}$, 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 Fig.s 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 decreasing sequential processing time on the 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. Therefore, tasks are stacked in reverse, starting from the upper bound $\frac{3\lambda}{2}$. Applying this strategy allows us to derive the same structure as shown in Fig. 1. Then, we update the CPU's load accordingly.

At each step, the least loaded CPU has a load of at most λ: it would otherwise contradict the fact that the total work area of the tasks is bounded by $m\lambda$, because of Constraint (C_1). Hence, the idle time interval on the least loaded CPU has a length at least equal to $\frac{\lambda}{2}$, and it can be used to execute task T_s. This completes the proof of Lemma 5. □

Lemma 6. *If $W_C^{(ILP)} \leqslant m\lambda$, the tasks assigned to sets (5) and (6) occupy at most k GPUs, for 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 due to Constraint (C_5), and their processing time is at most λ: the dual approximation would otherwise reject the solution. The tasks of set (5) are scheduled first, one per GPU.

By construction, all tasks in set (6) have a processing time on a GPU at most $\frac{\lambda}{2}$, and thus 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 decreasing processing time on the 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 (same strategy as before). Update the GPU's load.

At each step, the least loaded GPU has a load of at most λ: it would otherwise contradict the fact that the total work area of the tasks is bounded by $k\lambda$, according to Constraint (C_4). Hence, the length of the idle time interval on the least loaded GPU is at least equal to $\frac{\lambda}{2}$ and can contain the task T_s. This completes the proof of Lemma 6. □

Lemmas 4, 5, 6 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 with 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 of at most $\frac{3\lambda}{2}$. □

If no solution exists, for which the computational area on the CPUs is at most $m\lambda$ (i.e., Constraint (C_1) cannot be fulfilled), the dual approximation algorithm rejects the current guess λ. Such a behavior is due to (ILP), as it minimizes the computational area $W_C^{(ILP)}$. As the seven-set structure allows solutions with makespan greater than the optimal, this implies that $W_C^{(ILP)} \leqslant W_C^*$. Let us assume that a solution with makespan of at most λ exists, then we would get $m\lambda < W_C^{(ILP)} \leqslant W_C^* \leqslant m\lambda$. This leads to a contradiction. Hence, no solution with a makespan of at most λ exists in that case, and rejecting the current guess λ is the correct behavior.

For a given guess λ of the dual approximation algorithm, we have proved so far that if (ILP) is infeasible ($W_C^{(ILP)} > m\lambda$), then there is no solution with makespan λ, and the guess has to be rejected. If the solution of (ILP) has a computational area on the CPUs of at most $m\lambda$, then we can construct a solution with makespan at most $\frac{3\lambda}{2}$ by using the partitioning of the CPUs and GPUs sets.

5.3 Correctness of the Dual Approximation Scheme

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

Lemma 8. *Suppose that a solution σ_{ref} with makespan at most λ exists. The assignment of tasks to 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 to 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 are interested in. □

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

Lemma 9. *If a solution σ_{ref} with makespan at most λ exists, then a solution σ_{struct} with the seven-set structure exists, 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, we prove that the big tasks of σ_{ref}, namely tasks with a processing time greater than $\frac{\lambda}{2}$, fit in the 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 $\frac{\lambda}{2}$, and thus, 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 pairs, one in shelf $(1)L$, the other in shelf $(1)R$. As a result, the tasks in set (1) in σ_{struct} use fewer processors than they would in σ_{ref}.
- The tasks assigned to sets (2) and (3) use their canonical number of CPUs for a time limit of 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 that is greater than λ, they use fewer processors than they would in σ_{ref}. Observe that the tasks assigned to set (2) use fewer processors than they do in σ_{ref} due to the relaxed time limit.

We now consider the tasks of σ_{ref} assigned to CPUs with a processing time at most $\frac{\lambda}{2}$. All the tasks with a sequential time at most $\frac{\lambda}{2}$ 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 at most $\frac{\lambda}{2}$. The monotony assumption ensures that they can fit in any of the sets (1), (2), (3), and (4), without increasing the computational load. In order to prove that such an assignment of these remaining tasks exists, we consider the integer linear program introduced in Section 4.2, but we relax it by removing Constraint (C_3). This allows set (4) to occupy as many CPUs as needed. Due to Lemma 8, tasks that have already be assigned to GPUs as in σ_{ref} have their corresponding variables in the integer linear program set according to their assignment. 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 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 proof of the first case 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—assigned to tasks from set (3)—is loaded for time at most λ but at least $\frac{3\lambda}{4}$. 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 now consider the case where some CPUs remain idle. We denote their number by $\mu_\emptyset$.

(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 five-set structure 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 minimizing the CPU load. Therefore, only a fraction of a single task can be assigned to set (4) while its 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 we suppose that 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

set $(3)\Delta$ the subset of set (3) that shares processors with 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 assigned task T_Δ to set (4). As set (4) is the one creating the most work among sets (1), (2), (3), and (4), this choice had to be made because constraints would have been violated otherwise. We know for sure that $\mu_{(3)\Delta} > 0$, otherwise this would contradict Step 1. Moreover, as T_Δ was not assigned to $\mu_\emptyset$ 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:

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

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 show that the previous step leads to a contradiction, hence $\overline{\sigma}_{\text{struct}}$ fits into m CPUs. Inequality (3) 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 results in a contradiction, as this would mean that the total work is greater than $m\lambda$. □

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

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

5.4 Building the Schedule

We have described the core step of the dual approximation algorithm with a fixed guess. A binary search is employed to approach the optimal makespan with successive guesses. By 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 for reducing the search space size of the ILP, as a task can be assigned to at most four sets instead of seven (cf. Fig. 3). Moreover, as the number of tasks increases, every task's relative processing time shrinks. Thus, for large instances, most of the tasks will be assigned to sets (0) and (6) only.

One could also employ dynamic programming to solve the allocation problem, as it would result in an approximation algorithm of polynomial-time complexity. However, the search space is so large that such scheme would be practically infeasible. By adapting the techniques proposed by Bleuse et al. [7], such an algorithm would have complexity of $\mathcal{O}(n^2m^4k^2)$.

6 Algorithm APPROX-2

As stated in Section 5.4, the run-time of APPROX-3/2 is not proved to be polynomial. To get more insight on our dual approximation algorithm, we devise a simpler, polynomial-time approximation algorithm APPROX-2, which provides an approximation ratio of $2 + \epsilon$. APPROX-2 uses the same principles as APPROX-3/2: it partitions the computing resources, allocates the tasks to a partition, and schedules the tasks within their partition. Note that APPROX-2 is presented here under the assumption of having monotonic tasks, but it does not rely on such an assumption. The algorithm can easily be adapted for any kind of moldable task model, by considering for each task the allocation on CPUs that has a processing time of at most λ and minimizes the work.

6.1 Sketch of APPROX-2

We consider a guess λ of the optimal makespan. The scheduling problem on the CPUs is simplified by forcing each task to use its canonical number of CPUs, with respect to λ, i.e., $\gamma(j, \lambda)$. If a task with processing time greater than λ on both architectures exists, the guess λ is trivially rejected. Otherwise, the algorithm works as follows:

1) Allocate the tasks that possess a running time at most λ on only one type of architecture (i.e., these tasks are larger than λ on the other type of architecture).
2) Sort the remaining tasks by decreasing work ratio $\frac{w_{j,\gamma(j,\lambda)}}{p_j}$. The approximation ratio derives from this sorting, as will be explained in Lemma 12.
3) Allocate the sorted tasks on the GPUs until each GPU has a load of more than λ.
4) Schedule the remaining rigid tasks on the CPUs—which are allotted $\gamma(j, \lambda)$ CPUs—with a 2-approximation algorithm. List scheduling algorithms or strip-packing algorithms are viable alternatives for this step.

If the tasks do not fit within a makespan of at most 2λ, then the algorithm rejects the guess. Otherwise, we have found a valid schedule.

6.2 Analysis of APPROX-2

We now analyze some properties of APPROX-2. First, we study its approximation ratio, then its time 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 a solution with makespan at most λ exists, 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 in an 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 a solution with makespan at most λ exists. The makespan bound follows from the fact that we schedule the remaining tasks with a list scheduling [25] or a strip-packing [26] algorithm. The strip-packing algorithm would provide a contiguous solution. □

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

APPROX-2 is an algorithm of low polynomial complexity. It 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, the complexity of the algorithm is $\mathcal{O}\left(n\left[\log(n)+\log(k)+m\log(m)\right]\right)$.

7 Simulation Results and Performance Evaluation

After providing the theoretical foundation for solving the given scheduling problem, we now examine the applicability of our approach. To this end, we evaluate both the makespan of the schedules and the run-time to compute the solutions of APPROX-3/2 and APPROX-2. In our evaluation, we also consider scheduling solutions from other heuristics, for which we used adaptations of the classical Heterogeneous Earliest Finish Time algorithm (HEFT) [27].

7.1 Problem Instances

Finding the right problem instances for evaluating scheduling algorithms through simulation is generally a hard problem, and real-world instances are often considered essential for such an analysis. However, testing an algorithm on only a small set of real-world instances will most likely not support the claim with enough empirical evidence that an algorithm is generally well applicable. The StarPU run-time system supports the dispatching of jobs to either CPUs and GPUs [28], and it can potentially execute moldable tasks. However, until now run-time systems like StarPU or ParSEC [29] schedule single-processor (sequential) tasks on available CPUs and GPUs. For that reason, to the best of our knowledge, no real-world instances for our scheduling problem exist yet. Another problem is that influencing factors, such as the number of tasks or the size of tasks, are most often fixed in real-world instances. As a consequence, evaluating the impact of different factors is often impossible. Hence, we decided to generate instances that allow us to study the general applicability of our algorithms by varying different factors.

To generate new instances, we first fix the main parameters of the scheduling problem, which are the number of tasks (n), the number of CPUs (m), and the number of GPUs (k). Then, the instance generator decides on the processing time of all tasks on several CPUs and one GPU. The intuition behind the generation process is the following:

1) First, we randomly pick the sequential processing time of a task on a single CPU.
2) Then, we choose the sequential fraction of this task, which defines its speedup model. The idea is that the sequential fraction (between 0 and 1) 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 [30]).
3) Last, we pick a speedup factor for this task on the GPU, which defines how much faster (or possibly slower) a particular task runs on a GPU compared to being executed on all m CPUs.

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

Step 1: The sequential processing 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 moldable tasks. The law states that the possible speedup of a parallel program is bounded by its sequential fraction. We select the sequential fraction β_j of each task, where β_j follows a uniform distribution in $[\beta^{min}, \beta^{max}]$. The knowledge of the sequential processing time $\overline{p_{j,1}}$ and the sequential fraction β_j allows us to compute the parallel processing 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 when executed 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 on all m CPUs ($\overline{p_{j,m}}$). To obtain the time on the GPU ($\underline{p_j}$), we draw a speedup factor g_j for task T_j and set $\underline{p_j} = g_j\,\overline{p_{j,m}}$, where the value of g_j follows a normal distribution with mean g^{mean} and standard deviation g^{sd}. The idea is that most tasks should be faster on the GPU on average [1]. But since Lee et al. [1] report that several applications may also experience a slowdown on a GPU, we allow tasks to be slower on the GPU than when being executed on all CPUs. We also limit the maximum speedup or maximum slowdown of each task on the GPU in order to generate realistic processing times, as the normal distribution is unbounded. We introduce the variables g^{min} and g^{max} to denote the minimum (maximum speedup) and maximum value of g_j (maximum slowdown) of a task when being executed on a GPU. That means, if the drawn speedup factor g_j is outside the interval $[g^{min}, g^{max}]$ then we draw another value from the normal distribution. This process is repeated until the value of g_j lies within the interval.

Table 1 shows the set of parameters that were used to produce the simulation results shown in the present paper. Ten problem instances were generated for a combination of parameters n, m, and k using the method described before. Notice that we did not generate instances for all possible combinations, as they would often not be useful. For example, for an instances with $n = 10$ tasks, we restricted the number of processors to $m = 16$ and the number of GPUs to $k = 1$. Similarly, the number of CPUs and GPUs was restricted to $m = 64, k = 4$ and $m = 512, k = 32$, for instances with $n = 100$ and $n = 1000$ tasks, respectively. The sequential processing time of each task lies within 1 and 100 units of

TABLE 1: Parameter settings used to generate scheduling instances.

description	variable	values
number of tasks	n	$\{10, 50, 100, 1000\}$
number of CPUs	m	$\{4, 16, 64, 256, 512\}$
number of GPUs	k	$\{1, 2, 4, 8, 16, 32\}$
minimum sequential processing time of tasks	$\overline{p}^{min}$	1
maximum sequential processing time of tasks	$\overline{p}^{max}$	100
minimum sequential fraction of a task	β^{min}	0
maximum sequential fraction of a task	β^{max}	0.9
mean speedup factor for tasks on GPUs	g^{mean}	0.2
standard deviation of speedup factor for tasks on GPUs	g^{sd}	0.5
minimum speedup factor for tasks on GPUs	g^{min}	0.1 (10× speedup on the GPU)
maximum speedup factor for tasks on GPUs	g^{max}	1.5 (50% slowdown on the GPU)

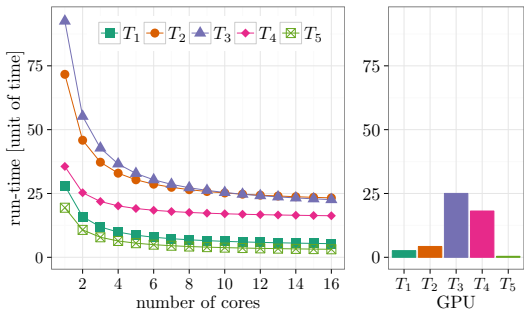


Fig. 4: Example instance ($n = 5$, $m = 16$, $k = 1$): Each of the five tasks exhibits a different parallel scalability on the multi-core machine (left) and performs differently on the GPU (right).

time. The actual unit of time is irrelevant in the simulations. What matters is that, in this case, there are two orders of magnitude between different processing times. Thus, the sequential processing times are very heterogeneous, similar to what we find in batch scheduling systems on current parallel machines [31]. The mean speedup factor g_j is based on the speedup graphs presented by Lee et al. [1]. Despite the fact that Lee et al. only showed an average speedup of 2 to 3 on the GPU with their set of benchmarks, we used an average GPU speedup of 5 (speedup factor of 0.2).

Fig. 4 depicts one of the problem instances, which is composed of only five tasks for the sake of readability. Each task has a different scalability behavior (caused by a different sequential fraction). We can also observe a different speedup behavior on the GPU. The processing times of tasks T_3 and T_4 increase on the GPU, whereas the other tasks experience a performance improvement when being mapped onto the GPU.

We have also experimented with other sets of parameters, including different distributions for the sequential processing time, e.g., using a beta distribution, such that processing times become more homogeneous. We also modified the intervals for the sequential processing time and the speedup factor. Last, we tested the influence of the sequential fraction of the tasks. In many other scenarios, the results were similar to the ones shown in the present paper. However, we found that heterogeneity in the problem instance favors our APPROX-3/2 algorithm, i.e., the more heterogeneous the instances become the better the schedules of APPROX-3/2 get in comparison to the other algorithms.

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 these approaches with practically relevant algorithms, we include HEFT-like algorithms in our evaluation. We call them HEFT-like algorithms as they work similar to the original HEFT algorithm [27], but target a slightly different scheduling problem. Such algorithms are used in practice, for example, in the run-time system StarPU, which implements a very similar algorithm (called MCT for minimum completion time) to schedule tasks on CPUs and GPUs [28].

Now, we describe the variants and implementations of the HEFT-like algorithms for scheduling moldable tasks on system containing multiple CPUs and several GPUs. Our implementation resembles the original algorithm proposed by Topcuoglu et al. [27], except that—since precedence constraints are absent—we change the priority function used to sort the tasks. Similar to HEFT, our algorithm places the highest priority task on either a subset of CPUs or on one of the GPUs, such that the finish time of a task is minimized. We call this strategy *earliest finish time* (**EFT**). We expected that HEFT-like algorithms are most likely 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 tasks in decreasing order of their processing times (**L**ongest **P**rocessing **T**ime),
2) **SPT**: This strategy sorts tasks in increasing order of their processing times (**S**hortest **P**rocessing **T**ime), and
3) **RATIO**: This strategy sorts tasks in decreasing order of the following ratio: processing time on the CPUs over the processing time on a GPU, i.e., $\frac{\overline{p_{j,l}}}{\overline{p_j}}$, where l is either 1 for sequential tasks or m for parallel tasks.

For the strategies **LPT** and **SPT**, the processing time of task T_j is computed as $\min(\overline{p_{j,l}}, \overline{p_j})$, for $l \in \{1, m\}$. By using the minimum, the partial schedules on the CPUs or GPUs will be roughly in **LPT** or **SPT** order, which would not be the case when using $\max(\overline{p_{j,l}}, \overline{p_j})$. Notice that we have also performed simulations using the maximum of the two processing times, but the HEFT-like algorithms computed better solutions using the minimum.

Now, the remaining question is: 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

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**)

to a task ($l = 1$). Considering the monotonic assumption for the processing time of moldable tasks (i.e., being non-increasing in the number of CPUs), the strategy **PAR** is a greedy way of minimizing the processing time of a task. The second strategy (**SEQ**) favors task parallelism and minimizes the work. It is certainly possible to improve these HEFT-like heuristics, but such considerations are outside the scope of this paper. In total, we have implemented six different HEFT-like heuristics, called Heuristic 1–6, which are listed in Table 2.

7.3 Implementation Details

We implemented APPROX-3/2[1] using the programming languages Julia [32] and Python. Logically, the algorithm APPROX-3/2 consists of two steps: (i) finding the best λ by applying the bisection method to partition the tasks into sets, and (ii) building 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. It allows us to write the ILP using the JuMP API[2] only once, and then we are able to use different solvers to compute a solution. The second step, the building of the schedule, has been implemented in Python.

The algorithm APPROX-2 has been entirely implemented in Julia and also relies on a bisection search to find the best λ. However, since it maps tasks directly to devices (CPUs or GPUs), instead of relying on the solution of an ILP, the actual schedule is built on the fly.

As stated above, the lower and the upper bound of the scheduling problem get 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 simulations.

The HEFT-like heuristics have been implemented in Python. Similarly to the implementation of APPROX-2, the actual schedule is directly built, as no previous partitioning step is required.

We have used the following software packages for obtaining the simulation results presented in this paper: Julia $0.4.3$, Python $2.7.11$, JuMP $0.12.2$, Gurobi binding for JuMP $0.2.1$, Gurobi Optimizer for OS X $6.5.1$, and Mac OS X $10.10.5$.

7.4 Simulation Results

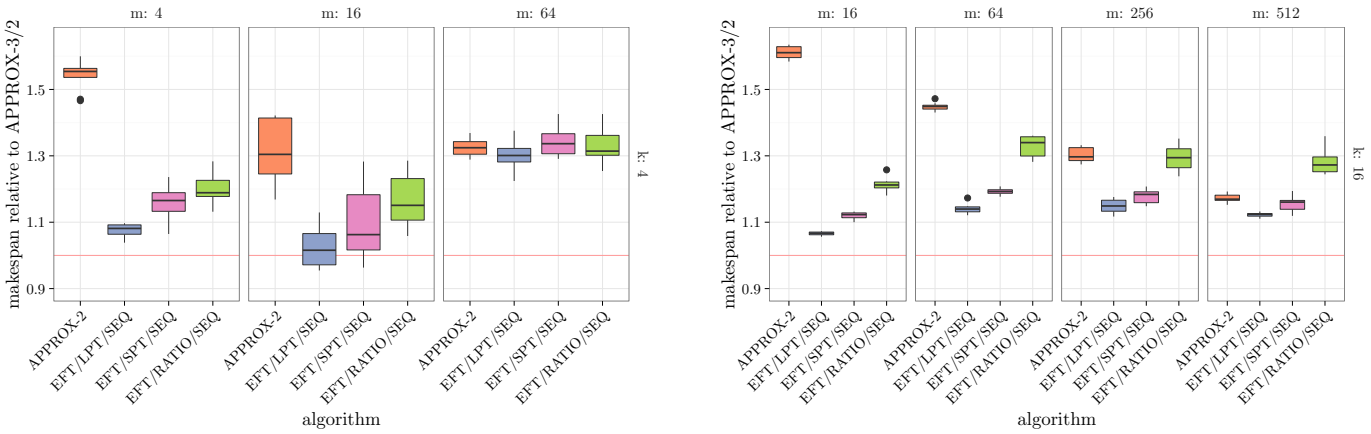
Fig. 5 compares the makespans of the schedules that were computed by APPROX-3/2, APPROX-2, and the six different HEFT-like heuristics. For a better comparability, we normalize the makespan for each individual scheduling instance to the makespan obtained from APPROX-3/2. Thus, the algorithm APPROX-3/2 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 is shorter than the one obtained from 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 for smaller instances (cf. Fig. 5a, case m=16 and k=4). If the number of tasks, CPUs, and GPUs increases, the results in Fig. 5b provide evidence that APPROX-3/2 produces significantly shorter schedules than its competitors. The results of the heuristics 4 to 6 using the **PAR** strategy (cf. Table 2) have been omitted for the sake of readability, 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 ones of 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 solving 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.

Fig. 6 compares the 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 simulations were conducted on a quad-core Intel i7-3615QM with a clock speed of 2.3 GHz. We recorded the run-time of each algorithm on each instance ten times and computed the median run-time over the ten trials. Then, we aggregated all instances by unique values of n, m, and k, i.e., we compute the median run-time over the 10 runs for one particular instance, and then compute the mean over all instances for a specific tuple of values (n, m, k).

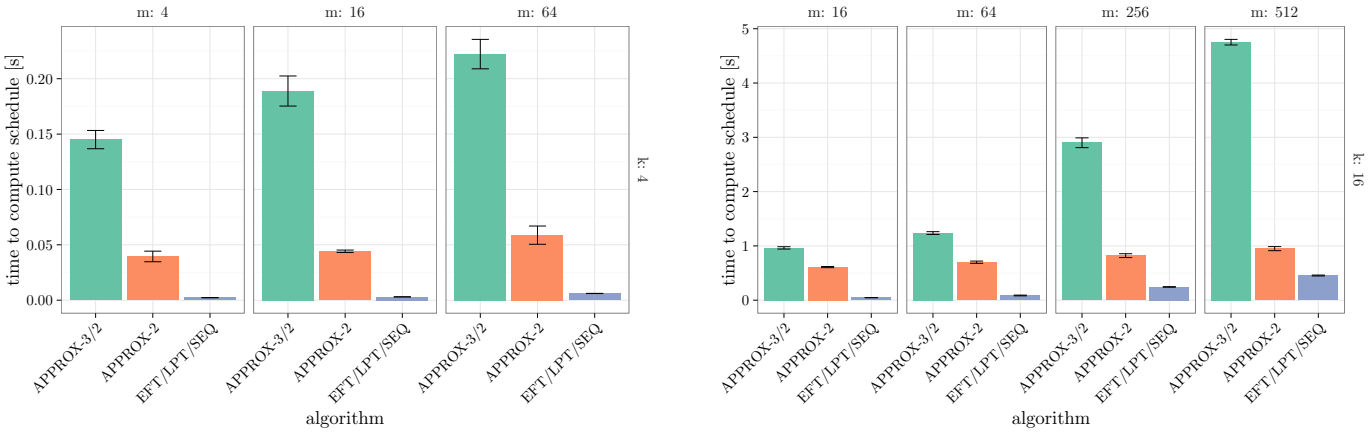
Since the run-times of the various HEFT-like heuristics were very similar, as only the prioritization function needs to be changed, we only show the run-time for Heuristic 1 (**EFT/LPT/SEQ**). As expected, the run-time of Heuristic 1 has been found to be the shortest among the three scheduling algorithms 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 in the cases considered. However, we can observe that the run-time of APPROX-3/2 grows sub-linearly with problem parameter m, i.e., increasing the number of cores m by two does not lead to a twice slower run-time. Therefore, although APPROX-3/2 is an ILP-based algorithm,

1. source code available at https://github.com/hunsa/moldableILP
2. Application Programming Interface



(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 algorithm to compute the solutions (n tasks, m CPUs, k GPUs).

it computes solutions relatively quickly; it takes less than five seconds to compute the schedule for larger instances in our test set (e.g., $n = 1000$, $m = 512$, $k = 16$, cf. Fig. 6b). This run-time is certainly too large to schedule relatively short, fine-grained tasks on CPUs or GPUs, but it is a promising alternative when trying to schedule long-running tasks (or even different parallel applications).

We have also studied the effectiveness of the filtering step that we introduced in Section 4.2.2. We recall that the internal ILP finds 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 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. Fig. 7 shows the number of available partitions for increasing values of n. The "mean number of possible partitions" is computed over all tasks of one iteration. For

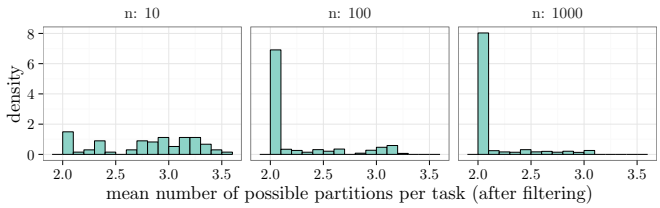


Fig. 7: Distribution of the (mean) number of possible partitions per task after the filtering has been applied for APPROX-3/2. The graphs show distributions for all values of m and k presented in Table 1.

example, for one problem instance, the mean number of possible partitions (over all the tasks) may be 2 in iteration 1 and 2.5 in iteration 2. In such a case, the distributions shown in Fig. 7 will contain the values 2 and 2.5. We observe that, except in the case of $n = 10$, the ILP only needs to decide between two partitions for the majority of tasks (on average).

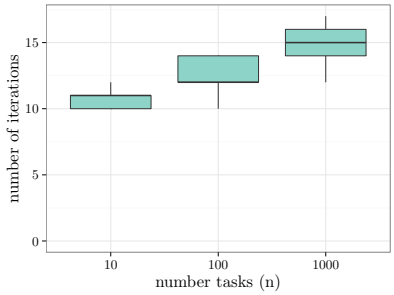


Fig. 8: Distribution of iterations (of the bisection method) performed by APPROX-3/2 to converge to a solution.

This supports our claim that the filtering step is very effective and important for obtaining a practically relevant running time.

Fig. 8 complements the previous results by an analysis of the number of iterations required until the bisection method converges. In our simulations, 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 schedules 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 compared to the other scheduling algorithms. The simulation results also provide evidence that using moldable tasks can indeed improve performance. As OpenMP applications are examples of moldable tasks in practice, a next step could be to evaluate our algorithm in an experimental setting, e.g., computing and executing a static schedule on top of StarPU.

8 Conclusions

In this paper, we presented a new scheduling algorithm using a generic methodology (the opposite of specific ad-hoc algorithms) for hybrid architectures (a 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 find an adequate partition of tasks on the CPUs and the GPUs, by using a dual approximation scheme and integer linear programming. We were not able to show that the running time of our algorithm is polynomial in the input size. Nonetheless, we show that our algorithm is efficient by comparing it to a polynomial-time algorithm with approximation ratio $2 + \epsilon$. A simulation analysis on realistic instances has been provided to assess the computational efficiency and the schedule quality of the proposed method when compared to adaptations of the classical HEFT algorithm. The main conclusion is that the ILP-based algorithm is stable because of its approximation guaranty, and it runs in a reasonable time. Moreover, the proposed algorithm outperforms all HEFT-like algorithms when dealing with instances of larger size, which is often the case on real 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, and S. Kato, "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] 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.

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

[12] 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.

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

[14] É. Blayo, L. Debreu, G. Mounié, and D. Trystram, *Euro-Par'99*. Springer Berlin Heidelberg, 1999, ch. Dynamic Load Balancing for Ocean Circulation Model with Adaptive Meshing, pp. 303–312.

[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," *Algorithmica*, vol. 32, no. 3, pp. 507–520, 2002.

[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] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

[29] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. J. Dongarra, "PaRSEC: Exploiting heterogeneity to enhance scalability," *Computing in Science and Engineering*, vol. 15, no. 6, pp. 36–45, 2013.

[30] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the AFIPS '67 Spring Joint Computer Conference*, 1967, pp. 483–485.

[31] E. Shmueli and D. G. Feitelson, "On simulation and design of parallel-systems schedulers: Are we doing the right thing?" *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 7, pp. 983–996, 2009.

[32] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *CoRR*, vol. abs/1411.1607, 2014. [Online]. Available: http://arxiv.org/abs/1411.1607

Raphaël Bleuse holds a M.Sc. degree in Computer Science from Grenoble INP, Grenoble, France. He is a Ph.D. candidate in Computer Science at the University Grenoble Alpes, Grenoble, France.

Sascha Hunold received the Ph.D. degree in Computer Science from the University of Bayreuth, Bayreuth, Germany, and the M.Sc. degree in Computer Science from the University of Halle- Wittenberg, Halle, Germany. He is an assistant professor at the TU Wien, Vienna, Austria.

Safia Kedad-Sidhoum is a member of the LIP6 (Laboratoire d'Informatique de Paris 6) as Assistant Professor in the team « Operations Research » of Pierre et Marie Curie University. She received a Ph.D. degree in Operations Research in January 1997 from Ecole Centrale Paris, and an HdR degree from Pierre et Marie Curie University in November 2010. She spent two years (97-99) in Dynasys as a Project Manager in Supply Chain planning. She received a CNRS-Google Focused Research Award in 2011. She is a member of the steering committee of the International Workshop on Lot-Sizing (IWLS) since 2010 (every year). She is a member of the organizing committee of the challenge ROADEF/EURO since 2012. She participated to several research projects including ANR (LMCO), FUI (RCSM, DematFactory) and PEPS (COOL) projects. Her research interests include combinatorial optimization, supply chain, scheduling theory, planning and lot-sizing.

Florence Monna holds a Ph.D. degree in Computer Science from the UPMC, Sorbonnes University, Paris, France, and a M.Sc. degree in Applied Mathematics (Optimization) from ENSTA ParisTech, Paris, France. She now works as an engineer in the Research and Development department at Solent SAS, Nanterre, France.

Grégory Mounié received the Ph.D. degree from Grenoble Institute of Technology, France, in 2000. He is associate professor at the Univ. Grenoble, Alpes. (dept. Grenoble-INP/Ensimag), France, since 2001. He lectures on operating systems, concurrent programming and distributed programming. His research focuses on scheduling for high performance computing, parallel computing and communication optimization.

Denis Trystram is professor at Grenoble Institute of Technology since 1991 and is now distinguished professor in this Institute. He was nominated as a senior member of the Institut Universitaire de France in 2010. Denis' current research activities concern the design and analysis of efficient approximation algorithms for multi-objective scheduling problems applied to parallel and distributed processing (from many-cores with accelerators to exascale systems). He is interested in problems concerning reliability and energy optimization and started collaborations with the French company BULL to accompany them to the exascale. He is also involved in the editorial board of Parallel Computing, Journal of Parallel and Distributed Computing and IEEE TPDS. He served on the program committees of major international conferences in the field. He has published several books, more than 90 papers in international journals and 150 contributions in international conferences.