

Generic Algorithms for Scheduling Applications on Heterogeneous Platforms

Marcos Amaris, Giorgio Lucarelli, Clément Mommessin, Denis Trystram

▶ **To cite this version:**

Marcos Amaris, Giorgio Lucarelli, Clément Mommessin, Denis Trystram. Generic Algorithms for Scheduling Applications on Heterogeneous Platforms. Concurrency and Computation: Practice and Experience, 2018, 10.1002/cpe.4647 . hal-01896868v1

HAL Id: hal-01896868

https://inria.hal.science/hal-01896868v1

Submitted on 16 Oct 2018 (v1), last revised 17 Oct 2018 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Generic Algorithms for Scheduling Applications on Heterogeneous Platforms*

Marcos Amaris Giorgio Lucarelli Clément Mommessin
Denis Trystram†

October 16, 2018

Abstract

We study the problem of executing an application represented by a precedence task graph on a parallel machine composed of standard computing cores and accelerators. Both off-line and on-line settings are addressed by proposing generic scheduling approaches. In the first case, we establish strong lower bounds on the worst-case performance of a known approach based on Linear Programming and replace the greedy List Scheduling policy used in this approach by a better task ordering. Although this modification leads to the same approximability guarantees, it performs much better in practice. We also extend this algorithm to more types of computing units, achieving an approximation ratio which depends on the number of different types. In the on-line case, tasks arrive in any order which respects the precedence relations and the scheduler has to take irrevocable decisions about their allocation and execution. We propose the first on-line scheduling algorithm taking into account precedences, which is based on adequate rules for selecting the type of processor where to allocate the tasks. Finally, all the previous algorithms have been experimented on a large number of simulations built on actual libraries, assessing their good practical behavior with respect to the state-of-the-art solutions and baseline algorithms.

1 Introduction

The parallel and distributed platforms available today become more and more *heterogeneous*. Such heterogeneous architectures, composed of several kinds of computing units, have a growing impact on performance

*An extended abstract of a preliminary version of this paper has been presented in Euro-Par 2017 [2]

†Institute of Mathematics and Statistics, University of São Paulo, São Paulo, Brazil. amaris@ime.usp.br
Univ. Grenoble Alpes, CNRS, Inria, LIG, F-38000 Grenoble, France. {giorgio.lucarelli, clement.mommessin, denis.trystram}@inria.fr

in high-performance computing. Hardware accelerators, such as General Purpose Graphical Processing Units (in short GPUs), are often used in conjunction with multiple computing units (CPUs) on the same chip sharing the same common memory [14]. As an instance of this, the number of platforms of the TOP500 equipped with accelerators has significantly increased during the last years [20]. In the future it is expected that the nodes of such platforms will be even more diverse than today: they will be composed of fast computing nodes, hybrid computing nodes mixing general purpose units with accelerators, I/O nodes, nodes specialized in data analytics, etc. The interconnect of a huge number of such nodes will also lead to more heterogeneity. Using heterogeneous platforms would lead to better performances through the use of more appropriate resources depending on the computations to perform, but it has a cost in terms of code development and more complex resource management.

In this work, we present efficient algorithms for scheduling an application represented by a precedence task graph on hybrid and more general heterogeneous computing resources. We are interested in designing generic approaches for efficiently implementing parallel applications where the scheduling is not explicitly part of the application. In this way, the code is portable and can be adapted to the next generation of machines.

Underlying architecture.
We consider a hybrid multi-core machine composed of two different types of resources. Each type is composed of a set of identical processors. An application consists of tasks that are linked by precedence relations. Each task is characterized by two processing times depending on which type of processor it is assigned to. We assume that an exact estimation of both these processing times is available to the scheduler. This assumption can be justified by several existing models to estimate the execution times of tasks [1]. In several applications we always observe an acceleration of the tasks if they are executed on a GPU compared to their execution on a CPU. However, we consider the more general case where the relation between the two processing times can differ for different tasks. This work focuses on the analysis of the qualitative behavior induced by heterogeneity since it may be assumed that the computations dominate local shared memory costs. Thus, no memory assignment or overhead for data management are considered, nor communication times between the shared memory and the processors, or between two processors of different types. Without loss of generality, we denote in the following by CPU and GPU the two types of processors.

As the application developers are mainly looking for performance, the objective of a scheduler is usually to minimize the completion time of the last finishing task, which is one of the most commonly studied objectives [11]. In an heterogeneous context, minimizing the makespan of an application corresponds to minimizing the maximum between the makespan of the tasks assigned on each set of processors.

Definition and notations.
We consider a parallel application which should be scheduled on m identical CPUs and k identical GPUs. Without loss of generality, we assume

that $m \geq k$. The application is represented by a Directed Acyclic Graph $G = (V, E)$ whose nodes correspond to sequential tasks and arcs correspond to precedence relations among the tasks. We denote by $\mathcal{T}$ the set of all tasks. The execution of a task needs a different amount of time if it is performed by a CPU or by a GPU. Let $\overline{p_j}$ (resp. $\underline{p_j}$) be the processing time of a task T_j if it is executed on any CPU (resp. GPU). Given a schedule S, we denote by C_j the completion time of a task T_j in S. In any feasible schedule, for each arc $(i, j) \in E$, the task T_j cannot start its execution before the completion of T_i. We say that T_i is a *predecessor* of T_j and we denote by $\Gamma^-(T_j)$ the set of all predecessors of T_j. Similarly, we say that T_j is a *successor* of T_i and we denote by $\Gamma^+(T_i)$ the set of all successors of T_i. We call *descendant* of T_j each task T_i for which there is a path from j to i in G.

The objective is to create a feasible non-preemptive schedule of minimum makespan. In other words, we seek a schedule that respects the precedence constraints among tasks, does not interrupt their execution and minimizes the completion time of the last finishing task, i.e., $C_{\max} = \max_{T_j}\{C_j\}$. Extending the three-fields notation for scheduling problems introduced by Graham, this problem can be denoted as $(CPU, GPU) \mid prec \mid C_{\max}$.

Contributions and outline.
In this paper we study the above problem on both off-line and on-line settings. The goal is to design algorithms through a solid theoretical analysis that can be practically implemented in actual systems.

Contrarily to most existing approaches (e.g., [22]), we address the problem by separately focusing on the following two phases, as in [16]:

- *allocation*: each task is assigned to a type of resources, either CPU or GPU;
- *scheduling*: each task is assigned to a specific pair of resource and time interval respecting the decided allocation as well as the precedence constraints.

In the off-line mode, we aim to study the two phases separately motivated by the fact that there are strong lower bounds on the approximability of known single-phase algorithms. For example, the approximation ratio of the well-known Heterogeneous Earliest Finish Time (HEFT) algorithm [22] cannot be better than $\Omega(\frac{m}{k^2})$ when $k \leq \sqrt{m}$ (Section 3). On the other hand, it can be easily shown that List Scheduling policies have arbitrarily large approximation ratio, even if we consider some enhanced order of tasks, like prioritizing the task of the largest acceleration.

The two-phases approach has been used by Kedad-Sidhoum et al. [16] where a linear program (which we call Heterogeneous Linear Program or simply HLP) in conjunction with a rounding have been proposed for the allocation phase, while the greedy Earliest Starting Time (EST) policy has been applied to schedule the tasks. This algorithm, called HLP-EST, achieves an approximation ratio of 6 and we show in Section 3 that this ratio is tight. In fact, our worst-case example does not depend on the scheduling policy applied in the second phase.

Based on this negative result, we propose to revisit both phases. In Section 4.1, we replace the EST policy in HLP-EST by a specific ordering of tasks combined to a classical List Scheduling. The task ordering is based on both the allocation decisions taken during the first phase (linear program) and the critical path. This refined algorithm, denoted by HLP-OLS, preserves the tight approximation ratio of 6 and achieves good practical performances.

In Section 4.2 we study the on-line version of the problem, where tasks arrive in any order that respects the precedence constraints, and the scheduler has to take irrevocable decisions for their execution at the time of their arrival. We present a combination of low-complexity rules for deciding the allocation of a task upon its arrival, which takes into account the actual schedule and the relation between its processing times. We show that these rules, combined with List Scheduling, lead to an algorithm of competitive ratio $\Theta(\sqrt{\frac{m}{k}})$. This is the first on-line algorithm when precedence constraints are considered in the hybrid context.

In Section 5 we propose an extension of HLP-EST and HLP-OLS and their analysis for the case where $Q \geq 2$ types of identical processors are available. We show that both algorithms achieve a tight approximation ratio of $Q(Q+1)$.

Section 6 describes the generation of a benchmark used to perform the experimental evaluation of the discussed off-line and on-line algorithms, which is presented in Section 7. In the off-line setting, experiments showed that the new scheduling method based on HLP (HLP-OLS) outperforms HLP-EST on both contexts with 2 or 3 resource types with an average improvement of 10%. Comparisons with HEFT showed that HLP-OLS offered similar performances with an improvement of 2% on average for 2 resource types, but a decrease of 4% on average for 3 resource types. In the on-line setting, results showed that our proposed algorithm outperformed a greedy policy by an average improvement of 16% but was outperformed by the Earliest Finish Time (EFT) policy by 10% on average.

Before continuing, we present in Section 2 the work related to our setting and, finally, we conclude in Section 8.

2 Related work

Most papers of the huge existing literature about GPUs concern specific applications. There are only few papers dealing with generic scheduling in mixed CPU/GPU architectures, and very few of them consider precedence constraints.

From a theoretical perspective, the problem of scheduling tasks on two types of resources is more complex than the problem on parallel identical machines, $P \mid prec \mid C_{\max}$, but it is easier than the problem on unrelated machines, $R \mid prec \mid C_{\max}$. Moreover, if all tasks are accelerated by the same factor in the GPU side, then $(CPU, GPU) \mid prec \mid C_{\max}$ coincides with the problem of scheduling on uniformly-related parallel machines, $Q \mid prec \mid C_{\max}$. In this sense, we can say that the former is more general than the latter one; however, in our problem all tasks have only two different processing times, that makes it simpler.

For $P \mid prec \mid C_{\max}$, Graham's List Scheduling algorithm [15] is a 2-approximation, while it is NP-hard to have a better approximation ratio assuming a particular variant of the Unique Games Conjecture [21]. Chudak and Shmoys [9] developed a polynomial-time $O(\log m)$-approximation algorithm for $Q \mid prec \mid C_{\max}$ while Chekuri and Bender [7] proposed a faster polynomial-time approximation algorithm with the same order of worst-case performance. Note that no approximation algorithm exists for $R \mid prec \mid C_{max}$, while algorithms of polylogarithmic approximation ratios are proposed by Shmoys et al. [19] for the special case of $R \mid chain \mid C_{max}$ and by Kumar et al. [17] for $R \mid forest \mid C_{max}$.

For hybrid architectures, a 6-approximation algorithm was proposed by Kedad-Sidhoum et al. [16]. In the case of independent tasks there is a $(\frac{4}{3} + \frac{1}{3k})$-approximation algorithm [5] based on dynamic programming. Several other faster algorithms have been proposed, achieving approximation ratio of 2 [6] and 3.41 [4]. If the tasks arrive in an on-line order, a 4-competitive algorithm was presented by Chen et al. [8] for hybrid architectures without precedence relations.

A closely related problem, in which the architecture consists of $Q \geq 2$ different types of resources and each task can be executed only on some of them, has also been studied in the literature. This problem generalizes the dedicated processors case if each processor consists of several identical cores, while a $(Q + 1)$-approximation algorithm has been proposed for it [18]. Note that given an allocation, the problem of scheduling in hybrid machines reduces to the above generalized dedicated processors problem.

On a more practical side, there exist some work about off-line scheduling, such as the well-known algorithm HEFT introduced by Topcuoglu et al. [22], which has been implemented on the run-time system starPU [3].

In the case of independent tasks, a systematic comparison of various heuristics has been performed by Braun et al. [13]. Specifically, the authors examined 11 different heuristics and provide a good basis for comparison and insights on circumstances why a technique outperforms another. Finally, experimental comparisons of the algorithms in the more recent literature for independent tasks can be found [4, 5, 6].

3 Preliminaries

In this section we briefly present the two basic existing approaches for scheduling on heterogeneous/hybrid platforms and we discuss their theoretical efficiency by presenting lower bounds on their performance.

The first approach is the scheduling-oriented algorithm HEFT [22]. According to HEFT, the tasks are initially prioritized with respect to their precedence relations and their average processing times. Then, following this priority, tasks are scheduled with possible backfilling on the available pair of processor and time interval in which they feasibly complete as early as possible. Note that HEFT is a heuristic that works for platforms with several heterogeneous resources and also takes into account possible communication costs. However, even for the simpler setting without communication costs, with only two types of resources and $k = 1$, HEFT cannot have a worst-case approximation guarantee better than $\frac{m}{2}$ [5]. This

result depends only on the number of CPUs, since the example provided uses just one GPU. The following theorem slightly improves the above result for the case of a single GPU. More interestingly, it expresses the lower bound to the approximation ratio of HEFT using both the number of CPUs and of GPUs.

Theorem 1. *For any $k \leq \sqrt{m}$, the worst-case approximation ratio for HEFT is at least $\frac{m+k}{k^2}\left(1-\frac{1}{e^k}\right)$, even in the hybrid model with independent tasks.*

Proof. We describe an instance that consists of independent tasks, and hence no communication costs are defined. We also consider the hybrid platform model where we only have a set of m identical CPUs and a set of k identical GPUs. Then, the rank of each task $T_j \in \mathcal{T}$ computed by HEFT is simplified as follows

$$rank(T_j) = \frac{m\overline{p_j} + k\underline{p_j}}{m+k}$$

HEFT considers the tasks in non-increasing order with respect to their rank and assigns each task to the CPU or GPU where its completion time is minimized. In case of ties, we assume, without loss of generality, that HEFT prefers to assign the task to a GPU, while it chooses arbitrarily between CPUs or GPUs. Notice that, since all tasks are independent, no idle times are introduced in the schedule.

Our instance consists of $2m$ sets of $km + m^2$ tasks in total, as shown in Table 1.

Sets of tasks	# tasks per set	$\overline{p_j}$	$\underline{p_j}$
A_i, $1 \leq i \leq m$	k	$\left(\frac{m}{m+k}\right)^i$	$\left(\frac{m}{m+k}\right)^i$
B_i, $1 \leq i \leq m$	m	$\left(\frac{m}{m+k}\right)^i$	$\frac{k}{m^2}\left(\frac{m}{m+k}\right)^m$

Table 1: Sets of tasks composing the instance for which HEFT achieves an approximation ratio of $\frac{m+k}{k^2}\left(1-\frac{1}{e^k}\right)$.

The rank of each task $T_j \in A_i$, $1 \leq i \leq m$, is

$$rank(T_j) = \frac{(m+k)\left(\frac{m}{m+k}\right)^i}{m+k}$$

while the rank of each task $T_j \in B_i$, $1 \leq i \leq m$, is

$$rank(T_j) = \frac{m\left(\frac{m}{m+k}\right)^i + \frac{k^2}{m^2}\left(\frac{m}{m+k}\right)^m}{m+k}$$

According to the above ranks, HEFT will schedule all tasks in A_{i+1} (resp. B_{i+1}) after all tasks in A_i (resp. B_i), $1 \leq i \leq m-1$. Moreover,

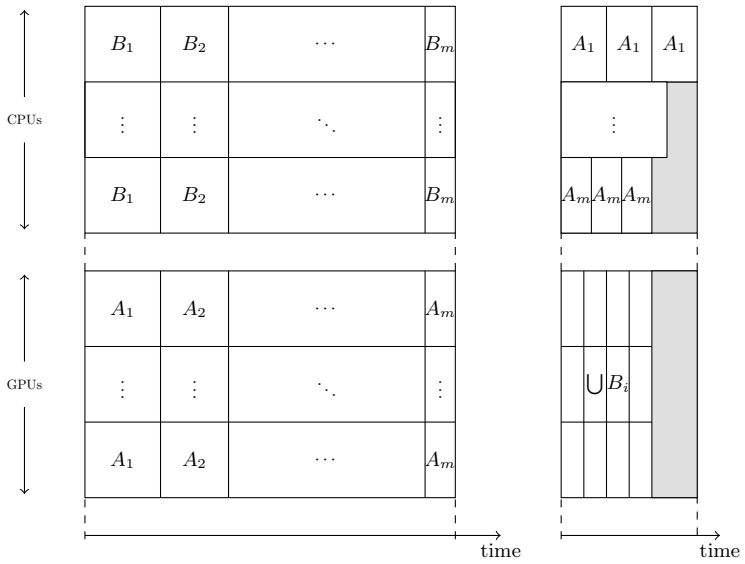


Figure 1: Possible schedule of HEFT (left) and optimal schedule (right). Notice that the gray area represents idle time.

for any $T_j \in A_i$ and $T_{j'} \in B_i$, $1 \leq i \leq m$, we have

$$\begin{aligned}
&(m+k)\left(rank(T_j) - rank(T_{j'})\right) \\
&= (m+k)\left(\frac{m}{m+k}\right)^i - m\left(\frac{m}{m+k}\right)^i - \frac{k^2}{m^2}\left(\frac{m}{m+k}\right)^m \\
&= k\left(\left(\frac{m}{m+k}\right)^i - \frac{k}{m^2}\left(\frac{m}{m+k}\right)^m\right) \\
&\geq k\left(\left(\frac{m}{m+k}\right)^m - \frac{k}{m^2}\left(\frac{m}{m+k}\right)^m\right) > 0
\end{aligned}$$

where the last inequality holds since $k \leq m$. For any $T_j \in B_i$ and $T_{j'} \in A_{i+1}$, $1 \leq i \leq m-1$, we have

$$\begin{aligned}
&(m+k)\left(rank(T_j) - rank(T_{j'})\right) \\
&= m\left(\frac{m}{m+k}\right)^i + \frac{k^2}{m^2}\left(\frac{m}{m+k}\right)^m - (m+k)\left(\frac{m}{m+k}\right)^{i+1} \\
&> \left(\frac{m}{m+k}\right)^i \left(m - (m+k)\frac{m}{m+k}\right) = 0
\end{aligned}$$

Based on the above, HEFT will consider the sets of tasks according to the following order

$$A_1 \prec B_1 \prec A_2 \prec B_2 \prec \cdots \prec A_i \prec B_i \prec A_{i+1} \prec \cdots \prec A_m \prec B_m$$

Initially, HEFT will schedule the k tasks in A_1 in a different GPU. Hence, to minimize the completion times of the m tasks in B_1, each one

should be scheduled on a different CPU. Note that, all tasks in $A_1 \cup B_1$ finish at the same time, i.e., at time $\frac{m}{m+k}$. Similarly, the tasks in A_2 will be scheduled on a different GPU, the tasks in B_2 on a different CPU, and all of them will finish at the same time, i.e., at time $\frac{m}{m+k} + \left(\frac{m}{m+k}\right)^2$. The scheduling procedure continues in the same way for the tasks in the remaining sets. The left-hand side of Figure 1 shows a schedule produced by HEFT. In this schedule, all machines finish their execution at time

$$\sum_{i=1}^{m} \left(\frac{m}{m+k}\right)^i = \frac{1 - \left(\frac{m}{m+k}\right)^{m+1}}{1 - \frac{m}{m+k}} - 1 \simeq \frac{1 - \frac{m}{m+k}\frac{1}{e^k}}{1 - \frac{m}{m+k}} - 1$$
$$= \frac{(m+k)e^k - m}{ke^k} - 1 = \frac{me^k - m}{ke^k} = \frac{m}{k}\left(1 - \frac{1}{e^k}\right)$$

On the other hand, we can create a schedule of makespan at most $\frac{km}{m+k}$. To see this, we assign all tasks of A_i, $1 \leq i \leq m$, on CPU i, while to each of the k GPUs we assign $\frac{m^2}{k}$ different tasks of $\bigcup_{i=1}^{m} B_i$. The right-hand side of Figure 1 visualizes such a schedule, whose makespan is dominated either by the load of CPU 1 or by the load of any of the GPUs. Specifically, the makespan will be equal to

$$\max\left\{k\left(\frac{m}{m+k}\right), \frac{m^2}{k}\frac{k}{m^2}\left(\frac{m}{m+k}\right)^m\right\} \leq \frac{km}{m+k}$$

Since an optimal schedule could have an even smaller makespan the theorem follows. □

The second approach is proposed by Kedad-Sidhoum et al. [16] and distinguishes the allocation and the scheduling decisions. For the allocation phase, an integer linear program is proposed which decides the allocation of tasks to the CPU or GPU side by optimizing the standard lower bounds for the makespan of a schedule which are proposed by Graham [15], namely the critical path and the load. To present this integer linear program, let x_j be a binary variable which is equal to 1 if a task T_j is assigned to the CPU side, and zero otherwise. Let also C_j be a variable that indicates the completion time of T_j and λ the variable that corresponds to the maximum over all lower bounds used., i.e., to a lower bound of the makespan. Then, the Heterogeneous Linear Program (HLP)

is as follows:

$$\text{minimize } \lambda$$

$$C_i + \overline{p_j} x_j + \underline{p_j}(1 - x_j) \leq C_j \qquad \forall T_j \in \mathcal{T}, T_i \in \Gamma^-(T_j) \qquad (1)$$

$$\overline{p_j} x_j + \underline{p_j}(1 - x_j) \leq C_j \qquad \forall T_j \in \mathcal{T} : \Gamma^-(T_j) = \emptyset \qquad (2)$$

$$C_j \leq \lambda \qquad \forall T_j \in \mathcal{T} \qquad (3)$$

$$\frac{1}{m} \sum_{T_j \in \mathcal{T}} \overline{p_j} x_j \leq \lambda \qquad (4)$$

$$\frac{1}{k} \sum_{T_j \in \mathcal{T}} \underline{p_j}(1 - x_j) \leq \lambda \qquad (5)$$

$$x_j \in \{0, 1\} \qquad \forall T_j \in \mathcal{T} \qquad (6)$$

$$C_j \geq 0 \qquad \forall T_j \in \mathcal{T}$$

Constraints (1), (2) and (3) describe the critical path, while Constraints (4) and (5) impose that the makespan cannot be smaller than the average load on CPU and GPU sides. Note that the particular problem of deciding the allocation to minimize the maximum over the three lower bounds is NP-hard, since it is a generalization of the PARTITION problem to which reduces if all tasks are independent, $m = k$, and $\overline{p_j} = \underline{p_j}$ for each T_j.

After relaxing the integrality Constraint (6), a fractional allocation can be found in polynomial time. To get an integral solution, the variables x_j are rounded as follows: if $x_j \geq \frac{1}{2}$ then T_j is assigned to the CPU side, otherwise T_j is assigned to the GPU side.

Finally, the Earliest Starting Time (EST) policy is applied for scheduling the tasks: at each step, the ready task with the earliest possible starting time is scheduled respecting the precedence relations and the decided allocation. We call this algorithm HLP-EST.

HLP-EST achieves an approximation ratio of 6 [16]. The following theorem shows that this ratio is tight.

Theorem 2. *There is an instance for which HLP-EST achieves an approximation ratio of* $6 - O(\frac{1}{m})$*. Hence, the ratio for HLP-EST is tight.*

Proof. Consider a hybrid system with an equal number of CPUs and GPUs, i.e, $m = k$. The instance consists of $2m + 3$ tasks that are partitioned into 3 sets as shown in Table 2.

Sets of tasks	# tasks per set	$\overline{p_j}$	$\underline{p_j}$
A	1	$\frac{m(2m+1)}{m-1}$	∞
B_1	$2m+1$	$2m-1$	1
B_2	$2m+1$	1	$2m-1$

Table 2: Sets of tasks composing the instance for which HLP-EST achieves an approximation ratio of $6 - O(\frac{1}{m})$.

The only precedence relations exist between tasks of B_1 and B_2. Specifically, for each task $T_j \in B_2$ we have that $\Gamma^-(T_j) = B_1$, that is no task in B_2 can be executed before the completion of all tasks in B_1. Note that there are no precedences between tasks of the same set.

Any optimal solution of the relaxed HLP for the above instance will assign the task T_A on a CPU, i.e., $x_A = 1$. Hence, the objective value of any optimal solution will be at least $\frac{m(2m+1)}{m-1}$ due to Constraints (2) and (3). The following technical proposition shows that an optimal solution for the relaxed HLP has exactly this objective value, by describing a feasible fractional assignment for the remaining tasks.

Proposition 1. *There is a small constant $\epsilon > 0$ for which the assignment $x_A = 1$, $x_j = \frac{1}{2}$ for each $T_j \in B_1$, $x_j = \frac{1}{2} - \epsilon$ for each $T_j \in B_i$, and $\lambda = \frac{m(2m+1)}{m-1}$ corresponds to a feasible solution for the relaxed HLP.*

Proof. We will show that every constraint of the relaxed HLP is satisfied by the assignment of the binary variables x_j proposed in the statement and by setting $\lambda = \frac{m(2m+1)}{m-1}$. But before this, we need to feasibly define C_j, for each $T_j \in \mathcal{T}$, based on Constraints (1) and (2).

For the task T_A, we set

$$C_A = \frac{m(2m+1)}{m-1}$$

for each task $T_j \in B_1$, we set

$$C_j = \frac{1}{2}(2m-1) + \frac{1}{2} = m$$

while for each task $T_j \in B_2$, we set

$$C_j = m + (\frac{1}{2} - \epsilon) + (\frac{1}{2} + \epsilon)(2m-1) = 2m + 2\epsilon(m-1)$$

satisfying by definition Constraints (1) and (2).

To show the feasibility of Constraint (3), it suffices to prove it for T_A as well as for a task $T_j \in B_2$. For these cases, we have

$$C_A = \frac{m(2m+1)}{m-1} = \lambda$$
$$C_j = 2m + 2\epsilon(m-1) \leq \lambda$$

where the last inequality holds for arbitrarily small ϵ, and hence Constraint (3) is satisfied.

For Constraint (4), we have

$$\begin{aligned}\sum_{T_j \in \mathcal{T}} \overline{p_j} x_j &= \overline{p_A} x_A + \sum_{T_j \in B_1 \cup B_2} \overline{p_j} x_j \\ &= \frac{m(2m+1)}{m-1} + (2m+1)\frac{2m-1}{2} + (2m+1)(\frac{1}{2} - \epsilon) \\ &< \frac{m(2m+1)}{m-1} + m(2m+1) \\ &= m\frac{m(2m+1)}{m-1} = m\lambda\end{aligned}$$

and hence it is satisfied.

For Constraint (5), we have

$$\begin{aligned}\sum_{T_j \in \mathcal{T}} \underline{p_j}(1 - x_j) &= \underline{p_A}(1 - x_A) + \sum_{T_j \in B_1 \cup B_2} \underline{p_j}(1 - x_j) \\ &= 0 + (2m+1)\frac{1}{2} + (2m+1)(2m-1)(\frac{1}{2} + \epsilon) \\ &< m(2m+1) + \epsilon(4m^2 - 1) \\ &\leq m\lambda = k\lambda\end{aligned}$$

where the last inequality is true for an arbitrarily small ϵ, and hence the constraint is satisfied.

Concluding, all constraints are satisfied with $\lambda = \frac{m(2m+1)}{m-1}$, and thus the proposition holds. □

Given the optimal fractional assignment proposed above, HLP-EST will round the fractional variables and allocate the tasks as follows: the task T_A is assigned to the CPU side, each task $T_j \in B_1$ is assigned to the CPU side, and each task $T_j \in B_2$ is assigned to the GPU side. Then, HLP-EST schedules the tasks according to the EST policy. However, we will argue here for any scheduling policy and thus any possible schedule.

Assuming that an algorithm has scheduled the task T_A on any CPU during any interval $[t, t + \overline{p_A})$ and $m \geq 3$, there is only one meaningful family of schedules for the tasks in $B_1 \cup B_2$. Specifically, the $2m+1$ tasks of B_1 will be scheduled during the interval $[0, 3(2m-1))$ on the m CPUs, while at least one of them completes at time $3(2m-1)$. Then, the $2m+1$ tasks of B_2 will be scheduled during the interval $[3(2m-1), 6(2m-1))$ on the $k = m$ GPUs, while at least one of them completes at time $6(2m-1)$. Clearly, we should define t such that $t + \overline{p_A} \leq 6(2m-1)$. An illustration of the above schedule is given in Figure 2.

The makespan of the created schedule is equal to $6(2m-1)$, while Proposition 1 implies a feasible solution for the relaxed HLP of objective value $\frac{m(2m+1)}{m-1}$. Hence, the approximation ratio achieved for this instance is

$$\frac{6(2m-1)}{\frac{m(2m+1)}{m-1}} = 6 - O\left(\frac{1}{m}\right)$$

and the theorem follows. □

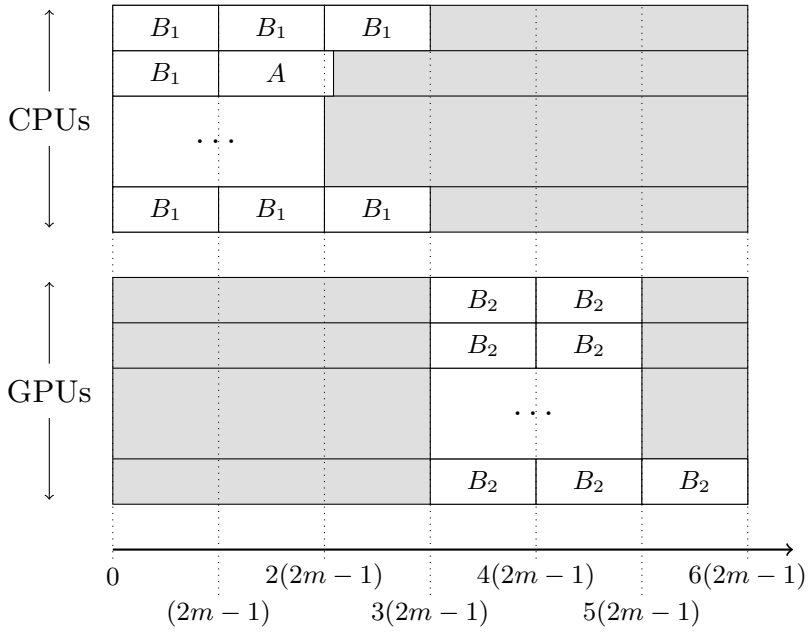


Figure 2: Resulting schedule of HLP-EST for the proposed instance. Notice that the gray areas represent idle times.

Note that the proof of the previous theorem implies a stronger result since the worst case example does not depend on which scheduling policy will be applied after the allocation step, and hence the following corollary holds.

Corollary 1. *Any scheduling policy which is applied after the allocation decisions taken by the rounding of an optimal solution of the relaxed HLP leads to an approximation algorithm of ratio at least* $6 - O(\frac{1}{m})$.

4 Algorithms

In this section we propose algorithms for both the off-line and on-line settings of the addressed problem.

4.1 Off-line setting

We propose in the following a new scheduling policy which prioritizes the tasks based on the solution obtained for HLP after the rounding step. The motivation of assigning priorities to the tasks is for taking into account the precedence relations between them. More specifically, we want to prioritize the scheduling of *critical tasks*, i.e., the tasks on the critical path, before the remaining (less critical) tasks.

To do this, we define for each task T_j a rank $Rank(T_j)$ in the same sense as in HEFT. However, in our case, the rank of each task depends on the allocation given by HLP, while in HEFT it depends on the average processing time of the task. Specifically, the rank of each task T_j is computed after the rounding operation of the assignment variable x_j and corresponds to the length, in the sense of processing time, of the longest path between this task and its last descendant in the precedence graph. Thus, each task will have a larger rank than all its descendants. The rank of the task T_j is recursively defined as follows:

$$Rank(T_j) = \overline{p_j}x_j + \underline{p_j}(1 - x_j) + \max_{i \in \Gamma^+(T_j)} \{Rank(T_i)\}$$

After ordering the tasks in non-increasing order with respect to their ranks, we apply the standard List Scheduling algorithm adapted to two types of resources and taking into account the rounding of the assignment variables x_j. We call the above described policy Ordered List Scheduling (OLS), while the newly defined algorithm (including the allocation) is denoted by HLP-OLS.

Although this policy performs well in practice, as we will see in Section 7, its approximation ratio cannot be better than 6 due to the lower bound presented in Theorem 2. On the other hand, it is quite easy to see that HLP-EST and HLP-OLS have the same approximation ratio by following the same reasoning as in Lemmas 4 and 5 of Kedad-Sidhoum et al. [16].

Consider the schedule produced by HLP-OLS and partition the time interval $I = [0, C_{max})$ into two subsets I_{CP} and I_W. The set I_{CP} contains every time slot where at least one processor of each type is idle, while the set I_W consists of the remaining time slots in I, i.e., $I_W = I \backslash I_{CP}$. We then can divide the set I_W into two possibly non-disjoint subsets I_{CPU} (resp. I_{GPU}) containing the time slots where all the CPUs (resp. GPUs) are busy. Denoting by $|I|$ the number of unitary time slots in an interval I we have

$$|I_{CP}| \leq CP$$
$$|I_{CPU}| \leq \frac{W_{CPU}}{m}$$
$$|I_{GPU}| \leq \frac{W_{GPU}}{k}$$

where CP, W_{CPU} and W_{GPU} denote respectively the length of the critical path, the total work load on all CPUs and the total work load on all GPUs based on the assignment decided by the rounding of the fractional solution of HLP. Because of the rounding, each of these three quantities are bounded above by twice the objective value λ of the optimal solution of HLP. Since λ is smaller than the feasible optimal makespan C^*_{max} we

deduce the following bound for the makespan of HLP-OLS:

$$\begin{aligned} C_{max} &\leq |I_{CPU}| + |I_{GPU}| + |I_{CP}| \\ &\leq \frac{W_{CPU}}{m} + \frac{W_{GPU}}{k} + CP \\ &\leq 6C^*_{max} \end{aligned}$$

Corollary 2. *HLP-OLS achieves an approximation ratio of 6. This ratio is tight.*

4.2 On-line setting

In the HLP-EST algorithm, an integer linear program is used to find an efficient allocation of each task to the CPU or GPU side. Although this program optimizes the classical lower bounds for the makespan, and hence informally optimizes the allocation, the resolution of its relaxation has a high complexity in practice and cannot be used in an on-line setting.

In what follows, we present an algorithm for the on-line scheduling problem in the hybrid context. Our algorithm first decides the allocation of a task to a resource type upon its arrival by using a set of rules. These rules take into account both the actual schedule and the relation between the processing times of a task, in a similar way as in the 4-competitive algorithm proposed by Chen et al. [8] for the on-line problem with independent tasks. Then, a List Scheduling is applied to schedule each task respecting the decided allocation and precedence relations.

To describe the new rules, we define τ_{gpu} to be the earliest time when at least one GPU is idle. Let also $R_{j,gpu} = \max\{\tau_{gpu}, \max_{T_i \in \Gamma^-(T_j)}\{C_i\}\}$ be the *ready time* of T_j for GPUs, i.e., the earliest time at which T_j can be executed on a GPU. Then, the set of rules is defined as follows:

Step 1: If $\overline{p_j} \geq R_{j,gpu} + \underline{p_j}$ then assign T_j to the GPU side.

Step 2: Else if $\frac{\overline{p_j}}{\sqrt{m}} \leq \frac{\underline{p_j}}{\sqrt{k}}$ then assign T_j to the CPU side. Otherwise assign T_j to the GPU side.

This set of rules is combined with a greedy List Scheduling policy that schedules each task as early as possible on the CPU or GPU side already decided by the rules. We call the algorithm obtained by this combination ER-LS (Enhanced Rules - List Scheduling). In the following, we give upper and lower bounds for the competitive ratio of ER-LS.

Theorem 3. *ER-LS achieves a competitive ratio of* $4\sqrt{\frac{m}{k}}$.

Proof. Let W_{CPU}, W_{GPU} and CP be the total load on all CPUs, the total load on all GPUs and the length of the critical path produced by the allocation of tasks decided by the rules, respectively. With the same reasoning as in Section 4.1, for the makespan of the schedule produced by ER-LS we can observe that

$$C_{max} \leq \frac{W_{CPU}}{m} + \frac{W_{GPU}}{k} + CP \tag{7}$$

In the following, we bound the average load of both sides ($\frac{W_{CPU}}{m} + \frac{W_{GPU}}{k}$) by $3\sqrt{\frac{m}{k}}C^*_{max}$ and the length of the critical path by $\sqrt{\frac{m}{k}}C^*_{max}$. Recall that C^*_{max} denotes the makespan of the optimal off-line solution of the instance.

We denote by $\mathcal{SA}_{cpu}$ (resp. $\mathcal{SA}_{gpu}$) the set containing the tasks placed on the CPU (resp. GPU) side in both a solution of the algorithm and the optimal solution, by $\mathcal{SB}_{gpu}$ the set containing tasks placed by Step 1 on the GPU side in a solution of the algorithm but on the CPU side in the optimal solution, and by $\mathcal{SC}_{cpu}$ (resp. $\mathcal{SC}_{gpu}$) the set containing tasks placed by Step 2 on the CPU (resp. GPU) side in a solution of the algorithm but on the GPU (resp. CPU) side in the optimal solution.

We also denote by sa_{cpu}, sa_{gpu}, sb_{gpu}, sc_{cpu} and sc_{gpu} the sum of processing times of all tasks in the sets $\mathcal{SA}_{cpu}$, $\mathcal{SA}_{gpu}$, $\mathcal{SB}_{gpu}$, $\mathcal{SC}_{cpu}$ and $\mathcal{SC}_{gpu}$, respectively. Note that we use here the processing times according to the allocation of ER-LS.

Bounding the loads.
Consider T_{j_0} to be the last finishing task in $\mathcal{SB}_{gpu}$. Since the task is scheduled according to Step 1, we know that $\overline{p_{j_0}} \geq R_{j_0,gpu} + \underline{p_{j_0}} \geq \frac{sb_{gpu}}{k}$. We also know that T_{j_0} is scheduled on a CPU in the optimal solution so we have $\overline{p_{j_0}} \leq C^*_{max}$ and thus: $\frac{sb_{gpu}}{k} \leq C^*_{max}$.

Each task in $\mathcal{SC}_{gpu}$ is scheduled on the CPU side in the optimal solution. According to Step 2, the total processing times of tasks in $\mathcal{SC}_{gpu}$ in the optimal solution is at least $\sqrt{\frac{m}{k}}sc_{gpu}$, so we have for the cpu side $\frac{sa_{cpu}+\sqrt{\frac{m}{k}}sc_{gpu}}{m} \leq C^*_{max}$. The same reasoning for the GPU side gives $\frac{sa_{gpu}+\sqrt{\frac{k}{m}}sc_{gpu}}{k} \leq C^*_{max}$.

By adding the three inequalities we have the following:

$$\frac{sb_{gpu}}{k} + \frac{sa_{cpu} + \sqrt{\frac{m}{k}}sc_{gpu}}{m} + \frac{sa_{gpu} + \sqrt{\frac{k}{m}}sc_{cpu}}{k} \leq 3C^*_{max}$$

By separating the loads on CPU and on GPU on the left-hand side of the above inequality and taking into account that $m \geq k$ we have

$$\frac{sa_{cpu}}{m} + \frac{sc_{cpu}}{\sqrt{mk}} \geq \frac{sa_{cpu} + sc_{cpu}}{m} \geq \sqrt{\frac{k}{m}}\frac{sa_{cpu} + sc_{cpu}}{m}$$

and

$$\frac{sa_{gpu} + sb_{gpu}}{k} + \frac{sc_{gpu}}{\sqrt{mk}} \geq \frac{sa_{gpu} + sb_{gpu}}{k} + \frac{sc_{gpu}}{k}\sqrt{\frac{k}{m}} \geq \sqrt{\frac{k}{m}}\frac{sa_{gpu} + sb_{gpu} + sc_{gpu}}{k}$$

Summing these two bounds we finally obtain

$$\sqrt{\frac{k}{m}}\left(\frac{sa_{cpu} + sc_{cpu}}{m} + \frac{sa_{gpu} + sb_{gpu} + sc_{gpu}}{k}\right) \leq 3C^*_{max}$$

and thus

$$\frac{W_{CPU}}{m} + \frac{W_{GPU}}{k} \leq 3\sqrt{\frac{m}{k}}C^*_{max} \tag{8}$$

Bounding the critical path.
Consider the sets $\mathcal{SA}^{CP}_{cpu} \subseteq \mathcal{SA}_{cpu}$, $\mathcal{SA}^{CP}_{gpu} \subseteq \mathcal{SA}_{gpu}$, $\mathcal{SB}^{CP}_{gpu} \subseteq \mathcal{SB}_{gpu}$, $\mathcal{SC}^{CP}_{cpu} \subseteq \mathcal{SC}_{cpu}$ and $\mathcal{SC}^{CP}_{gpu} \subseteq \mathcal{SC}_{gpu}$ to be the sets containing only the tasks belonging to the critical path obtained by the algorithm, with the same notation in lower case for the sum of processing times of all tasks in each set and the same notation with a star * for the sum of processing times of all tasks in the optimal solution.

For the sets $\mathcal{SA}^{CP}_{cpu}$ and $\mathcal{SA}^{CP}_{gpu}$, by definition, we have

$$sa^{CP}_{cpu} + sa^{CP}_{gpu} = sa^{CP^*}_{cpu} + sa^{CP^*}_{gpu}$$

According to Step 1, every task in $\mathcal{SB}^{CP}_{gpu}$ has a processing time smaller than that in the optimal solution, so $sb^{CP}_{gpu} \leq sb^{CP^*}_{gpu}$. According to Step 2, every task T_j in $\mathcal{SC}^{CP}_{cpu}$ (resp. $\mathcal{SC}^{CP}_{gpu}$) verifies $\overline{p_j} \leq \sqrt{\frac{m}{k}}\underline{p_j}$ (resp. $\underline{p_j} \leq \sqrt{\frac{k}{m}}\overline{p_j}$), so we have $sc^{CP}_{cpu} \leq \sqrt{\frac{m}{k}} sc^{CP^*}_{cpu}$ and $sc^{CP}_{gpu} \leq \sqrt{\frac{m}{k}} sc^{CP^*}_{gpu}$.

By summing the previous inequalities for the critical path we get

$$\begin{aligned} CP &= sa^{CP}_{cpu} + sa^{CP}_{gpu} + sb^{CP}_{gpu} + sc^{CP}_{cpu} + sc^{CP}_{gpu} \\ &\leq \sqrt{\frac{m}{k}}(sa^{CP^*}_{cpu} + sa^{CP^*}_{gpu} + sb^{CP^*}_{gpu} + sc^{CP^*}_{cpu} + sc^{CP^*}_{gpu}) \leq \sqrt{\frac{m}{k}} CP^* \end{aligned}$$

Since $CP^* \leq C^*_{max}$, we have $CP \leq \sqrt{\frac{m}{k}} C^*_{max}$ and, combining this inequality with Equations (7) and (8), the theorem follows.

□

As the following theorem shows, the competitive ratio of ER-LS is almost tight and we cannot expect a much better analysis for its upper bound.

Theorem 4. *There is an instance for which ER-LS achieves a competitive ratio of* $\sqrt{\frac{m}{k}}$.

Proof. Consider a hybrid system with m CPUs and $k \leq m$ GPUs. The instance consists of $m + k$ tasks that are partitioned into 2 sets as shown in Table 3.

Type	Number of task	Processing time on CPU/GPU
A	k	$\sqrt{m}$ / $\sqrt{m}$
B	m	$\sqrt{m}$ / $\sqrt{k}$

Table 3: Sets of tasks composing the instance for which ER-LS achieves a competitive ratio of $\sqrt{\frac{m}{k}}$.

The k tasks of type A are independent to each other and the m tasks of type B are with precedence constraints as follows:

$$B_1 \prec B_2 \prec \cdots \prec B_m$$

The tasks are ordered in a list by first taking all tasks of type A and then the tasks of type B respecting the precedences.

The ER-LS algorithm will first place the k tasks of type A on a GPU according to Step 1. The completion time of these tasks is then $\sqrt{m}$. Then, since $\sqrt{m} \leq \sqrt{m} + \sqrt{k}$, the task B_1 will be placed on a CPU according to Step 2, with completion time $\sqrt{m}$. The task B_2 will also be placed on a CPU according to Step 2, starting at time $\sqrt{m}$ and completing at time $2\sqrt{m}$. With the same reasoning, each task B_i, $i \in \{1, m\}$ is placed on a CPU according to Step 2 starting at time $(i-1)\sqrt{m}$ and completing at time $i\sqrt{m}$.

Thus, the schedule produced by ER-LS for this instance has a makespan of $C_{max} = m\sqrt{m}$.

An optimal schedule would have all tasks of type A placed on the CPU side with a completion time for each task of $\sqrt{m}$. The tasks of type B would be placed on the GPU side with a completion time for each task B_i, $i \in \{1, m\}$, of $i\sqrt{k}$. Thus, the optimal makespan is $C^*_{max} = m\sqrt{k}$.

Hence, ER-LS achieves a competitive ratio $\frac{C_{max}}{C^*_{max}} = \sqrt{\frac{m}{k}}$ for this instance and the theorem holds. □

5 Generalization on Q Resource Types

In this section we generalize the addressed scheduling problem for $Q \geq 2$ different types of identical processors. Specifically, we explain how to extend HLP-EST and HLP-OLS, which are designed for the case $Q = 2$, to handle more types of computing resources. In this direction, we first present a modified linear program and a new rounding method to decide the allocation for $Q \geq 2$. Then, we explain the modifications made on the scheduling policy OLS. Note that the EST policy is directly generalized for multiple resource types. Finally, we give an upper bound on the approximation ratio of these two new algorithms.

Before continuing, we need some additional notations. Let M_q be the set of processors of type q, $1 \leq q \leq Q$, and $m_q = |M_q|$ its size. The execution of a task $T_j \in \mathcal{T}$ on a processor of type q, $1 \leq q \leq Q$, takes $p_{j,q}$ time units.

In what follows, we extend the linear program HLP to take into account more resource types. To do this, we replace the binary variable x_j by $x_{j,q}$ which indicates if the task $T_j \in \mathcal{T}$ is assigned to the resource type q. As before, let C_j be a variable corresponding to the completion time of T_j and λ be the variable that represents a lower bound to the makespan. Then, we consider the following modification of HLP, which

we call QHLP:

$$\text{minimize } \lambda$$

$$C_i + \sum_{q=1}^{Q} p_{j,q} x_{j,q} \leq C_j \qquad \forall T_j \in \mathcal{T}, T_i \in \Gamma^-(T_j) \tag{9}$$

$$\sum_{q=1}^{Q} p_{j,q} x_{j,q} \leq C_j \qquad \forall T_j \in \mathcal{T} : \Gamma^-(T_j) = \emptyset \tag{10}$$

$$C_j \leq \lambda \qquad \forall T_j \in \mathcal{T} \tag{11}$$

$$\frac{1}{m_q} \sum_{T_j \in \mathcal{T}} p_{j,q} x_{j,q} \leq \lambda \qquad 1 \leq q \leq Q \tag{12}$$

$$\sum_{q=1}^{Q} x_{j,q} = 1 \qquad \forall T_j \in \mathcal{T} \tag{13}$$

$$x_{j,q} \in \{0,1\} \qquad \forall T_j \in \mathcal{T},\ 1 \leq q \leq Q \tag{14}$$

$$C_j \geq 0 \qquad \forall T_j \in \mathcal{T}$$

$$\lambda \geq 0$$

The main difference here concerns Constraint (13) which assures that each task is integrally assigned to exactly one type of resources.

After relaxing the integrality Constraint (14) of QHLP, we can solve in polynomial time the obtained relaxation. To get an integral allocation, we assign each task T_j to the resource type q' for which the assignment variable $x_{j,q}$ has the greatest value, i.e., $q' = \text{argmax}_{1 \leq q \leq Q}\{x_{j,q}\}$. In other words, for such q' we set $x_{j,q'} = 1$ and $x_{j,q} = 0$ for any $q \neq q'$. In case of ties, we give priority to the resource type for which T_j has the smallest processing time. Once the assignment step is done, we use the Earliest Starting Time policy taking into account the precedence constraints as well as the allocation provided by the rounding of $x_{j,q}$ variables. We call this algorithm QHLP-EST.

For the scheduling policy OLS, we modify the computation of the rank for each task as follows:

$$Rank(T_j) = \sum_{q=1}^{Q} p_{j,q} x_{j,q} + \max_{i \in \Gamma^+(T_j)} \{Rank(T_i)\}$$

The algorithm HLP-OLS can be extended using the linear program QHLP and the new rounding method to get an integral allocation of the tasks. We then obtain the algorithm QHLP-OLS using the modified OLS policy.

In the following, we show that the approximation ratio of QHLP-EST is $Q(Q+1)$ and that this ratio is tight. Note that, as in the case $Q = 2$ presented in Section 4.1, the same reasoning can be applied for QHLP-OLS leading to the same result.

Theorem 5. *QHLP-EST achieves an approximation ratio of $Q(Q+1)$. This ratio is tight.*

Proof. We analyze the structure of a schedule produced by the algorithm to give an upper bound on the approximation ratio.

We denote by W_q, $1 \leq q \leq Q$, the total load on all processors of type q in the obtained schedule. We also denote by C^R_{max}, W^R_q and L^R the objective value, the total load on all processors of type q and the length of the longest path in the fractional optimal solution of the relaxed QHLP, respectively. Finally, we define by $C^*_{\max}$ the optimal makespan over all feasible schedules for our problem. Then, the following inequalities hold.

$$L^R \leq C^R_{\max} \leq C^*_{\max} \tag{15}$$

$$\frac{W^R_q}{m_q} \leq C^R_{\max} \leq C^*_{\max},\ 1 \leq q \leq Q \tag{16}$$

To analyze the structure of the schedule, we partition the time interval of the schedule $I = [0, C_{max})$ into two disjoint subsets of intervals I_{CP} and I_W. The set I_{CP} contains every time slot where at least one processor of each type is idle, while the set I_W consists of the remaining time slots in I, i.e., $I_W = I \backslash I_{CP}$. We then can divide the set I_W into Q, possibly non-disjoint, subsets I_q, $1 \leq q \leq Q$, which contain respectively every time slot where all processors of type q are busy. Henceforth, we denote by $|I|$ the length of I, i.e. the number of unitary time slots in I. Then, we have that

$$C_{\max} = |I| \leq |I_{CP}| + \sum_{q=1}^{Q} |I_q|$$

In the following, we will bound above by QC^*_{max} the length of the subset I_{CP} and each subset I_i, $1 \leq i \leq Q$.

Due to the rounding policy, we know that if $x_{j,q} = 1$ then $x^R_{j,q} \geq \frac{1}{Q}$. Hence, we have

$$x_{j,q} \leq Q \cdot x^R_{j,q} \quad \forall T_j \in \mathcal{T}, 1 \leq q \leq Q \tag{17}$$

Consider first the subset of intervals I_{CP}. There is a directed path $\mathcal{P}$ of tasks being executed during any time slot in I_{CP}. The construction of $\mathcal{P}$ is the same as described by Graham [15] and Kedad-Sidhoum *et al.* [16]. Since the directed path $\mathcal{P}$ covers every time slot in I_{CP}, the length of I_{CP} is smaller than the length of $\mathcal{P}$ and the length of $\mathcal{P}$ in the optimal solution of $QHLP$, noted $\mathcal{P}^R$, is smaller than L^R. Thus, using the inequalities (15) and (17), we have the following bound:

$$\begin{aligned} |I_{CP}| \leq |\mathcal{P}| &\leq \sum_{j \in \mathcal{P}} \sum_{q=1}^{Q} p_{j,q} x_{j,q} \\ &\leq Q \sum_{j \in \mathcal{P}} \sum_{q=1}^{Q} p_{j,q} x^R_{j,q} = Q \cdot |\mathcal{P}^R| \\ &\leq Q \cdot L^R \leq Q \cdot C^*_{\max} \end{aligned}$$

Consider now each subset I_q, $1 \leq q \leq Q$. For each time slot in I_q all processors of type q are busy, so $|I_q|$ is smaller than the average load on

all the processors of type q. Using the inequalities (16) and (17), we have the following bounds:

$$\begin{aligned} |I_q| \leq \frac{W_q}{m_q} &\leq \frac{1}{m_q} \sum_{x_{j,q}=1} p_{j,q} \\ &\leq \frac{Q}{m_q} \sum_{j \in V} p_{j,q} x^R_{j,q} \\ &\leq Q \cdot \frac{W^R_q}{m_q} \leq Q \cdot C^*_{\max} \end{aligned}$$

Thus, by combining the calculated bounds we get

$$\begin{aligned} C_{\max} = |I| &\leq |I_{CP}| + \sum_{q=1}^{Q} |I_q| \\ &\leq Q(Q+1) C^*_{max} \end{aligned}$$

The tightness comes directly from Theorem 2, and hence the theorem follows. □

6 Benchmark Creation

In this section we describe the generation of a general benchmark for evaluating scheduling algorithms for the addressed problem. The benchmark is composed of 5 applications generated by *Chameleon*, a dense linear algebra software which is part of the MORSE project [12], and a more irregular application (*fork-join*) generated using *GGen*, a library for generating directed acyclic graphs [10].

The applications of *Chameleon*, named *getrf*, *posv*, *potrf*, *potri* and *potrs*, are composed of multiple sequential basic tasks of linear algebra. Different number, denoted by *nb_blocks*, and sizes, denoted by *block_size*, of sub-matrices have been used for the applications; specifically we set $nb_blocks \in \{5, 10, 20\}$ and $block_size \in \{64, 128, 320, 512, 768, 960\}$, for a total of 18 instances per application. Table 4 shows the total number of tasks for each application and each value of *nb_blocks*. Notice that the value of *block_size* does not impact the number of tasks.

For the setting with 2 resource types, the applications were executed with the runtime StarPU [3] on a machine with two Dual core Xeon E7 v2 with a total of 10 physical cores with hyper-threading of 3 GHz and 256 GB of RAM. The machine had 4 GPUs NVIDIA Tesla K20 (Kepler architecture) with each 5 GB of memory and 200 GB/s of bandwidth.

For 3 resource types, the applications were executed with the runtime StarPU on an Intel Dual core i7-5930k machine with a total of 6 physical cores with hyper-threading of 3.5 GHz and 12 GB of RAM. This machine had 2 NVIDIA GPUs: a GeForce GTX-970 (Maxwell architecture) with 4 GB of memory and 224 GB/s of bandwidth; and a Quadro K5200 (Kepler architecture) with 8 GB of memory and 192 GB/s of bandwidth. We forced each task to run first on CPU and then on GPU (or on both GPUs for the case with 3 resource) and stored the processing times for each type

Nb_blocks \ Apps	getrf_nopiv	posv	potrf	potri	potrs
5	55	65	35	105	30
10	385	330	220	660	110
20	2870	1960	1540	4620	420

Table 4: Number of tasks for each instance of the Chameleon applications

Nb_phases \ Width	100	200	300	400	500
2	203	403	603	803	1003
5	506	1006	1506	2006	2506
10	1011	2011	3011	4011	5011

Table 5: Number of tasks for each instance of the fork-join application.

of resource.

The *fork-join* application corresponds to a real situation where the execution starts sequentially and then forks to *width* parallel tasks. The results are aggregated by performing a join operation, completing a phase. This procedure can be repeated p times, the number of phases. For this benchmark, we set $p \in \{2, 5, 10\}$ and $width \in \{100, 200, 300, 400, 500\}$, for a total of 15 instances.

The processing time of each task on CPU was computed using a Gaussian distribution with center p and standard deviation $\frac{p}{4}$. To generate the processing times of a task on each of the two types of GPUs, we define the notion of *acceleration factor* with respect to the processing time on CPU already computed. Our goal is to create a more irregular application than the 5 from Chameleon to study the importance of the allocation decision. For this reason, 5% of the parallel tasks of each phase are highly decelerated when executed on a GPU by choosing uniformly at random an acceleration factor for each of them in $[0.1, 0.5]$. For each of the remaining tasks, we randomly chose an acceleration factor in $[0.5, 50]$, which corresponds to the range of acceleration factors observed for the 5 applications of Chameleon. Note that the 5% of highly decelerated tasks are separately selected at random for each of the two types of GPUs. Table 5 shows the total number of tasks for each instance of the fork-join application.

The data sets and other information are available under Creative Commons Public License[1].

[1]Hosted at: `https://github.com/marcosamaris/heterogeneous-SWF`, last visited on Mar. 2018.

7 Experiments

In this section we compare the performance of various scheduling algorithms by a simulation campaign using a benchmark composed of 6 parallel applications, presented in Section 6. First, we compare the off-line algorithms for the studied problem with 2 and 3 resource types. We then compare the on-line algorithms for 2 resource types.

7.1 Off-line setting

Algorithms and machine configurations.
We compared the performance, in terms of makespan, of HLP-OLS (Section 4.1) with HLP-EST (Section 3) and HEFT ([22]). The algorithms were implemented in Python (v. 2.7.6). The command-line *glpsol* (v. 4.52) solver of the GLPK package was used for the linear program. Each algorithm was implemented with a second version adapted for 3 types of resources, using the generalization of the algorithms presented in Section 5 for the 2 linear program-based algorithms. We denote by QHLP-EST, QHLP-OLS and QHEFT these algorithms for 3 resource types.

For the machine settings, we determined different sets of pairs (Nb_CPUs, Nb_GPUs). Specifically, we used 16, 32, 64 and 128 CPUs with 2, 4, 8 and 16 GPUs for a total of 16 machine settings for the case with 2 resource types. For the case with 3 resource types, we determined different sets of triplets (Nb_CPUs, Nb_GPU1s, Nb_GPU2s) with the same numbers of CPUs and for either types of GPUs, for a total of 64 machine settings.

We define a *configuration* to be a combination of an instance of application and a machine setting. We executed the algorithms only once with each configuration since all algorithms are deterministic. For each run, we stored the optimal objective solution of the linear program, denoted by LP^*, and the makespans of the six algorithms. For the application instance giving the highest number of tasks, the linear program resolution took about 100 seconds while the running time of each algorithm took at most 10 seconds, once a solution of HLP was found for the linear program-based algorithms.

Results for 2 resource types.
To study the performance of the 3 algorithms we computed the ratio between each makespan and the optimal solution LP^* of the linear program HLP, which corresponds to a good lower bound of the optimal makespan. Fig. 3 shows the ratio of each configuration. Notice that the bigger dot represents the mean value of the ratio for each application. We can see that HLP-EST is outperformed, on average, by the two other algorithms. The performances of HLP-OLS and HEFT are quite similar, on average, but we observe that HEFT creates more outliers.

Fig. 4 compares in more detail the two HLP-based algorithms (left), and the algorithms HLP-OLS and HEFT (right), by showing the ratio between the makespans of the two algorithms. We can see that HLP-OLS outperforms HLP-EST, except for a few configurations with the application *potri*, with an improvement close to 8% on average. Comparing HLP-OLS and HEFT we notice that, even if the two algorithms have

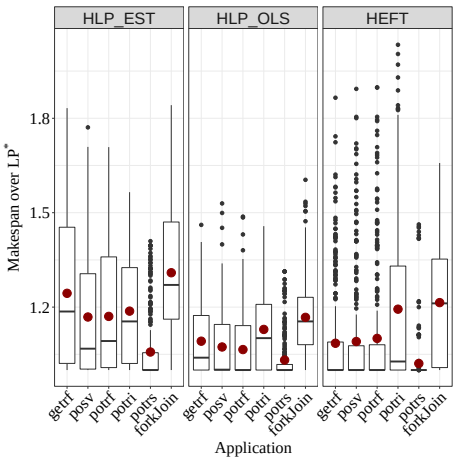


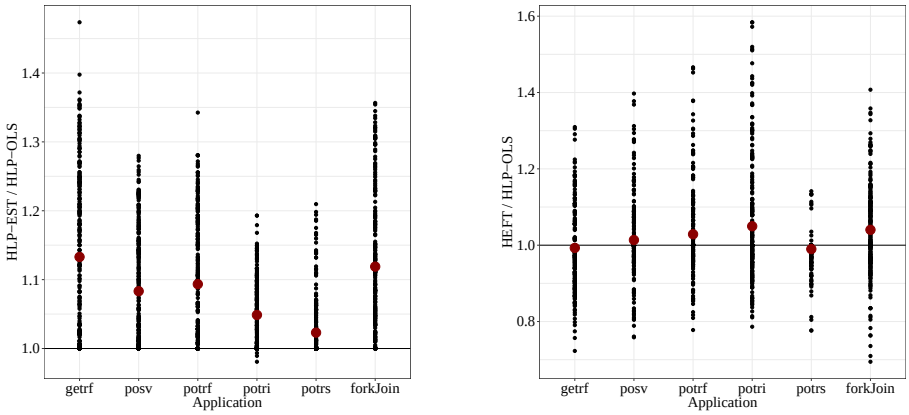
Figure 3: Ratio of makespan over LP^* for each configuration grouped by application for the off-line algorithms with 2 resource types.

similar performances, HEFT is on average outperformed by HLP-OLS by 2%, with a maximum of 60% of improvement for HLP-OLS with some configurations of *potri*. Moreover, HEFT has a significantly worse performance than HLP-OLS in strongly heterogeneous applications where there is a bigger perturbation in the (dis-)acceleration of the tasks on the GPU side, like *forkJoin*, since in these irregular cases the allocation problem becomes more critical.

Results for 3 resource types.
To study the performance of the 3 algorithms we computed the ratio between each makespan and the optimal solution LP^* of the linear program QHLP, which corresponds to a good lower bound of the optimal makespan. Fig. 5 (left) shows the ratio of each configuration. Notice that the bigger dot represents the mean value of the ratio for each application. We can see that QHLP-EST is on average outperformed by the two other algorithms. We also observe that even if QHEFT presents many outliers for the applications *getrf*, *posv* and *potrf*, the algorithms outperforms on average QHLP-OLS.

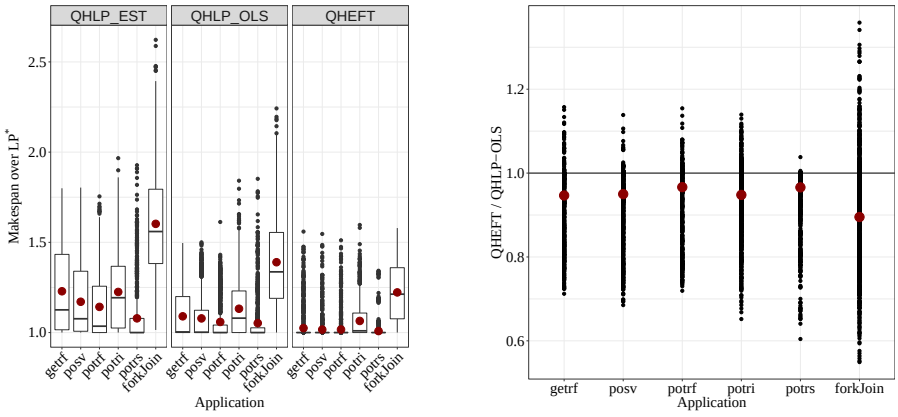
Fig. 5 (right) compares in more detail QHEFT and QHLP-OLS, by showing the ratio between the makespans of the two algorithms. Comparing QHLP-OLS and QHEFT, we can see that QHEFT presents an improvement over QHLP-OLS of 5% on average. We also observe that the ratios for the most irregular application, *fork-join*, are spread with configurations favorable to QHEFT (up to 45% of improvement) and other configurations favorable to QHLP-OLS (up to 36% of improvement). Similar results were observed between QHLP-EST and QHLP-OLS as for 2 resource types and thus are not showed here.

Finally, we notice that the approximation ratios, computed with a lower bound of the optimal makespan, do not exceed 2 and thus are far



(a) (b)

Figure 4: Ratio between the makespans of HLP-EST and HLP-OLS (left), and HEFT and HLP-OLS (right) for each configuration, grouped by application.



(a) (b)

Figure 5: Ratio of makespan over LP^* for each configuration (left), grouped by application for the algorithms generalized for 3 resource types. Ratio between the makespan of QHEFT and QHLP-OLS for each configuration (right), grouped by application.

from the theoretical bounds of the algorithms, even for the case with 3 resource types.

7.2 On-line setting

Algorithms.
We compared the performance, in terms of makespan, of the algorithm ER-LS (Section 4.2) with 3 baseline algorithms: *EFT*, which schedules a task on the processor which gives the earliest finish time for that task; *Greedy*, which allocates a task on the processor type which has the smallest processing time for that task; and *Random*, which randomly assigns a task to the CPU or GPU side. For the algorithms Greedy and Random, we used a List Scheduling algorithm to schedule the tasks once the allocations has been made. The algorithms were implemented in Python (v. 2.8.6). For the machine settings, we used the same sets of pairs as for the off-line setting with 16, 32, 64 and 128 CPUs, and 2, 4, 8 and 16 GPUs.

We executed the algorithms only once with each configuration since all algorithms are deterministic, except Random. The running times of the algorithms were similar and took at most 5 seconds for the application instance giving the highest number of tasks.

Results.
Fig. 6 (left) compares the ratios between the makespan of each of the on-line algorithms and LP^*. Due to large differences between the performances of Random and the 3 other algorithms, we kept only the algorithms ER-LS, EFT and Greedy. Results show that Greedy is on average outperformed by ER-LS and EFT, and that EFT creates less outliers than the 2 other algorithms. Fig. 7 compares in more detail Greedy and ER-LS (left), and EFT and ER-LS (right), by showing the ratio between the makespans of the two algorithms. We can see that ER-LS outperforms Greedy on average, with a maximum for the potri application where ER-LS performs 11 times better than Greedy. More specifically, there is an improvement of between 8% and 36% on average for ER-LS depending on the application considered, except for *potrs* whose makespans are on average 10% greater than for Greedy.

Comparing EFT and ER-LS, we can see that ER-LS is outperformed by EFT with a decrease of 11% on average, and up to 60% for certain configurations of fork-join. However, the worst-case competitive ratio for EFT can be directly obtained from the proof of the worst-case approximation ratio for HEFT, presented in Section 3. More specifically, if the adversary presents to EFT the list of tasks ordered as in the counter-example for HEFT (Theorem 1), i.e., by decreasing order of the rank, then we obtain the same lower bound.

We also study the performance of the 3 algorithms with respect to the theoretical upper bound given in Section 4.2. Fig. 6 (right) shows the mean competitive ratio of ER-LS, EFT and Greedy along with the standard error as a function of $\sqrt{\frac{m}{k}}$ associated to each configuration. To simplify the lecture, we only present the applications *potri* and *fork-join*, since other *Chameleon* applications showed similar results. We observe that the competitive ratio is smaller than $\sqrt{\frac{m}{k}}$ and far from the theoretical

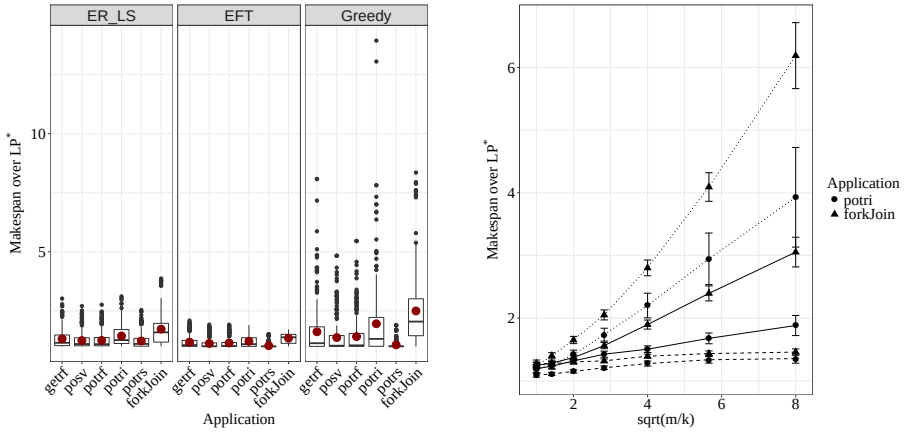


Figure 6: Ratio of makespan over LP^* for each configuration, grouped by application for the on-line algorithms with 2 resource types (left). Mean competitive ratio of ER-LS (plain), EFT (dashed) and Greedy (dotted) as a function of $\sqrt{\frac{m}{k}}$ (right).

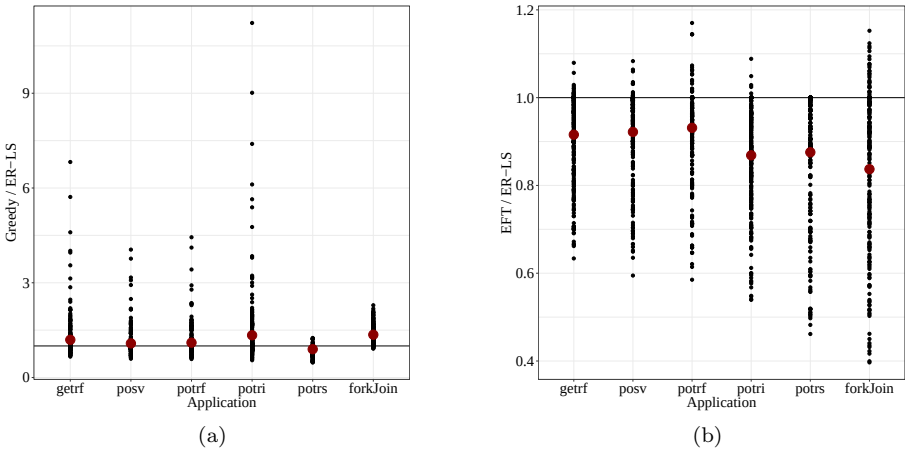


Figure 7: Ratio between the makespans of Greedy and ER-LS (left), and EFT and ER-LS (right) for each configuration, grouped by application.

upper bound of $4\sqrt{\frac{m}{k}}$ for ER-LS.

8 Conclusions

We studied the problem of scheduling parallel applications, represented by a precedence task graph, on heterogeneous multi-core machines. We focused on generic approaches, non depending on the particular application, by distinguishing the allocation and the scheduling phases and proposed efficient algorithms with worst-case performance guarantees for both off-line and on-line settings. In the off-line case, motivated by new lower bounds on the performance of existing algorithms, we refined the scheduling phase of the best known approximation algorithm and we presented a new algorithm that preserves the approximation ratio and performs better in our experiments. We also extended this methodology for the more general case where the architecture is composed of $Q \geq 2$ types of resources. In the on-line case, we presented a $\Theta(\sqrt{\frac{m}{k}})$-competitive algorithm based on adequate rules. This is the first on-line result dealing with precedence constraints on hybrid machines.

From the practical point of view, an extensive simulation campaign on representative benchmarks constructed by real applications showed that it is possible to outperform the classical HEFT algorithm keeping reasonable running times for the case with 2 resource types. With more resource types, results showed that a simple generalization of an algorithm could present similar performances as HEFT.

For the on-line case, the algorithm based on rules is a good trade-off since it delivers a solution close to the optimal while keeping a running time similar to pure greedy algorithms. We aim to implement it on a real run-time system (such as StarPU [3]) which currently uses HEFT on successive sets of independent tasks.

In this work we assumed that the communications between CPUs, GPUs and the shared memory are neglected. Our next step is to introduce communication costs in the algorithms, which should not be too hard in both integer program and greedy rules.

Acknowledgments

This work was partially supported by FAPESP (São Paulo Research Foundation, grant #2012/23300-7) and ANR Moebus Project. All the experiments were performed on the Froggy platform of the CIMENT infrastructure (https://ciment.ujf-grenoble.fr), which is supported by the Rhône-Alpes region (GRANT CPER07_13 CIRA) and the Equip@Meso project (reference ANR-10-EQPX-29-01) of the program "Investissements d'Avenir" supervised by the French Research Agency (ANR). We would like to thank Florent Pruvost and Mathieu Faverge, from Inria team MORSE, for the collaboration in this work.

References

[1] M. Amaris, R. Y. de Camargo, M. Dyab, A. Goldman, and D. Trystram. A comparison of GPU execution time prediction using machine learning and analytical modeling. In *IEEE 15th International Symposium on Network Computing and Applications*, pages 326–333, Oct 2016.

[2] M. Amaris, G. Lucarelli, C. Mommessin, and D. Trystram. Generic algorithms for scheduling applications on hybrid multi-core machines. In *Euro-Par: International Conference on Parallel and Distributed Computing*, pages 220–231, 2017.

[3] 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*, 23(2):187–198, 2011.

[4] O. Beaumont, L. Eyraud-Dubois, and S. Kumar. Approximation proofs of a fast and efficient list scheduling algorithm for task-based runtime systems on multicores and GPUs. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 768–777, 2017.

[5] 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*, 27(6):1625–1638, 2015.

[6] L.-C. Canon, L. Marchal, and F. Vivien. Low-cost approximation algorithms for scheduling independent tasks on hybrid platforms. In *Euro-Par: International Conference on Parallel and Distributed Computing*, pages 232–244, 2017.

[7] C. Chekuri and M. Bender. An efficient approximation algorithm for minimizing makespan on uniformly related machines. *Journal of Algorithms*, 41(2):212–224, 2001.

[8] L. Chen, D. Ye, and G. Zhang. Online scheduling of mixed CPU-GPU jobs. *International Journal of Foundations of Computer Science*, 25(06):745–761, 2014.

[9] F. A. Chudak and D. B. Shmoys. Approximation algorithms for precedence-constrained scheduling problems on parallel machines that run at different speeds. *Journal of Algorithms*, 30:323–343, 1999.

[10] D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J.-M. Vincent, and F. Wagner. Random graph generation for scheduling simulations. In *ICST (SIMUTools)*, 2010.

[11] Maciej Drozdowski. *Scheduling for Parallel Processing*. Springer Publishing Company, 2009.

[12] E. Agullo et al. Poster: Matrices over runtime systems at exascale. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 1330–1331, Nov 2012.

[13] T. D. Braun et al. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel Distributed Computing*, 61(6):810–837, 2001.

[14] V. W. Lee et al. Debunking the 100x GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU. *SIGARCH Computer Architecture News*, 38:451–460, 2010.

[15] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal On Applied Mathematics*, 17(2):416–429, 1969.

[16] S. Kedad-Sidhoum, F. Monna, and D. Trystram. Scheduling tasks with precedence constraints on hybrid multi-core machines. In *HCW - IPDPS Workshops*, pages 27–33, 2015.

[17] S. Kumar, M. Marathe, S. Parthasarathy, and A. Srinivasan. Scheduling on unrelated machines under tree-like precedence constraints. In *Algorithmica*, volume 55, pages 609–609, Aug 2005.

[18] J. W. Liu and C. L. Liu. Performance analysis of multiprocessor systems containing functionally dedicated processors. *Acta Informatica*, 10(1):95–104, 1978.

[19] D. B. Shmoys, C. Stein, and J. Wein. Improved approximation algorithms for shop scheduling problems. *SIAM Journal on Computing*, 23(3):617–632, 1994.

[20] TOP500 Supercomputer. `http://www.top500.org` (last visited on Nov. 2017).

[21] O. Svensson. Hardness of precedence constrained scheduling on identical machines. *SIAM Journal on Computing*, 40(5):1258–1274, 2011.

[22] H. Topcuoglu, S. Hariri, and Min-You Wu. Task scheduling algorithms for heterogeneous processors. In *Heterogeneous Computing Workshop (HCW)*, pages 3–14, 1999.