

Online Scheduling of Task Graphs on Heterogeneous Platforms

Louis-Claude Canon, Loris Marchal, Bertrand Simon, Frédéric Vivien

▶ **To cite this version:**

Louis-Claude Canon, Loris Marchal, Bertrand Simon, Frédéric Vivien. Online Scheduling of Task Graphs on Heterogeneous Platforms. IEEE Transactions on Parallel and Distributed Systems, 2020, 31 (3), pp.721-732. 10.1109/TPDS.2019.2942909 . hal-02291268

HAL Id: hal-02291268

https://inria.hal.science/hal-02291268

Submitted on 18 Sep 2019

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.

Online Scheduling of Task Graphs on Heterogeneous Platforms

Louis-Claude Canon, Loris Marchal, Bertrand Simon, and Frédéric Vivien

Abstract—Modern computing platforms commonly include accelerators. We target the problem of scheduling applications modeled as task graphs on hybrid platforms made of two types of resources, such as CPUs and GPUs. We consider that task graphs are uncovered dynamically, and that the scheduler has information only on the available tasks, i.e., tasks whose predecessors have all been completed. Each task can be processed by either a CPU or a GPU, and the corresponding processing times are known. Our study extends a previous $4\sqrt{m/k}$-competitive online algorithm by Amaris et al. [Euro-Par, 2017], where m is the number of CPUs and k the number of GPUs ($m \geq k$). We prove that no online algorithm can have a competitive ratio smaller than $\sqrt{m/k}$. We also study how adding flexibility on task processing, such as task migration or spoliation, or increasing the knowledge of the scheduler by providing it with information on the task graph, influences the lower bound. We provide a $(2\sqrt{m/k}+1)$-competitive algorithm as well as a tunable combination of a system-oriented heuristic and a competitive algorithm; this combination performs well in practice and has a competitive ratio in $\Theta(\sqrt{m/k})$. We also adapt all our results to the case of multiple types of processors. Finally, simulations on different sets of task graphs illustrate how the instance properties impact the performance of the studied algorithms and show that our proposed tunable algorithm performs the best among the online algorithms in almost all cases and has even performance close to an offline algorithm.

Index Terms—Scheduling, heterogeneous computing, task graphs, online algorithms.

1 Introduction

Modern computing platforms increasingly use specialized hardware accelerators, such as GPUs or Xeon Phis: 102 of the supercomputers in the TOP500 list include such accelerators, while several of them include several accelerator types. The increasing complexity of such computing platforms makes it hard to predict the exact execution time of computational tasks or of data movement. Thus, dynamic runtime schedulers are often preferred to static ones, as they are able to adapt to variable running times and to cope with inaccurate predictions. Indeed, with the widespread heterogeneity of computing platforms, many scientific applications now rely on runtime schedulers such as OmpSs, XKaapi or StarPU. Most of these frameworks model an application as a task graph, and more precisely a Directed Acyclic Graph (DAG) of tasks, where nodes represent tasks and edges represent dependencies between tasks. While task graphs have been widely studied in the theoretical scheduling literature, most of the existing studies concentrate on static scheduling in the offline context: both the graph and the running times of the tasks are known beforehand.

We believe that there is a crucial need for online schedulers, that is, of scheduling algorithms that do not rely on the structure of the graph. First, not all graphs are fully available at the beginning of the computation: sometimes the graph itself depends on the data being processed, different inputs may result in different task graphs. This is especially the case when the behavior of an iterative application depends on the accuracy of the output. Second, in most existing runtimes, even if the graph does not depend on the input data, it is not fully submitted at the beginning of the computation; instead, tasks are dynamically uncovered during the computation. Finally, even if part of the graph is available, runtime schedulers usually avoid traversing large parts of the graph each time they take a decision in order to strongly limit the time needed to make decisions.

In the present paper, we concentrate on the online scheduling of task graphs on a hybrid platform composed of 2 types of processors that we call CPU and GPU for convenience. There are m CPUs and k GPUs, where $m \geq k \geq 1$. Note that we do not make any assumptions on the CPUs and GPUs (i.e., on the processing times of each task), so that these results may be symmetrically applied to the converse case with more GPUs. The objective is to schedule a DAG G of tasks, so as to minimize the total completion time, or makespan. Each task can be assigned either to a single CPU or to a single GPU. We adopt the notations of [1]: the processing time of task T_i on a CPU is noted by $\overline{p_i}$ and on a GPU by $\underline{p_i}$.

We consider the following online problem. At the beginning, the algorithm is aware of all the input tasks of the graph, and can schedule each one on either a CPU or on a GPU. A task is released and becomes available to the scheduler only when all its predecessors completed. At any given point in the computation, the scheduler is totally unaware of tasks that have not yet been released, but it knows the processing times $\overline{p_i}$ and $\underline{p_i}$ of all available tasks: we assume that tasks correspond to well-known computational kernels whose processing times have been acquired through extensive benchmarking; this happens for example in linear

- *L. Marchal and F. Vivien are with CNRS, INRIA and University of Lyon, LIP, ENS Lyon, 46 allée d'Italie, Lyon, France. E-mail: {loris.marchal,frederic.vivien}@ens-lyon.fr*
- *L.-C. Canon is with FEMTO-ST Institute – Université de Bourgogne Franche-Comté, 16 route de Gray, 25 030 Besançon, France. E-mail: louis-claude.canon@univ-fcomte.fr*
- *B. Simon is with University of Bremen, Bibliothekstr., 28359 Bremen. E-mail: bsimon@uni-bremen.de*

algebra applications. We do not take into account the time needed for moving data and assume that there is no delay between the release of a task and the start of its processing. We denote this problem as $(Pm, Pk)|prec, online|C_{max}$.

Contributions

This paper extends the work of Amaris et al. [1] on the very same problem, which provides a $4\sqrt{m/k}$-competitive algorithm. We recall that an online algorithm is x-competitive if the makespan returned by this algorithm on any instance is at most x times larger than the optimal makespan (which can be computed by an offline algorithm). The present paper brings the following contributions:

- We prove that the competitive ratio of any online algorithm is lower-bounded by $\sqrt{m/k}$ for any value of m and k, and by $\lfloor\sqrt{m/k}\rfloor + 1$ when k is large. We study how the knowledge of the task graph and the flexibility of the tasks may influence the lower bound; we especially prove that knowing the bottom-level of any task (i.e., the critical path length from this task to the end of the graph) or having preemptive tasks does not help much, whereas the knowledge of the number of descendants allows reducing the lower bound to $\frac{1}{2}(m/k)^{1/4}$ (Section 3). We also extend these results to randomized algorithms.
- We propose a $(2\sqrt{m/k}+1)$-competitive algorithm, QA, by refining both the algorithm and the analysis of Amaris et al. [1] (Section 4.1).
- We study the generalized problem with multiple types of processors, on which adapt the lower bounds and the online algorithm (Section 6);
- We propose a simple heuristic (Section 4.2) based on QA and the system-oriented heuristic EFT, which is both a competitive algorithm and performs well in practice, as we show with a comprehensive simulation set (Section 7).

Note that a preliminary version of this work was presented at the Euro-Par 2018 conference [2].

2 Related work

We briefly position our contributions in comparison to the existing work, starting with the offline case when the whole scheduling problem (both task dependencies and running times) is known beforehand.

Offline algorithms

Several schedulers for independent tasks on hybrid platforms have been proposed. Bleuse et al. [3] designed a polynomial but expensive $\left(\frac{4}{3} + \frac{1}{3k}\right)$-approximation. Low complexity algorithms, which are closer to our work, have been studied [4], [5] and achieve approximation ratios respectively equal to 2 and $2 + \sqrt{2}$. For tasks with precedence constraints, Kedad-Sidhoum et al. [6] provided a tight 6-approximation based on linear programming, while Beaumont et al. [7] extend their previous algorithm [5] to the (offline) scheduling of task graphs.

In a different context, Chudak and Shmoys [8] provided a $\log(p)$-approximation for the $Q \mid prec \mid C_{max}$ problem: scheduling a graph of tasks on p machines with different speeds. Considering independent moldable tasks (tasks can be assigned to multiple CPUs or one GPU), Bleuse et al. [9] provide a 2-approximation. It was later adapted by Aba et al. [10] to account for data movement between CPUs and GPUs.

Online algorithms

When tasks with precedences are released over time, Graham's List Scheduling algorithm [11] is $(2 - 1/m)$-competitive on homogeneous processors. Svensson [12] proved that no polynomial offline algorithm can have a better competitive ratio assuming $P \neq NP$ and a variant of the Unique Games Conjecture.

On the problem of scheduling independent tasks on two sets of processors, Imreh [13] proposed a $(4 - 2/m)$-competitive algorithm. The principle of this algorithm is to schedule a task T_i on GPU if $\overline{p_i}/\underline{p_i} \geq m/k$, or if T_i can complete on GPU given the current schedule before time $\overline{p_i}$. Chen et al. [14] later refine a similar algorithm by introducing four tunable parameters. A combination of parameters leads to an improved competitive ratio equal to 3.85. They also show that any online algorithm has a competitive ratio at least 2.

Based on this work, Amaris et al. [1], [15] exhibited an online algorithm for precedence constraints, achieving a competitive ratio of $4\sqrt{m/k}$. The main difference with the previous algorithms is that a task is scheduled on GPU if it is accelerated by a factor at least $\sqrt{m/k}$ (and not at least m/k).

Runtime strategies

Actual runtime schedulers usually rely on low-complexity scheduling policies to limit the time needed to allocate tasks. For instance, StarPU [16] builds a performance model of tasks that enables to predict their processing times. When a new task is submitted, it is allocated to the resource that will complete it the soonest (when using the **dm** policy, previously called **heft-tm** [17]), which corresponds to the classical Earliest Finish Time (EFT) scheduling policy [18]. Other strategies have been proposed that take into account communication times, or precomputed task priorities, depending on the descendants of each task. This has motivated the consideration of such information in the design of the lower bounds on competitive ratios (Section 3).

3 Lower bound on online algorithms competitiveness

In this section, we provide a lower bound on the competitive ratio of any online algorithm: no online algorithm has a competitive ratio smaller than $\tau = \sqrt{m/k}$ for any values of m and k (Theorem 1) and smaller than $\lfloor\tau\rfloor + 1$ when k is large (Remark 1). We also study how adding flexibility to task processing or giving some knowledge of the graph to the scheduler impacts this lower bound.

Intuitively, the main difficulty for this problem arises from choosing on which type of resource (CPU or GPU) a given task should be processed, and not to come up with the final schedule. This is indeed proven in Theorem 7, Section 5: if the allocation of the tasks is fixed, any online list scheduling algorithm is $\left(3 - \frac{1}{m}\right)$-competitive, and this competitive ratio cannot be surpassed.

Table 1
Summary of the lower bounds results obtained for various versions of online models. τ^* represents the largest triangular number such that $\tau^* \leq \tau$. If τ is large, we have $\lfloor\sqrt{2\tau^*}\rfloor \geq \sqrt{\tau}$.

Flexibility	Knowledge	Lower bound	Proof	Note
None or Spoliation	None	τ	Th. 1	$\lfloor\tau\rfloor + 1$ for large k
	Bottom Level	τ	Th. 1	$\frac{1}{2}\tau$ if $k = 1$; $\lfloor\tau\rfloor + 1$ for large k
	BL + descendants	$\frac{1}{2}\lfloor\sqrt{2\tau^*}\rfloor$	Th. 2	–
Migration	None	$\frac{1}{2}\tau$	Th. 1	–
	BL	$\frac{1}{2}\tau$	Th. 1	$\frac{1}{4}\tau$ if $k = 1$
	BL + descendants	$\frac{1}{4}\lfloor\sqrt{2\tau^*}\rfloor$	Th. 2	–

The proof of Theorem 1 heavily relies on the fact that an online algorithm has no information on the successors of each task. In practice, it is sometimes possible to get some information on the task graph, for example by precomputing some information offline before submitting the tasks. For instance, offline schedulers usually rank available tasks with priorities based on the graph structure. On homogeneous platforms, the *bottom-level* of a task is commonly used, and is defined as the maximum length of a path from this task to an exit node, where nodes of the graphs are weighted with the processing time of the corresponding tasks. In the heterogeneous case, the priority scheme used in the standard HEFT algorithm [19] is to set the weight of each node as the average processing time of the corresponding task on all resources.

Knowing the bottom-level does not change the lower-bounds of Theorem 1 and Remark 1, see the last statement of Theorem 1. The only benefit is a diminution by a factor 2 if there is exactly one GPU. An interesting component of this proof is that all the tasks are equivalent (same CPU and GPU computing times) so other heterogeneous variants of the bottom level are also captured.

When the online scheduler is given the knowledge of the number of descendants of each submitted task in addition to their bottom-level, the lower bound of Theorem 1 is reduced to $\frac{1}{2}\sqrt{\tau}$ when m/k is large enough (see Theorem 2), so no constant-factor competitive algorithm exists. Note that all the tasks are also equivalent in this proof; so it also captures, for instance, the knowledge of the CPU and GPU computing times of all the descendants; only the pattern of precedence relations remains unknown. Note that, however, no algorithm has been proposed that reaches this bound.

Another interesting question is whether adding flexibility on how tasks are processed changes this bound. Allowing task spoliation (where tasks can be canceled and restarted on another resource, as done in [5]) does not change any lower bound and our results hold both when spoliation is authorized or forbidden. Allowing task migration (where tasks can be interrupted and resumed on another resource) divides the lower bounds obtained by a factor 2.

Table 1 summarizes the results for all combination of knowledge given to the scheduler and flexibility on the task processing. The best known competitive ratio for every setting is smaller than $2\tau + 1$, and is achieved by the QA algorithm we design in Section 4.1. This algorithm does not use all the knowledge or flexibility as it does not practice spoliation or migration, does not use any information on the bottom level or the descendants, and schedules tasks one by one, without looking at other available tasks.

First, we consider algorithms that are not aware of the bottom level of the tasks.

Theorem 1. *No online algorithm has a competitive ratio smaller than τ for the problem $(Pm, Pk)|prec, online|C_{max}$. If preemption with migration is authorized, no online algorithm has a competitive ratio smaller than $\frac{\tau}{2}$. These results sill hold if the bottom level of each task is known and $k \geq 2$, and are halved if $k = 1$.*

Proof. We first focus on the case where migration is not authorized and the bottom level is unknown.

Consider an online algorithm $\mathcal{A}$, making use of spoliations. We assume for the moment that τ is an integer. We consider an integer n as large as we want. A large n will lead to a large graph and a competitive ratio closer to τ. We will use an adversary proof, by building a graph composed of nm tasks denoted by T_i^j, with $1 \leq j \leq n\tau$ and $1 \leq i \leq k\tau$. The CPU processing time of each task equals τ and the GPU processing time equals 1.

The procedure can be cut into $n\tau$ phases. During the jth phase, tasks T_i^j for i from 1 to $k\tau$ are independent and available. The adversary selects the task that $\mathcal{A}$ completes the latest, breaking ties arbitrarily. Let T_*^j be this task. The $k\tau$ tasks of the next phase are then made successors of T_*^j. See Figure 1 for an illustration of a built graph.

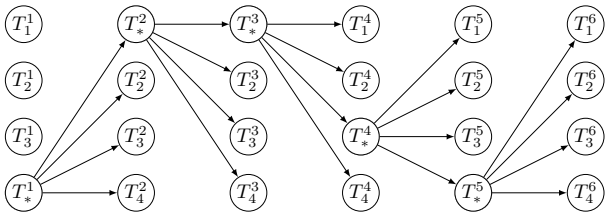


Figure 1. Example of built graph with $\tau = 2$, $k = 2$, $n = 3$.

We now show how to build an efficient (offline) schedule $\mathcal{S}$ of the resulting graph. A *bucket* is defined as a set of processors, a starting time and a duration time. We use buckets to book some processors for an amount of time, and schedule a set of tasks in a given bucket. We consider $n + 1$ buckets, as illustrated in Figure 2. Buckets B_i for i from 1 to n each concerns all m CPUs, lasts a time τ, and starts at time

$i\tau$. Note that m tasks fit into each bucket. The last bucket, B concerns one GPU, starts at time 0 and lasts a time $n\tau$.

$\mathcal{S}$ schedules the $n\tau$ tasks T_*^j successively on a single GPU, which fit into bucket B. In parallel, $\mathcal{S}$ schedules the remaining tasks on CPU. More precisely, it puts in bucket B_ℓ tasks T_i^j such that $(\ell-1)\tau < j \leq \ell\tau$, except for tasks T_*^j. They all fit into the bucket as there are less than $\tau \times k\tau \leq m$ such tasks. Moreover, task $T_*^{\ell\tau}$ completes at time $\ell\tau$. Therefore, every task T_i^j with $(\ell-1)\tau < j \leq \ell\tau$ can be started at time $\ell\tau$, and thus can be scheduled into bucket B_ℓ. Therefore, $\mathcal{S}$ achieves a makespan equal to $(n+1)\tau$. Note that this schedule delays the executions of tasks on CPUs in order to start m tasks simultaneously a the beginning of each bucket. Another solution would be to start each task allocated to CPU as soon as possible, which does not improve the makespan.

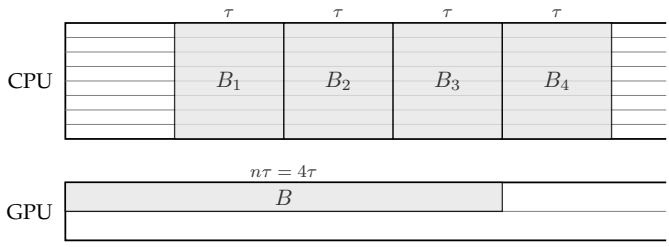


Figure 2. Buckets used by $\mathcal{S}$ with $n = 4$.

Now, we consider algorithm $\mathcal{A}$, and we show that the makespan obtained is at least $n\tau^2$. At each phase, the adversary reveals the next phase only when all the tasks of the current phase are completed. If one task of the phase is scheduled on CPU, it takes a time τ. Otherwise, all $k\tau$ tasks are scheduled on GPU, and the last one completes at time at least $k\tau/k = \tau$. Therefore, $\mathcal{A}$ completes each phase in time at least τ. As there are $n\tau$ phases, the whole graph cannot be scheduled in time smaller than $n\tau^2$. The competitive ratio of $\mathcal{A}$ is then at least:

$$\frac{n\tau^2}{(n+1)\tau} \quad \underset{n\to\infty}{\longrightarrow} \quad \tau.$$

Therefore, we have proved this first result: for any m and k such that τ is integer, no online algorithm has a competitive ratio smaller than τ.

Now, consider an algorithm $\mathcal{A}'$ that makes use of preemption with migration. The adversary strategy and the schedule $\mathcal{S}$ is unchanged. We first prove by contradiction that $\mathcal{A}'$ cannot complete a phase in a makespan smaller than $\tau/2$. Assume that one phase is completed in time $\tau/2$. We consider the fraction of each task performed on a CPU. All tasks have a processing time of τ on CPU, so for each task, this fraction cannot be larger than one half. Therefore, at least half of each task is executed on a GPU, which takes a time $1/2$ for each task, so it takes $k\tau/2$ units of GPU computing time. As we assumed that the phase is completed in time $\tau/2$, there is no more than $k\tau/2$ work units available on the k GPUs, which thus cannot process more than one half of each task. Therefore, at least half of each task is processed on CPUs, from the very beginning to the very end of the phase. This requires executing each task simultaneously on a CPU and on a GPU, which is not possible even with migration. Therefore, $\mathcal{A}'$ cannot complete one phase in time $\tau/2$ (and a fortiori in a shorter time). Thus, $\mathcal{A}'$ requires a time larger than $n\tau^2/2$ to complete all $n\tau$ phases. The competitive ratio of $\mathcal{A}'$ is then at least:

$$\frac{\frac{1}{2}n\tau^2}{(n+1)\tau} \quad \underset{n\to\infty}{\longrightarrow} \quad \frac{\tau}{2}.$$

In a last step, we relax the constraint that τ is an integer. Let q be an integer as large as we want, and $r = \lfloor q\tau \rfloor$, so that $r/q \leq \tau \leq (r+1)/q$. A large q will lead to a greater precision. We adapt the graph in the following way: there are now nr phases each containing $k\lfloor\tau\rfloor + 1$ tasks. Each task has a CPU computing time equal to τ/q and a GPU computing time equal to $1/q$.

We now adapt the schedule $\mathcal{S}$, which still fits inside the buckets (as previously defined). The tasks T_*^j all fit inside bucket B. Indeed, the time needed to process them sequentially is equal to $nr/q \leq n\tau$. For bucket B_1, we execute the tasks T_i^j for $j = 1, \ldots, r$ except tasks T_*^j. The corresponding tasks T_*^j are completed before the start of bucket B_1. Inside bucket B_1, we need to execute $kr\lfloor\tau\rfloor$ tasks of CPU computing time τ/q. The number of processors needed is then $\lceil kr\lfloor\tau\rfloor/q\rceil \leq k\tau^2 = m$, so the first r phases fit into bucket B_1. The same reasoning applies to the following buckets, so the makespan of $\mathcal{S}$ is $(n+1)\tau$.

Concerning the algorithm $\mathcal{A}$, each phase needs at least a time τ/q to complete as computing all tasks on GPU take a time $(\lfloor\tau\rfloor+1)/q$. The total makespan is then at least $nr\tau/q \geq n\tau\left(\tau - \frac{1}{q}\right)$, so the competitive ratio tends to τ when q and n grow. If $\mathcal{A}$ uses preemption with migration, this ratio tends to $\tau/2$, which terminates the proof.

Finally, we illustrate on Figure 3 the proof of the last step of the theorem, when the online algorithm has access to the bottom level of each task. The detailed proof of this result and the following ones have been deferred to the web supplementary material.

By adding $n\tau$ tasks to the graph of Figure 1, it is possible to build a graph in which all tasks of the same phase have the same bottom level. An offline algorithm can compute these additional tasks in parallel on a separate GPU if $k \geq 2$. Hence, the theorem.

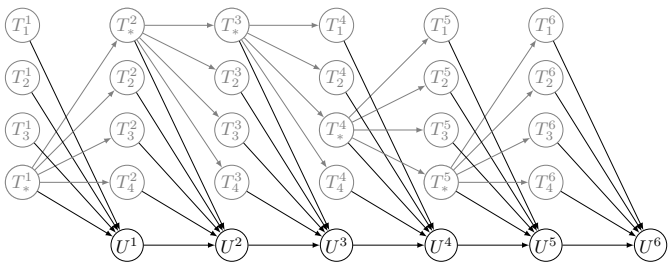


Figure 3. Example of built graph with $\tau = 2$, $k = 2$, $n = 3$. In gray, the tasks and dependencies existing in the previous proof.

□

In the proof of Theorem 1, we use the fact that the bottom level is no longer useful to differentiate tasks T_*^j from other tasks T_i^j. However, the former may have many more descendants than the latter ($\Theta(nm)$ compared to $\Theta(n\tau)$ for

small j). Some heuristic may thus rely on the total weight of the descendants of a task in order to decrease its competitive ratio. We nevertheless prove in Theorem 2 that this knowledge cannot lead to constant-factor approximation, as we prove a lower-bound in $\Theta(\sqrt{\tau})$. As discussed at the beginning of this section, all the tasks used in the proof of the following theorem are identical, so the *weight* can actually capture several functions such as the number of descendants, the average computing time, etc. We recall for the theorem statement that τ is a triangular integer if we have $\tau = 1 + 2 + \ldots + \left\lfloor \sqrt{2\tau} \right\rfloor$.

Theorem 2. *No online algorithm has a competitive ratio smaller than $\frac{1}{2}\left\lfloor \sqrt{2\tau^*} \right\rfloor$ for the problem $(Pm, Pk)|prec, online|C_{max}$, even when both the bottom level and the total weight of the descendants of each task is known. If preemption with migration is authorized, no online algorithm has a competitive ratio smaller than $\frac{1}{4}\left\lfloor \sqrt{2\tau^*} \right\rfloor$. In these bounds, τ^* is the largest triangular integer not larger than τ.*

Proof idea. Similarly to the last part of Theorem 1, this result is proved by adding tasks to the focused graphs. However, the number of phases has to be small in this case so that the (many) additional tasks can be computed quickly in the offline schedule. This explains why the lower bounds are significantly smaller. □

Theorem 1 proves a lower bound valid for any value of m and k. In the following remark (proven in the web supplementary material), we show that the lower bound can be improved from τ to $\lfloor \tau \rfloor + 1$ if k is large enough compared to m. However, it should be noted that this result is still not tight, as Lemma 3 provides a better bound $(3 - 1/m)$ for small values of τ and large values of m.

Remark 1. *When $4m \leq (k-2)^3$, no online algorithm has a competitive ratio smaller than $\lfloor \tau \rfloor + 1$, even when the bottom level of each task is known.*

We conclude this part by adapting our results to randomized algorithms. We have focused so far on establishing lower bounds on the approximation ratio of deterministic online algorithms. The competitive ratio of a randomized online algorithm is defined as the maximum over all graphs of the following ratio: the expected makespan obtained by the algorithm on a given graph divided by the offline minimum makespan of this graph.

Theorem 3. *The lower bounds proved in Theorems 1 and 2 on the competitive ratio of deterministic algorithms not using migration hold for randomized algorithms, divided by a factor 2.*

Proof. Consider a randomized online algorithm $\mathcal{B}$ not using preemption with migration. We focus on a *phase* of $k\lfloor \tau \rfloor$ independent and indistinguishable tasks of CPU computing time τ and GPU computing time 1, where only one task T_* among these has successors. Select T_* as the task with the highest expected completion time under $\mathcal{B}$. In any schedule computed by $\mathcal{B}$, the average completion time of all tasks is at least $(\lfloor \tau \rfloor + 1)/2 \geq \tau/2$.

Plugging this result into the proofs of Theorems 1 and 2, we get that the expected makespan obtained by $\mathcal{B}$ is at least half of the smallest makespan that can be guaranteed by a deterministic online algorithm. Hence, the results. □

4 Competitive algorithms

4.1 The Quick Allocation (QA) algorithm

Amaris et al. [1] designed an online algorithm named ER-LS composed of two phases. For each available task, it first decides whether it should be allocated to CPUs or GPUs, and then schedules it on the appropriate resource type. More precisely, ER-LS can be described as follows:

1) Take any available task T_i.
   a) If T_i can complete on GPU in the current schedule before time $\overline{p_i}$, allocate it to GPU.
   b) If $\overline{p_i}/\underline{p_i} \leq \tau = \sqrt{m/k}$, then allocate T_i to CPU, otherwise assign it to GPU.
2) Schedule T_i as soon as possible on the assigned type of processor
3) If there are remaining tasks, return to Step 1.

This algorithm is proved to be 4τ-competitive. We propose here a simplified version of this algorithm, for which we prove a better competitive ratio. The improvement comes both from the simplification and a tighter analysis. We define the algorithm QA, which stands for Quick Allocation. Rule 1a of ER-LS is deleted, so the allocation phase is then simplified to:

- Take any available task T_i. If $\overline{p_i}/\underline{p_i} \leq \tau$, then allocate T_i to the CPU side, otherwise allocate it to the GPU side.

Note that this allocation phase does not take into account any precedence relation or current schedule. Once this task is allocated to the CPU or GPU side, it is scheduled on the processor of this side that has completed its tasks the soonest.

One could wonder why the ratio τ is the best choice in the allocation phase. Intuitively, there are more CPUs than GPUs, so if $\overline{p_i}/\underline{p_i} < 1$, task T_i is executed faster on CPU, which is a lesser rare resource, therefore task T_i should be allocated to CPU. On the contrary, if $\overline{p_i}/\underline{p_i} > m/k$, then not only task T_i is executed faster on GPU, but if there are many independent tasks with the same processing times to compute, they will be executed faster all on GPUs than all on CPUs. Therefore we can safely allocate T_i to GPUs although this is a rare resource. When this ratio is between 1 and m/k, the loss is minimized when switching resource at the geometric mean of 1 and m/k, which is equal to τ.

We now show that QA is $\left(2\tau + 1 - \frac{1}{k\tau}\right)$ – competitive and that the analysis of this ratio is almost tight, as we provide an example on which QA leads to a makespan $\left(2\tau + 1 - \frac{1}{k}\right)$ times larger than the optimal solution.

Theorem 4. QA *is $\left(2\tau + 1 - \frac{1}{k\tau}\right)$ – competitive.*

Proof. We consider a graph, an online instance of this graph, and the schedule $\mathcal{S}$ computed by QA, of makespan C_{max}. We also consider an optimal schedule of this graph (later referred to by *the* optimal solution), and we let OPT be its makespan.

Let W_c (resp. W_g) be the total load on the CPUs (resp. GPUs). Let cp be a critical path of the task graph given the allocation of $\mathcal{S}$, and CP be the sum of the processing times of the tasks of cp in $\mathcal{S}$.

The objective is to prove that:

$$C_{max} \leq \left(2\tau + 1 - \frac{1}{k\tau}\right) OPT.$$

We first use Lemma 1 to bound C_{max} using the processor loads (W_c and W_g) and the length of the critical path (CP). Then, we bound the expression obtained in function of OPT to prove the result.

Lemma 1.

$$C_{max} \leq \frac{W_c}{m} + \frac{W_g}{k} + \left(1 - \frac{1}{m}\right) CP.$$

Proof Sketch. The proof of this lemma is deferred to the web supplementary material. The idea is very similar to the proof of Graham's list scheduling algorithm [11]. □

Bounding the loads

We denote by A_c (resp. A_g) the set of tasks placed on the CPU (resp. GPU) both by $\mathcal{S}$ and in the optimal solution. We denote by C_c (resp. C_g) the set of tasks placed on CPU (resp. GPU) by $\mathcal{S}$ but not in the optimal solution. The lowercase denotes the sum of the processing times of these sets.

The optimal makespan OPT is at least equal to the average work on CPU (and on GPU) in the optimal solution. In this solution, the tasks executed on CPU are tasks of the sets A_c and C_g. Tasks of A_c have the same processing time in $\mathcal{S}$ and in the optimal solution, as they are executed on CPU in both cases. Tasks of C_g are completed faster in $\mathcal{S}$ than in the optimal solution. More precisely, the allocation phase ensures that any task T_i of C_g verifies $\overline{p_i} \geq \tau \underline{p_i}$. Therefore, bounding OPT by the average work on CPU than on GPU gives the following inequalities:

$$OPT \geq \frac{1}{m}\left(a_c + \tau c_g\right), \tag{1}$$

$$OPT \geq \frac{1}{k}\left(a_g + \frac{c_c}{\tau}\right). \tag{2}$$

Using the fact that $k\tau \leq m$, we multiply by $m/k\tau$ both sides of Equation (1) to get:

$$\frac{m}{k\tau} OPT \geq \frac{m}{k\tau}\frac{a_c}{m} + \frac{m}{k\tau}\frac{\tau c_g}{m} \geq \frac{a_c}{m} + \frac{c_g}{k}.$$

We then simplify Equation (2) using that $k\tau \leq m$:

$$OPT \geq \frac{1}{k}\left(a_g + \frac{c_c}{\tau}\right) \geq \frac{a_g}{k} + \frac{c_c}{m}.$$

Summing these two inequalities, we get:

$$\left(1 + \frac{m}{k\tau}\right) OPT \geq \frac{a_c + c_c}{m} + \frac{a_g + c_g}{k} \geq \frac{W_c}{m} + \frac{W_g}{k}. \tag{3}$$

Bounding the critical path

We now bound the length of the critical path produced: every task of this critical path is also scheduled in the optimal schedule, and forms a path. Each task can be accelerated by a factor at most τ in the optimal schedule, so the time dedicated to process this path in the optimal schedule is at least CP/τ. Therefore, we have:

$$CP \leq \tau OPT. \tag{4}$$

Conclusion of the proof

Finally, from Lemma 1 we get:

$$C_{max} \leq \frac{W_c}{m} + \frac{W_g}{k} + \left(1 - \frac{1}{m}\right) CP.$$

Equations (3) and (4) lead to:

$$\begin{aligned} C_{max} &\leq \left(1 + \frac{m}{k\tau}\right) OPT + \left(\tau - \frac{\tau}{m}\right) OPT \\ &\leq \left(1 + \tau + \tau - \frac{1}{k\tau}\right) OPT. \end{aligned} \tag{5}$$

Hence, the theorem. □

We note that in the definition of QA, choosing a ratio slightly larger than τ can lead to a slightly smaller provable competitive ratio using the same proof as above up to Equation (5). However, this only leads to a marginal improvement of the upper bound of the competitive ratio, at the price of a more complex and less elegant criterion. Moreover, due the non-perfectly tight approximation ratio of both versions, we cannot decide which one has the actual best (tight) competitive ratio. Therefore, we only state this result as a remark.

Remark 2. *When $m > k$, choosing a separation ratio equal to $\tau\sqrt{\frac{m}{m-1}}$ instead of τ in the definition of* QA *leads to an algorithm that is $\left(1 + 2\sqrt{\frac{m-1}{k}}\right)$-competitive.*

We now prove that the competitive ratio of QA is almost tight in the following theorem.

Theorem 5. *The competitive ratio of* QA *is at least $\left(2\tau + 1 - \frac{1}{k}\right)$.*

Proof. We let ε be a small processing time and we define $q = (k-1)k$.

Consider a graph composed of $q + mk + 2$ tasks, labeled by T_i for i from 1 to $q+mk+2$. The first $q+mk+1$ tasks are all independent. These tasks are composed of four groups:

- The first q tasks have an infinite CPU processing time and a GPU processing time equal to $x = (k-1)/q = 1/k$.
- The next mk tasks have a CPU processing time of $(1 + \varepsilon)/k$ and a GPU processing time of $1/\sqrt{mk}$.
- The next task, T_{q+mk+1} has an infinite CPU processing time and a GPU processing time of ε.
- The last task of the graph, T_{q+mk+2} is a successor of T_{q+mk+1}. Its CPU processing time is equal to τ, and its GPU time is equal to $1 + \varepsilon$.

We consider the online setting in which the tasks T_i arrive in the order given by i. The ratio of CPU time over GPU time is larger than τ for every task except the last one. Then, QA schedules the first q tasks on k GPUs in time $qx/k = (k-1)/k$. Then, it schedules the next mk tasks on k GPUs in time $m/\sqrt{mk} = \tau$. Task T_{q+mk+1} is then scheduled on GPU in a time ε, after which the last task is scheduled on CPU. The makespan obtained is then equal to:

$$M_{\text{QA}} = 2\tau + \frac{k-1}{k} + \varepsilon.$$

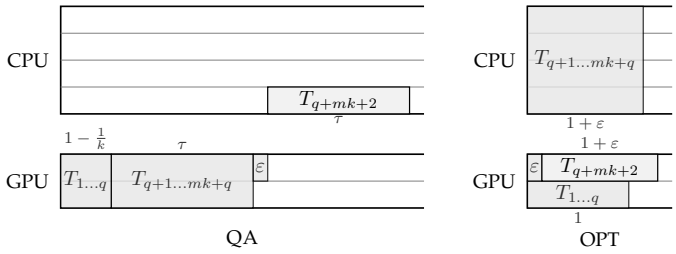


Figure 4. Schedule obtained by QA (left) and the optimal one (right).

Another possibility consists in scheduling first T_{q+mk+1} then T_{q+mk+2} on a single GPU, which takes a time $1+2\varepsilon$. In parallel, tasks T_1 to T_q are scheduled on the remaining $(k-1)$ GPUs, which takes a time $qx/(k-1)=1$. In parallel, we schedule tasks T_{q+1} to T_{q+mk} on m CPUs, which are then completed at time $1+\varepsilon$. The makespan obtained is then:

$$M = 1 + 2\varepsilon.$$

The schedules obtained are illustrated on Figure 4.

The ratio of the makespan obtained by QA divided by M is then equal to:

$$\frac{M_{\text{QA}}}{M} = \frac{2\tau + \frac{k-1}{k} + \varepsilon}{1+2\varepsilon} \quad \underset{\varepsilon\to 0}{\longrightarrow} \quad 2\tau + \frac{k-1}{k}.$$

□

4.2 A tunable competitive algorithm which performs well in practice

EFT, which stands for Earliest Finish Time, is one of the most intuitive algorithm to solve this problem: it schedules each task on a resource on which it will be completed the soonest. If several tasks are available, they are scheduled in an arbitrary order. This algorithm has good performance in practice as demonstrated in our experiments, because the load between resources is maintained balanced. However, in some instances, it can achieve makespans $m/k+2-\frac{1}{k}$ times longer than the optimal solution or the one computed by QA, with any tie-breaking strategy, as proved in Lemma 2.

Lemma 2. *The competitive ratio of* EFT *is at least* $(m/k+2-\frac{1}{k})$*, for any ordering policies when several tasks are available.*

Proof. Let ε be arbitrary small. We assume that k divides m and $m > 1$.

We first prove a weaker result, by exposing an instance on which EFT achieves a makespan equal to m/k where the optimal result is arbitrarily close to 1.

Consider $n := (m+k+1)m/k$ tasks composed of three types. m tasks of type A have a CPU computing time equal to $1+\varepsilon$ and a GPU computing time equal to 1. m^2/k tasks of type B, have a CPU computing time equal to $1-\varepsilon/2$ and a GPU computing time equal to ε. The remaining tasks, of type ε have a computing time of $\varepsilon/2$ on CPU and ε on GPU.

The online instance is decomposed into m/k phases, each starting by k tasks of type A and one task of type ε, which is the predecessor of m tasks of type B. The tasks of the next phase are the successors of one task of type B.

A nearly-optimal schedule executes all the tasks B and ε on GPUs then, once they are all completed, each task A on a single CPU. After the m/k phases, this achieves a makespan smaller than $1+n\varepsilon$, where n is the number of tasks in the graph.

EFT allocates the first k tasks A on GPU as they complete faster (1 versus $1+\varepsilon$), and the task ε on CPU. Then, all the GPUs are busy until time 1, so EFT allocates the next m tasks of type B on CPU, starting at time $\varepsilon/2$ and completing at time 1. Therefore, at the end of the first phase, all the processors are busy until time 1. Consequently, after the m/k phases, EFT achieves a makespan equal to m/k. We have then proved the first result.

This instance can now be modified to prove the lemma. Split the last phase into $k-1$ sub-phases, where each sub-phase contains the same tasks as one phase of the previous instance, but which computing time are divided by k. EFT schedules this phase in time $1-\frac{1}{k}$, using all processors, achieving a makespan equal to $(m-1)/k$. A nearly-optimal schedule executes all B and ε subtasks on GPUs, then executes the $k(k-1)$ A subtasks on $k-1$ CPUs. Therefore, compared to the previous instance, the makespan achieved by EFT so far is slightly smaller than before, but OPT has one idle CPU. Now, add a new phase at the end of the instance composed of one task A and one task ε followed by k tasks of type C, which have an infinite CPU computing time and a GPU computing time equal to 1. The graph and schedules obtained are represented in Figures 5 and 6. The last A task is noted A' to differentiate it from the previous A tasks.

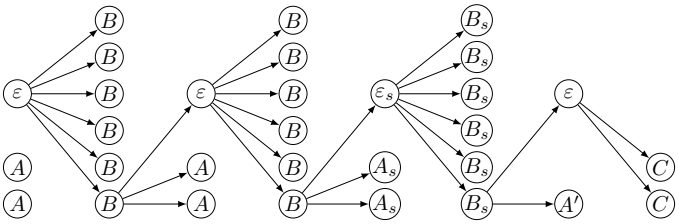


Figure 5. Example of instance built with $m=6$, $k=2$. Note that this instance contains 1 subphase of subtasks denoted by the subscript s.

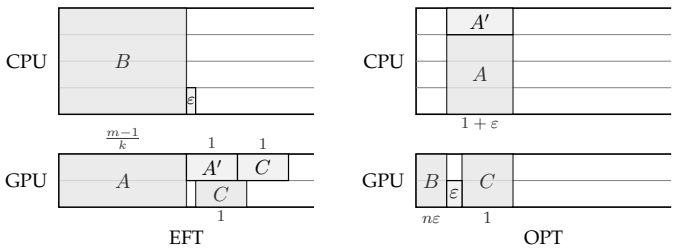


Figure 6. Schedule obtained by EFT (left) and an nearly-optimal one (right). The ε tasks except the last one are not depicted for simplicity.

The optimal schedule executes the task A on the last idling CPU, and each task C on a GPU. The makespan obtained is then at most $1+n\varepsilon$, where n is the number of tasks in the graph.

EFT schedules $k+1$ tasks of this phase on GPU. Its makespan is then increased by 2, to reach a value of $m/k+2-\frac{1}{k}$; hence, the lemma. □

We propose a new tunable algorithm, named MIXEFT that benefits both from the performance of EFT on most instances, and from the robustness of QA on the hardest graphs. The idea is to improve EFT by switching to a guaranteed algorithm if EFT does not perform well enough. The algorithm is composed of two phases. In the first phase, it is equal to EFT except that it also simulates the schedule that QA would have produced on the same instance. If the makespan obtained by EFT is more than λ times larger than the makespan obtained by the simulated QA (for a fixed positive parameter λ) we switch to the second phase, and MIXEFT from this point behaves as QA. A small λ leads to a smaller competitive ratio, but may degrade the performance of MIXEFT in practice. We propose to use a value of λ between 1 and 2. The pseudocode is provided in Algorithm 1. The time complexity to schedule a task is dominated by the computation of the schedule that would be achieved by QA, which is $O(n \log m)$. Note that this algorithm assumes that dependencies are learned once they are met. This yields the partial DAG required to simulate QA.

Algorithm 1: MIXEFT (λ)

```
StayEFT ← True
while there is a new task T_i do
    if StayEFT then
        C_EFT ← makespan obtained by scheduling T_i as EFT
        C_QA ← makespan that QA would have obtained in this instance
        StayEFT ← (C_EFT ≤ λ C_QA)
    if StayEFT then
        Schedule T_i as soon as possible on the resource on which it completes the earliest
    else
        Schedule T_i as soon as possible on CPU if p̄_i/p_i ≤ τ and on GPU otherwise
```

Theorem 6. MIXEFT *is* $(\lambda+1)(2\tau+1)$*-competitive.*

Proof. Let OPT be the length of the optimal schedule. QA solves the whole graph in less than $(2\tau+1)OPT$. Therefore it completes any subset of the graph in less than this time (as the optimal makespan cannot increase when removing tasks). Therefore, the time to complete the first phase is less than $\lambda(2\tau+1)OPT$. The time to complete the second phase (which may be empty) it then smaller than $(2\tau+1)OPT$. The whole graph is then completed in less than $(\lambda+1)(2\tau+1)OPT$. □

5 THE ALLOCATION IS MORE DIFFICULT THAN THE SCHEDULE

The proofs of the lower bounds presented in Section 3 and the competitive algorithms designed in Section 4 focus substantially more on the allocation decisions (i.e., deciding whether to execute a task on CPUs or on GPUs) than on the scheduling decisions (e.g., if a task is allocated to CPUs, deciding its starting time and the CPU). In this section, we support this observation by showing that if the allocation decisions are given by an oracle, then any online list scheduling algorithm is $(3-\frac{1}{m})$-competitive, which is the best competitive ratio achievable.

A corollary of this result is that, when $m = k$, QA has the best competitive ratio achievable. This also shows that the best lower bound proved in Remark 1 (which holds if $m = k \geq 4$) is not tight, as it is only equal to 2.

Theorem 7. *If the allocation of each task is fixed, any online list scheduling algorithm is* $\left(3-\frac{1}{m}\right)$*-competitive for the problem* $(Pm, Pk)|prec, online|C_{max}$.

Proof. Consider a graph where each task has a fixed allocation, an online instance of this graph, and the schedule $\mathcal{S}$ computed by any online list scheduling algorithm, of makespan C_{max}. Let W_c (resp. W_g) be the total load on the CPUs (resp. GPUs). Let cp be a critical path of $\mathcal{S}$, and CP be the sum of the processing times of the tasks of cp in $\mathcal{S}$.

The result of Lemma 1 stated in the proof of Theorem 4 holds, therefore we have:

$$C_{max} \leq \frac{W_c}{m} + \frac{W_g}{k} + \left(1-\frac{1}{m}\right) CP.$$

Let OPT be the optimal makespan given the fixed allocation. The m CPUs have to execute tasks whose execution time sum to W_c, so $OPT \geq W_c/m$. Similarly, $OPT \geq W_g/k$, and as CP is the length of the critical path, we have $OPT \geq CP$. Therefore, we conclude that:

$$C_{max} \leq \left(3-\frac{1}{m}\right) OPT.$$

□

We now show that this upper bound is tight.

Lemma 3. *If the allocation of each task is fixed, no online scheduling algorithm has a competitive ratio smaller than* $\left(3-\frac{1}{m}\right)$ *for the problem* $(Pm, Pk)|prec, online|C_{max}$.

Proof. We assume $m \geq 2$. Note that the result also holds for $m = 1$, with a simpler example without the second group of tasks built below. Let $\mathcal{A}$ be an online scheduling algorithm. We let n be an integer multiple of $km(m-1)$ and an adversary will build a graph G composed of the $2n+1$ following tasks:

- tasks T_1 to T_n have a GPU computing time equal to k/n and an infinite CPU computing time;
- tasks T_{n+1} to T_{2n} have a CPU computing time equal to $(m-1)/n$ and an infinite GPU computing time;
- task T_{2n+1} has a CPU computing time equal to 1 and an infinite GPU computing time.

In the graph G, there will exist $i \in [1, n]$ and $j \in [n+1, 2n]$ such that the dependencies of G are from task T_i to tasks $T_{n+\ell}$ for every $\ell > 0$ and from task T_j to task T_{2n+1}.

Every such graph can be scheduled in time $1+(k+m-1)/n$: schedule each task as soon as possible starting task T_i at time 0, task T_j at time k/n and task T_{2n+1} at time $(k+m-1)/n$, see Figure 7. Tasks $T_{1\ldots n}$ are completed on k GPUs in time $n/k * k/n = 1$, tasks $T_{n+1\ldots 2n}$ are completed on $m-1$ CPUs in time $(m-1)/n * n/(m-1) = 1$, and task T_{2n+1} in time 1.

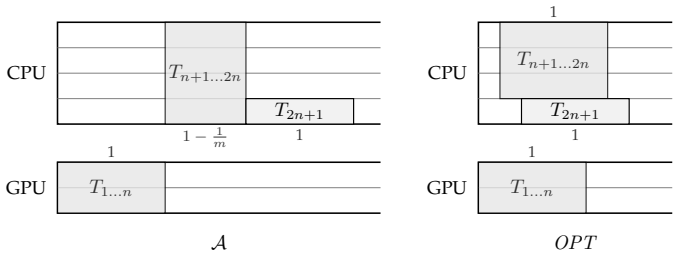


Figure 7. Schedules obtained by $\mathcal{A}$ and OPT.

Now, consider algorithm $\mathcal{A}$. The adversary selects the last task of $T_{1\dots n}$ to complete as the predecessor of every task $T_{n+\ell}$ for every $\ell > 0$. Similarly, it selects the last task of $T_{n+1\dots 2n}$ to complete as the predecessor of task T_{2n+1}. The makespan obtained is then at least the time necessary to complete $T_{1\dots n}$ on k GPUs, plus the time to complete $T_{n+1\dots 2n}$ on m CPUs, plus the time to complete T_{2n+1} on 1 CPU, see Figure 7:

$$1 + \frac{n}{m}\,\frac{m-1}{n} + 1 = 3 - \frac{1}{m}.$$

Therefore, the competitive ratio of $\mathcal{A}$ is at least:

$$\frac{3-\frac{1}{m}}{1+\frac{1}{n}(k+m-1)} \quad \underset{n\to\infty}{\longrightarrow} \quad 3-\frac{1}{m}.$$

□

6 Extension to multiple types of processors

In this section, we generalize our study to $Q \geq 2$ types of processors, which allows modeling a platform composed of several accelerator types. For comparison, in the offline setting, Amaris et al. [15] provide a $Q(Q+1)$-approximation. We denote by m_q the number of processors of type q, and we assume that they are ordered such that $m_q \geq m_{q+1}$. The computing time of task T_i on processor type q is denoted by $p_{i,q}$.

Our first result directly extends the lower bounds of Section 3 for Q processor types. We only detail the proof of Theorem 8 here, but the same generalization can be done for every lower bound presented in Table 1, replacing τ by $\sqrt{\sum_{q=1}^{Q-1} m_q/m_Q}$.

Theorem 8. *No online algorithm for Q processor types has a competitive ratio smaller than $\sqrt{\sum_{q=1}^{Q-1} m_q/m_Q}$.*

Proof. Let $\mathcal{P}$ be the target platform composed of Q types of processors. Consider an alternative platform $\mathcal{P}'$ composed of 2 types of processors, m' CPUs and k' GPUs, with $m' = \sum_{q=1}^{Q-1} m_q$ and $k' = m_Q$.

Any instance G' on $\mathcal{P}'$ can be simulated by an instance G on $\mathcal{P}$, which has the same vertices and edges as G'. The processing times of the tasks of G are defined as follows: for any task T_i , $p_{i,Q}$ is equal to the GPU processing time of T_i on $\mathcal{P}'$ and $p_{i,q}$, for $q = 1, \dots, Q-1$, is equal to its CPU processing time. Therefore, a schedule of G on $\mathcal{P}$ can be adapted as a schedule of G' on $\mathcal{P}'$ achieving the same makespan, and vice-versa: the processor types 1 to $Q-1$ are equivalent in $\mathcal{P}$ and can be mapped to the CPUs of $\mathcal{P}'$.

Suppose by contradiction that an online algorithm has a competitive ratio smaller than $\sqrt{\sum_{q=1}^{Q-1} m_q/m_Q} = \sqrt{m'/k'}$ on $\mathcal{P}$. Its competitive ratio on $\mathcal{P}'$ is then smaller than $\sqrt{m'/k'}$, which violates Theorem 1. □

We also adapt the QA algorithm (and thus MixEFT) for this setting, by changing its allocation phase:

- Allocate T_i to a processor type q that minimizes $p_{i,q}\,/\sqrt{m_q}$.

Note that with $Q = 2$, this algorithm is equal to the original QA. Theorem 9 (proved below) generalizes the competitive ratio. However, the gap with the lower bound proved above increases with Q, as the lower bound is roughly the square root of the sum of ratios m_q/m_Q whereas the competitive ratio of QA is roughly the sum of the square roots of the same ratios.

Theorem 9. *On Q types of processors,* QA *is* $\left(\sqrt{\frac{m_1}{m_Q}} + \sum_{q=1}^{Q}\sqrt{\frac{m_q}{m_Q}}\right)$ *-competitive.*

In comparison, there is an instance similar to the one of Lemma 2 on which EFT achieves a ratio larger than $\sum_{q=1}^{Q} m_q/m_Q$. Indeed, by setting identical computing times to processor types 1 to $Q-1$, EFT behaves as if there were $\sum_{q=1}^{Q-1} m_q$ CPUs and m_Q GPUs, which leads to this result.

The generalization of QA may seem straightforward, but it brings new insights on the underlying principles. We can see that the value $\tau = \sqrt{m/k}$ hides several concepts. Comparing the processing times ratio $\overline{p_i}/\underline{p_i}$ to τ is actually a way to select the resource type q that minimizes $p_{i,q}\,/\sqrt{m_q}$. The competitive ratio of QA on two types of processors, $2\tau+1$, is the sum of two terms. The first one is the maximal deceleration of a task compared to the optimal schedule, which is equal to τ on two types of processors and generalized to $\sqrt{m_1/m_Q}$. The second one is the maximal loss when scheduling too many tasks on the fastest resource type while all the others may idle, which is equal to $\tau+1$ on two types of processors, and generalizes to $\left(\sum_{q=1}^{Q}\sqrt{m_q}/\sqrt{m_Q}\right)$. The gap with the lower bound of Theorem 8 is explained by the fact that the proposed lower bound does not exploit the different execution times of tasks on the $Q-1$ first resource types. The construction proposed in Section 3 actually strongly relies on the fact that tasks have only two different processing times: either all tasks are executed on GPU, or at least one of them is not executed on GPU. Both cases must lead to the same processing time for the lower bounds to hold. It should be noted that this generalization exhibits another meaning of the value τ: it is equal to the value $\sqrt{\sum_{q=1}^{Q-1} m_q/m_Q}$ when $Q = 2$.

Proof of Theorem 9. We consider a graph G and the schedule $\mathcal{S}$ computed by QA, of makespan C_{max}. We consider also an optimal offline solution, to which we will refer as *the* optimal solution, of makespan OPT. Let W_q be the total load on the processors of type q for each $q \in \{1, \dots, Q-1\}$. Let cp be a critical path of $\mathcal{S}$, and CP be the sum of the processing times of the tasks of cp in $\mathcal{S}$.

First, we prove that:

$$C_{max} \leq \sum_{q=1}^{Q} \frac{W_q}{m_q} + CP. \tag{6}$$

As in Theorem 4, consider a path p of tasks of G whose execution starts the soonest and completes exactly at time C_{max} in $\mathcal{S}$. When no task of p is being executed, one type of processor is necessarily busy because of the scheduling strategy, which always schedules an available task if one processor of each type is idle. The total amount of time during which at least one type of processor has no idle resource is at most $\sum_{q=1}^{Q} \frac{W_q}{m_q}$; hence, the result.

We now bound CP. Consider a task T_i that is executed on processor type ℓ in QA and q in the optimal solution. We have, by definition of QA, m_1 and m_Q:

$$p_{i,\ell} \leq \sqrt{\frac{m_\ell}{m_q}}\, p_{i,q} \leq \sqrt{\frac{m_1}{m_Q}}\, p_{i,q}. \tag{7}$$

Summing over the tasks of cp, we obtain:

$$CP \leq \sqrt{\frac{m_1}{m_Q}} OPT. \tag{8}$$

We consider the workload W_q^* on processor type q in the optimal solution, which is not larger than $m_q OPT$. For any processor type ℓ, let C_q^ℓ be the sum of the computing times on processors of type ℓ of tasks allocated to processor type ℓ in QA and to processor type q in the optimal solution. We can lower bound W_q^* in the optimal solution by the quantities C_ℓ^q, using the first inequality of Equation (7):

$$OPT \geq \frac{W_q^*}{m_q} \geq \frac{1}{m_q} \sum_{\ell=1}^{Q} \sqrt{\frac{m_q}{m_\ell}} C_q^\ell$$

$$\sqrt{\frac{m_q}{m_Q}} OPT \geq \sum_{\ell=1}^{Q} \frac{C_q^\ell}{\sqrt{m_Q m_\ell}} \geq \sum_{\ell=1}^{Q} \frac{C_q^\ell}{m_\ell}.$$

Now, summing over all processor types q, we get:

$$\begin{aligned} \frac{1}{\sqrt{m_Q}} \left(\sum_{q=1}^{Q} \sqrt{m_q} \right) OPT &\geq \sum_{q=1}^{Q} \sum_{\ell=1}^{Q} \frac{C_q^\ell}{m_\ell} \geq \sum_{\ell=1}^{Q} \frac{1}{m_\ell} \sum_{q=1}^{Q} C_q^\ell \\ &\geq \sum_{\ell=1}^{Q} \frac{W_\ell}{m_\ell}. \end{aligned} \tag{9}$$

Finally, combining Equations (6), (8) and (9), we get the result:

$$C_{max} \leq \frac{1}{\sqrt{m_Q}} \left(\sqrt{m_1} + \sum_{q=1}^{Q} \sqrt{m_q} \right) OPT. \quad \square$$

7 Simulations

We now provide simulations to illustrate the performance of both competitive algorithms and simple heuristic strategies on various task graphs.

7.1 Baseline heuristics

In addition to the four online algorithms discussed above (ER-LS from [1], QA, EFT, and MixEFT, a combination of EFT and QA implemented with $\lambda = 2$ unless otherwise specified), we consider two simple strategies that follow the same scheme as QA, with a different allocation criteria: Quickest allocates each task to the resource type on which its computing time is smaller; Ratio allocates a task on GPUs if and only if its GPU computing time is at least m/k times smaller than its CPU computing time. Intuitively, Quickest should perform well on graphs on which the critical path is preponderant. On the opposite, Ratio should perform well on graphs with a high parallelism throughout the execution.

We also used the offline HEFT algorithm [19], which is known to perform well in practice, as a baseline to compare all online strategies. Moreover, backfilling is performed following HEFT insertion policy.

7.2 Experimental setup

Experiments have been performed on a platform consisting of $m = 20$ CPUs and $k = 2$ GPUs. We used three types of instances: realistic DAGs corresponding to the Cholesky factorization, random DAGs used in the literature, and ad hoc instances designed to be difficult for this problem and specifically for QA.

Cholesky factorization is a linear algebra application whose parallel implementation usually uses a blocked algorithm on a tiled matrix for performance issues. We consider matrix sizes ranging from 2×2 tiles to 15×15 tiles, which leads to DAGs with 4 to 680 tasks. Tasks correspond to four linear algebra kernels: GEMM, SYRK, TRSM, and POTRF. Their respective processing times on a CPU are set to 170ms, 95ms, 88ms, and 33ms, and on a GPU to 5.95ms, 3.65ms, 8.11ms, and 15.6ms, which corresponds to measures [20], [21] made using the Chameleon software [22].

The random instances come from the STG set [23], which is often used in the literature to compare the performance of scheduling strategies. The set contains instances with 50 to 5000 nodes. We report here the simulations made with 180 graphs of 300 nodes each. Simulations performed with sizes 100 and 500 lead to similar conclusions. We consider that the cost generated by the STG random generator is the processing time of the corresponding task on a GPU. Based on the previous measures for linear algebra kernels, we assume that the average speedup between CPU and GPU is around 15 with a large variance. Thus, to obtain the processing time of a task on CPU, we multiply its cost on GPU by a random value with expected value 15 and standard deviation 15. For that, we use a gamma distribution because it has been advocated for modeling job runtimes [24], it is positive and it is possible to specify its expected value and standard deviation by adjusting its parameters. In this paper, we do not focus on the differences between the four random DAG generator. We refer the interested reader to the web supplementary material for this discussion.

Finally, specific random instances have been designed to test the limitations of QA. These ad hoc instances consist of a chain of tasks together with a set of independent tasks, such that all cores are expected to finish simultaneously if a GPU

is dedicated to the chain and all independent tasks are load-balanced on the other cores. These instances are assumed to be problematic for online strategies that are unaware of the critical path. The expected processing time of a task on a GPU is 1 (with a standard deviation of 0.1) and the expected processing time on a CPU varies from $(m/k)^{-1/4}$ to $(m/k)^{5/4}$ (with a standard deviation equal to 10% of this expected value). For a given expected CPU cost μ, the number of tasks in the chain is $\lceil \frac{n}{m/\mu+k} \rceil$, where $n = 300$ is the total number of tasks. Therefore, the larger μ, the longer the chain. The DAG size only affects the variability of the results (larger variability with $n = 100$ and lower with $n = 500$).

7.3 Results

Figure 8 depicts the performance of the six online scheduling algorithms. Except when varying its parameter (Figure 8(d)), MIXEFT performs exactly as EFT (and is thus omitted for better readability) because none of our instances leads to a switch to QA. A more detailed and complete analysis can be found in the web supplementary material. In particular, Figure 8(a) is extended to different numbers of CPUs and GPUs, Figure 8(b) is separated in four based on the type of DAGs in the STG set and Figure 8(d) is broken down into the three considered types of instances.

On Cholesky DAGs (Figure 8(a)), EFT (and thus MIXEFT) is always the best strategy. The only difference between QA and ER-LS concerns the first tasks (as we removed Step 1a in QA), which explains why their behaviour is similar for large graphs. QA, ER-LS, and RATIO all put POTRF tasks on the CPU, which leads to performance loss when the graph is small because its parallelism is limited and the GPUs are often idle. However, it is acceptable for larger graphs in which many tasks may be executed in parallel on the GPUs. On the contrary, QUICKEST puts all tasks on the GPUs. This is efficient for small graphs with low parallelism but it is worse than RATIO for large graphs.

Figure 8(b) shows similar trends on the random graphs from STG set: EFT (and thus MIXEFT) gives the best results, followed by QA and ER-LS. Note that with graphs from STG set of different sizes, RATIO may be more efficient than QUICKEST due to DAG characteristics that depend on the size in the STG data set.

Figure 8(c) first shows that EFT (and MIXEFT) is almost always the best online heuristic for ad hoc graphs. For extreme values of the expected CPU processing time μ (significantly smaller than 1 or larger than m/k), all four other heuristics are equivalent and perform well. Otherwise, when μ is slightly larger than 1, the instance contains many independent tasks and QUICKEST is almost m/k worst than HEFT because scheduling independent tasks on GPUs is not efficient. Symmetrically, when μ is slightly smaller than m/k, the instance contains a large critical path and RATIO shows poor performance, because it schedules the critical path on CPUs. QA and ER-LS take the best of these two strategies, and have a worst performance $\sqrt{m/k} \approx 3$ times larger than HEFT, when μ is close to $\sqrt{m/k}$.

Figure 8(d) shows that MIXEFT behaves like QA when its λ parameter is smaller than 1, and rapidly changes to mimic EFT when the parameter increases and exceeds 1.

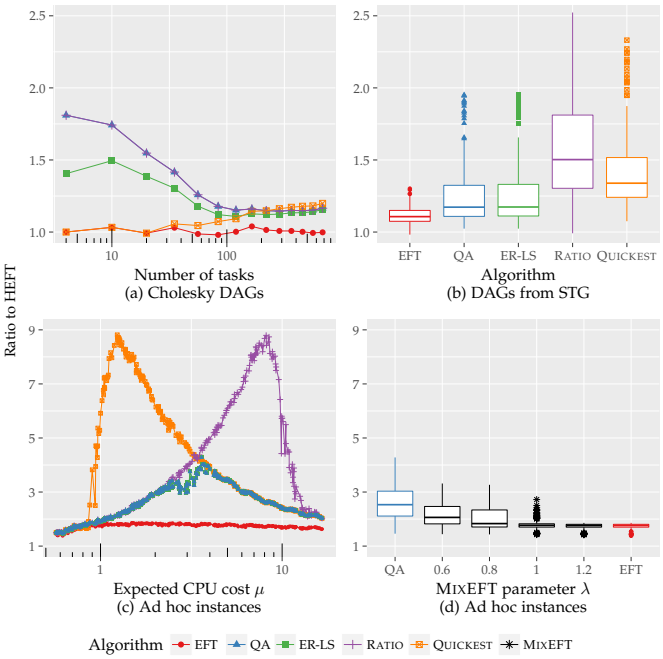


Figure 8. Ratios of the makespan over HEFT for EFT, QA, ER-LS, RATIO, QUICKEST, and MIXEFT with $m = 20$ CPUs and $k = 2$ GPUs. Except in Figure (d), MIXEFT is not shown because it performs exactly as EFT. In Figure (d), ER-LS, RATIO, and QUICKEST are discarded.

Note that in all studied instances, EFT was never far from HEFT and that there is no practical gain of using MIXEFT rather than EFT. The main advantage of MIXEFT lies in its competitive ratio inherited from QA, whereas EFT can lead to very large makespans on specific instances.

These simulations show that QA achieves the same performance as ER-LS, except for Cholesky DAGs where the latter is better, especially for small instances. However, both these algorithms, despite their worst-case performance, are significantly outperformed by EFT and MIXEFT, even on difficult instances such as the ad hoc ones.

8 CONCLUSION

In this paper, we have focused on the problem of scheduling task graphs on hybrid platforms made of two types of processors, such as CPUs and GPUs. We have studied the online case, when only the tasks whose predecessors are all completed are known to the scheduler, and the graph is thus gradually discovered. We proved that no scheduling algorithm can have a competitive ratio smaller than $\sqrt{m/k}$, and studied how this ratio varies when more knowledge on the graph is given to the scheduler and/or tasks may be migrated between processors. We have proposed a $(2\sqrt{m/k}+1)$-competitive algorithm as well as a mixed strategy, which is both $\Theta(\sqrt{m/k})$-competitive and performs as well as the best heuristics in practice. This is demonstrated through an extensive set of simulations. We have also extended the lower bounds and the competitive algorithms to the case with more types of processors.

Future work includes several directions. For independent tasks, there is still a gap between the best lower bound on online algorithms competitive ratio (2) and the best online algorithm (3.85-competitive) [14]. The best algorithm to

schedule offline DAGs of tasks is a 6-approximation relying on linear programming [1]. Improving this algorithm, on the approximation factor or the complexity, is therefore an open problem. Another research direction consists in exploiting task parallelism. A 2-approximation has been exhibited for offline scheduling of independent moldable tasks on hybrid platforms in [9], but this problem with precedence constraints remains unexplored. Finally, in order to model more closely realistic problems, it remains to take into account communication times when moving data from/to the GPUs, and to cope with inaccurate processing time estimates.

REFERENCES

[1] M. Amaris, G. Lucarelli, C. Mommessin, and D. Trystram, "Generic algorithms for scheduling applications on hybrid multi-core machines," in *Euro-Par 2017: Parallel Processing*, 2017, pp. 220–231.

[2] L.-C. Canon, L. Marchal, B. Simon, and F. Vivien, "Online scheduling of task graphs on hybrid platforms," in *Euro-Par 2018: Parallel Processing*, 2018, pp. 192–204.

[3] 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.

[4] L.-C. Canon, L. Marchal, and F. Vivien, "Low-cost approximation algorithms for scheduling independent tasks on hybrid platforms," in *Euro-Par 2017: Parallel Processing*, 2017, pp. 232–244.

[5] 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 *IEEE IPDPS*, 2017, pp. 768–777.

[6] S. Kedad-Sidhoum, F. Monna, and D. Trystram, "Scheduling tasks with precedence constraints on hybrid multi-core machines," in *IEEE IPDPS Workshops*, 2015, pp. 27–33.

[7] O. Beaumont, L. Eyraud-Dubois, and S. Kumar, "Fast approximation algorithms for task-based runtime systems," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 17, 2018.

[8] 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*, vol. 30, no. 2, pp. 323–343, 1999.

[9] R. Bleuse, S. Hunold, S. Kedad-Sidhoum, F. Monna, G. Mounié, and D. Trystram, "Scheduling independent moldable tasks on multi-cores with gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 9, pp. 2689–2702, 2017.

[10] M. A. Aba, L. Zaourar, and A. Munier, "Approximation algorithm for scheduling applications on hybrid multi-core machines with communications delays," in *IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops*, 2018, pp. 36–45.

[11] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969.

[12] O. Svensson, "Hardness of precedence constrained scheduling on identical machines," *SIAM Journal on Computing*, vol. 40, no. 5, pp. 1258–1274, 2011.

[13] C. Imreh, "Scheduling problems on two sets of identical machines," *Computing*, vol. 70, no. 4, pp. 277–294, 2003.

[14] L. Chen, D. Ye, and G. Zhang, "Online scheduling of mixed CPU-GPU jobs," *Int. Journal of Foundations of Computer Science*, vol. 25, no. 06, pp. 745–761, 2014.

[15] M. Amaris, G. Lucarelli, C. Mommessin, and D. Trystram, "Generic algorithms for scheduling applications on heterogeneous multi-core platforms," CoRR, Tech. Rep., 2017.

[16] 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.

[17] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst, "Data-aware task scheduling on multi-accelerator based platforms," in *ICPADS*, Dec 2010, pp. 291–298.

[18] J. Y. Leung, *Handbook of scheduling: algorithms, models, and performance analysis*. CRC Press, 2004.

[19] H. Topcuoglu, S. Hariri, and M. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE TPDS*, vol. 13, no. 3, pp. 260–274, 2002.

[20] E. Agullo, O. Beaumont, L. Eyraud-Dubois, and S. Kumar, "Are static schedules so bad? A case study on Cholesky factorization," in *IPDPS*. IEEE, 2016.

[21] O. Beaumont, T. Cojean, L. Eyraud-Dubois, A. Guermouche, and S. Kumar, "Scheduling of linear algebra kernels on multiple heterogeneous resources," in *HiPC*, 2016.

[22] "Chameleon, a dense linear algebra software for heterogeneous architectures," https://project.inria.fr/chameleon.

[23] T. Tobita and H. Kasahara, "A standard task graph set for fair evaluation of multiprocessor scheduling algorithms," *Journal of Scheduling*, vol. 5, no. 5, pp. 379–394, 2002.

[24] D. Feitelson, "Workload modeling for computer systems performance evaluation," *Book Draft, Version 1.0.1*, pp. 1–601, 2014.



Louis-Claude Canon received the PhD degree in computer science from the University of Nancy in 2010. He is an associate professor of computer science at the University of Franche-Comté and conducting research in the FEMTO-ST laboratory. His main research interests include scheduling, stochastic optimization, and reproducible research.



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



Bertrand Simon graduated in Computer Sciences and received his PhD from École Normale Supérieure de Lyon (ENS Lyon, France). He is now a researcher at the University of Bremen (Germany). His research interests include data structures, parallel computing and scheduling.



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