Online Scheduling of Task Graphs on Hybrid Platforms

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

Online Scheduling of Task Graphs on Hybrid Platforms

Louis-Claude Canon[1,2], Loris Marchal[2], Bertrand Simon[2], and Frédéric Vivien[2]

[1] FEMTO-ST Institute – Université de Bourgogne Franche-Comté
16 route de Gray, 25 030 Besançon, France
louis-claude.canon@univ-fcomte.fr
[2] Univ Lyon, CNRS, ENS de Lyon, Inria, Université Claude-Bernard Lyon 1, LIP
UMR5668, F-69342, LYON Cedex 07, France
loris.marchal@ens-lyon.fr, frederic.vivien@inria.fr,
bertrand.simon@ens-lyon.fr

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 [2], 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 $\Theta(\sqrt{m/k})$. 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.

Keywords: 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 [24]. 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 [22], XKaapi [7], or StarPU [4]. Most of these frameworks model an application as a Directed Acyclic Graph (DAG) of tasks, where nodes represent tasks and edges represent dependences between tasks. While task graphs have been widely studied in the theoretical scheduling literature [14], 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 rely neither on the structure of the graph nor on the knowledge of tasks' running times. 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. Third, even if part of the graph is available, schedulers (such as StarPU [4]) usually avoid traversing large parts of the graph each time they take a decision in order to strongly limit the time needed to take decisions. Finally, tasks' processing times are not always known beforehand, and the occasionally available predictions may not be very accurate, as two successive executions of the same task may result in slightly different timings.

There has recently been an effort of the scheduling community to fill the gap between the assumptions used in theoretical studies and the information available to the underlying schedulers of runtime systems (see details in Section 2). Schedulers for independent tasks on hybrid platforms have first been proposed [5, 8, 11]. Some of them have been adapted for task graphs: [20] extends the algorithm of [5] to the (offline) scheduling of task graphs, while [2] adapts an online scheduler for independent tasks on hybrid platforms [17] to obtain a competitive online scheduler for task graphs.

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 [2]: 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 are terminated. 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 kernels whose processing times have been acquired through extensive benchmarking; this happens in particular in linear 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.

The closer related work considering the very same problem is [2], which provides a $4\sqrt{m/k}$-competitive algorithm for this problem. 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 $\sqrt{m/k}$. 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 to reduce the lower bound to $\frac{1}{2}(m/k)^{1/4}$ (Section 3).
- We propose a $(2\sqrt{m/k}+1)$-competitive algorithm, by refining both the algorithm and the analysis of [2] (Section 4.1).
- We propose a simple heuristic (Section 4.2) based on 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 5).

2 Related Work

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

Offline algorithms. Several schedulers for independent tasks on hybrid platforms have been proposed. Bleuse et al. [8] 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 in [5, 11] and achieve approximation ratios respectively equal to 2 and $2+\sqrt{2}$. For tasks with precedence constraints, Kedad-Sidhoum et al. [18] provided a tight 6-approximation algorithm based on linear programming. In a different setting, Raravi et al. [21] also consider the same platform composed of two types of processors, on which the objective is to schedule a set of chains of tasks, with each task having a release date and a deadline. They design an algorithm that schedules the tasks of each chain on the same processor, under some assumptions such as the existence of a valid schedule on slightly slower processors.

Online algorithms. When tasks with precedences are released over time, Graham's List Scheduling algorithm [16] is 2-competitive on homogeneous processors (note that this is also the best offline approximation for this problem). On our model with two sets of processors, Imreh [17] and Chen et al. [13] proposed an algorithm to schedule independent tasks with a competitive ratio smaller than 4. Based on this work, Amaris et al. [2] exhibited an online algorithm for precedence constraints, achieving a competitive ratio of $4\sqrt{m/k}$.

Runtime strategies. Actual runtime schedulers usually rely on low-complexity scheduling policies to limit the time needed to allocate tasks. For instance, StarPU [4] 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** in [3]), which corresponds to the classical Earliest Finish Time (EFT) scheduling policy [19]. Other strategies have been proposed that take into account communication times, or precomputed task priorities, depending on the descendants of each task. We include similar information in the design of the lower bounds on competitive ratios (Section 3).

3 Lower bound on competitive algorithms

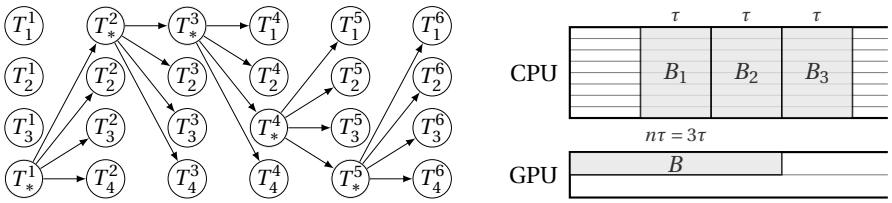
In this section, we provide a lower bound on the competitive ratio of any online algorithm, as outlined in the following theorem. We also study how adding flexibility to task processing or giving some knowledge of the graph to the scheduler impacts this lower bound.

Theorem 1. *No online algorithm has a competitive ratio smaller than* $\sqrt{m/k}$.

Proof. We prove this result here only when $\tau = \sqrt{m/k}$ is an integer. The proof for the general case can be found in the corresponding research report [10]. Consider an online algorithm $\mathcal{A}$. We fix an integer n, which will later be made as large as we want for the competitive ratio to get closer to τ. We use an adversary proof: an adversary dynamically builds the graph depending on the current schedule produced by $\mathcal{A}$. This results in a graph composed of nm tasks denoted by T_i^j, with $j = 1, \ldots, n\tau$ and $i = 1, \ldots, k\tau$. Each task has a CPU processing time of τ and a GPU processing time of 1.

The procedure consists of $n\tau$ phases. During the jth phase, $k\tau$ tasks are released (tasks T_i^j for $i = 1, \ldots, k\tau$), without dependences between these tasks. 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 1a for an illustration of the resulting graph.

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 1b. Each bucket B_i for $i = 1, \ldots, n$ contains all m CPUs, has a duration



(a) Graph built by the adversary. (b) Buckets used by $\mathcal{S}$.

Fig. 1. Illustration of the graph and the buckets for $\tau = 2$, $k = 2$, $n = 3$.

of τ, and starts at time $i\tau$. Note that m tasks fit into each of these buckets. The last bucket, B, contains one GPU, starts at time 0 and lasts for 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 $(n+1)\tau$.

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:

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

It seems from the above proof that 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 the following lemma, which states that given an allocation of the tasks to the two types of resources, scheduling them among the $m+k$ resources can be done with constant competitive ratio (for the proof, please refer to [10]).

Lemma 1. *If each task can be processed on a single type of resource, then any online list scheduling algorithm is* $(3 - \frac{1}{m})$*-competitive, and no online algorithm has a smaller competitive ratio.*

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 pre-computing some information offline before submitting the tasks. For instance, offline schedulers

Table 1. Lower bounds for various combinations of flexibility in task processing and knowledge given to the scheduler (BL stands for bottom-level).

Flexibility	Knowledge	Lower bound	Special cases
None or Spoliation	None or BL	$\sqrt{m/k}$	if BL and $k = 1$: $\frac{1}{2}\sqrt{m/k}$
	BL + descendants	$\frac{1}{2}(m/k)^{1/4}$	
Migration	None or BL	$\frac{1}{2}\sqrt{m/k}$	if BL and $k = 1$: $\frac{1}{4}\sqrt{m/k}$
	BL + descendants	$\frac{1}{4}(m/k)^{1/4}$	

usually rank available tasks with priorities based on the dependences. 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 [25] 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-bound of Theorem 1: it is possible to transform the above proof using an adversary that submits tasks with identical bottom-levels in each phase (see details in the corresponding research report [10]). When there is exactly one GPU, the lower bound is decreased to $\frac{1}{2}\sqrt{m/k}$. 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 result in the same lower bounds.

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 $\frac{1}{2}(m/k)^{1/4}$ when m/k is large enough, so no constant-factor competitive algorithm exists. Note that all the tasks are equivalent in this proof. The lower bound is thus also valid if the knowledge of the CPU and GPU computing times of all the descendants is given to the scheduler and 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 any resource, as done in [5]) does not help, and allowing task migration (where tasks can be preempted and resumed on any resource) only halves the bounds. Table 1 summarizes the lower bounds obtained for all combination of knowledge given to the scheduler and flexibility on the task processing (for proofs, please refer to [10]).

4 Competitive algorithms

4.1 The Quick Allocation (QA) algorithm

Amaris et al. [2] designed an online algorithm named ER-LS, which is proved to be $4\sqrt{m/k}$-competitive. The results of Section 3 show that this ratio can only be improved by a constant factor, as no online algorithm can be better than $\sqrt{m/k}$-competitive. ER-LS applies the following processing to each available task T_i:

1. (a) If T_i can be completed on a GPU before time $\overline{p_i}$, then assign it to GPUs.
   (b) Else, if $\overline{p_i}/\underline{p_i} \leq \sqrt{m/k}$, then assign T_i to CPUs, else assign it to GPUs.
2. Schedule T_i as soon as possible on the allocated type of resource.

The main objective of Step 1a is to avoid allocating the first tasks on a slow resource, which intuitively is desirable only on small graphs. Such a technique enables a similar online algorithm to be constant-factor competitive for independent tasks, see [13]. However, it actually increases the competitive factor with precedence constraints. We propose to simplify the allocation phase by suppressing Step 1a. The resulting algorithm QA (which stands for Quick Allocation) is then defined by Steps 1b and 2. Along with a rigorous analysis, this simplification allows us to reach a competitive ratio smaller than $2\sqrt{m/k}+1$, which is almost tight, as outlined in the following theorems. The complete proofs of the following results are available in [10].

Theorem 2. QA *is* $\left(2\sqrt{m/k}+1-(mk)^{-1/2}\right)$ *– competitive.*

Proof sketch. Consider a graph G and the schedule $\mathcal{S}$ obtained by QA, of makespan C_{max}. Let W_c (resp. W_g) be the sum of the processing times of the tasks scheduled on CPU (resp. GPU) by $\mathcal{S}$, and CP be the computing time of a critical path of G, given the allocation of $\mathcal{S}$. We first prove that:

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

Now, focusing first on the workload in the optimal solution, and then on the length of the critical path in the optimal solution, we can show the following inequalities and conclude:

$$\frac{W_c}{m} + \frac{W_g}{k} \leq \left(1+\sqrt{\frac{m}{k}}\right) \mathrm{OPT} \quad \text{and} \quad CP \leq \sqrt{\frac{m}{k}}\mathrm{OPT}. \qquad \square$$

Theorem 3. *The competitive ratio of* QA *is at least* $\left(2\sqrt{m/k}+1-\frac{1}{k}\right)$.

Proof sketch. Let ε be a small processing time. Consider the graph composed of the three groups of tasks below. The online instance will reveal the tasks in the same order. The only dependence is from task ε to task d.
Group A $k(k-1)$ tasks with $\overline{p_i} = \infty$ and $\underline{p_i} = 1/k$.

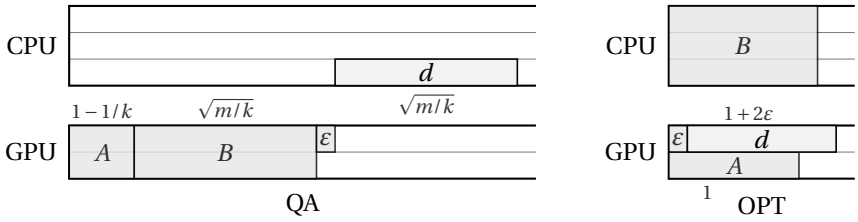


Fig. 2. Schedule obtained by QA (left) and the optimal one (right).

Group B mk tasks with $\overline{p_i} = (1+\varepsilon)/k$ and $\underline{p_i} = 1/\sqrt{mk}$.

Group C Task ε, with $\overline{p_\varepsilon} = \infty$ and $\underline{p_\varepsilon} = \varepsilon$, and task d, with $\overline{p_d} = \sqrt{m/k}$ and $\underline{p_d} = 1+\varepsilon$.

As depicted in Figure 2, QA will schedule groups A and B and Task ε on GPU, then task d on CPU, for a total makespan equal to $2\sqrt{m/k} + 1 - \frac{1}{k} + \varepsilon$. The optimal solution schedules only group B on CPU, for a total makespan equal to $1 + 2\varepsilon$, hence the result. □

The proofs of these two results give some intuition on why choosing a ratio equal to $\sqrt{m/k}$ is the best choice in Step 1b. With a smaller ratio (closer to 1), more tasks would be allocated to GPU. This would allow tasks on the critical path to be processed faster. However, the GPUs, which can be seen as a rare resource (since $m \geq k$), may be wasted on tasks that are not accelerated enough. For instance, if the GPU computing time of the tasks of group B in the proof of Theorem 3 were larger, such an algorithm would perform worse than QA. On the opposite, with a larger ratio (closer to m/k), the GPU would not be wasted on such tasks and the loads would be divided more equally on both types of resources. But computing the critical path, such as task d in the example graph, could be more expensive because such a task would be inefficiently executed on CPUs. Intuitively, the geometric mean between these two bounds (1 and m/k) is then the best solution.

4.2 A competitive algorithm that performs well in practice

Although the QA algorithm has the best known competitive ratio, the greedy strategy EFT (see Section 2) actually leads to better schedules on most realistic instances because it balances the load among the resources. However, its performance can be $2 + (m-1)/k$ times worse than the optimal solution (see [10] for a proof of this result).

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.

The competitive ratio of this algorithm is in $O(\lambda\sqrt{m/k})$. Indeed, the first phase cannot lead to a schedule more than λ times worse than QA, and the second phase has the competitive ratio of QA. Therefore, the algorithm is $(\lambda + 1)(2\sqrt{m/k}+1)$-competitive (see [10] for more details). Note that this competitive ratio is not tight. The worst performance observed so far is $\max(\lambda, 2\sqrt{m/k}+1)$.

5 Simulations

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

5.1 Baseline heuristics

In addition to the four online algorithms discussed above (ER-LS from [2], QA, EFT, and MixEFT, 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 as it minimizes the execution time of each task. On the opposite, Ratio should perform well on graphs with a high parallelism throughout the execution, as it will execute more tasks concurrently on the CPUs. We also used the offline HEFT algorithm [25], which is known to perform well in practice, as a baseline to compare all online strategies.

5.2 Experimental setup

We used three types of instances: realistic DAGs corresponding to a linear algebra application, namely 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 [1, 6] made using the Chameleon software [12].

The random instances come from the STG set [23], which is often used in the literature to compare the performance of scheduling strategies. We report here the simulations made with 180 graphs of 300 nodes each. 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 [15], it is positive and it is possible to specify its expected value and standard deviation by adjusting its parameters.

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. The expected processing time of each task on a GPU is 1 (with a standard deviation of 0.1). Each instance is parameterized by a number μ, which represents the expected processing time on a CPU, and varies from $(m/k)^{-1/4}$ to $(m/k)^{5/4}$ (the standard deviation of the CPU processing times is equal to 10% of μ). For a given expected CPU cost μ, the number of tasks in the chain is equal to $\lceil \frac{n}{m/\mu+k} \rceil$, where $n = 300$ is the total number of tasks. Therefore, the larger μ, the longer the chain.

We have performed simulations for various platform sizes, whose results are available in [10]. As expected from the theoretical analysis, the behaviors of the heuristics mainly depend on the value m/k. For the sake of brevity, we only report here the results obtained for $m = 20$ CPUs and $k = 2$ GPUs, as it is representative of the results for relatively large values of m/k. The code and scripts used for the simulations and the data analysis are available online [9].

5.3 Results

Figure 3 depicts the performance of the six online scheduling algorithms. Except when varying its parameter (Figure 3(d)), MixEFT performs exactly as EFT (and is thus omitted for better readability). On Cholesky DAGs (Figure 3(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 becomes worse than Ratio for large graphs.

Figure 3(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.

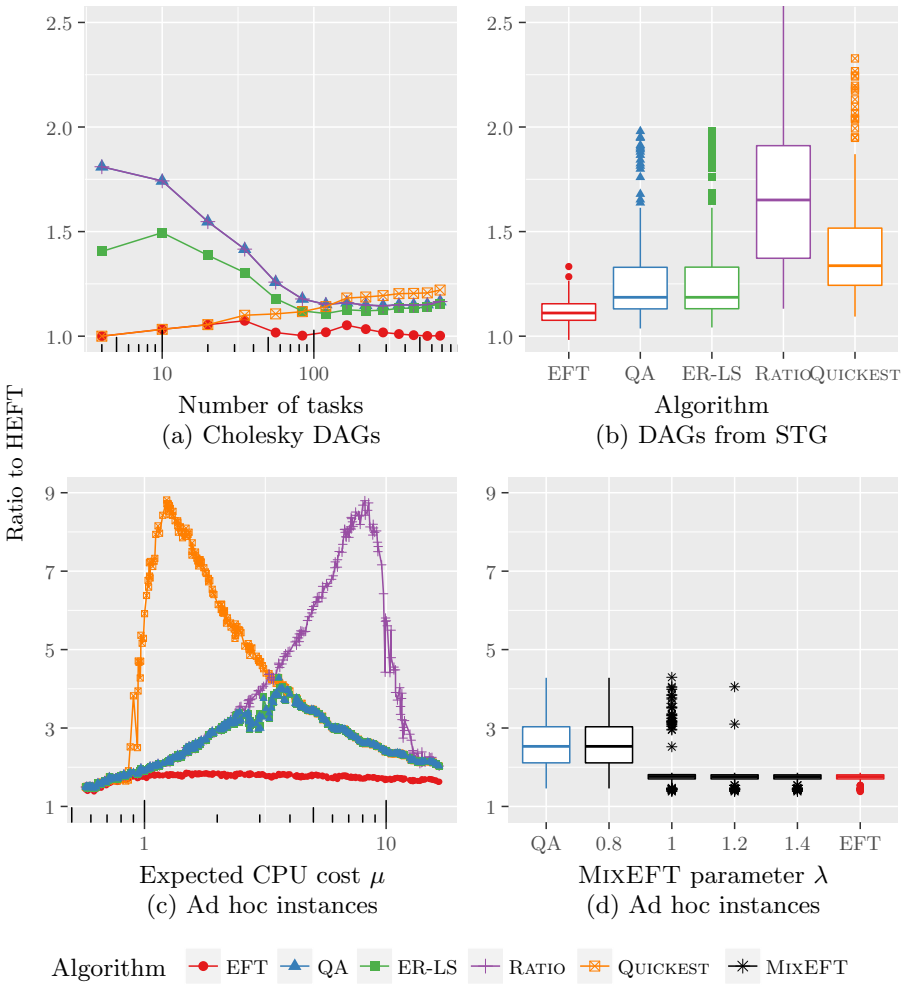


Fig. 3. Ratios of the makespan over HEFT for EFT, QA, ER-LS, Ratio, Quickest, and MixEFT with $m = 20$ CPUs and $k = 2$ GPUs. Except in Figure (d), MixEFT is not shown because it performs exactly as EFT. In Figure (d), ER-LS, Ratio, and Quickest are discarded.

Figure 3(c) first shows that EFT (and MixEFT) is almost always the best online heuristic for these 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 3(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. 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 whereas EFT can lead to very large makespans on specific instances.

6 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. Our future work includes taking into account communication times when moving data from/to the GPUs, and coping with inaccurate processing time estimates.

Data Availability Statement and Acknowledgments

The datasets generated during and/or analyzed during the current study are available in the Figshare repository: `https://doi.org/10.6084/m9.figshare.6353456`.

This work was supported by the SOLHAR project (ANR-13-MONU-0007) which is operated by the French National Research Agency (ANR).

References

1. Agullo, E., Beaumont, O., Eyraud-Dubois, L., Kumar, S.: Are static schedules so bad? A case study on Cholesky factorization. In: IPDPS. IEEE (2016)
2. Amaris, M., Lucarelli, G., Mommessin, C., Trystram, D.: Generic algorithms for scheduling applications on hybrid multi-core machines. In: Euro-Par 2017: Parallel Processing. pp. 220–231 (2017)
3. Augonnet, C., Clet-Ortega, J., Thibault, S., Namyst, R.: Data-aware task scheduling on multi-accelerator based platforms. In: ICPADS. pp. 291–298 (Dec 2010)

4. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Conc. and Comp.: Practice and Experience 23(2), 187–198 (2011)
5. Beaumont, O., Eyraud-Dubois, L., Kumar, S.: Approximation proofs of a fast and efficient list scheduling algorithm for task-based runtime systems on multicores and GPUs. In: IEEE IPDPS. pp. 768–777 (2017)
6. Beaumont, O., Cojean, T., Eyraud-Dubois, L., Guermouche, A., Kumar, S.: Scheduling of linear algebra kernels on multiple heterogeneous resources. In: HiPC (2016)
7. Bleuse, R., Gautier, T., Lima, J.V., Mounié, G., Trystram, D.: Scheduling data flow program in XKaapi: A new affinity based algorithm for heterogeneous architectures. In: Euro-Par 2014: Parallel Processing. pp. 560–571 (2014)
8. Bleuse, R., Kedad-Sidhoum, S., Monna, F., Mounié, G., Trystram, D.: Scheduling independent tasks on multi-cores with GPU accelerators. Concurrency and Computation: Practice and Experience 27(6), 1625–1638 (2015)
9. Canon, L.C., Marchal, L., Simon, B., Vivien, F.: Code for simulating online scheduling of task graphs on hybrid platforms, figshare, code (2018), `https://doi.org/10.6084/m9.figshare.6353456`
10. Canon, L.C., Marchal, L., Simon, B., Vivien, F.: Online scheduling of task graphs on hybrid platforms. Research report 9150, INRIA (Feb 2018)
11. Canon, L.C., Marchal, L., Vivien, F.: Low-cost approximation algorithms for scheduling independent tasks on hybrid platforms. In: Euro-Par 2017: Parallel Processing. pp. 232–244 (2017)
12. Chameleon, a dense linear algebra software for heterogeneous architectures. `https://project.inria.fr/chameleon`
13. Chen, L., Ye, D., Zhang, G.: Online scheduling of mixed CPU-GPU jobs. Int. Journal of Foundations of Computer Science 25(06), 745–761 (2014)
14. Drozdowski, M.: Scheduling parallel tasks – algorithms and complexity. In: Leung, J. (ed.) Handbook of Scheduling. Chapman and Hall/CRC (2004)
15. Feitelson, D.: Workload modeling for computer systems performance evaluation. Book Draft, Version 1.0.1 pp. 1–601 (2014)
16. Graham, R.L.: Bounds on multiprocessing timing anomalies. SIAM journal on Applied Mathematics 17(2), 416–429 (1969)
17. Imreh, C.: Scheduling problems on two sets of identical machines. Computing 70(4), 277–294 (2003)
18. Kedad-Sidhoum, S., Monna, F., Trystram, D.: Scheduling tasks with precedence constraints on hybrid multi-core machines. In: IEEE IPDPS Workshops. pp. 27–33 (2015)
19. Leung, J.Y.: Handbook of scheduling: algorithms, models, and performance analysis. CRC Press (2004)
20. Olivier, B., Lionel, E., Suraj, K.: Fast approximation algorithms for task-based runtime systems. Concurrency and Computation: Practice and Experience `https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4502`, online Version of Record before inclusion in an issue
21. Raravi, G., Andersson, B., Nélis, V., Bletsas, K.: Task assignment algorithms for two-type heterogeneous multiprocessors. Real-Time Systems 50(1), 87–141 (2014)
22. Sainz, F., Mateo, S., Beltran, V., Bosque, J.L., Martorell, X., Ayguadé, E.: Leveraging OmpSs to exploit hardware accelerators. In: SBAC-PAD. pp. 112–119 (2014)
23. Tobita, T., Kasahara, H.: A standard task graph set for fair evaluation of multiprocessor scheduling algorithms. Journal of Scheduling 5(5), 379–394 (2002)

24. TOP500 Supercomputer Site, http://www.top500.org, List of November 2017
25. Topcuoglu, H., Hariri, S., Wu, M.: Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE TPDS 13(3), 260–274 (2002)