Fast Approximation Algorithms for Task-Based Runtime Systems

Olivier Beaumont, Lionel Eyraud-Dubois, Suraj Kumar

Received: 00 Month 0000 | Revised: 00 Month 0000 | Accepted: 00 Month 0000

DOI: xxx/xxxx

ARTICLE TYPE

Fast Approximation Algorithms for Task-Based Runtime Systems

Olivier Beaumont[1,2] | Lionel Eyraud-Dubois*[1,2] | Suraj Kumar[1,2,3]

[1]RealOPT, Inria, Bordeaux, France
[2]University of Bordeaux, France
[3]Ericsson Research, India

Correspondence
*Lionel Eyraud-Dubois, Email: Lionel.Eyraud-Dubois@inria.fr

Abstract

In High Performance Computing, heterogeneity is now the norm with specialized accelerators like GPUs providing efficient computational power. Resulting complexity led to the development of task-based runtime systems, where complex computations are described as task graphs, and scheduling decisions are made at runtime to perform load balancing between all resources of the platforms. Developing good scheduling strategies, even at the scale of a single node, and analyzing them both theoretically and in practice is expected to have a very high impact on the performance of current HPC systems. The special case of two kinds of resources, typically CPUs and GPUs is already of great practical interest. The scheduling policy HeteroPrio has been proposed in the context of fast multipole computations (FMM), and has been extended to general task graphs with very promising results. In this paper, we provide a theoretical study of the performance of HeteroPrio, by proving approximation bounds compared to the optimal schedule, both in the case of independent tasks and in the case of general task graphs. Interestingly, our results establish that spoliation (a technique that enables resources to restart uncompleted tasks on another resource) is enough to prove bounded approximation ratios for a list scheduling algorithm on two unrelated resources, which is known to be impossible otherwise. This result holds true both for independent and dependent tasks graphs. Additionally, we provide an experimental evaluation of HeteroPrio on real task graphs from dense linear algebra computation, that establishes its strong performance in practice.

KEYWORDS:
List scheduling; Approximation proofs; Runtime systems; Heterogeneous scheduling; Dense linear algebra

1 | INTRODUCTION

Accelerators such as GPUs are more and more commonplace in processing nodes due to their massive computational power, usually beside multicores. When trying to exploit both CPUs and GPUs, several phenomena are added to the inherent complexity of the underlying NP-hard optimization problem. First, multicores and GPUs are highly unrelated resources, even in the context of regular linear algebra kernels. Indeed, depending on the kernel, the performance of the GPUs with respect to the one of CPUs may be much higher, close or even worse. In the scheduling literature, unrelated resources are known to make scheduling problems harder (see (1) for a survey on the complexity of scheduling problems, (2) for the specific simpler case of independent tasks scheduling and (3) for a recent survey in the case of CPU and GPU nodes). Second, the variety of existing architectures of processing nodes has increased dramatically with the combination of available resources and the use of both multicores and

accelerators. In this context, developing optimized hand-tuned kernels for all these architectures turns out to be extremely costly. Third, CPUs and GPUs of the same node share many resources (caches, buses,...) and exhibit complex memory access patterns (due to NUMA effects in particular). Therefore, it becomes extremely difficult to predict precisely the durations of both tasks and data transfers, on which traditional offline schedulers rely to allocate tasks, even on very regular kernels such as linear algebra.

On the other hand, this situation favors dynamic scheduling strategies where decisions are made at runtime based on the (dynamic) state of the machine and on the (static) knowledge of the application. The state of the machine can be used to allocate a task close to its input data, whereas the knowledge of the application can be used to favor tasks that are close to the critical path. In the recent years, several such task-based runtime systems have been developed, such as StarPU (4), StarSs (5), SuperMatrix (6), QUARK (7), XKaapi (8) or PaRSEC (9). All these runtime systems model the application as a DAG, where nodes correspond to tasks and edges to dependencies between these tasks. These runtime systems are typically designed for linear algebra applications, and the task typically corresponds to a linear algebra kernel whose granularity is well suited for all types of resources. At runtime, the scheduler knows (i) the state of the different resources (ii) the set of tasks that are currently processed by all non-idle resources (iii) the set of ready (independent) tasks whose dependencies have all been solved (iv) the location of all input data of all tasks (v) possibly an estimation of the duration of each task on each resource and of each communication between each pair of resources and (vi) possibly priorities associated to tasks that have been computed offline. Therefore, the runtime system faces the problem of deciding, given an independent set of tasks (and possibly priorities computed by an offline algorithm), given the characteristics of these tasks on the different resources (typically obtained through benchmarking), where to place and to process them.

In the case of independent tasks, on the theoretical side, several solutions have been proposed for this problem, including a PTAS (see for instance (10)). Nevertheless, in the context of runtime systems, the dynamic scheduler must take its decisions at runtime and is itself on the critical path of the application. This reduces the spectrum of possible algorithms to very fast ones, whose complexity to decide which task to process next is typically sublinear in the number of ready tasks.

Several scheduling algorithms have been proposed in this context and can be classified in several classes. The first class of algorithms is based on (variants of) HEFT (11), where the priority of tasks is computed based on their expected distance to the last node, with several possible metrics to define the expected durations of tasks (given that tasks can be processed on heterogeneous resources) and data transfers (given that input data may be located on different resources). To the best of our knowledge there is no approximation ratio for this class of algorithms on unrelated resources, and Bleuse et al. (3) have exhibited an example on m CPUs and 1 GPU where HEFT algorithm achieves a makespan $\Omega(m)$ times worse the optimal. The second class of scheduling algorithms is based on more sophisticated ideas that aim at minimizing the makespan of the set of ready tasks (see for instance (3, 12)). In this class of algorithms, the main difference lies in the compromise between the quality of the scheduling algorithm (expressed as its approximation ratio when scheduling independent tasks) and its cost (expressed as the complexity of the scheduling algorithm). At last, a third class of algorithms has recently been proposed (see for instance (13, 14, 15, 16)), in which scheduling decisions are based on the affinity between tasks and resources, *i.e.* try to process the tasks on the best suited resource for it. We will discuss the related work on independent and dependent tasks scheduling in more detail in Section 3.

In this paper, we focus on HETEROPRIO (13, 15, 16)) that belongs to the third class. This algorithm has been proposed in practical settings to benefit as much as possible from platform heterogeneity (17), and has obtained good performance in this context. We are interested in a theoretical analysis of its performance, and we start with the simpler setting of independent tasks. More specifically, we define precisely in Section 2 a variant called HeteroPrioIndep that combines the best of all worlds. Indeed, after introducing notations and general results in Section 4, we first prove that contrarily to HEFT variants, HeteroPrioIndep achieves a bounded approximation ratio in Section 5 while having a low complexity. We also provide a set of proven and tight approximation results, depending on the number of CPUs and GPUs in the node. In Section 6, we analyse HeteroPrioDep, an extension to general task graphs, by applying it at each scheduling stage to the set of independent ready tasks, and we prove that this extended version also achieves a bounded (with respect to the number of resources) approximation ratio. At last, we provide in Section 7 a set of experimental results showing that, besides its very low complexity, HETEROPRIO achieves better performance than the other schedulers based either on HEFT or on approximation algorithms for independent tasks scheduling. Concluding remarks are given in Section 8.

	DPOTRF	DTRSM	DSYRK	DGEMM
CPU time / GPU time	1.72	8.72	26.96	28.80

TABLE 1 Acceleration factors for Cholesky kernels (tile size 960).

2 | PRESENTATION OF HETEROPRIO

2.1 | Affinity Based Scheduling

HETEROPRIO has been proposed in the context of task-based runtime systems responsible for allocating tasks onto heterogeneous nodes typically consisting of a few CPUs and GPUs (17).

Historically, most systems use scheduling strategies inspired by the well-known HEFT algorithm: tasks are ordered by priorities (which are computed offline) and the highest priority ready task is allocated onto the resource that is expected to complete it first, given the expected transfer times of its input data and the expected processing time of this task on this resource. These systems have shown some limits (13) in strongly heterogeneous and unrelated systems, which is the typical case of nodes consisting of both CPUs and GPUs. Indeed, with such nodes, the relative efficiency of accelerators, that we denote as the affinity in what follows, strongly differs from one task to another. Let us for instance consider the case of Cholesky factorization, where 4 types of tasks (kernels DPOTRF, DTRSM, DSYRK and DGEMM) are involved, on our local platform consisting of 20 CPU cores of two Haswell Intel® Xeon® E5-2680 processors and 4 Nvidia K40-M GPUs. The acceleration factors are depicted in Table 1 .

In the rest of this paper, acceleration factor is always defined as the ratio between the processing times on a CPU and on a GPU, so that the acceleration factor can be smaller than 1 if CPUs turn out to be faster on a given kernel. From this table, we can observe the main characteristics that will influence our model and the design of HETEROPRIO. The acceleration factor strongly depends on the kernel. Some kernels, like DSYRK and DGEMM are almost 29 times faster on GPUs, while DPOTRF is only slightly accelerated. This justifies the interest of runtime scheduling policies where the affinity between tasks and resources plays the central role. HETEROPRIO belongs to this class, and its principle is the following: when a resource becomes idle, HETEROPRIO selects among the ready tasks the one for which it has a maximal affinity. For instance, in the case of Cholesky factorization, among the ready tasks, CPUs will prefer DPOTRF to DTRSM to DSYRK and to DGEMM while GPUs will prefer DGEMM to DSYRK to DTRSM to DPOTRF. With this simple definition, HETEROPRIO behaves like a list scheduling algorithm: if some resource is idle and ready tasks exist, HETEROPRIO assigns one of the ready tasks to the idle resource. It is known that list scheduling algorithms are prone to pathological cases with unrelated processors: for a given $X > 0$, consider an instance with one CPU, one GPU, and two tasks. Task T_1 has length $1 + \epsilon$ on the CPU and $1/X$ on the GPU, and task T_2 has length X on the CPU and 1 on the GPU. Since task T_1 has a better acceleration factor, it is processed on the GPU at time 0, and the idle CPU is forced to process the (very long) task T_2, leading to a makespan of X. But scheduling both tasks on the GPU yields a makespan of $1 + 1/X$, and thus an arbitrarily high ratio.

For this reason, we propose to enhance HETEROPRIO with a spoliation mechanism, defined in the following way. When a fast resource becomes idle and no ready task is available, if it is possible to restart a task already started on another resource and to complete it earlier, then the task is cancelled and restarted on the fast resource. We call this mechanism a *spoliation* in order to avoid a possible confusion with preemption. Indeed, it does not correspond to preemption since all the progress made on the slow resource is lost. It is therefore less efficient than preemption but it can be implemented in practice, unlike preemption on heterogeneous resources like CPUs and GPUs. This spoliation mechanism can be viewed in two different ways. If it is technically feasible to keep a copy of all data used by tasks running on slow resources, then spoliating tasks running on them is possible and HETEROPRIO in this context is a completely online (non-clairvoyant) algorithm. This mechanism does not exist in StarPU, but it would be feasible at the cost of additionnal memory costs; in larger scale systems (like Hadoop), it is actually used to cope with nodes that have unexpectedly long computation times. Otherwise, we can see HETEROPRIO as a completely offline algorithm: we see spoliation as a way to simplify the presentation of the algorithm, and we use it to compute an offline schedule in which spoliated tasks are actually never started. This schedule is thus completely valid in the standard scheduling model.

This HETEROPRIO allocation strategy has been studied in the context of StarPU for several linear algebra kernels and it has been proven experimentally that it enables to achieve a better utilization of slow resources (13) than other standard HEFT-based strategies.

In the rest of the paper, since task based runtime systems see a set of independent tasks, we will concentrate first on this problem. We present HeteroPrioIndep, a version of HETEROPRIO targeted at independent tasks, for which we prove approximation ratios under several scenarios for the composition of the heterogeneous node (namely 1 GPU and 1 CPU, 1 GPU and several CPUs and several GPUs and several CPUs) in Section 5. We extend HETEROPRIO to task graphs and obtain HeteroPrioDep, for which we prove (much weaker) approximation ratios in Section 6.

2.2 | HETEROPRIO Algorithm for a set of Independent Tasks

Algorithm 1 The HeteroPrioIndep Algorithm for a set of independent tasks

1: Sort Ready Tasks in Queue Q by non-increasing acceleration factors
2: **while** there exists an unprocessed task **do**
3: **if** all workers are busy **then**
4: continue
5: **end if**
6: Select an idle worker W
7: **if** $Q \neq \emptyset$ **then**
8: Consider a task T from the beginning (resp. the end) of Q if W is a GPU (resp. CPU) worker.
9: Start processing T on W.
10: **else**
11: Consider tasks running on the other type of resource in decreasing order of their expected completion time. If the expected completion time of T running on a worker W' can be improved on W, then T is spoliated and W starts processing T.
12: **end if**
13: **end while**

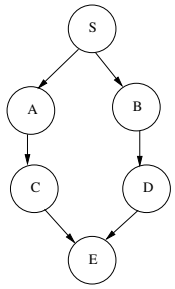
When priorities are associated to tasks, then Line 1 of Algorithm 1 takes them into account for breaking ties among tasks with the same acceleration factor and places the highest priority task first in the scheduling queue. Approximation factors proven in this paper do not depend on how ties are broken and do not depend on task priorities. However, taking task priorities into account is known to improve makespan in practice and we will therefore use priorities in the experiments performed in Section 7.

The queue Q of ready tasks in Algorithm 1 can be implemented as a heap. Therefore, the time complexity of Algorithm 1 on m CPUs and n GPUs is $\mathcal{O}(N \log N)$, where N is the number of ready tasks and the amortized cost of a scheduling decision is therefore of order $\mathcal{O}(\log N)$. We will prove results on the approximation ratio achieved by Algorithm 1 on different platform configurations for independent tasks in Sections 4 and 5.

2.3 | HETEROPRIO for Task Graphs

As already mentioned in Section 1, our goal is to extend HeteroPrioIndep designed for independent tasks, to graphs of dependent tasks, and to use HeteroPrioIndep as a building block used by the runtime system to allocate the set of ready tasks. However, in order to obtain guarantees, we need to slightly augment the set of ready tasks. We thus define more precisely HeteroPrioDep: when a GPU (resp. CPU) becomes idle, it picks the task with the highest (resp. lowest) acceleration factor among the ready tasks of Q or the tasks currently running on the CPU (resp. GPU), provided that it can complete them before their expected completion time. In the latter case, the currently running task is spoliated, *i.e.* aborted on the CPU (resp. GPU) and restarted on the GPU (resp. CPU).

Indeed, such an adaptation of HeteroPrioIndep is necessary to achieve a bounded approximation ratio for task graphs, as shown by the following example. Let us consider a simple platform consisting of 1 CPU and 1 GPU, and the task graph depicted on Figure 1 , together with the expected time of the different tasks on both resource types.



Task	CPU Time	GPU Time	acceleration ratio
S	1	1	1
A	1	1	1
B	2	2	1
C	ρ_c	1	ρ_c
D	1	$\rho_c - 1$	$\frac{1}{\rho_c - 1}$
E	1	1	1

FIGURE 1 A task graph and duration of different tasks.

Let us analyze the behavior on this task graph with an algorithm for which spoliation is forbidden while there exists a ready task. Since tasks S, A, B and C have the same acceleration factor, let us assume[1] that S and A are processed on the CPU (that becomes available at time 2) and task B by the GPU (that becomes available at time 3). At time 2, C becomes ready and the CPU is available, so that it processes C until time $2 + \rho_c$. Then at time 3, the GPU becomes ready and D is the only ready task at that time. If the GPU only considers the set of ready tasks, and not the tasks running on the other resource, it misses the opportunity to process C for which it is very well suited since $\rho \gg 1$, and processes D, for which it is badly suited, until time $2 + \rho_c$. Then both resources are available at time $2 + \rho_c$ and one picks task E, so that the makespan is $3 + \rho_c$ if spoliation is forbidden while there exists a ready task.

Moreover, the minimum length of path $S \to B \to D \to E$ is 5, so that $C_{\max}^{Opt} \geq 5$. Similarly, if S, B, C, E are scheduled on the GPU and A, D on the CPU, then the completion time is 5, so that $C_{\max}^{Opt} = 5$, and the approximation ratio achieved when spoliation is forbidden while there exists a ready task would be $\frac{\rho_c+3}{5}$, which can be made arbitrarily large with ρ_c large enough.

On the other hand, let us consider HeteroPrioDep (Algorithm 2) where spoliation is allowed even if there exists a ready task. Then, at time 3, the best acceleration ratio among the ready tasks (task D) is $\frac{1}{\rho_c - \epsilon} \ll 1$, the best acceleration ratio among the tasks running on the CPU (*i.e.* task C) is $\rho_c \gg 1$ and the GPU is able to complete C at time $3 + 1 \ll 2 + \rho_c$, so that C is spoliated and processed on the GPU. In this case, the makespan of HeteroPrioDep is $C_{\max}^{Opt} = 5$, so that it is optimal on this instance. We will prove general approximation ratios for HeteroPrioDep in Section 6 and evaluate its performance in Section 7.

It is worth noting that spoliating while there exist ready tasks is never useful in the case of independent tasks: indeed, since all tasks are available from the beginning, the situation where both types of resources process tasks for which they are badly suited cannot happen. This is why in the case of independent tasks, we consider the version HeteroPrioIndep, as depicted in Algorithm 1.

3 | RELATED WORKS

3.1 | Independent Tasks Scheduling

The problem considered in this paper is a special case of the standard unrelated scheduling problem $R||C_{\max}$. This problem is made easier by preemption, *i.e.* the ability of stopping a task and storing its current state on a given machine and then to restart it from its current state onto another machine. In this context, Lawler et al. (18) have proved that the problem can be solved in polynomial time using linear programming. Unfortunately, in particular in the context of CPUs and GPUs, preemption is not feasible in practice. This is why we concentrate in this paper on spoliation, where a task is stopped and restarted from the beginning onto another resource. Without preemption, Lenstra et al (2) proposed a PTAS for the general problem with a fixed number of machines, and propose a 2-approximation algorithm, based on the rounding of the optimal solution of the linear program which describes the preemptive version of the problem. This result has recently been improved (19) to a $2 - \frac{1}{m}$ approximation. However, the time complexity of these general algorithms is too high to allow using them in the context of runtime systems.

The more specialized case with a small number of types of resources has been studied in (10) and a PTAS has been proposed, which also contains a rounding phase whose complexity makes it impractical, even for 2 different types of resources. Greedy

[1] Strictly speaking, this depends on how ties are broken ; however this behavior can be forced by changing the acceleration ratios by arbitrarily small values. We do not include these small values for simplicity.

Algorithm 2 The HeteroPrioDep Algorithm for a set of dependent tasks.

```
Sort input tasks in Queue Q by non-increasing acceleration factors
Create an empty Queue of running tasks Q' by non-increasing acceleration factors
while there exists an unprocessed task do
    if all workers are busy then
        continue
    end if
    Select an idle worker W
    if Q' ≠ ∅ then
        Consider tasks from the beginning (resp. the end) of Q' if W is a GPU (resp. CPU) worker
        Until finding a candidate task C' whose completion time would be smaller if restarted on W.
    end if
    if Q ≠ ∅ then
        Consider a candidate task C from the beginning (resp. the end) of Q if W is a GPU (resp. CPU) worker.
    end if
    Among C and C', choose the task T with the highest acceleration factor if W is a GPU (resp. the smallest acceleration
    factor if W is a CPU)
    if a worker W' of the other type is idle then
        Process task T on its favorite resource between W and W'
    else
        Process task T on W (potentially using spoliation).
    end if
    Update Q and Q'.
end while
```

approximation algorithms for the online case have been proposed by Imreh on two different types of resources (20) and by Chen et al. in (21). In (21), a 3.85-approximation algorithm (CYZ_5) is provided in the case of online scheduling on two types of resources. These algorithms have linear complexity, however most of their decisions are based on comparing task processing times on both types of resources and not on trying to balance the load. Since these algorithms are tailored towards worst-case behavior, their average performance on practical instances is not satisfying (in practice all tasks are scheduled on the GPUs, see Section 7). For example, it is possible to show that there exist instances with arbitrarily small tasks for which the ratio to the optimal solution is bounded by 1.5, whereas the approximation ratio of all other algorithms tends to 1 when the size of tasks tends to 0.

Additionally, Bleuse et al (3, 22) have proposed algorithms with varying approximation factors ($\frac{4}{3}$, $\frac{3}{2}$ and 2) based on dynamic programming and dual approximation techniques. These algorithms have better approximation ratios than the ones proved in this paper, but their time complexity is much higher, as we will discuss in Section 7. Furthermore, we also show in Section 7 that their actual performance is not as good when used iteratively on the set of ready tasks in the context of task graph scheduling. We also exhibit that HETEROPRIO performs better on average than the above mentioned algorithms, despite its higher worst case approximation ratio.

Finally, there has recently been a number of papers targeting both a low computing cost and a bounded approximation ratio, such as (14, 12, 15). In (14), Cherière et al. propose a 2 approximation algorithm under the assumption that no task is longer on any resource than the optimal makespan. It is worth noting that under this restrictive assumption, Lemma 3 establishes the same approximation ratio for HeteroPrioIndep. The algorithm CLB2C proposed in (14) is based on the same underlying ideas as HETEROPRIO, as tasks are ordered by non-increasing acceleration factors and the highest accelerated task and the less accelerated task compete for being allocated respectively to the least loaded GPU and the least loaded CPU, under a strategy that slightly differs from HeteroPrioIndep. In (15), a preliminary version of this paper is proposed, that concentrate on independent tasks only. In (12), Canon et al. propose two strategies that also achieve a 2 approximation ratio for independent task scheduling. All these results are summarized in Table 2 . We describe these algorithms in detail in Section 7, where we also provide a detailed comparison both in terms of running time and in terms of average approximation ratio on realistic instances. Another

Name	Complexity	Approximation ratio
CYZ_5 (21)	$N \log(m+n)$	3.85
LG (20)	$N \log(m+n)$	$2 + \frac{m-1}{n}$
MG(1,1) (20)	$N \log(m+n)$	$4 - \frac{2}{m}$
DualHP (22)	$N \log(Nmn) \log(\Delta)$	2
DualDynamic (22)	$N^2 m^2 k^3 \log(\Delta)$	$\frac{4}{3} + \frac{1}{3k}$
DualAccel (22)	$Nm \log(N) \log(\Delta)$	$\frac{3}{2}$*
CLB2C (14)	$N \log(Nmn)$	2**
BalancedEstimate (12)	$N \log(Nmn)$	2
BalancedMakespan (12)	$N^2 \log(Nmn)$	2
HeteroPrioIndep	$N \log(N) + (N+m+n) \log(m+n)$	3.42

TABLE 2 Summary of complexity and approximation ratios of different algorithms from the literature. We consider instances with N tasks, and two sets of resources of sizes n and m, with $n \leq m$. The parameter Δ is the range of possible makespan values tested by the dual approximation algorithms, and is bounded by $\sum_i \max(p_i, q_i) - \max_i \min(p_i, q_i)$. Remarks: * DualAccel assumes that all tasks execute faster on the GPUs. ** CLB2C assumes that $\forall i, \max(p_i, q_i) \leq C_{\max}^{Opt}$.

similar experimental comparison is available in (12), but it is limited to the case of independent tasks and based on less realistic randomly-generated instances.

3.2 | Task Graph Scheduling

The literature on approximation algorithms for task graph scheduling on heterogeneous machines is more limited. In the case of related resources, Chudak et al. (23) and Chekuri et al. (24) propose a $\log m$ approximation algorithm, where m denotes the number of machines. To the best of our knowledge, these results are still the best approximation algorithms for the case of related machines, even though better approximation ratios are known for special cases, such as chains (25).

The special case of platforms consisting of two kinds of unrelated resources, typically CPUs and GPUs, has recently received a lot of attention. In (26), Kedad-Sidhoum et al. establish the first constant approximation ratio (*i.e.* 6) in this context. The same approximation ratio (but a better behavior in practice) has also been established in (27). These algorithms are based on a two phases approach, where the assignment of tasks onto resource types is computed using the rounding of the solution of a linear program and then the schedule is computed using a list scheduling approach. We will compare in detail the results obtained by this algorithm in comparison to all variants of HETEROPRIO both in terms of running time and in terms of average approximation ratio on realistic instances in Section 7.

4 | NOTATIONS AND FIRST RESULTS

4.1 | General Notations

In this paper, we prove approximation ratios achieved by both HeteroPrioIndep for independent tasks and HeteroPrioDep for task graphs scheduling. The input of the scheduling problem we consider is thus a platform of n GPUs and m CPUs, a set $\mathcal{I}$ of independent tasks (or a task graph G), where task T_i has processing time p_i on CPU and q_i on GPU. Then, the goal is to allocate and schedule those tasks (or a task graph) onto the resources so as to minimize the makespan. We define the acceleration factor of task T_i as $\rho_i = \frac{p_i}{q_i}$. $C_{\max}^{Opt}(\mathcal{I})$ and $C_{\max}^{Opt}(G)$ denote respectively the optimal makespan of set $\mathcal{I}$ and task graph G. Throughout this paper, even in the case of task graph scheduling, we neglect the communication times that happen when data has to be exchanged from CPU to GPU. Indeed, taking them into account would make this hard problem even more complex, and in many practical cases they can be overlapped by the computational tasks. A more precise model which handles communications is left for future work.

4.2 | HETEROPRIO Schedule For a Set of Independent Tasks

To analyze the behavior of HeteroPrioIndep, it is useful to consider the list schedule obtained before any spoliation attempt. We will denote this schedule $\mathcal{S}_{\text{HP}}^{\text{NS}}$, and the final HeteroPrioIndep schedule is denoted $\mathcal{S}_{\text{HP}}$. Figure 2 shows $\mathcal{S}_{\text{HP}}^{\text{NS}}$ and $\mathcal{S}_{\text{HP}}$ for a set of independent tasks $\mathcal{I}$. We define $t_{FirstIdle}$ as the first moment any worker is idle in $\mathcal{S}_{\text{HP}}^{\text{NS}}$, which is also the first time any spoliation can occur. Therefore, after time $t_{FirstIdle}$, each worker processes at most one task in $\mathcal{S}_{\text{HP}}^{\text{NS}}$. Finally, we define $C_{\max}^{HP}(\mathcal{I})$ as the makespan of $\mathcal{S}_{\text{HP}}$ on instance $\mathcal{I}$.

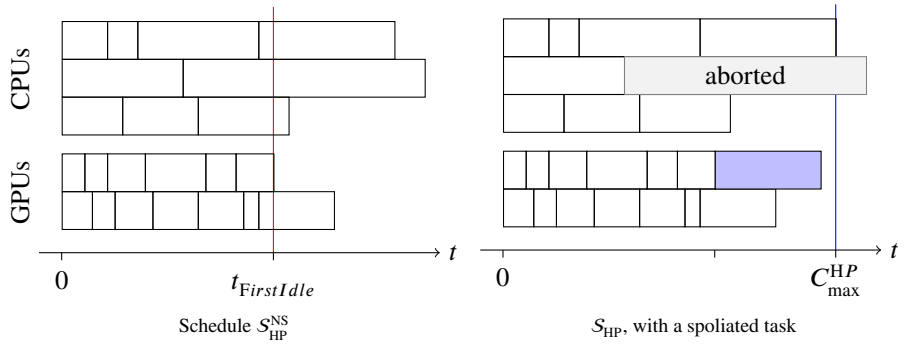


FIGURE 2 Example of a HeteroPrioIndep schedule.

4.3 | Area Bound For a Set of Independent Tasks

Let us now present and characterize a lower bound on the optimal makespan. This lower bound is obtained by assuming that tasks are fully divisible, *i.e.* can be processed in parallel on any number of resources without any efficiency loss. More specifically, any fraction x_i of task T_i is allowed to be processed on CPUs, and this fraction overall consumes CPU resources for $x_i p_i$ time units. Then, the lower bound $AreaBound(\mathcal{I})$ for a set of tasks $\mathcal{I}$ on m CPUs and n GPUs is the solution (in rational numbers) of the following linear program.

$$\text{Minimize } AreaBound(\mathcal{I}) \text{ such that}$$

$$\sum_{i \in I} x_i p_i \leq m \cdot AreaBound(\mathcal{I}) \tag{1}$$

$$\sum_{i \in I} (1 - x_i) q_i \leq n \cdot AreaBound(\mathcal{I}) \tag{2}$$

$$0 \leq x_i \leq 1$$

Since any valid solution to the scheduling problem can be converted into a solution of this linear program, it is clear that $AreaBound(\mathcal{I}) \leq C_{\max}^{Opt}(\mathcal{I})$. Another immediate bound on the optimal is $\forall T \in \mathcal{I}, \min(p_T, q_T) \leq C_{\max}^{Opt}(\mathcal{I})$. By contradiction and with simple exchange arguments, one can prove the following two lemmas.

Lemma 1. *In the area bound solution, the completion time on each type of resources is the same, i.e. constraints* (1) *and* (2) *are both equalities.*

Proof. Let us assume that one of the inequality constraints in the area solution is not tight. Without loss of generality, let us assume that Constraint (1) is not tight. Then some load from the GPUs can be transferred to the CPUs, what in turn decreases the value of $AreaBound(\mathcal{I})$*, in contradiction with the optimality of the solution.* □

Lemma 2. *In* $AreaBound(\mathcal{I})$*, the assignment of tasks is based on the acceleration factor, i.e.* $\exists k > 0$ *such that* $\forall i, x_i < 1 \Rightarrow \rho_i \geq k$ *and* $x_i > 0 \Rightarrow \rho_i \leq k$.

Proof. Let us assume $\exists(T_1,T_2)$ such that (i) T_1 is partially processed on GPUs (i.e., $x_1 < 1$), (ii) T_2 is partially processed on CPUs (i.e., $x_2 > 0$) and (iii) $\rho_1 < \rho_2$.

Let WC and WG denote respectively the overall work on CPUs and GPUs in $AreaBound(\mathcal{I})$. If we transfer a fraction $0 < \epsilon_2 < min(x_2, \frac{(1-x_1)p_1}{p_2})$ of T_2 work from CPU to GPU and a fraction $\frac{\epsilon_2 q_2}{q_1} < \epsilon_1 < \frac{\epsilon_2 p_2}{p_1}$ of T_1 work from GPU to CPU, the overall loads WC' and WG' become

$$WC' = WC + \epsilon_1 p_1 - \epsilon_2 p_2$$
$$WG' = WG - \epsilon_1 q_1 + \epsilon_2 q_2$$

Since $\frac{p_1}{p_2} < \frac{\epsilon_2}{\epsilon_1} < \frac{q_1}{q_2}$, then both $WC' < WC$ and $WG' < WG$ hold true, and hence the $AreaBound(\mathcal{I})$ is not optimal. Therefore, $\exists$ a positive constant k such that $\forall i$ on GPU, $\rho_i \geq k$ and $\forall i$ on CPU, $\rho_i \leq k$. □

This area bound can be extended to the case of non-independent tasks (13, 26) by adding a variable s_i representing the start time of each task i, and the following set of constraints:

$$\forall i, j \text{ such that } i \to j, \quad s_i + x_i p_i + (1 - x_i) q_i \leq s_j \tag{3}$$
$$\forall i, \quad s_i + x_i p_i + (1 - x_i) q_i \leq AreaBound(\mathcal{I}) \tag{4}$$

Constraint (3) ensures that the starting times of all tasks are consistent with their dependencies ($x_i p_i + (1 - x_i) q_i$ represent the processing time of task i), and constraint (4) conveys the fact that the makespan is not smaller than the completion time of all tasks. This mixed bound (which takes into account both area and dependencies) will not be used in our proofs, but is used in Section 7 to analyze the quality of the schedules obtained by all algorithms.

4.4 | Summary of Approximation Results

This paper presents several approximation results for HeteroPrioIndep depending on the number of CPUs and GPUs. The following table presents a quick overview of the main results for a set of independent tasks proven in Section 5.

(#CPUs, #GPUs)	Approximation ratio	Worst case ex.
(1,1)	$\frac{1+\sqrt{5}}{2}$	$\frac{1+\sqrt{5}}{2}$
(m,1)	$\frac{3+\sqrt{5}}{2}$	$\frac{3+\sqrt{5}}{2}$
(m,n)	$2+\sqrt{2} \approx 3.41$	$2+\frac{2}{\sqrt{3}} \approx 3.15$

We also present $(m + n)$-approx proof for HeteroPrioDep on m CPUs and n GPUs, and a tight worst-case example on 1 CPU and 1 GPU in Section 6.

5 | PROOF OF HeteroPrioIndep APPROXIMATION RESULTS

5.1 | General Lemmas

The following lemma gives a characterization of the work performed by HeteroPrioIndep at the beginning of the processing, and shows that HeteroPrioIndep performs as much work as possible when all resources are busy. For any time instant t, let us define $\mathcal{I}'(t)$ as the sub-instance of $\mathcal{I}$ obtained by removing the fractions of tasks that have been processed between time 0 and time t by HeteroPrioIndep. Then, a schedule beginning like HeteroPrioIndep (until time t) and ending like $AreaBound(\mathcal{I}'(t))$ completes in $AreaBound(\mathcal{I})$.

Lemma 3. *At any time $t \leq t_{FirstIdle}$ in S^{NS}_{HP},*

$$t + AreaBound(\mathcal{I}'(t)) = AreaBound(\mathcal{I})$$

Proof. HeteroPrioIndep assigns tasks based on their acceleration factors. Therefore, at instant t, $\exists k_1 \leq k_2$ such that (i) all tasks (at least partially) processed on GPUs have an acceleration factor larger than k_2, (ii) all tasks (at least partially) allocated on CPUs have an acceleration factor smaller than k_1 and (iii) all tasks not assigned yet have an acceleration factor between k_1 and k_2.

After t, $AreaBound(\mathcal{I}')$ satisfies Lemma 2, and thus $\exists k$ with $k_1 \leq k \leq k_2$ such that all tasks of $\mathcal{I}'$ with acceleration factor larger than k are allocated on GPUs and all tasks of $\mathcal{I}'$ with acceleration factor smaller than k are allocated on CPUs.

Therefore, combining above results before and after t, the assignment $\mathcal{S}$ beginning like HeteroPrioIndep (until time t) and ending like $AreaBound(\mathcal{I}'(t))$ satisfies the following property: $\exists k > 0$ such that all tasks of $\mathcal{I}$ with acceleration factor larger than k are allocated on GPUs and all tasks of $\mathcal{I}$ with acceleration factor smaller than k are allocated on CPUs. This assignment $\mathcal{S}$, whose completion time on both CPUs and GPUs (according to Lemma 1) is $t+AreaBound(\mathcal{I}')$ clearly defines a solution of the fractional linear program defining the area bound solution, so that $t + AreaBound(\mathcal{I}') \geq AreaBound(\mathcal{I})$.

Similarly, $AreaBound(\mathcal{I})$ satisfies both Lemma 2 with some value k' and Lemma 1 so that in $AreaBound(\mathcal{I})$, both CPUs and GPUs complete their work simultaneously. If $k' < k$, more work is assigned to GPUs in $AreaBound(\mathcal{I})$ than in $\mathcal{S}$, so that, by considering the completion time on GPUs, we get $AreaBound(\mathcal{I}) \geq t + AreaBound(\mathcal{I}')$. Similarly, if $k' > k$, by considering the completion time on CPUs, we get $AreaBound(\mathcal{I}) \geq t + AreaBound(\mathcal{I}')$.

□

Since $AreaBound(\mathcal{I})$ is a lower bound on $C_{\max}^{Opt}(\mathcal{I})$, the above lemma implies that

1. at any time $t \leq t_{FirstIdle}$ in $\mathcal{S}_{\mathrm{HP}}^{\mathrm{NS}}$, $t + AreaBound(\mathcal{I}'(t)) \leq C_{\max}^{Opt}(\mathcal{I})$,
2. $t_{FirstIdle} \leq C_{\max}^{Opt}(\mathcal{I})$, and thus all tasks start before $C_{\max}^{Opt}(\mathcal{I})$ in $\mathcal{S}_{\mathrm{HP}}^{\mathrm{NS}}$,
3. if $\forall i \in \mathcal{I}, \max(p_i, q_i) \leq C_{\max}^{Opt}(\mathcal{I})$, then $C_{\max}^{HP}(\mathcal{I}) \leq 2C_{\max}^{Opt}(\mathcal{I})$.

Another interesting characteristic of HeteroPrioIndep is that spoliation can only take place from one type of resource to the other. Indeed, since assignment in $\mathcal{S}_{\mathrm{HP}}^{\mathrm{NS}}$ is based on the acceleration factors of the tasks, and since a task can only be spoliated if it can be processed faster on the other resource type, we get the following lemmas.

Lemma 4. *If, in $\mathcal{S}_{HP}^{NS}$, a resource type processes a task whose processing time is not larger on the other resource type, then no task is spoliated from the other resource type.*

Proof. Without loss of generality let us assume that there exists a task T processed on a CPU in $\mathcal{S}_{HP}^{NS}$, such that $p_T \geq q_T$. We prove that in this case, there is no spoliated task on CPUs, which is the same thing as there being no aborted task on GPUs.

T is processed on a CPU in $\mathcal{S}_{HP}^{NS}$, and $\frac{p_T}{q_T} \geq 1$, therefore by definition of HeteroPrioIndep, all tasks on GPUs in $\mathcal{S}_{HP}^{NS}$ have an acceleration factor at least $\frac{p_{T'}}{q_{T'}} \geq 1$. Non spoliated tasks running on GPUs after $t_{FirstIdle}$ are candidates to be spoliated by the CPUs. But for each of these tasks, the processing time on CPU is at least as large as the processing time on GPU. It is thus not possible for an idle CPU to spoliate any task running on GPUs, because this task would not complete earlier on the CPU. □

Lemma 5. *In HeteroPrioIndep, if a resource type executes a spoliated task then no task is spoliated from this resource type.*

Proof. Without loss of generality let us assume that T is a spoliated task processed on a CPU. From the HeteroPrioIndep definition, $p_T < q_T$. It also indicates that T was processed on a GPU in $\mathcal{S}_{HP}^{NS}$ with $q_T \geq p_T$. By Lemma 4, CPUs do not have any aborted task due to spoliation. □

Finally, we will also rely on the following lemma, that provides the worst case performance of a list schedule when all tasks lengths are large (*i.e.* $> C_{\max}^{Opt}$) on one type of resource.

Lemma 6. *Let $\mathcal{B} \subseteq \mathcal{I}$ such that the processing time of each task of $\mathcal{B}$ on one resource type is larger than $C_{\max}^{Opt}(\mathcal{I})$, then any list schedule of $\mathcal{B}$ on $k \geq 1$ resources of the other type has length at most $(2 - \frac{1}{k})C_{\max}^{Opt}(\mathcal{I})$.*

Proof. Without loss of generality, let us assume that the processing time of each task of set $\mathcal{B}$ on CPU is larger than $C_{\max}^{Opt}(\mathcal{I})$. All these tasks must therefore be processed on the GPUs in an optimal solution. If scheduling this set $\mathcal{B}$ on k GPUs can be done in time C, then $C \leq C_{\max}^{Opt}(\mathcal{I})$. The standard list scheduling result from Graham (28) implies that the length of any list schedule of the tasks of $\mathcal{B}$ on GPUs is at most $(2 - \frac{1}{k})C \leq (2 - \frac{1}{k})C_{\max}^{Opt}(\mathcal{I})$. □

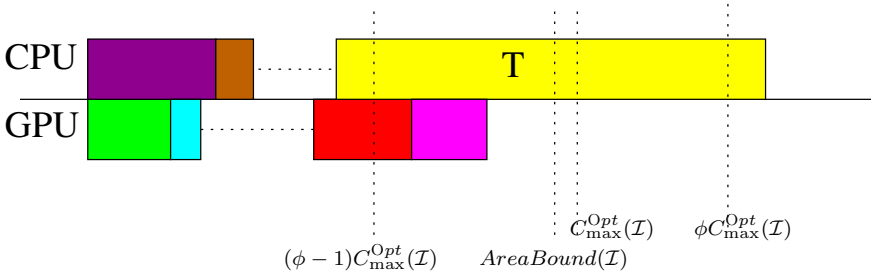


FIGURE 3 Situation where T ends on CPU after $\phi C^{\mathrm{O}pt}_{\max}(\mathcal{I})$.

5.2 | Approximation Ratio with 1 GPU and 1 CPU

With the above lemmas, we are able to prove an approximation ratio of $\phi = \frac{1+\sqrt{5}}{2}$ for HeteroPrioIndep when the node is composed of 1 CPU and 1 GPU. We will also prove that this result is the best achievable by providing a task set $\mathcal{I}$ for which the approximation ratio of HeteroPrioIndep is ϕ.

Theorem 7. *For any instance $\mathcal{I}$ with 1 CPU and 1 GPU, $C^{\mathrm{HP}}_{\max}(\mathcal{I}) \leq \phi C^{\mathrm{O}pt}_{\max}(\mathcal{I})$.*

Proof. Without loss of generality, let us assume that the first idle time (at instant $t_{FirstIdle}$) occurs on the GPU and the CPU is processing the last remaining task T. We will consider two main cases, depending on the relative values of $t_{FirstIdle}$ and $(\phi-1)C^{\mathrm{O}pt}_{\max}$.

The easy case is $t_{FirstIdle} \leq (\phi-1)C^{\mathrm{O}pt}_{\max}$. Indeed, in S^{NS}_{HP}, the completion time of task T is at most $t_{FirstIdle} + p_T$. If task T is spoliated by the GPU, its execution time is $t_{FirstIdle} + q_T$. In both cases, the completion time of task T is at most $t_{FirstIdle} + \min(p_T, q_T) \leq (\phi-1)C^{\mathrm{O}pt}_{\max} + C^{\mathrm{O}pt}_{\max} = \phi C^{\mathrm{O}pt}_{\max}$.

On the other hand, we assume now that $t_{FirstIdle} > (\phi-1)C^{\mathrm{O}pt}_{\max}$. If T ends before $\phi C^{\mathrm{O}pt}_{\max}$ on the CPU in S^{NS}_{HP}, since spoliation can only improve the completion time, this ends the proof of the theorem. In what follows, we assume that the completion time of T on the CPU in S^{NS}_{HP} is larger than $\phi C^{\mathrm{O}pt}_{\max}(\mathcal{I})$, as depicted in Figure 3 .

It is clear that T is the only unprocessed task after $C^{\mathrm{O}pt}_{\max}$. Let us denote by α the fraction of T processed after $C^{\mathrm{O}pt}_{\max}$ on the CPU. Then $\alpha p_T > (\phi-1)C^{\mathrm{O}pt}_{\max}$ since T ends after $\phi C^{\mathrm{O}pt}_{\max}$ by assumption. Lemma 3 applied at instant $t = t_{FirstIdle}$ implies that the GPU is able to process the fraction α of T by $C^{\mathrm{O}pt}_{\max}$ (see Figure 4) while starting this fraction at $t_{FirstIdle} \geq (\phi-1)C^{\mathrm{O}pt}_{\max}$ so that $\alpha q_T \leq (1-(\phi-1))C^{\mathrm{O}pt}_{\max} = (2-\phi)C^{\mathrm{O}pt}_{\max}$. Therefore, the acceleration factor of T is at least $\frac{\phi-1}{2-\phi} = \phi$. Since HeteroPrioIndep assigns tasks on the GPU based on their acceleration factors, all tasks in S assigned to the GPU also have an acceleration factor at least ϕ.

Let us now prove that the GPU is able to process $S \bigcup \{T\}$ in time $\phi C^{\mathrm{O}pt}_{\max}$. Let us split $S \bigcup \{T\}$ into two sets S_1 and S_2 depending on whether the tasks of $S \bigcup \{T\}$ are processed on the GPU (S_1) or on the CPU (S_2) in the optimal solution. By construction, the processing time of S_1 on the GPU is at most $C^{\mathrm{O}pt}_{\max}$ and the processing of S_2 on the CPU takes at most $C^{\mathrm{O}pt}_{\max}$. Since the acceleration factor of tasks of S_2 is larger than ϕ, the processing time of tasks of S_2 on the GPU is at most $C^{\mathrm{O}pt}_{\max}/\phi$ and the overall processing of $S \bigcup \{T\}$ takes at most $C^{\mathrm{O}pt}_{\max} + C^{\mathrm{O}pt}_{\max}/\phi = \phi C^{\mathrm{O}pt}_{\max}$. This concludes the proof of the theorem. □

Theorem 8. *The bound of Theorem 7 is tight, i.e. there exists an instance $\mathcal{I}$ with 1 CPU and 1 GPU for which HeteroPrioIndep achieves a ratio of ϕ with respect to the optimal solution.*

Proof. Let us consider the instance $\mathcal{I}$ consisting of 2 tasks X and Y, with $p_X = \phi$, $q_X = 1$, $p_Y = 1$ and $q_Y = \frac{1}{\phi}$, such that $\rho_X = \rho_Y = \phi$.

The minimum length of task X is 1, so that $C^{\mathrm{O}pt}_{\max} \geq 1$. Moreover, allocating X on the GPU and Y on the CPU leads to a makespan of 1, so that $C^{\mathrm{O}pt}_{\max} \leq 1$ and finally $C^{\mathrm{O}pt}_{\max} = 1$.

On the other hand, let us consider the following valid HeteroPrioIndep schedule, where CPU first selects X and the GPU first selects Y. GPU becomes available at instant $\frac{1}{\phi} = \phi - 1$ but does not spoliate task X since it cannot complete X earlier than its expected completion time on the CPU. Therefore, the completion time of HeteroPrioIndep is $\phi = \phi C^{\mathrm{O}pt}_{\max}$. □

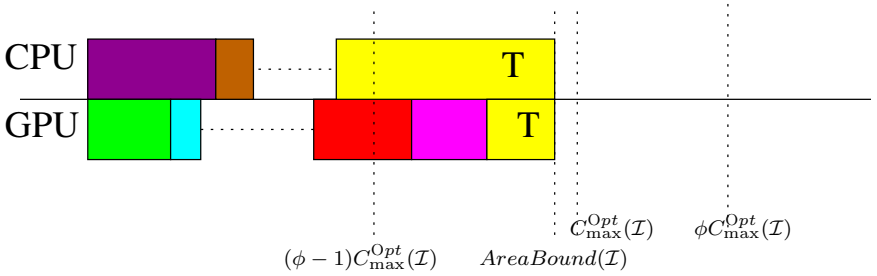


FIGURE 4 Area bound consideration to bound the acceleration factor of T.

5.3 | Approximation Ratio with 1 GPU and m CPUs

In the case of a single GPU and m CPUs, we prove in this Section in Theorem 9 that the approximation ratio of HeteroPrioIndep becomes $1+\phi = \frac{3+\sqrt{5}}{2}$, and we show in Theorem 11 that this bound is tight, asymptotically when m becomes large. Note that these results are also valid for the symmetric case (1 CPU and n GPUs), since we make no difference from both types of resources, except by their name. This Section presents the 1 GPU and m CPUs case for simplicity, and because it is more common in practice.

Theorem 9. *HeteroPrioIndep achieves an approximation ratio of* $(1+\phi) = \frac{3+\sqrt{5}}{2}$ *for any instance* $\mathcal{I}$ *on m CPUs and 1 GPU.*

Proof. Let us assume by contradiction that there exists a task T *whose completion time is larger than* $(1+\phi)C_{\max}^{Opt}$. *We know that all tasks start before* $C_{\max}^{Opt}$ *in* S_{HP}^{NS}. *If* T *is processed on the GPU in* S_{HP}^{NS}, *then* $q_T > C_{\max}^{Opt}$ *and thus* $p_T \leq C_{\max}^{Opt}$. *Since at least one CPU is idle at time* $t_{FirstIdle}$, T *should have been spoliated and processed by* $2C_{\max}^{Opt}$.

We know that T *is processed on a CPU in* S_{HP}^{NS}, *and completes later than* $(1+\phi)C_{\max}^{Opt}$ *in* S_{HP}. *Let us denote by* S *the set of all tasks spoliated by the GPU (from a CPU to the GPU) before considering* T *for spoliation in the processing of HeteroPrioIndep and let us denote by* $S' = S \bigcup \{T\}$. *We now prove the following claim.*

Claim 10. The following holds true:

- $p_i > C_{\max}^{Opt}$ for all tasks i of S',
- the acceleration factor of T is at least ϕ,
- the acceleration factor of tasks running on the GPU in $S_{\mathrm{HP}}^{\mathrm{NS}}$ is at least ϕ.

Proof of Claim 10. Since all tasks start before $t_{FirstIdle} \leq C_{\max}^{Opt}$ *in* S_{HP}^{NS}, *and since* T *completes after* $(1+\phi)C_{\max}^{Opt}$ *in* S_{HP}^{NS}, *then* $p_T > \phi C_{\max}^{Opt}$. *Since HeteroPrioIndep performs spoliation of tasks in decreasing order of their completion time, the same applies to all tasks of* S': $\forall i \in S', p_i > \phi C_{\max}^{Opt}$, *and thus* $q_i \leq C_{\max}^{Opt}$.
Since $p_T > \phi C_{\max}^{Opt}$ *and* $q_T \leq C_{\max}^{Opt}$, *then* $\rho_T > \phi$. *Since* T *is processed on a CPU in* S_{HP}^{NS}, *all tasks processed on GPU in* S_{HP}^{NS} *have an acceleration factor at least* ϕ. □

Since T *is processed on the CPU in* S_{HP}^{NS} *and* $p_T > q_T$, *Lemma 4 applies and no task is spoliated from the GPU. Let* A *be the set of tasks running on GPU right after* $t_{FirstIdle}$ *in* S_{HP}^{NS}. *We consider only one GPU, therefore* $|A| \leq 1$.

1. *If* $A = \{a\}$ *with* $q_a \leq (\phi-1)C_{\max}^{Opt}$, *then Lemma 6 applies to* S' *(with* $n = 1$*) and the overall completion time is* $\leq t_{FirstIdle} + q_A + C_{\max}^{Opt} \leq (\phi+1)C_{\max}^{Opt}$.
2. *If* $A = \{a\}$ *with* $q_a > (\phi-1)C_{\max}^{Opt}$, *since* $\rho_a > \phi$ *by Claim 10,* $p_a > \phi(\phi-1)C_{\max}^{Opt} = C_{\max}^{Opt}$. *Lemma 6 applies to* $S' \bigcup A$, *so that the overall completion time is bounded by* $t_{FirstIdle} + C_{\max}^{Opt} \leq 2C_{\max}^{Opt}$.
3. *If* $A = \emptyset$, *Lemma 6 applies to* S' *and get* $C_{\max}^{HP}(\mathcal{I}) \leq t_{FirstIdle} + C_{\max}^{Opt} \leq 2C_{\max}^{Opt}$.

Therefore, in all cases, the completion time of task T *is at most* $(\phi+1)C_{\max}^{Opt}$. □

Theorem 11. *Theorem 9 is tight, i.e. for any* $\delta > 0$, *there exists an instance* $\mathcal{I}$ *such that* $C_{\max}^{HP}(\mathcal{I}) \geq (\phi+1-\delta)C_{\max}^{Opt}(\mathcal{I})$.

Proof. For any fixed $\epsilon > 0$, let us set $x = (m-1)/(m+\phi)$ and consider the following set:

Task	*CPU Time*	*GPU Time*	*# of tasks*	*acceleration ratio*
T_1	1	$1/\phi$	*1*	ϕ
T_2	ϕ	1	*1*	ϕ
T_3	ϵ	ϵ	$(mx)/\epsilon$	1
T_4	$\epsilon\phi$	ϵ	x/ϵ	ϕ

The minimum length of task T_2 is 1, so that $C_{\max}^{Opt} \geq 1$. Moreover, if T_1, T_4 and T_3 (in this order) are scheduled on CPUs, then the completion time is at most $1+\epsilon$ (since the total work is 1 and the last finishing task belongs to T_3). With task T_2 on the GPU, this yields a schedule with makespan at most $1+\epsilon$, so that $C_{\max}^{Opt} \leq 1+\epsilon$.

Consider the following valid HeteroPrioIndep schedule. The GPU first selects tasks from T_4 and the CPUs first select tasks from T_3. All resources become available at time x. Now, the GPU selects task T_1 and one of the CPUs selects task T_2, with a completion time of $x+\phi$. The GPU becomes available at $x+1/\phi$ but does not spoliate T_2 since it would not complete before $x+1/\phi+1 = x+\phi$. The makespan of HeteroPrioIndep is thus $x+\phi$, and since x tends towards 1 when m becomes large, the approximation ratio of HeteroPrioIndep on this instance can be arbitrarily close to $1+\phi$. □

5.4 | Approximation Ratio with *n* GPUs and *m* CPUs

In the most general case with *n* GPUs and *m* CPUs, the approximation ratio of HeteroPrioIndep is at most $2+\sqrt{2}$, which will be proven in Theorem 12. To establish this result, we rely on the same techniques as in the case of a single GPU, but the result of Lemma 6 is weaker for $n > 1$, which explains why the approximation ratio is larger than in Theorem 9. We have not been able to prove, as previously, that this bound is tight, but we provide in Theorem 14 a family of instances for which the approximation ratio is arbitrarily close to $2+\frac{2}{\sqrt{3}}$.

Theorem 12. *$\forall \mathcal{I}$, $C_{\max}^{HP}(\mathcal{I}) \leq (2+\sqrt{2}) C_{\max}^{Opt}(\mathcal{I})$.*

Proof. We prove this by contradiction. Let us assume that there exists a task T whose completion time in S_{HP} is larger than $(2+\sqrt{2})C_{\max}^{Opt}$. Without loss of generality, we assume that T is processed on a CPU in S_{HP}^{NS}. In the rest of the proof, we denote by S the set of all tasks spoliated by GPUs in the HeteroPrioIndep solution, and $S' = S \cup \{T\}$. We now prove the following claim.

Claim 13. The following holds true
(i) $\forall i \in S', p_i > C_{\max}^{Opt}$
(ii) $\forall T'$ processed on GPU in $S_{\mathrm{HP}}^{\mathrm{NS}}$, $\rho_{T'} \geq 1+\sqrt{2}$.

Proof of Claim 13. In S_{HP}^{NS}, all tasks start before $t_{FirstIdle} \leq C_{\max}^{Opt}$. By assumption, T ends after $(2+\sqrt{2})C_{\max}^{Opt}$ in S_{HP}, and spoliation occurs only to improve the completion time of tasks, T ends after $(2+\sqrt{2})C_{\max}^{Opt}$ in S_{HP}^{NS} as well. This implies $p_T > (1+\sqrt{2})C_{\max}^{Opt}$. The same applies to all spoliated tasks that complete after T in S_{HP}^{NS}. If T is not considered for spoliation, no task that complete before T in S_{HP}^{NS} is spoliated, and the first result holds. Otherwise, let s_T denote the instant at which T is considered for spoliation. The completion time of T in S_{HP} is at most $s_T + q_T$, and since $q_T \leq C_{\max}^{Opt}$, $s_T \geq (1+\sqrt{2})C_{\max}^{Opt}$. Since HeteroPrioIndep handles tasks for spoliation in decreasing order of their completion time in S_{HP}^{NS}, task T' is spoliated after T has been considered and not processed at time s_T, and thus $p_{T'} > \sqrt{2}C_{\max}^{Opt}$.

Since $p_T > (1+\sqrt{2})C_{\max}^{Opt}$ and $q_T \leq C_{\max}^{Opt}$, then $\rho_T \geq (1+\sqrt{2})$. Since T is processed on a CPU in S_{HP}^{NS}, all tasks processed on GPU in S_{HP}^{NS} have acceleration factor at least $1+\sqrt{2}$. □

Let $\mathcal{A}$ be the set of tasks processed on GPUs after time $t_{FirstIdle}$ in S_{HP}^{NS}. We partition $\mathcal{A}$ into two sets $\mathcal{A}_1$ and $\mathcal{A}_2$ such that $\forall i \in \mathcal{A}_1, q_i \leq \frac{C_{\max}^{Opt}}{\sqrt{2}+1}$ and $\forall i \in \mathcal{A}_2, q_i > \frac{C_{\max}^{Opt}}{\sqrt{2}+1}$.

Since there are n GPUs, $|\mathcal{A}_1| \leq |\mathcal{A}| \leq n$. We consider the schedule induced by HeteroPrioIndep on the GPUs with the tasks $\mathcal{A} \bigcup S'$ (if T is spoliated, this schedule is actually returned by HeteroPrioIndep, otherwise this is what HeteroPrioIndep builds when attempting to spoliate task T). This schedule is not worse than a schedule that processes all tasks from $\mathcal{A}_1$ starting at time $t_{FirstIdle}$, and then performs any list schedule of all tasks from $\mathcal{A}_2 \bigcup S'$. Since $|\mathcal{A}_1| \leq n$, the

first part takes time at most $\frac{C^{Opt}_{\max}}{\sqrt{2}+1}$. *For all* T_i *in* $\mathcal{A}_2$, $\rho_i \geq 1+\sqrt{2}$ *and* $q_i > \frac{C^{Opt}_{\max}(\mathcal{I})}{\sqrt{2}+1}$ *imply* $p_i > C^{Opt}_{\max}$. *Thus, Lemma 6 applies to* $\mathcal{A}_2 \bigcup S'$ *and the second part takes at most* $2C^{Opt}_{\max}$. *Overall, the completion time on GPUs is bounded by* $t_{FirstIdle} + \frac{C^{Opt}_{\max}}{\sqrt{2}+1} + (2-\frac{1}{n})C^{Opt}_{\max} < C^{Opt}_{\max} + (\sqrt{2}-1)C^{Opt}_{\max} + 2C^{Opt}_{\max} = (\sqrt{2}+2)C^{Opt}_{\max}$, *which is a contradiction.* □

Theorem 14. *The approximation ratio of HeteroPrioIndep is at least* $2+\frac{2}{\sqrt{3}} \approx 3.15$.

Proof. We consider an instance $\mathcal{I}$, *with* $n = 6k$ *GPUs and* $m = n^2$ *CPUs, containing the following tasks.*

Task	*CPU Time*	*GPU Time*	*# of tasks*	*acceleration ratio*
T_1	n	$\frac{n}{r}$	n	r
T_2	$\frac{rn}{3}$	*see below*	*see below*	$\frac{r}{3} \leq \rho \leq r$
T_3	1	1	mx	1
T_4	r	1	nx	r

where $x = \frac{(m-n)}{m+nr}n$ *and* r *is the solution of the equation* $\frac{n}{r} + 2n - 1 = \frac{nr}{3}$. *Note that the highest acceleration factor is* r *and the lowest is* 1 *since* $r > 3$. *The set* T_2 *contains tasks with the following processing time on GPU: one task of length* $n = 6k$, *and for all* $0 \leq i \leq 2k-1$, *six tasks of length* $2k+i$.

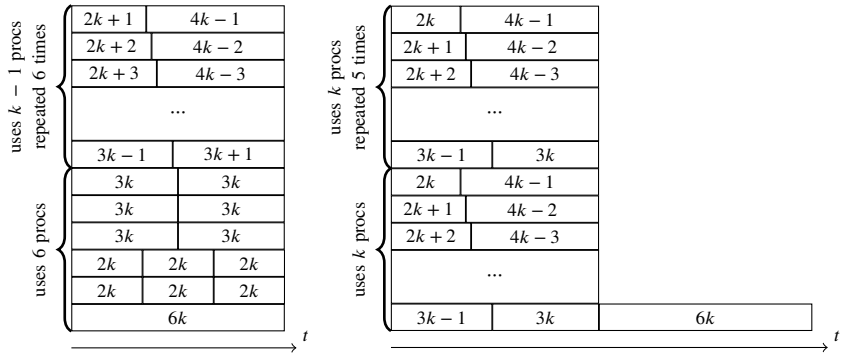


FIGURE 5 2 schedules for task set T_2 on $n = 6k$ homogeneous processors, tasks are labeled with processing times. Left one is an optimal schedule and right one is a possible list schedule.

This set T_2 *of tasks can be scheduled on n GPUs in time n (see Figure 5).* $\forall 1 \leq i < k$, *each of the six tasks of length* $2k+i$ *can be combined with one of the six tasks of length* $2k+(2k-i)$, *occupying* $6(k-1)$ *processors; the tasks of length* $3k$ *can be combined together on* 3 *processors, and there remains* 3 *processors for the six tasks of length* $2k$ *and the task of length* $6k$. *On the other hand, the worst list schedule may achieve makespan* $2n-1$ *on the n GPUs.* $\forall 0 \leq i \leq k-1$, *each of the six tasks of length* $2k+i$ *is combined with one of the six tasks of length* $4k-i-1$, *which occupies all* $6k$ *processors until time* $6k-1$, *then the task of length* $6k$ *is processed. The fact that there exists a set of tasks for which the makespan of the worst case list schedule is almost twice the optimal makespan is a well known result (28). However, the interest of set* T_2 *is that the smallest processing time is* $C^{Opt}_{\max}(T_2)/3$, *which allows these tasks to have a large processing time on CPU in instance* $\mathcal{I}$ *(without having a too large acceleration factor).*

Figure 6 a shows an optimal schedule of length n for this instance: the tasks from set T_2 *are scheduled optimally on the n GPUs, and the sets* T_1, T_3 *and* T_4 *are scheduled on the CPUs. Tasks* T_3 *and* T_4 *fit on the* $m-n$ *CPUs because the total work is* $mx + nxr = x(m+nr) = (m-n)n$ *by definition. On the other hand, Figure 6 b shows a possible HeteroPrioIndep schedule*[2]

[2]Once again, it is possible to make sure that this is the *only* possible HeteroPrioIndep schedule with arbitrarily small changes to execution times: it is enough to make sure that tasks T_4 have a larger acceleration factor than tasks T_1, and that the CPU execution times of some tasks in T_2 are slightly increased to ensure that spoliation results in the schedule at the right of Figure 5 .

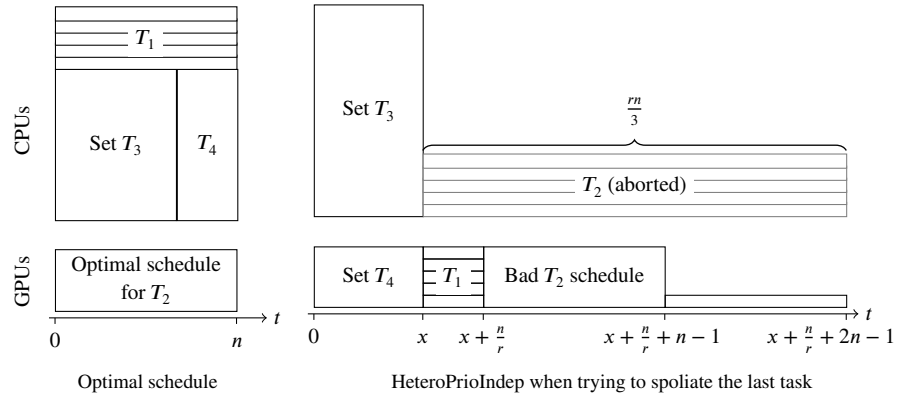


FIGURE 6 Optimal and HeteroPrioIndep on Theorem 14 instance.

for $\mathcal{I}$. The tasks from set T_3 have the lowest acceleration factor and are scheduled on the CPUs, while tasks from T_4 are scheduled on the GPUs. All resources become available at time x. Tasks from set T_1 are scheduled on the n GPUs, and tasks from set T_2 are scheduled on m CPUs. At time $x + \frac{n}{r}$, the GPUs become available and start spoliating the tasks from set T_2. Since they all complete at the same time, the order in which they get spoliated can be arbitrary, and it can lead to the worst case behavior of Figure 5 , where the task of length n is processed last. In this case, spoliating this task does not improve its completion time, and the resulting makespan for HeteroPrioIndep on this instance is $C^{HP}_{\max}(I) = x + \frac{n}{r} + 2n - 1 = x + \frac{nr}{3}$ by definition of x. The approximation ratio on this instance is thus $C^{HP}_{\max}(I)/C^{Opt}_{\max}(I) = x/n + r/3$. When n becomes large, x/n tends towards 1, and r tends towards $3 + 2\sqrt{3}$. Hence, the ratio $C^{HP}_{\max}(I)/C^{Opt}_{\max}(I)$ tends towards $2 + 2/\sqrt{3}$. □

6 | PROOF OF HeteroPrioDep APPROXIMATION RESULTS

6.1 | Approximation Guarantee

In this Section, we prove an $(n + m)$ approximation ratio for HeteroPrioDep (Algorithm 2, page 6), the version of HETEROPRIO adapted to task graphs. To analyze the schedule S produced by HeteroPrioDep, let us first define *happy* tasks and prove a few lemmas.

Definition 1. A task T is said to be *happy* in schedule S at time t if it runs on its favorite resource type.

We can note that by definition, a spoliated task is always happy after having been spoliated. The definition of HeteroPrioDep allows to state the following lemma.

Lemma 15. *In the schedule S, if a task completes at time t on resource type R, then at least one of the following holds:*

- *the schedule completes at time t,*
- *there is at least one happy task running just after time t,*
- *there is one task running on another resource, which was not a candidate for spoliation at time t or earlier.*

Proof. Let us assume that task T completes at time t. If the schedule does not complete at time t, then there are remaining tasks to run and not all resources will stay idle. Furthermore, if no task is running alongside T before time t, then resources of both types are idle at time t and the last test in Algorithm 2 ensures that at least one of the next running tasks will be happy.

Thus, if the schedule does not complete at time t and no happy task runs after time t, there was a task running alongside T, that was not stolen at time t (otherwise it would have become happy). We can identify two cases:

- *Case 1: some task T' is running on the other type of resource at time t. Note that any unhappy task running on R has a worse acceleration factor for R than T'. Since HeteroPrioDep decided either to run no task after time t on resource R, or to run an unhappy task, it means that HeteroPrioDep did not consider T' for spoliation, and thus that it was not a candidate.*

- *Case 2: no task is running on the other type of resource. Denote as T' a (unhappy) task running on resource type R at time t. Since T' is unhappy, all resources of the other type R' were busy when it started, and since they are idle at time t, there exists $t' < t$ where one resource of type R' becomes idle. Since T' was not spoliated at time t', it implies that it was not candidate for spoliation at time $t' < t$.*

□

Let us consider the following (trivial) lower bound on the optimal makespan.

Lemma 16. *For a task graph G on m CPUs and n GPUs, we have:*

$$\sum_{T \in G} \min(p_T, q_T) \leq (m+n) C_{\max}^{Opt}(G)$$

Proof. Any assignment has to process all tasks, with the duration of task T being at least $\min(p_T, q_T)$. Since $n+m$ resources are available, the result follows. □

Theorem 17. *The length of the schedule returned by HeteroPrioDep as defined in Algorithm 2 on a task graph G on m CPUs and n GPUs is at most $(m+n)$ times the optimal makespan.*

Proof. From the HeteroPrioDep schedule S, we build intervals in the following way: start at time $t = 0$, when a happy task T starts. Then,

1. *Create an interval from t to the end of task T, and associate this interval to task T. Let t denote the ending time of T.*
2. *If t denotes the end of the schedule, stop.*
3. *If there exists a happy task T' running after t, set $T = T'$ and go back to point 1.*
4. *Otherwise, from Lemma 15, there exists an unhappy task which was not candidate for spoliation. Call it task T and go to point 1. Please note that since this task T was not candidate for spoliation at time t or earlier, it is not candidate at any later time and thus will run to completion on its current resource.*

This procedure partitions the time from 0 to C (the completion time of S) in intervals. Each interval is associated to a task, and each task is associated to at most one interval. Then, the following claim holds true.

Claim 18. If an interval I is associated with task T, then the length of I is at most $\min(p_T, q_T)$.

Proof. If task T is happy, then its processing time is $\min(p_T, q_T)$, and the result follows directly from the fact that I does not start before task T starts being processed.

If task T is unhappy, let us denote by l the left point of I, and $e(T)$ the ending time of T (and of interval I). Since task T was not candidate for spoliation at time l or earlier, we have is $l + \min(p_T, q_T) \geq e(T)$. Thus the length of I is $e(T) - l \leq \min(p_T, q_T)$. □

By summing over all intervals, we get $C = \sum_I len(I) \leq \sum_{T \text{ associated to } I} \min(p_T, q_T) \leq \sum_T \min(p_T, q_T)$. □

We provide in Figure 7 an example depicting the decomposition of a HeteroPrioDep schedule S into intervals. In this schedule, tasks B, C, D, E and F complete on their favorite resource type, while A, G and H complete on their less favorite resource type. This HeteroPrioDep schedule exhibits all possible kinds of behaviors (both resources process their favorable (resp. less favorable) tasks, a resource executes a favorable (resp. less favorable) task and the other resource is idle, a task is spoliated).

A possible interval sequence for this HeteroPrioDep schedule as described in Theorem 17 is the following: $[x_0, x_1]$, $[x_1, x_2]$, $[x_2, x_3]$, $[x_3, x_4]$, $[x_4, x_6]$, $[x_6, x_7]$, $[x_7, x_8]$. Claim 18 shows the following inequalities:

- $x_1 - x_0 \leq \min(p_B, q_B)$, $x_3 - x_2 \leq \min(p_C, q_C)$, $x_4 - x_3 \leq \min(p_D, q_D)$, $x_6 - x_4 \leq \min(p_F, q_F)$ because tasks B, C, D and F are happy;
- $x_2 - x_1 \leq \min(p_A, q_A)$, $x_7 - x_6 \leq \min(p_G, q_G)$, $x_8 - x_7 \leq \min(p_H, q_H)$ because tasks A, G and H were not spoliated at the start of their interval.

The completion time $C = x_8 - x_0$ of HeteroPrioDep thus satisfies:

$$C = x_8 - x_0 \leq \sum_{i \in \{A,B,C,D,F,G,H\}} \min(p_i, q_i) \leq \sum_{i \in \{A,B,C,D,E,F,G,H\}} \min(p_i, q_i).$$

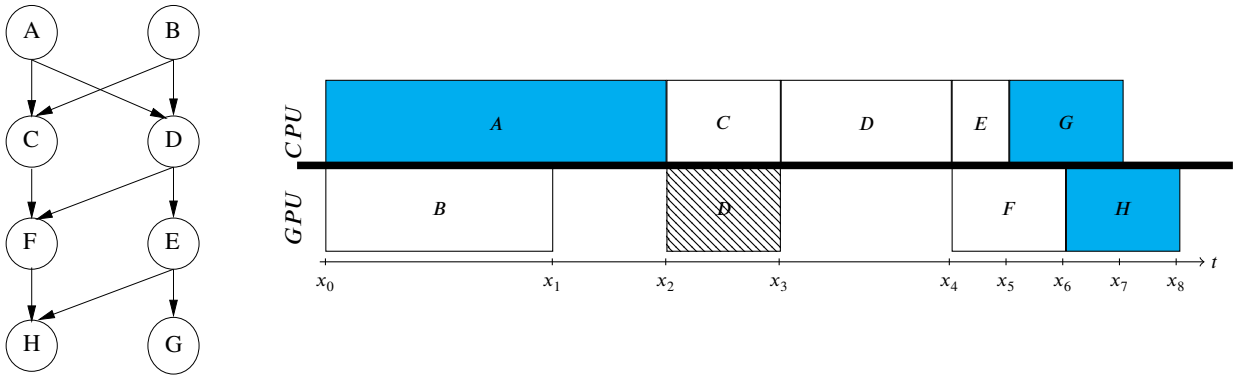


FIGURE 7 A task graph and its HeteroPrioDep schedule on 1 CPU and 1 GPU. An aborted task is shown in a pattern box. Blue-colored tasks in the schedule are unhappy.

6.2 | Worst-case example for $n = m = 1$

We now exhibit a worst case example which shows that our bound is tight for the case with $n = 1$ GPU and $m = 1$ CPU. We consider an instance with two types of tasks: tasks of type A have a processing time of $p_A = 1$ on CPU and $q_A = 2$ on GPU, and tasks of type B have $p_B = 2$ and $q_B = 1$. The dependencies are made of two chains (see Figure 8): a chain of k tasks of type A, and a chain with one task of type A and $k - 1$ tasks of type B.

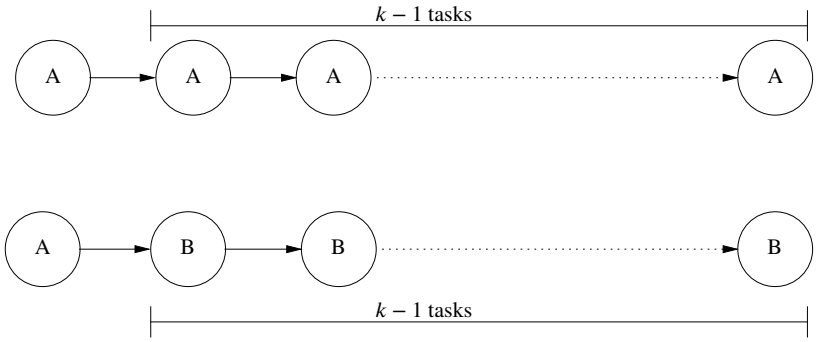


FIGURE 8 Two chains of tasks.

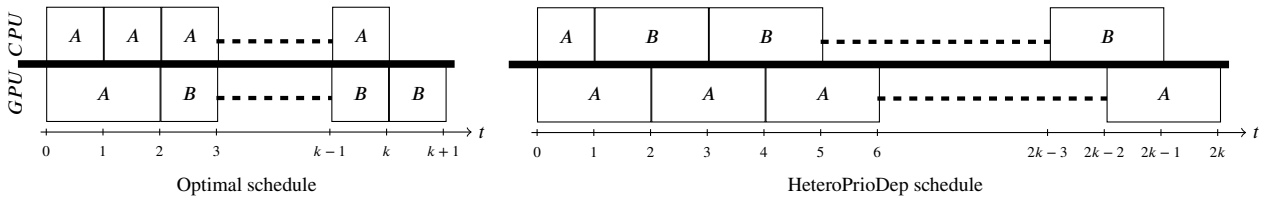


FIGURE 9 Optimal and HeteroPrioDep schedules for the worst case instance in Figure 8 .

Possible optimal and HeteroPrioDep schedules are depicted on Figure 9 . In a HeteroPrioDep schedule, since the first tasks are identical with respect to acceleration factor, it is possible that the CPU executes all tasks of the second chain, and the GPU all tasks of the first chain, with almost only unhappy tasks, each time a task completes, the unhappy task running on the other resource is not spoliated because this would not improve its completion time. The makespan is thus $2k$. On the other hand, it

is possible to schedule the first chain on the CPU, and the second chain on the GPU, which leads to a makespan of $k + 1$. This shows that the ratio 2 is tight in that case.

7 | EXPERIMENTAL EVALUATION

In this section, we propose an experimental evaluation of HETEROPRIO on instances coming from the dense linear algebra library Chameleon (29). We evaluate our algorithms in two contexts, (i) with independent tasks and (ii) with dependencies, which is closer to real-life settings and is ultimately the goal of the HETEROPRIO algorithm. In this section, we use task graphs from Cholesky, QR and LU factorizations, which provide interesting insights on the behavior of the algorithms. The Chameleon library is built on top of the StarPU runtime (4), and implements tiled versions of many linear algebra kernels expressed as graphs of tasks. Before the execution, the processing times of the tasks are measured on both types of resources, which then allows StarPU scheduler to have a reliable prediction of each task's processing time. In this section, we use this data to build input instances for our algorithms, obtained on a machine with 20 CPU cores of two Haswell Intel® Xeon® E5-2680 processors and 4 Nvidia K40-M GPUs. We consider Cholesky, QR and LU factorizations with a tile size of 960, and a number of tiles N varying between 4 and 64. All the code and data necessary to reproduce these experiments are available in (30).

We compare several algorithms from the literature :

- HETEROPRIO, with three variants that differ on the spoliation rule. HeteroPrioIndep is the variant for which an approximation ratio is proven in the case of independent tasks (in Section 5): when a resource selects a task for spoliation, it selects the task that will complete the latest. HeteroPrioDep is the variant used in the case with dependencies (see Algorithm 2 and Section 6), in which an idle resource may spoliate a task even if there still exists ready tasks, and the task with the best acceleration factor is selected. Finally, HeteroPrio is the original version from (13), which empirically obtains the best results, in which spoliation only happens when there is no ready task, and the task with highest priority is selected. Note that the priorities are not given by the user, but automatically computed from the task graph, as explained in more detail in Section 7.2.
- the well-known HEFT algorithm (11) (designed for the general $R|prec|C_{\max}$ problem).
- Several algorithms based on independent tasks scheduling
  - DualHP from (22) (specifically designed for CPU and GPU, with an approximation ratio of 2 for independent tasks). The DualHP algorithm works as follows : for a given guess λ on the makespan, it either returns a schedule of length 2λ, or ensures that $\lambda < C_{\max}^{Opt}$. To achieve this, any task with processing time more than λ on any resource type is assigned to the other resource type, and then all remaining tasks are assigned to the GPU by decreasing acceleration factor while the overall load is lower than $n\lambda$. If the remaining load on CPU is not more than $m\lambda$, the resulting schedule has makespan below 2λ. The best value of λ is then found by binary search.
  - BalancedEstimate and BalancedMakespan from (12), which also exhibits a 2-approximation guarantee for independent tasks. Both algorithms work by assigning each task to its favorite resource type, and then applying a selection rule to the tasks assigned to the most loaded resource type, keeping the best assignment met in the process. After task allocation, the tasks are then scheduled using LPT rule. The difference between both algorithms is the criteria used to determine the best assignment. In BalancedMakespan, the LPT schedule is computed for all assignment, whereas in BalancedEstimate, the largest average load on both types of resources is used as a cheaper estimate of the resulting makespan.
  - CYZ-5 is Algorithm Al5 from (21), an online algorithm with a 3.85 approximation guarantee, which works as follows: tasks with very long computation time on CPU go to GPU, tasks with very high or very low acceleration factor go on their best resource, and the remaining tasks are scheduled by attempting to balance their load between both resources. All these rules have parameter-based thresholds, whose values are chosen so as to optimize the approximation guarantee. For better fairness, since we do not consider an online context, we have added a sorting step before running this algorithm: we sort tasks by increasing and decreasing acceleration factor, fastest execution time, or average execution time, and we keep the best schedule out of all results.
- AREALIST is an algorithm designed for scheduling task graphs on two types of resources (26), with a 6-approximation guarantee. This algorithm first performs a rounding of the solution of a relaxed linear program to decide which tasks should

be allocated to each type of resource. Then all tasks are scheduled with a list scheduling algorithm. We also consider AREALISTSTEAL, which is the same algorithm except that idle GPUs are allowed to spoliate tasks from the CPUs. This avoids following too rigidly the allocation computed from the linear program, and in practice yields shorter schedules. Note that the 6-approximation guarantee of AREALIST does not trivially extend to AREALISTSTEAL.

7.1 | Independent Tasks

To obtain realistic instances with independent tasks, we have taken the actual measurements from tasks of each kernel (Cholesky, QR and LU) and considered these as independent tasks. For each instance, the performance of all three algorithms is compared to the area bound. Results are depicted in Figure 10 , where the ratio to the area bound is given for different values of the number of tiles N. Since the results for the different kernels are similar, the plot only shows the average values: for each algorithm and input size, we compute the average of its performance obtained with Cholesky, LU and QR instances.

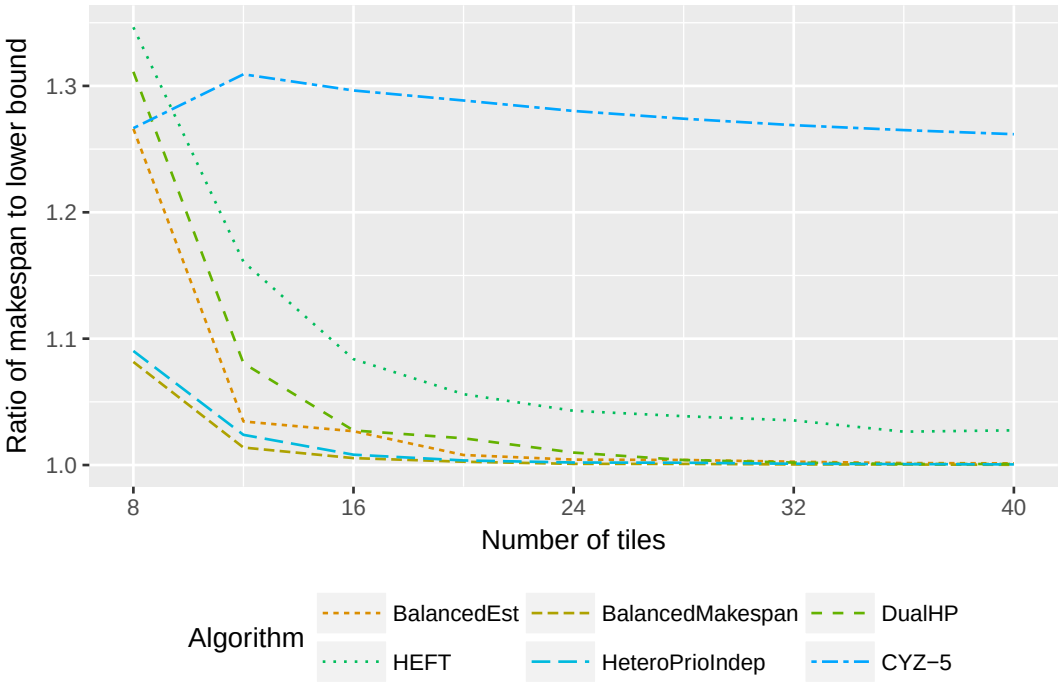


FIGURE 10 Results for independent tasks.

The results show that all algorithms except HEFT and CYZ-5 achieve close to optimal performance when N is large, but HeteroPrioIndep and BalancedMakespan achieve better results than BalancedEstimate and DualHP for small values of N (below 20). This may be surprising, since the approximation ratios of DualHP and BalancedEstimate are actually better than the one of HeteroPrioIndep. On the other hand, HeteroPrioIndep is primarily a list scheduling algorithm, that usually achieves good average case performance. In this case, it comes from the fact that DualHP and BalancedEstimate tend to balance the *load* between the set of CPUs and the set of GPUs, but for such values of N, the processing times of the different tasks on CPU are not negligible compared to the makespan. Thus, it happens that average loads are similar for both types of resources, but one CPU actually has significantly higher load than the others, and this results in a larger makespan. This does not happen with BalancedMakespan, and explains its very good performance for all input sizes. HEFT, on the other hand, has rather poor performance because it does not take acceleration factor into account, and thus assigns tasks to GPUs that would be better suited to CPUs, and vice-versa. CYZ-5 suffers from the issue mentioned in Section 3: since most tasks have high acceleration factor, they are all assigned to GPUs and the CPUs are kept idle. This satisfies the approximation guarantee, but leads to poor practical performance. A final

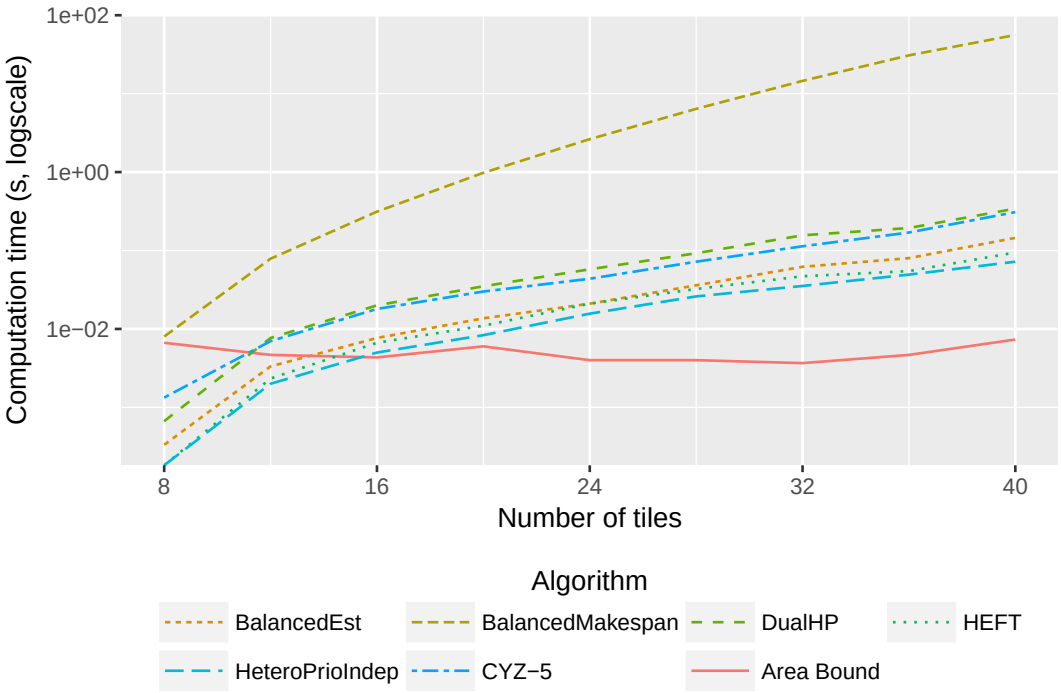


FIGURE 11 Computation time for independent tasks.

remark is that the independent-specific version HeteroPrioIndep has exactly the same performance as the generic, priority-based HeteroPrio (the lines of HeteroPrioIndep and HeteroPrio are superimposed in Figure 10).

Figure 11 shows the computational times of each algorithm, obtained as above with an average of the computation times on Cholesky, LU and QR instances. We observe that list algorithms (HEFT and HeteroPrioIndep, together with BalancedEstimate) have the fastest computation times. The binary search procedure of DualHP induces a (slightly) higher computation time. Finally, the BalancedMakespan algorithm has a higher cost, because it involves computing an actual LPT schedule for each of these possible assignments, resulting in a quadratic complexity. Finally, we note that these instances correspond to very large datasets: for $N = 40$, the instances contain more than $10,000$ tasks, and most algorithms still exhibit a processing time below half a second. To put these values into perspective, the actual execution time of a Cholesky factorization on the considered platform goes from 0.6 seconds for $N = 16$ to 20 seconds for $N = 64$.

Therefore, on our instances, HeteroPrioIndep produces the lowest makespan schedule among algorithms with reasonable computing time.

7.2 | Task Graphs

All the algorithms presented in the previous section can be adapted to take dependencies into account, by applying at any instant when a resource becomes idle the algorithm (for independent tasks) on the set of (currently) ready tasks. For DualHP, BalancedEstimate and BalancedMakespan, this implies recomputing the assignment of tasks onto resources each time a resource becomes idle (tasks which were already allocated but not started yet can be re-assigned by the algorithm), and also slightly modifying the algorithm to take into account the load of currently processed tasks. CYZ-5 can also be converted in this way, we denote the resulting algorithm as the batch version of CYZ-5. We also implemented an online version of CYZ-5, closer to the original algorithm from (21), which assigns tasks to CPUs or GPUs as soon as they become ready, and schedules tasks within each type of resource by list scheduling.

Finally, since HETEROPRIO is a list algorithm, its basic rule can be used to assign a ready task to any idle resource, as described in Algorithm 2. By abuse of notation, we will denote as HeteroPrioIndep the algorithm obtained this way, in which spoliation is made as in Algorithm 1 (only when no ready task is available, spoliate the task which finishes the latest). We also consider

Algorithm	Ranking Schemes	Spoliation	Sorting criterion
HeteroPrioIndep	`min`, `avg`, `area`	Yes	-
HeteroPrioDep	`min`, `avg`, `area`	Yes	-
HeteroPrio	`min`, `avg`, `area`	Yes	-
DualHP	`min`, `avg`, `area`, `fifo`	No	-
HEFT	`min`, `avg`, `area`	No	-
BalancedEstimate	`min`, `avg`, `area`, `fifo`, `LPT`	No	-
BalancedMakespan	`min`, `avg`, `area`, `fifo`, `LPT`	No	-
CYZ-5 (batch)	`min`	No	inc. or dec. `accel`, `min`, `avg`, `none`
CYZ-5 (online)	`min`, `avg`, `area`, `fifo`, `none`	Yes / No	-
AREALIST	-	No	-
AREALISTSTEAL	-	Yes	-

TABLE 3 List of all flavors for the 11 algorithms considered in the experimental evaluation

the generic version HeteroPrio, in which spoliation is also made when no ready task is available, but by selecting the task with highest priority.

When scheduling task graphs, a standard approach is to compute task priorities based on the dependencies. For homogeneous platforms, the most common priority scheme is to compute the *bottom-level* of each task, *i.e.* the maximum cost of a path from this task to the exit task, where nodes of the graph have a cost equal to the processing time of the corresponding task. In the heterogeneous case, the priority scheme used in the standard HEFT algorithm (11) is to set the cost of each node as the average processing time of the corresponding tasks on all resources. We will denote this scheme `avg`. A more optimistic view could be to set the cost of each node as the smallest processing time on all resources, hoping that tasks will get processed on their favorite resource type. We will denote this scheme `min`. A last possibility is to use the result from the area bound (ignoring dependencies), and to compute for each task its average execution time, where each resource is weighted with the fraction of this task computed on this resource. This scheme is denoted by `area`.

In both HETEROPRIO and DualHP, these ranking schemes are used to break ties. In HeteroPrio, whenever two tasks have the same acceleration factor, the highest priority task is assigned first; furthermore, when several tasks can be spoliated for some resource, the highest priority candidate is selected. In DualHP, once the assignment of tasks to CPUs and GPUs is computed, tasks are sorted by highest priority first and processed in this order. For DualHP, we also consider another ranking scheme, `fifo`, in which no priority is computed and tasks are assigned in the order in which they become ready. We have also added these priority schemes to the Balanced algorithms, in addition to the LPT rule which is used in the original paper.

We thus consider a total of 11 algorithms: the three variants of HETEROPRIO, DualHP, HEFT, BalancedEstimate, BalancedMakespan, the batch and online variants of CYZ-5, and AREALIST with its spoliation variant AREALISTSTEAL. Most of them are paired with a ranking scheme, which is one of `min`, `avg`, `area` or `fifo`. The batch variant of CYZ-5 also includes a sorting criterion for the ready tasks (accelaration factor, smallest execution time, average execution time, or none, as mentioned above). The online variant of CYZ-5 has two flavors in addition to ranking scheme: with or without spoliation. Finally, BalancedEstimate and BalancedMakespan can also use the `LPT` rule as specified in their original paper. All flavors of the algorithms are shown in Table 3 .

To simplify the presentation, for each of the 11 algorithms, we only present the result of the flavor (ranking scheme and/or sorting criterion) which achieves the lowest makespan. We again consider three types of task graphs: Cholesky, QR and LU factorizations, with the number of tiles N varying from 4 to 64. For each task graph, the makespan with each algorithm is computed, and we consider the ratio to the mixed area bound, which is a lower bound obtained by adding dependency constraints to the area bound (13, 26). In a first figure (Figure 12), we compare the results of the variants of HETEROPRIO. Results of all algorithms are depicted in Figure 13 .

The first conclusion from these results is that scheduling DAGs corresponding to small or large values of N is relatively easy, and all algorithms achieve a performance close to the lower bound. With small values of N, the makespan is constrained by the critical path of the graph, and executing all tasks on GPU is the best option. When N is large, the available parallelism is large

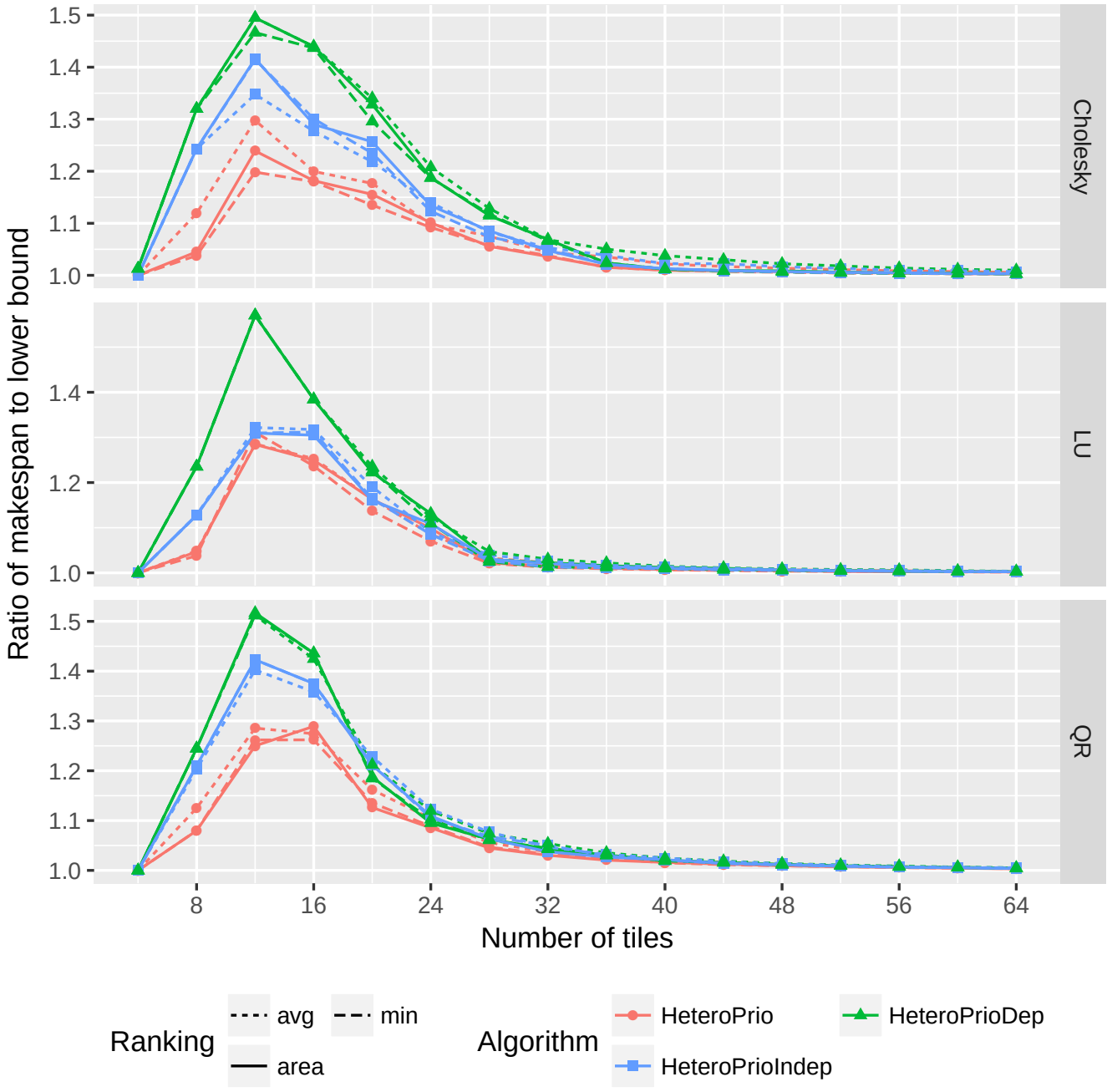


FIGURE 12 Comparison of the different HETEROPRIO versions for different DAGs.

enough, and the runtime is dominated by the available work. The interesting part of the results is thus for the intermediate values of N, between 10 and 30 or 40 depending on the task graph. In these cases, Figure 12 shows that the generic version HeteroPrio, despite not having an approximation ratio proof, obtains significantly better results than its more specialized counterparts. The version HeteroPrioDep tends to perform too many spoliations and thus wastes more computational power. On the other hand, HeteroPrioIndep does not consider priorities when spoliating tasks, which prevents from finding the most efficient schedules. Another interesting result is that in most cases, the `min` ranking scheme obtains better results for HETEROPRIO; this can be explained by the fact that indeed, in the resulting schedules, a very large fraction of the tasks are processed by GPUs. On Figure 13 , we have only included the generic version HeteroPrio and compare it with all other algorithms. On this graph, we can identify four groups of algorithms:

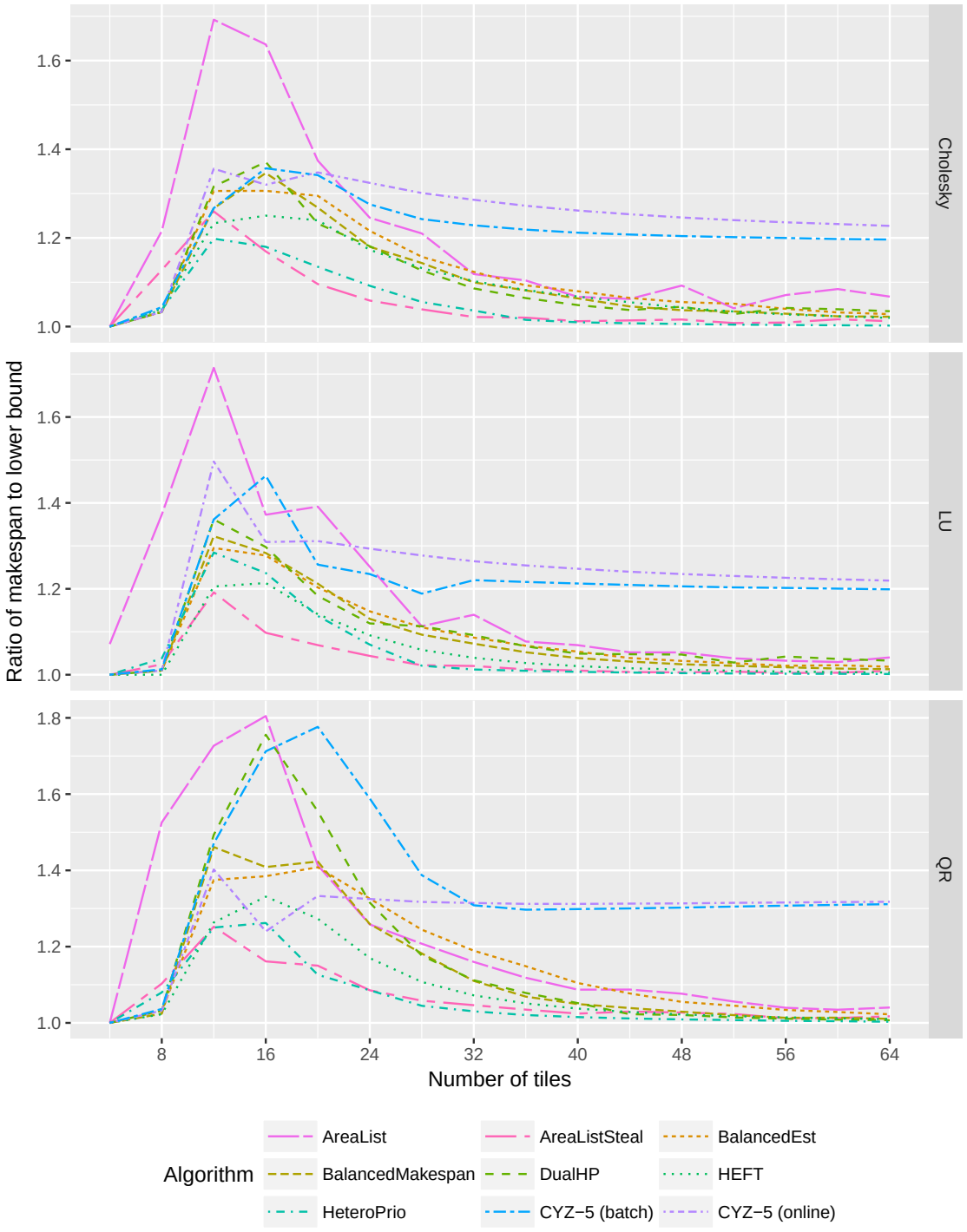


FIGURE 13 Results for different DAGs.

- The AREALIST algorithm obtains significantly worse performance than the others, despite being the only algorithm with a constant approximation guarantee. This comes from the fact that it relies strictly on an assignment of tasks to resource types, what induces long idle times on the GPUs when critical tasks have been assigned to the CPU.
- The CYZ-5 algorithms obtain reasonable performance for small N values (except for CYZ-5 batch for the QR task graph), but fail to achieve low makespan when N becomes large, similarly to the case with independent tasks.

- Algorithms based on successively solving an independent task scheduling problem (DualHP and the Balanced algorithms) have relatively close performance, with a factor to the lower bound ranging from 1.25 in the Cholesky case to 1.4 in the QR case (even 1.7 for DualHP).
- The three more "greedy" algorithms: HEFT, HeteroPrio, and AREALISTSTEAL are always within 30% of the lower bound. HeteroPrio outperforms HEFT in most cases, the only exception being for the LU task graph and N ranging from 8 to 20. In particular, HeteroPrio and AREALISTSTEAL consistently reach the lower bound as soon as $N = 32$ for all task graphs, whereas other algorithms require much larger values of N to achieve good performance.

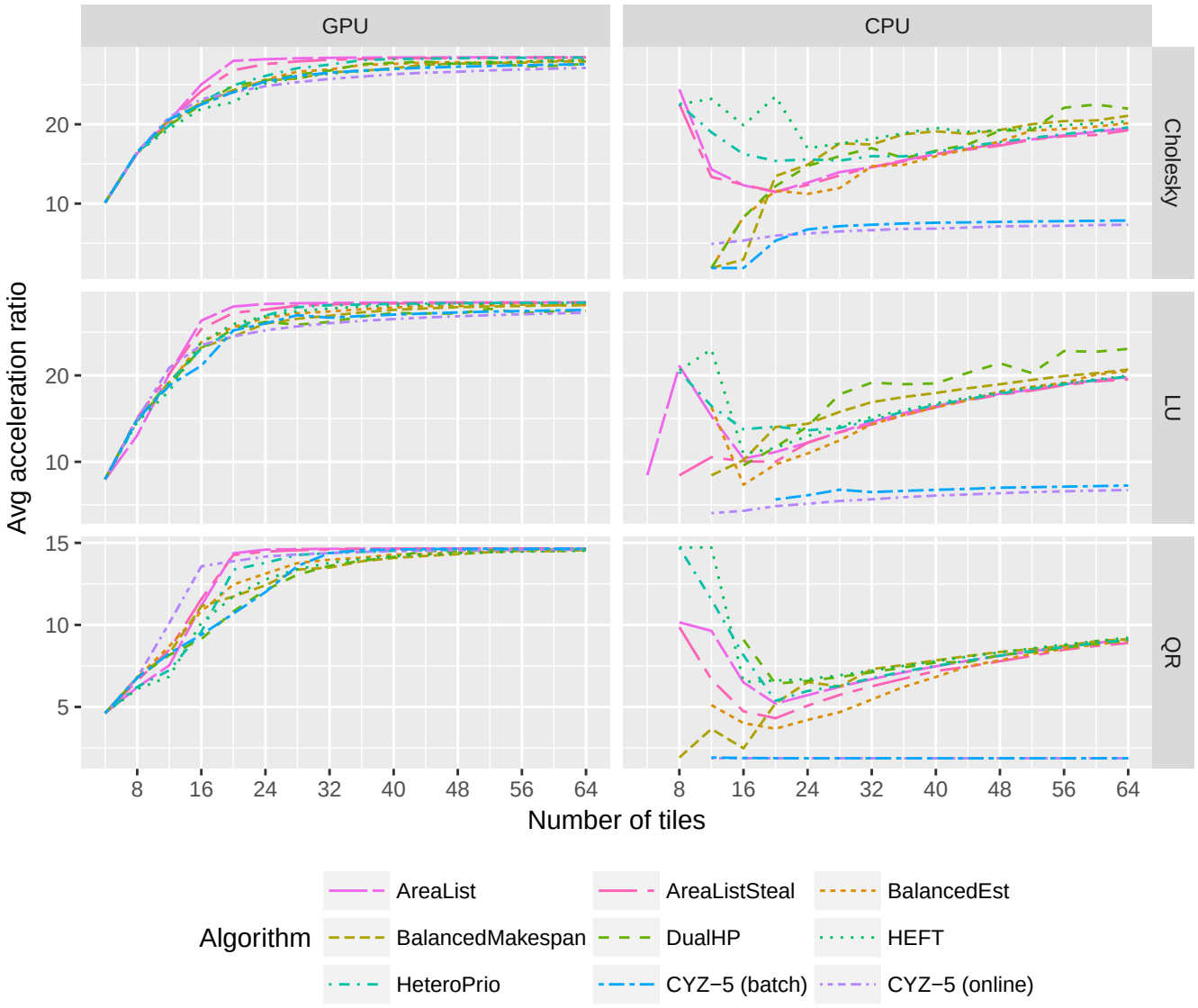


FIGURE 14 Equivalent acceleration factors.

To obtain a better insight on these results, we further analyze the schedules produced by each algorithm by focusing on two metrics. We first consider the amount of idle time on each type of resources (CPU and GPU – for fairness, any unfinished work processed on a spoliated task by HETEROPRIO contributes to idle time, so that all algorithms have the same amount of work to execute). We also consider the adequacy of task allocation, *i.e.* whether the tasks allocated to each resource type is a good fit

or not. To measure the adequacy of task allocation on a resource type r, we define the acceleration factor $\mathcal{A}_r$ of the "equivalent task" made of all the tasks assigned to that resource. let J be the set of tasks assigned to r, $\mathcal{A}_r = \frac{\sum_{i \in J} p_i}{\sum_{i \in J} q_i}$. A schedule has a good adequacy of task allocation if $\mathcal{A}_{\text{GPU}}$ is high and $\mathcal{A}_{\text{CPU}}$ is low. The values of equivalent acceleration factors for both resource types are shown on Figure 14 . On Figure 15 , the normalized idle time on each resource type is depicted, which is the ratio of the idle time on a resource type to the amount of that resource type used in the lower bound solution. Here also, to make graphs easier to read, we only included the variant of that algorithm which obtained the best makespan for that particular instance. This ensures that the results shown in these graphs are consistent with those shown on Figure 13 .

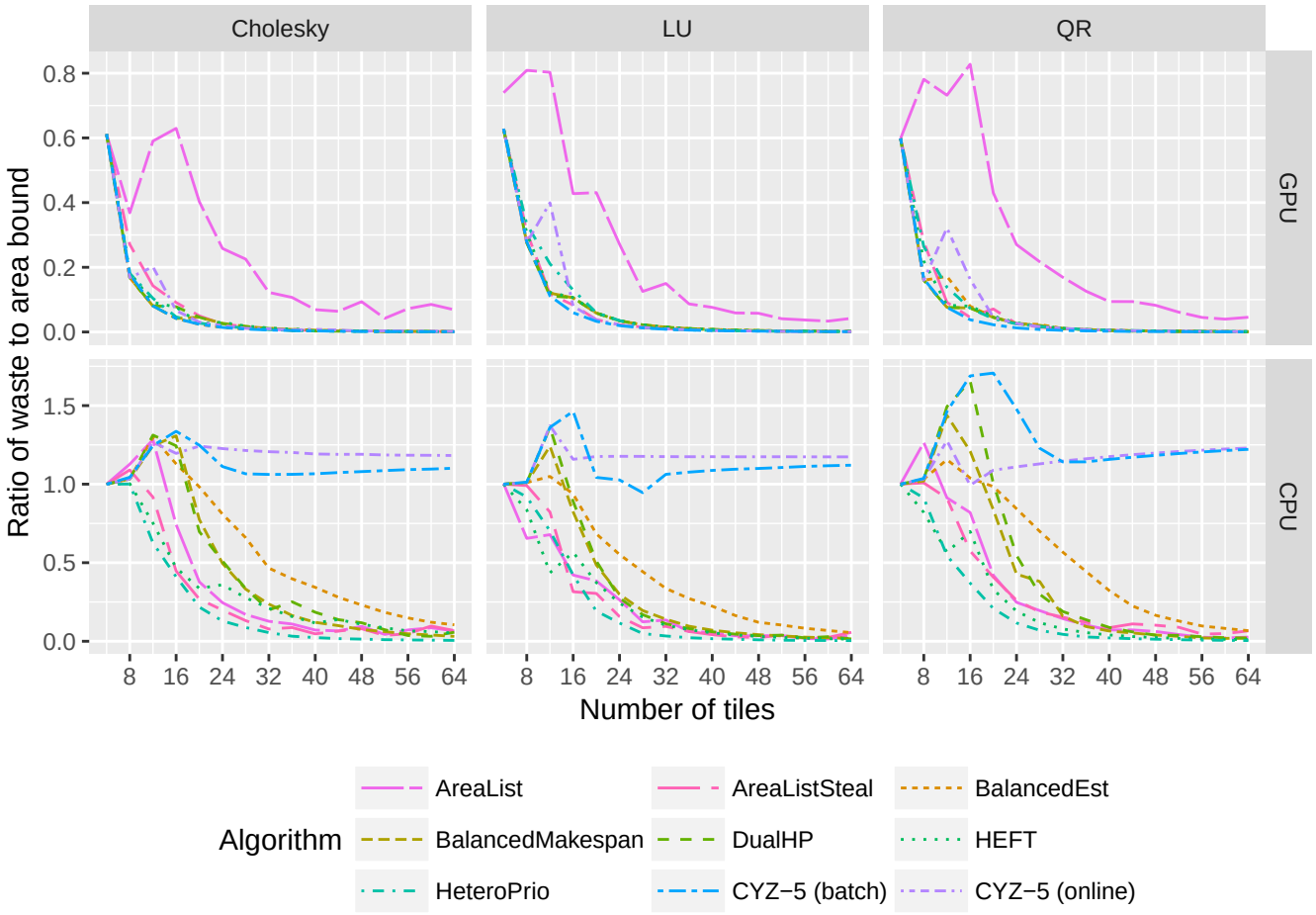


FIGURE 15 Normalized idle time.

On Figure 14 , we first observe the results for tasks allocated to the GPU (recall that in that case, higher values are better). We see that AREALIST schedules achieve the highest acceleration factors in all cases, and that HeteroPrio is among the highest value among the remaining algorithms in all cases. On the other hand, the acceleration factor of tasks assigned to the CPU between the different algorithms varies a lot. We can nevertheless observe the following trends. The CYZ algorithms assign tasks to the CPU with significantly lower acceleration factor (which is a good thing). DualHP and the Balanced algorithms obtain low values also when N is small, but get much higher when N is large. AREALIST schedules have a good assignment in that regard, in all cases. HEFT makes significantly worse assignments on the CPU, with higher acceleration factors. HeteroPrio is on the middle ground in most cases. In summary, AREALIST makes significantly better assignments, CYZ, DualHP and Balanced have a stronger emphasis on low CPU acceleration factors, HEFT obtains rather inadequate assignments, and HeteroPrio has a balanced assignment, in between HEFT and the other algorithms.

Let us now consider the idle time in the schedules. The first observation on Figure 15 is the high idle time on GPU for AREALIST, which explains its bad performance. Other algorithms all have rather similar (low) waste on the GPU, since this represents the critical resource type. We can observe how the CYZ algorithms fail to assign enough tasks to the CPUs, which explains their low performance. We can also see that DualHP and the Balanced algorithms induce much larger idle times on the CPU than other algorithms, whereas HEFT and HeteroPrio are able to keep relatively low idle times in all cases. The reason for this is that optimizing locally the makespan for the currently available tasks makes the algorithm too conservative, especially at the beginning of the schedule where there are not many ready tasks. These algorithms thus assign all tasks on the GPU because assigning one on the CPU would induce a larger completion time. HeteroPrio however is able to find a good compromise by keeping the CPU busy with the tasks that are not well suited for the GPU, and relies on the spoliation mechanism to ensure that bad decisions do not penalize the makespan.

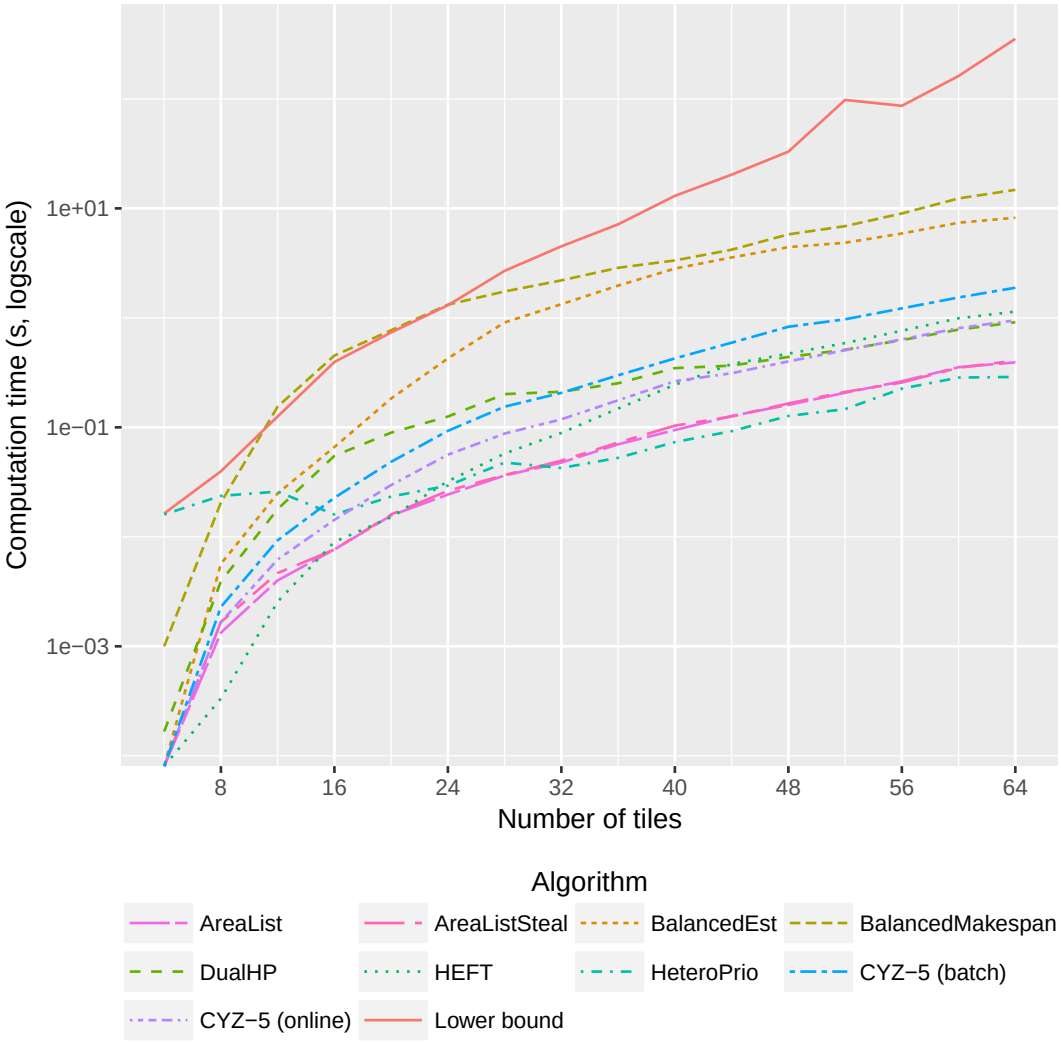


FIGURE 16 Computation time required to obtain the schedules.

Finally, we show on Figure 16 the computation times incurred by each algorithm to obtain the schedules. Similarly to the independent tasks case, we show here the average value obtained on the three applications (Cholesky, LU, QR). Most algorithms require less than one second to compute their schedules, with the exception of the Balanced algorithms which require a significantly higher time, up to 10 seconds. The AREALIST schedules are computed very quickly, but they require as input

the assignment computed from the mixed area bound, whose computation time is much higher and is counted separately in our experiments. The AREALIST algorithms have thus a much lower scalability than HETEROPRIO.

8 | CONCLUSION

We consider HETEROPRIO, an affinity based scheduling algorithm for both independent tasks and task graphs on heterogeneous and unrelated platforms consisting of CPUs and GPUs. The design of fast scheduling algorithm has a strong practical importance for the performance of task-based runtime systems, which are used nowadays to run high performance applications on nodes made of multicores and GPU accelerators. Indeed, in this case, the on-line scheduler is itself on the critical path of the application, and the problem consist in deciding, when a resource becomes idle, which ready task it should process. HETEROPRIO has been proposed in a practical context, and we provide in this paper theoretical worst-case approximation guarantees for independent tasks in all cases, and we prove that (almost all) the bounds are tight. In the case of task graphs, we are able to prove a worst-case $(n + m)$ approximation proof for HeteroPrioDep, an version of HETEROPRIO adapted to the very difficult context of unrelated resources and general task graphs (but with only 2 types of resources).

We also compare experimentally HETEROPRIO against state-of-the-art algorithms from the literature (both low and high complexity algorithms). We prove that the original, generic version HeteroPrio is the only algorithm among the candidates that is in the best category for both low computation time and high quality schedules, thus making it a very good candidate for inclusion in runtime systems.

This work opens several perspectives. First, it proves that despite the complexity of scheduling task graphs on unrelated resources, the problem becomes much easier when considering very few classes of processing resource types (which is a reasonable assumption in practice), thus opening the door to the design of other low cost approximation algorithms. It could be interesting to improve the approximation ratio for the case of dependent tasks, and to consider adding communication times to the problem formulation. Finally, it is well known that injecting some static knowledge about the task graph can improve the performance of the schedule by avoiding bad cases. The simplicity of HETEROPRIO makes it a good candidate for the design of such mixed (both static and dynamic) strategies.

References

[1] Peter Brucker, Sigrid Knust. Complexity results for scheduling problems Web document, URL: http://www2.informatik.uni-osnabrueck.de/knust/class/.

[2] Jan Karel Lenstra, David B Shmoys, Éva Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical programming.* 1990;.

[3] Raphael Bleuse, Safia Kedad-Sidhoum, Florence Monna, Grégory Mounié, Denis Trystram. Scheduling Independent Tasks on Multi-cores with GPU Accelerators. *Concurr. Comput. : Pract. Exper.*. 2015;27(6):1625–1638.

[4] Cédric Augonnet, Samuel Thibault, Raymond Namyst, Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009.* 2011;23:187–198.

[5] Judit Planas, Rosa M Badia, Eduard Ayguadé, Jesus Labarta. Hierarchical task-based programming with StarSs. *International Journal of High Performance Computing Applications.* 2009;23(3):284–299.

[6] Ernie Chan, Field G. Van Zee, Paolo Bientinesi, Enrique S. Quintana-Orti, Gregorio Quintana-Orti, Robert Geijn. SuperMatrix: A multithreaded runtime scheduling system for algorithms-by-blocks. In: *13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, :123–132; 2008.

[7] A. YarKhan, J. Kurzak, J. Dongarra. QUARK Users' Guide: QUeueing And Runtime for Kernels. UTK ICL2011.

[8] Everton Hermann, Bruno Raffin, François Faure, Thierry Gautier, Jérémie Allard. Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations. In: *Euro-Par (2)*, :235-246; 2010.

[9] George Bosilca, Aurélien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, Jack Dongarra. PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability. *Computing in Science and Engineering.* 2013;.

[10] Vincenzo Bonifaci, Andreas Wiese. Scheduling Unrelated Machines of Few Different Types. *CoRR.* 2012;abs/1205.0974.

[11] Haluk Topcuouglu, Salim Hariri, Min-you Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Trans. Parallel Distrib. Syst.*. 2002;13(3):260–274.

[12] Louis-Claude Canon, Loris Marchal, Frédéric Vivien. Low-Cost Approximation Algorithms for Scheduling Independent Tasks on Hybrid Platforms. In: *European Conference on Parallel Processing*, :232–244Springer; 2017.

[13] Emmanuel Agullo, Olivier Beaumont, Lionel Eyraud-Dubois, Suraj Kumar. Are Static Schedules so Bad? A Case Study on Cholesky Factorization. In: *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*, :1021–1030; 2016.

[14] Nathanaël Cheriere, Erik Saule. Considerations on distributed load balancing for fully heterogeneous machines: Two particular cases. In: *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, :6–16IEEE; 2015.

[15] Olivier Beaumont, Lionel Eyraud-Dubois, Suraj Kumar. Approximation Proofs of a Fast and Efficient List Scheduling Algorithm for Task-Based Runtime Systems on Multicores and GPUs. In: *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, ; 2017.

[16] Suraj Kumar. Scheduling of Dense Linear Algebra Kernels on Heterogeneous Resources. PhD thesisUniversité de Bordeaux2017.

[17] Emmanuel Agullo, Berenger Bramas, Olivier Coulaud, Eric Darve, Matthias Messner, Toru Takahashi. Task-based FMM for heterogeneous architectures. *Concurrency and Computation: Practice and Experience.* 2016;28(9).

[18] Eugene L Lawler, Jacques Labetoulle. On preemptive scheduling of unrelated parallel processors by linear programming. *Journal of the ACM (JACM).* 1978;25(4):612–619.

[19] Evgeny V. Shchepin, Nodari Vakhania. An optimal rounding gives a better approximation for scheduling unrelated machines. *Operations Research Letters.* 2005;.

[20] Csanád Imreh. Scheduling Problems on Two Sets of Identical Machines. *Computing.* 2003;70(4):277–294.

[21] Lin Chen, Deshi Ye, Guochuan Zhang. Online Scheduling of mixed CPU-GPU jobs. *International Journal of Foundations of Computer Science.* 2014;25(06):745–761.

[22] Raphaël Bleuse, Thierry Gautier, João V. F. Lima, Grégory Mounié, Denis Trystram. Scheduling Data Flow Program in XKaapi: A New Affinity Based Algorithm for Heterogeneous Architectures:560–571. Cham: Springer International Publishing 2014.

[23] Fabián A Chudak, David B Shmoys. Approximation algorithms for precedence-constrained scheduling problems on parallel machines that run at different speeds. *Journal of Algorithms.* 1999;30(2):323–343.

[24] Chandra Chekuri, Michael Bender. An efficient approximation algorithm for minimizing makespan on uniformly related machines. *Lecture Notes in Computer Science.* 1998;1412:383–393.

[25] Gerhard J Woeginger. A comment on scheduling on uniform machines under chain-type precedence constraints. *Operations Research Letters.* 2000;26(3):107–109.

[26] Safia Kedad-Sidhoum, Florence Monna, Denis Trystram. Scheduling tasks with precedence constraints on hybrid multi-core machines. In: *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, :27–33IEEE; 2015.

[27] Marcos Amaris, Giorgio Lucarelli, Clément Mommessin, Denis Trystram. Generic algorithms for scheduling applications on hybrid multi-core machines. In: *European Conference on Parallel Processing*, :220–231Springer; 2017.

[28] Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics.* 1969;17(2):416–429.

[29] *Chameleon, A dense linear algebra software for heterogeneous architectures.* https://project.inria.fr/chameleon; 2014.

[30] *Experimental repository for the present paper.* https://gitlab.inria.fr/eyrauddu/RAPD17-experiments; 2017.