Analysis of a List Scheduling Algorithm for Task Graphs on Two Types of Resources

Lionel Eyraud-Dubois, Suraj Kumar

► To cite this version:

Lionel Eyraud-Dubois, Suraj Kumar. Analysis of a List Scheduling Algorithm for Task Graphs on Two Types of Resources. IPDPS 2020 - 34th IEEE International Parallel and Distributed Procesing Symposium, May 2020, New Orleans / Virtual, United States. hal-02431810

HAL Id: hal-02431810
https://inria.hal.science/hal-02431810v1

Submitted on 8 Jan 2020

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

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

Analysis of a List Scheduling Algorithm for Task Graphs on Two Types of Resources

Lionel Eyraud-Dubois
Inria Bordeaux – Sud-Ouest
Université de Bordeaux, France
Email: lionel.eyraud-dubois@inria.fr

Suraj Kumar
Inria Paris, France
Email: suraj.kumar@inria.fr

***Abstract*—We consider the problem of scheduling task graphs on two types of unrelated resources, which arises in the context of task-based runtime systems on modern platforms containing CPUs and GPUs. In this paper, we focus on an algorithm named HeteroPrio, which was originally introduced as an efficient heuristic for a particular application. HeteroPrio is an adaptation of the well known list scheduling algorithm, in which the tasks are picked by the resources in the order of their acceleration factor. This algorithm is augmented with a spoliation mechanism: a task assigned by the list algorithm can later on be reassigned to a different resource if it allows to finish this task earlier.**

We propose here the first theoretical analysis of the HeteroPrio algorithm in the presence of dependencies. More specifically, if the platform contains m and n processors of each type, we show that the worst-case approximation ratio of HeteroPrio is between $1 + \max(\frac{m}{n}, \frac{n}{m})$ and $2 + \max(\frac{m}{n}, \frac{n}{m})$. Our proof structure allows to precisely identify the necessary conditions on the spoliation strategy to obtain such a guarantee. We also present an in-depth experimental analysis, comparing several such spoliation strategies, and comparing HeteroPrio with other algorithms from the literature. Although the worst case analysis shows the possibility of pathological behavior, HeteroPrio is able to produce, in very reasonable time, schedules of significantly better quality.

***Index Terms*—Unrelated Resources, Scheduling, Task Graphs, Approximation Proofs, Linear Algebra**

I. Introduction

In recent years, the HPC architectures have seen an increase in intra-node heterogeneity, with a wide adoption of GPU computing, and more generally of fast accelerators dedicated to specific types of computation. The current Top500 list [1] exhibits that 27% of systems use accelerators and 42% of overall performance is produced by accelerators. Exploiting the full potential of such hybrid platforms is challenging for several reasons. First, each architecture has different characteristics and often its own interface. Therefore, application developers encounter a steep learning curve to keep up with architectural changes, and applications often need to be redesigned to efficiently use all the available resources. Second, it is challenging to obtain a precise execution model for computation times and data transfer times due to shared buses, caches and parallel resources. Third, scheduling is a well known NP-complete optimization problem and hybrid resources make this problem harder. We point the reader to [2] for a study on the complexity of scheduling problems and to [3] for a recent study in the case of hybrid platforms. All these observations led to the development of different task based runtime systems. Among several runtimes, we may cite StarPU [4] from Inria Bordeaux (France), Legion [5] from Stanford Univ (USA), QUARK [6] and PaRSEC [7] from Univ. of Tennessee Knoxville (USA), StarSs [8] from Barcelona Supercomputing Center (Spain). In these frameworks, the application is expressed as a task graph, where vertices represent tasks to be executed and edges represent dependencies among them. Dependencies between tasks are either explicitly provided or implicitly discovered by the runtime system. During execution, the runtime system automatically handles synchronization and data movement to ensure a correct execution. Each framework also contains a scheduling component, whose aim is to automatically compute an efficient allocation of tasks to computing resources, taking into account both the heterogeneity of the platform and the task dependencies.

In this context, HeteroPrio has been proposed as a runtime scheduler based on the affinity of tasks and resources. It was initially used to schedule a large set of small independent tasks on their favorable resources during certain phases of the execution in Fast Multipole Methods (FMM) [9]. Later, several corrections are introduced in HeteroPrio to produce efficient schedules for a set of independent tasks on two types of resources [10]. Theoretical performance guarantees and worst case behavior of this scheduler for the specific case of independent tasks are studied in [11], [12]. In this paper, we extend HeteroPrio for task graphs (tasks with precedence constraints), and call it HeteroPrioDep throughout the text. We provide the first theoretical guarantee for any version of HeteroPrio with precedence constraints, and we prove that our guarantees are tight. Specifically, we show that the worst-case approximation ratio of HeteroPrioDep is between $1 + \max(\frac{m}{n}, \frac{n}{m})$ and $2 + \max(\frac{m}{n}, \frac{n}{m})$ on a platform of m CPUs and n GPUs. In particular, for the special case of an equal number of CPUs and GPUs, the approximation ratio of HeteroPrioDep is at most 3.

We also perform an in-depth comparison of HeteroPrioDep with sate-of-the-art schedulers for different linear algebra kernels, namely Cholesky, QR and LU factorizations, and synthetic fork-join applications. Our experiments show that the schedules produced by HeteroPrioDep are significantly more efficient than other scheduling algorithms, and their makespans are very close to the lower bounds.

The outline of the paper is the following. Section II describes previous work on scheduling a set of independent

tasks and task graphs on unrelated resources. In Section III, we first present our model and different notations required to express our problem, and then propose the HeteroPrioDep algorithm for task graphs. The approximation guarantee of HeteroPrioDep and its tightness proof are presented in sections IV and V respectively. In Section VI, we describe our experimental setup and perform an assessment of several spoliation strategies of HeteroPrioDep, as well as a comparison of HeteroPrioDep with state-of-the-art schedulers. We finally propose conclusions and perspectives in Section VII.

II. Related Work

There has been a lot of work on the scheduling a set of independent tasks on unrelated resources. Lenstra et al. presented a PTAS when the number of machines is fixed, as well as a 2-approximation scheduling algorithm obtained by rounding of the solution of the linear program [13]. The specific context of two types of resources has received added interest with the advent of GPU accelerators. Imreh [14] has proposed a linear time complexity algorithm for the online case, based on comparing the ratio of task processing times on both resources, with an approximation ratio of at most 4. Chen et al. have improved on this idea to obtain a 3.85 approximation ratio [15] for the online case as well. Bleuse et al. [16] have proposed offline algorithms with different approximation ratios, $\frac{4}{3}$ and $\frac{3}{2}$, based on dynamic programming techniques. As mentioned in the Introduction, the HeteroPrio algorithm was initially proposed in the context of FMM, in a specific phase where a large number of relatively short tasks are available [9]. This algorithm, based on affinity between tasks and resources, obtained very good experimental results for this application. Later, Beaumont et al. [11] introduced several modifications to this algorithm to improve it, but still for a set of independent tasks. The approximation ratio of this improved algorithm is 1.62 for a special case of 1 CPU and 1 GPU, and 3.41 for the general case. This algorithm also has been extended to handle multiple resources in [17], but without any theoretical performance guarantee.

In the context of task graphs, the well-known Graham's list scheduling algorithm has an approximation ratio of 2 on homogeneous resources [18]. This is the best approximation for homogeneous resources, but this result does not extend to unrelated resources. There is limited work on the scheduling of task graphs on unrelated resources. Most strategies implemented inside runtime systems are variants of the well known HEFT heuristic [19]. In these strategies, tasks are ordered by priority and the highest priority ready task is allocated to the resource on which it is expected to complete first. The task priorities are computed based on their expected distance to the last node. These strategies are generally very effective, but have shown limited success in systems with two very different types of resources. For example, if all tasks of an application are very well accelerated on GPUs, then these strategies often result in underutilization of CPUs, even if a large number of CPUs are available [10]. Kedad-Sidhoum et al. [20] proposed an algorithm with a 6 approximation ratio, which first computes an allocation of each task on CPU and GPU by rounding the solution of a linear program, and then uses list scheduling to schedule tasks on each resource type. Other noteworthy approaches on two types of resources are obtained by improving upon techniques for scheduling a set of independent tasks. Amaris et al. proposed an online $4\sqrt{max(\frac{m}{n},\frac{n}{m})}$-approximation algorithm based on comparison of processing times on both resources [21], which is similar to [14], [15] approaches for independent tasks. This algorithm and its analysis have been recently improved to obtain a $1+2\sqrt{max(\frac{m}{n},\frac{n}{m})}$ ratio [3].

III. Model and Notations

The problem studied in this paper is $P1,P2|prec|C_{\max}$. We consider a platform made of m CPUs and n GPUs, and a set $\mathcal{T}$ of tasks to be scheduled on this platform. Given a task $T_i \in \mathcal{T}$, we denote by p_i and q_i the processing times of task T_i on CPU and GPU respectively. The acceleration factor of task T_i on GPU is $\frac{p_i}{q_i}$. Tasks in $\mathcal{T}$ are ordered by precedence relations: $T_i \rightarrow T_j$ means that task j cannot start before task i has completed.

For a given instance, a valid solution provides an allocation $x_i \in \{0,1\}$ (x_i is 1 if T_i is processed on CPU, and 0 otherwise) and a schedule which specifies a starting time S_i and end time C_i for each task T_i, where $C_i = S_i + p_i x_i + q_i(1-x_i)$. This schedule needs to satisfy the precedence constraints, and must not use more than m CPUs and n GPUs at any time. The goal is to find a valid schedule with minimum makespan, *i.e.* the largest completion time among all tasks, $C_{\max} = \max_i C_i$.

Definition 1 (Favorable resource)**.** *If the processing time of task T on resource R is $\min(p_T, q_T)$, then we say that R is a* favorable *resource for this task, otherwise this resource is called* unfavorable *for this task.*

Note that if $p_T = q_T$, then both resources are favorable for task T. On the other hand, at most one resource can be unfavorable for any task T.

A. Area Bound

Let I be a set of independent tasks. Any fraction x_i of task T_i is allowed to be processed on CPU (resp. GPU) and this fraction overall consumes $x_i p_i$ (resp. $x_i q_i$) time units of CPU (resp. GPU) resource. The lower bound $Area(I)$ for a set of tasks I on m CPUs and n GPUs is the solution of the following linear program.

Minimize $Area(\mathcal{I})$ such that

$$\sum_{i\in I} x_i p_i \leq m Area(\mathcal{I}) \quad (1)$$

$$\sum_{i\in I} (1-x_i) q_i \leq n Area(\mathcal{I}) \quad (2)$$

$$0 \leq x_i \leq 1$$

Let $C^{\mathrm{Opt}}_{\max}(\mathcal{I})$ be the optimal makespan for a set of tasks I. Any valid schedule of I on m CPUs and n GPUs can be expressed as the solution of the above linear program. Hence, $Area(\mathcal{I}) \leq C^{\mathrm{Opt}}_{\max}(\mathcal{I})$.

Algorithm 1 The HeteroPrioDep Algorithm for a task graph.

1: Sort ready tasks in Queue Q by non-increasing acceleration factors
2: Create an empty Queue of running tasks on their unfavorable resource Q' sorted by **criterion X**
3: **while** there exists an unprocessed task **do**
4: **if** all workers are busy **then**
5: continue
6: **end if**
7: **if** a task completes **then**
8: Update Q and Q'.
9: **end if**
10: Select an idle worker W
11: **if** $Q \neq \emptyset$ **then**
12: Pick a task T from the beginning (resp. the end) of Q if W is a GPU (resp. CPU) worker.
13: **end if**
14: **if** $Q' \neq \emptyset$ and (T was not found or W is not favorable for T) **then**
15: Consider tasks from Q' running on a worker of the other type
16: Until finding a task T whose completion time would be smaller if restarted on W.
17: **end if**
18: **if** some task T was found **then**
19: **if** W is unfavorable for T, and a worker W' of the other type is idle **then**
20: Process task T on W'
21: **else**
22: Process task T on W (potentially using spoliation).
23: **end if**
24: Update Q and Q'.
25: **end if**
26: **end while**

Lemma 1. $\sum_{i\in I} \min(p_i, q_i) \leq (m+n)Area(\mathcal{I})$

Proof. Consider an optimal solution x^*. It is clear that $\forall i, \min(p_i, q_i) \leq x_i^* p_i + (1 - x_i^*)q_i$. The result comes from summing both inequalities (1) and (2). □

Lemma 2. *Both constraints of Area are tight. i.e.,* $Area(\mathcal{I}) = \frac{1}{m}\sum_{i\in I} x_i p_i = \frac{1}{n}\sum_{i\in I}(1 - x_i)q_i$.

This lemma can be proven by contradiction and simple exchange arguments. An exact proof is available in [12].

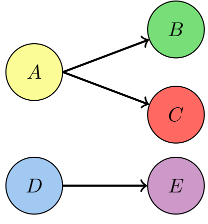
Let G be a DAG and IG is the set of its tasks. Let $C_{\max}^{\text{Opt}}(IG)$ represent the optimal makespan on m CPUs and n GPUs when precedence constraints are relaxed, and $C_{\max}^{\text{Opt}}(G)$ is the optimal makespan with precedence constraints. It is immediate that $Area(IG) \leq C_{\max}^{\text{Opt}}(IG) \leq C_{\max}^{\text{Opt}}(G)$.

B. Description of the HeteroPrio algorithm

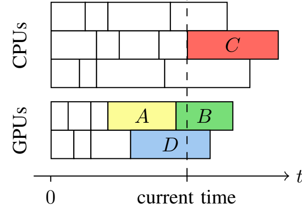
As mentioned earlier, the HeteroPrio algorithm was introduced in the context of FMM [9], which contains a large set of small independent tasks. This algorithm is based from the following fact: the optimal solution of the area bound is obtained when all the tasks assigned to GPUs have higher acceleration factors than the tasks assigned to CPUs. The first version of HeteroPrio is a list algorithm: whenever a resource is available, the most favorable ready task is selected to run on this resource. This works well with short tasks, but assigning a task with a long processing time to the wrong resource can lead to an arbitrarily large makespan. For this reason, it was later proposed to extend HeteroPrio with a *spoliation* mechanism: if there is no ready task, an idle resource can stop a task currently running on another resource and restart it from the beginning, if this allows to finish it earlier. When several spoliation candidates are available, choosing the one with the highest completion time allows to obtain a 3.41 approximation ratio for any set of independent tasks [11].

In this paper, we propose a slightly modified version of HeteroPrio, adapted to the case of task graphs, which we call HeteroPrioDep and describe in Algorithm 1. This version allows an idle resource to perform spoliation if there is no ready task for which it is a favorable resource (in HeteroPrio, spoliation only occurs if there is no ready task of any kind). Another difference is that HeteroPrioDep does not specify the criterion for choosing which task to spoliate when several are available (line 2): all proofs in this paper apply no matter which criterion is used. However, the practical performance of HeteroPrioDep depends on this criterion. In Section VI, we compare three possible choices: latest completion time, highest priority, most favorable. In most cases, highest priority is often the better choice.

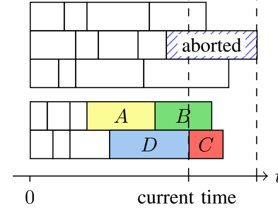
The queues used in Algorithm 1 can be implemented as heaps. Therefore, time complexity of Algorithm 1 is $\mathcal{O}(N \log N + E)$, where N is the number of tasks and E is the number of dependencies in the task graph. An example of the execution of HeteroPrioDep is illustrated in Figure 1.



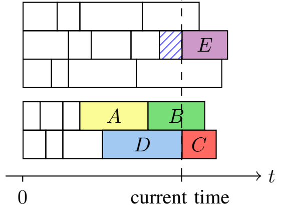
(a) Precedence constraints.



(b) HeteroPrioDep schedule when task C is first scheduled.



(c) HeteroPrioDep schedule after spoliation of task C.



(d) HeteroPrioDep schedule with first response of task E.

Fig. 1: An example of HeteroPrioDep algorithm. Tasks are such that $\frac{p_A}{q_A} > \frac{p_B}{q_B} > \frac{p_C}{q_C} > 1 > \frac{p_E}{q_E}$. (1b) The processing of tasks B and C first starts on GPU and CPU respectively. (1c) C is spoliated by a GPU (E is ready, but not favorable for the GPU). (1d) Task E is scheduled on the CPU left free by the spoliation.

Note on spoliation. In most runtime systems, aborting a task to restart it on another resource is not supported. Some promising approaches have been proposed to perform forward recovery of failed tasks in the context of resilience [22], but efficiently aborting and restarting tasks is a challenging feature. In this paper, we do not assume that spoliation is supported by the runtime system. Instead, HeteroPrio and HeteroPrioDep are seen as *offline* algorithms: the first phase computes a simulated schedule in which spoliation may occur, and the second phase implements this schedule on the runtime system. In the second phase, aborted parts of spoliated tasks are just seen as idle time for the resources.

IV. Approximation Proof of HeteroPrioDep

We consider the schedule produced by HeteroPrioDep on graph G.

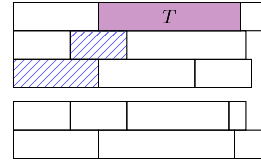
Definition 2 (First Response Time)**.** *For any task $T \in IG$, its First Response Time is the time at which task T starts its first processing in the HeteroPrioDep schedule. It is denoted by FRT_T.*

Note that FRT_T is not necessarily the start time of T in the final schedule if T is later spoliated.

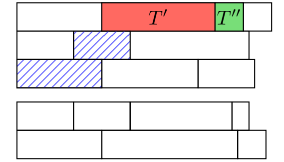
Let S be the set of tasks which complete on their unfavorable resources in the HeteroPrioDep schedule. For all $T \in S$, we construct two new tasks T' and T'' with the following timings. If processing of T completes on resource R, then the duration of T' on R is $max(p_T, q_T) - \min(p_T, q_T)$ and its duration on the other resource is 0. The duration of T'' on both resources is $\min(p_T, q_T)$. We denote by IG' the set of tasks obtained by replacing each task T of S by the associated tasks T' and T'': $IG' = (IG - S) \cup (\cup_{T \in S}\{T, T''\})$. The following result is direct:

Lemma 3. $Area(IG') \leq Area(IG)$.

Proof. Let $(x_i)_{i \in IG}$ be any solution to the area bound linear program for IG. We consider the solution $(y_i)_{i \in IG'}$ obtained by setting $y_{T'}$ and $y_{T''}$ equal to x_T for any $T \in S$, and $y_T = x_T$ for any $T \notin S$.



(a) Task T completes on its unfavorable resource.



(b) T is replaced with T' and T''.

Fig. 2: Task T, which completes on its unfavorable resource, is replaced with T' and T''. Aborted tasks are shown in pattern boxes.

Consider any task $T \in S$ whose unfavorable resource is the CPU ($\max(p_T, q_T) = p_T$). Then, since $p_{T''} = q_T$:

$$\begin{aligned} y_{T'}p_{T'} + y_{T''}p_{T''} &= y_{T'}(p_T - q_T) + y_{T''}q_T \\ &= x_T p_T \end{aligned}$$

And since $q_{T'} = 0$ and $q_{T''} = q_T$:

$$(1 - y_{T'})q_{T'} + (1 - y_{T''})q_{T''} = (1 - x_T)q_T$$

The same result applies symmetrically for any task $T \in S$ whose unfavorable resource is the GPU. This shows that $(y_i)_{i \in IG'}$ is a solution to the area bound linear program for IG' with the same cost. Since this is true for any solution x, in particular for the optimal solution of $Area(IG)$, we conclude that the cost of the optimal solution $Area(IG')$ is at most $Area(IG)$. □

Now we replace each task T of S with T' and T'' on the same resource in the HeteroPrioDep schedule in the following way. Remember that since task T completes on its unfavorable resource, it is not spoliated in this schedule. C_T represents the completion time of task T in the schedule.

$$\mathrm{FRT}_{T'} = \mathrm{FRT}_T \qquad C_{T'} = C_T - \min(p_T, q_T)$$
$$FRT_{T''} = C_{T'} \qquad C_{T''} = C_T$$

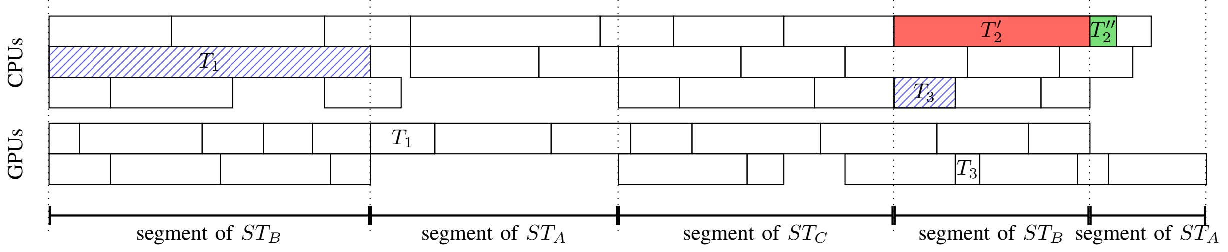


Fig. 3: Different segments of a modified HeteroPrioDep schedule. Aborted tasks are depicted in pattern boxes. GPU is the favorable resource for tasks T_1, T_2 and T_3. T_2 completes on its unfavorable resource in the original HeteroPrioDep schedule, hence it is replaced with T_2' and T_2'' in the modified schedule.

It is immediate that the above replacement procedure does not change the makespan of the original schedule. Figure 2 shows how a task is replaced with two newly constructed types of tasks in the schedule.

Lemma 4. *In the modified schedule, if a resource has a spoliated task or a task of type T' during time interval $[t_1, t_2]$, then the other resource is favorable for all tasks it processes during this interval.*

Proof. Without loss of generality, let us assume that a CPU has a spoliated task T_k or a T_k' task (whose original task is T_k) in $[t_1, t_2]$. Our goal is to prove that the GPU is a favorable resource for all tasks processed on GPU in this interval. This is equivalent to proving that no GPU processes a spoliated task or a task of type T'.

It is immediate that $t_1 \geq \text{FRT}_{T_k}$ and $t_2 \leq C_{T_k} - \min(p_k, q_k)$. Since T_k is spoliated or T_k' exists, we know that the CPU is the unfavorable resource for T_k. Now let us assume that the GPU processes a spoliated task T_s or a task T_s' (whose original task is T_s) in this duration. It indicates that T_s received its first response on a GPU; furthermore $\text{FRT}_{T_s} > \text{FRT}_{T_k}$, otherwise the CPU would have spoliated T_s before starting T_k. However, since $t_2 \leq C_{T_k} - \min(p_k, q_k)$, T_k is available for spoliation during the interval $[t_1, t_2)$. This implies that T_s cannot start in this interval, otherwise T_k would have been spoliated instead of starting T_s. We have a contradiction, so the GPU is a favorable resource for all tasks it processes during $[t_1, t_2]$. □

We now divide the modified schedule into different types of segments.

- ST_A: Segments where at least one worker is idle on both types of resources
- ST_B: Segments where at least one type of resource is fully busy and at least one task runs on its unfavorable resource (either a spoliated or a task of type T')
- ST_C: Segments where at least one type of resource is fully busy, and all tasks are on their favorable resources

These segment types are depicted on Figure 3. Let $C^{\text{HP}}_{\max}(G)$ denote the makespan of HeteroPrioDep schedule for task graph G, and $|ST_X|$ denote the sum of the durations of segments of type X. From the construction, $|ST_A| + |ST_B| + |ST_C| = C^{\text{HP}}_{\max}(G)$.

Now we try to bound $|ST_B| + |ST_C|$ by $max(1/m, 1/n) \sum_{T \in IG'} \min(p_T, q_T)$ and $|ST_A|$ by $C^{\text{Opt}}_{\max}(G)$. More precisely, we prove the Lemmas 5 and 6.

Lemma 5. $|ST_B| + |ST_C| \leq (1 + max(m/n, n/m)) C^{\text{Opt}}_{\max}(G)$.

Proof. For any interval I, we denote by x_T^I the fraction of non-aborted task T processed during this interval.

Let us consider one segment $I = [t_1, t_2]$ of ST_B. Without loss of generality, let us assume that a spoliated task or a task of type T' runs on a CPU. This implies that all the GPUs are busy (otherwise, this task would be spoliated to the GPU and this segment would have length 0). From Lemma 4, all GPUs are busy with favorable tasks in this interval. Hence, the total GPU Area of fractions of tasks processed on GPU in this segment is equal to $n|I|$. We thus obtain the following inequality:

$$|I| \leq \max\left(\frac{1}{m}, \frac{1}{n}\right) \left(\sum_T \min(p_T, q_T) * x_T^I\right)$$

The same inequality holds if the unfavorable task is processed on a GPU.

Similarly, let us look at the one segment I of ST_C. Without loss of generality, we can assume that all GPUs are busy; since $I \in ST_C$, they are busy with favorable tasks. Just like above, the total GPU Area of fractions of tasks completed on GPU in this segment is equal to $n|I|$, and we get:

$$|I| \leq \max\left(\frac{1}{m}, \frac{1}{n}\right) \left(\sum_T (\min(p_T, q_T) * x_T^I\right)$$

Since segments from ST_B and ST_C are non-overlapping, the total sum of fractions of tasks is at most 1: $\forall T, \sum_{I \in ST_B \cup ST_C} x_T^I \leq 1$. Since Lemma 1 states that $\sum_{T \in IG'} \min(p_T, q_T) \leq (m+n) Area(IG')$, we conclude:

$$|ST_B| + |ST_C| \leq \max\left(\frac{1}{m}, \frac{1}{n}\right) \left(\sum_{T \in IG'} \min(p_T, q_T)\right)$$

$$\leq \max\left(\frac{1}{m}, \frac{1}{n}\right)(m+n)Area(IG')$$
$$= \left(1+\max\left(\frac{m}{n}, \frac{n}{m}\right)\right) Area(IG')$$
$$\leq \left(1+\max\left(\frac{m}{n}, \frac{n}{m}\right)\right) Area(IG)$$
$$\leq \left(1+\max\left(\frac{m}{n}, \frac{n}{m}\right)\right) C^{\mathrm{Opt}}_{\max}(G)$$

□

Lemma 6. $|ST_A| \leq C^{\mathrm{Opt}}_{\max}(G)$.

Proof. We look at the segments of ST_A. At least one worker is idle on both types of resources, hence these segments contain no spoliated task, and no task of type T' (otherwise such tasks would have been spoliated by an idle worker of the other resource). We find a set of tasks which covers all ST_A segments, by going through the HeteroPrioDep schedule backward. We start with a task T_k of ST_A segments which finishes last. If T_k does not completely cover the last ST_A segment, then we look for another task T_{k-1} in the same segment on which T_k depends (there has to be one, otherwise T_k would have been started sooner). If T_k completely covers this segment, we look for another task on which T_k directly or indirectly depends in the preceding ST_A segment. If there are several possible such tasks, we select the one whose completion time is the latest: it can not end before the end of the segment (otherwise task T_k or one of its predecessors would have started immediately). This ensures that when the above procedure of selecting tasks finishes, all corresponding ST_A segments are covered by one or several tasks. Let $IPO = \{T_1, T_2, \cdots, T_k\}$ be the set of tasks obtained in this way.

Since the ST_A segments contain no spoliated task and no task of type T', the maximum contribution of any task T_i for all corresponding ST_A segments is at most $\min(p_i, q_i)$.

$$|ST_A| \leq \sum_{i \in IPO} \min(p_i, q_i)$$
$$\leq \max_{P \in Path(G)} \left(\sum_{i \in P} \min(p_i, q_i)\right)$$
$$\leq C^{\mathrm{Opt}}_{\max}(G)$$

□

Theorem 1. *The approximation ratio of HeteroPrioDep on m CPUs and n GPUs is at most $2 + \max\left(\frac{m}{n}, \frac{n}{m}\right)$.*

Proof. By construction, $C^{\mathrm{HP}}_{\max}(G) = |ST_A| + |ST_B| + |ST_C|$.

Using Lemmas 5 and 6:

$$C^{\mathrm{HP}}_{\max}(G) \leq \left(1+\max(\frac{m}{n}, \frac{n}{m})\right) C^{\mathrm{Opt}}_{\max}(G) + C^{\mathrm{Opt}}_{\max}(G)$$
$$= \left(2+\max(\frac{m}{n}, \frac{n}{m})\right) C^{\mathrm{Opt}}_{\max}(G)$$

□

Corollary 1. *The approximation ratio of HeteroPrioDep when the number of CPUs equals to the number of GPUs is at most 3.*

V. Tightness Result

Finally, we also prove a tightness result, showing that a constant ratio independent of m and n is not achievable.

Theorem 2. *The approximation ratio of HeteroPrioDep is at least $1 + \max\left(\frac{m}{n}, \frac{n}{m}\right)$.*

Proof. Without loss of generality, let us assume that $m \geq n$ and consider the task graph G of Fig 4 on m CPUs and n GPUs, where m is a multiple of n. The graph is a succession of $1 + \frac{m}{n}$ repetitions of the same pattern, and each pattern contains one task of each type A, T_1, T_2, and n tasks of type B. The processing times of each type of tasks are given in the following table, where ϵ is an arbitrary positive value.

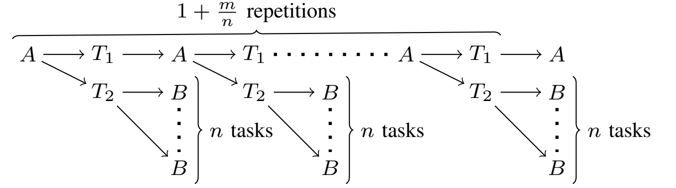


Fig. 4: A task graph for which HeteroPrioDep does not achieve a constant approximation ratio.

Task Name	#	GPU time	CPU time
A	$\frac{m}{n}+2$	2ϵ	$1-3\epsilon$
T_1	$\frac{m}{n}+1$	5ϵ	4ϵ
T_2	$\frac{m}{n}+1$	2ϵ	3ϵ
B	$m+n$	$1-\epsilon$	1

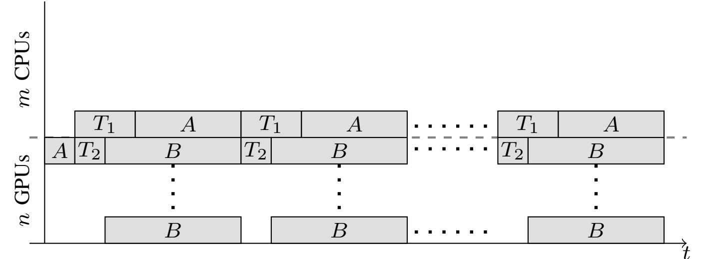


Fig. 5: A possible HeteroPrioDep schedule for the task graph of Fig 4.

A possible schedule produced by HeteroPrioDep for this instance is shown on Figure 5. Once the first task A is completed, the corresponding T_1 and T_2 tasks run on their favorite resources. When T_2 ends, the only available tasks are n tasks of type B. They are processed on GPUs. When T_1 ends and the next A task becomes available, no GPU is idle. The CPUs cannot spoliate any B task because they would not finish them earlier, so task A runs on a CPU and finishes at the same time as the B tasks. This pattern is repeated $1 + \frac{m}{n}$ times, and this yields a makespan of

$$C^{\mathrm{HP}}_{\max}(G) = (\frac{m}{n}+1)(1+\epsilon) + 2\epsilon.$$

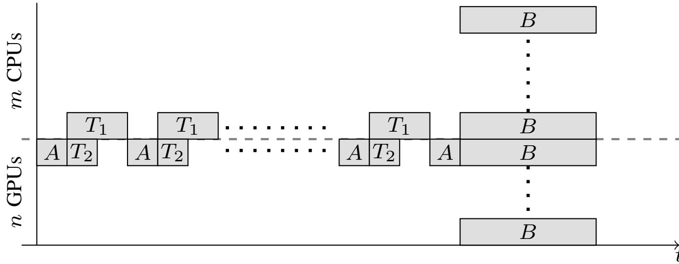


Fig. 6: A better schedule for the task graph of Fig 4.

Another possible schedule is shown on Figure 6, in which all B tasks are delayed until the end of the schedule, allowing the chain of tasks of type A and T_1 to be processed much earlier. This schedule has makespan $2\epsilon + 6\epsilon\left(\frac{m}{n}+1\right)+1$, and it provides a bound on the optimal makespan $C_{\max}^{\mathrm{Opt}}(G)$. Hence,

$$\frac{C_{\max}^{\mathrm{HP}}(G)}{C_{\max}^{\mathrm{Opt}}(G)} \geq \frac{(\frac{m}{n}+1)(1+\epsilon)+2\epsilon}{1+6\epsilon\left(\frac{m}{n}+1\right)+2\epsilon}$$

$$\lim_{\epsilon\to 0}\frac{C_{\max}^{\mathrm{HP}}(G)}{C_{\max}^{\mathrm{Opt}}(G)} \geq \frac{m}{n}+1$$

This implies that the approximation ratio of HeteroPrioDep is at least $1+\frac{m}{n}$. □

VI. Experimental Evaluation

In this section, we present an experimental evaluation to assess the average performance of HeteroPrioDep on realistic task graphs. We start by presenting the task graphs used in this evaluation. Then we propose and evaluate different spoliation orders for HeteroPrioDep. Finally, we compare the performance of HeteroPrioDep with other algorithms from the literature.

A. Task graphs

We consider instances introduced by Beaumont et al. [12] in a previous work on HeteroPrio. In order to obtain a representative mix of different kernels, applications from the Chameleon suite [23] (Cholesky, LU, QR) have been executed on the `sirocco` [1] platform with tile size of 960. Each of these applications consists in many calls to a few linear algebra kernels, which correspond to the individual tasks of the applications. We measure the average execution time of each kernel on each type of computing resource (CPU and GPU), and we use these values as the processing times of the corresponding tasks. The number of (960×960) tiles varies from 4 to 60. These instances are simulated on platforms with 4 GPUs, and a number of CPUs varying from 4 to 60, so as to evaluate how the performance of HeteroPrioDep varies with the ratio $\frac{m}{n}$.

We also consider fork-join instances used to evaluate the HLP algorithm [21]. The fork-join application corresponds to a real situation where the execution starts sequentially and then forks to width parallel tasks. The results are aggregated by performing a join operation, completing a phase. This procedure can be repeated p times, the number of phases. For this benchmark, we set $p = 20$ and $width \in \{100, 200, 300, 400, 500\}$. The processing time of each task on CPU was computed using a Gaussian distribution with center 20 and standard deviation 5. The processing time of a task on GPU is computed through the acceleration factor, in conjunction with the processing time on CPU already computed. The goal is to create a more irregular application than the 3 from Chameleon to study the importance of the allocation decision. For this reason, 5% of the parallel tasks of each phase are highly decelerated when processed on a GPU by choosing uniformly a random acceleration factor for each of them in $[0.1, 0.5]$. For each of the remaining tasks, an acceleration factor is randomly chosen in $[0.5, 50]$, which corresponds to the range of acceleration factors observed for the applications of Chameleon.

[1] https://www.plafrim.fr/

B. Variants of HeteroPrioDep

As we have noticed, the definition of HeteroPrioDep does not specify the ordering used for queue Q': any ordering provides the same worst-case approximation guarantee. In practice however, changing this ordering has an effect on the performance of the algorithm. We propose here three possible variants of HeteroPrioDep, depending on the ordering used for queue Q'.

- **latest** ordering: tasks in Q' are ordered in non-increasing order of their finish time. With this ordering, when the input contains no dependencies, HeteroPrioDep behaves like the HeteroPrio algorithm from [12], with a constant performance guarantee.
- **priority** ordering: tasks in Q' are ordered by their priority, computed as the bottom level in the task graph. This allows to finish high priority tasks as soon as possible.
- **acceleration factor** ordering: tasks in Q' are sorted by their acceleration factor, so that idle workers always pick the most favorable task, either from queue Q or Q'.

The results are provided on Figure 7. For each instance, we also compute a lower bound on the makespan, based on the same linear program as the one used for the HLP algorithm [21]. The plots present the normalized makespan, which is the ratio of the makespan obtained by each algorithm to the lower bound.

We can see that for smaller number of CPUs, all orderings are similar, because with fewer CPUs there are fewer candidates for spoliation, and thus the ordering does not really matter. When the number of CPUs increases, we see that the **priority** ordering outperforms the others for the Linear Algebra instances, but the **latest** ordering is more efficient on the fork-join instances. This can be explained by the fact that the fork-join instances are a sequence of independent tasks: we can not start the next batch until all tasks from the current batch have been finished. Spoliating the task that finishes last allows to finish the current batch faster, as was proven in the context of independent tasks [12].

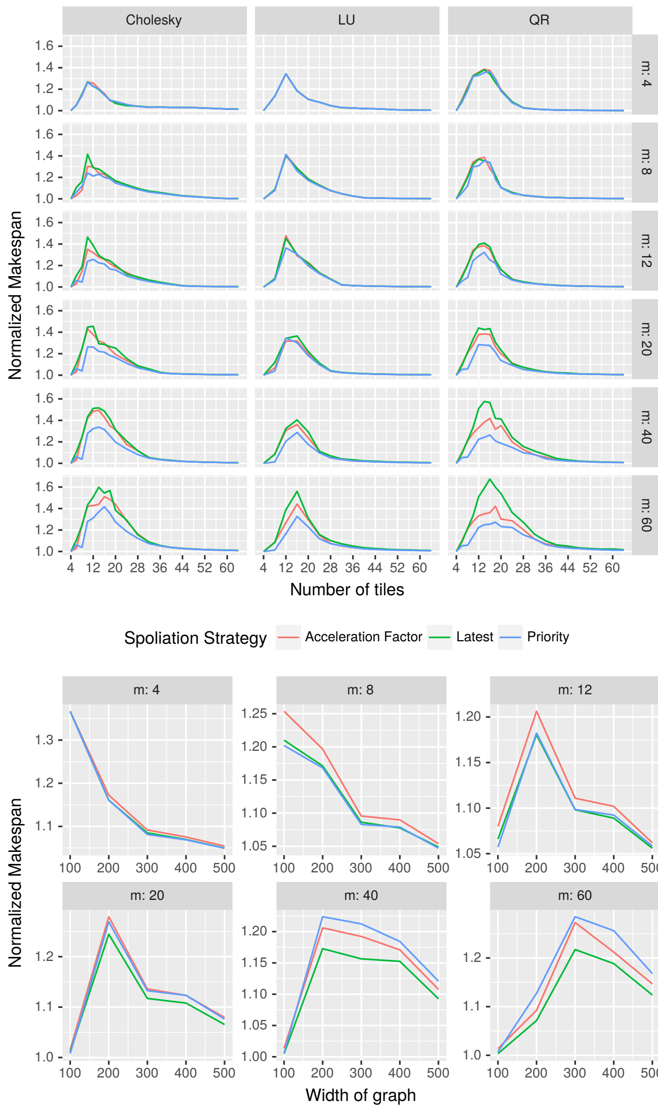


Fig. 7: Comparison of different HeteroPrioDep versions.

We also observe that, unlike what could be expected from the worst case results, the normalized makespan of HeteroPrioDep does not increase when the number of CPUs increases. Since the different variants of HeteroPrioDep have different performance depending on the instance, in the rest of this section we consider the HeteroPrioDep algorithm obtained by keeping the best schedule among all 3 variants.

C. Other algorithms from literature

We perform comparison of HeteroPrioDep with HLP [21], ER-LS [24] , QA [3], HEFT [19] and ECT [3] algorithms from the literature. ER-LS, QA and ECT are online algorithms while HLP and HEFT are offline algorithms.

- HLP: this algorithm determines the allocations of tasks on CPU and GPU by rounding the solution of a linear program. After that, a list scheduling approach is used to schedule tasks on both types of resources. This is a 6-approximation algorithm.
- ER-LS: this is a $4\sqrt{max(\frac{m}{n}, \frac{n}{m})}$-approximation algorithm, with simple decision rules. When task T_i becomes ready, if the CPU time of T_i is greater than its earliest completion time on GPU, then T_i is assigned to GPU. Otherwise, if $\frac{p_i}{\sqrt{m}} \leq \frac{q_i}{\sqrt{n}}$, then T_i is assigned to CPU, else it is assigned to GPU. After this allocation decision is made, T_i gets scheduled on the earliest available resource of the selected type.
- QA: this is a modified version of ER-LS algorithm, which achieves a $1+2\sqrt{max(\frac{m}{n}, \frac{n}{m})}$ approximation guarantee. This algorithm only uses the second decision rule of ER-LS, and a tighter analysis allows to prove a stronger approximation bound.
- HEFT: this is a classical offline scheduling algorithm for heterogeneous resources. In this algorithm, tasks are considered in the order of their bottom level priority (expected processing time on the longest path starting at this task). The highest priority task is scheduled on the resource which finishes it at the earliest time.
- ECT: this is an online version of HEFT algorithm. In this algorithm, tasks are considered in the order in which they become ready and get scheduled on the resources on which they finish the earliest. If several tasks are ready, they are considered in non-increasing order of their priority. This scheduling algorithm is very common in most modern runtime systems (in StarPU, it is implemented as the **dmdas** policy).

D. Comparison with other algorithms

Figure 8 depicts the performance comparison of all considered algorithms for Cholesky, QR, LU and fork-join applications. We also show, for each algorithm and each fixed number of tiles, the average performance over all instance types (Cholesky, QR and LU) and all platform configurations in Figure 9. Like before, these plots show the ratio of the makespan obtained by each algorithm to the lower bound obtained by the linear program.

Since allocation decisions of the ER-LS and QA algorithms are mostly based on comparing the processing times of tasks, and not on the current state of resources or available tasks, they obtain poor performance in most cases. The performance of ER-LS is slightly better than QA for small task graphs, because ER-LS avoids scheduling large tasks on the slow resources in the beginning. For large task graphs, ER-LS and QA both achieve similar performance. HEFT performs better than HLP for small and intermediate task graphs. Indeed, in such cases, the applications are mostly limited by the critical path of the graph, which is the main decision criterion for HEFT. For large task graphs, it is more important to schedule tasks on their favorable resources, which is the main focus of the HLP algorithm, and we can see that HLP outperforms HEFT in these cases. HEFT performs slightly better than ECT for task graphs which have multiple phases of high parallelism followed by synchronization tasks. For other tasks graphs, HEFT and ECT both achieve similar performance. HeteroPrioDep combines the desirable properties of both HLP

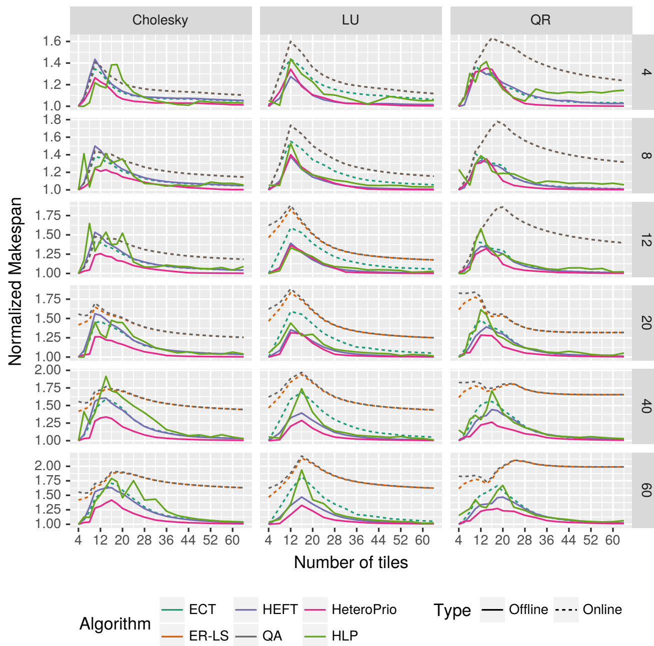


Fig. 8: Comparison of HeteroPrioDep with offline and online algorithms from the literature. Top: Chameleon instances, bottom: fork-join task graphs.

and HEFT algorithms. It schedules tasks on their favorable resources and its spoliation mechanism ensures that good progress is made along the critical path. Hence, it outperforms other algorithms in most cases and its performance is very close to the lower bounds.

Figure 10 depicts the running times of all considered algorithms, also averaged over instance types and platform configurations. The running time of HLP is dominated by the time needed to solve the linear program, and is significantly higher than other algorithms. This running time can be up to 200 seconds for 64 tiles on the Chameleon instances, whereas makepans of the corresponding kernels are between 8 seconds for QR and 25 seconds for LU. By contrast, the running time of HeteroPrioDep is at most 200ms, which is completely acceptable in this context. It should be noted that HeteroPrioDep is actually run three times with each different spoliation strategy to keep the best result, and that the running time shown on this plot is the sum of all three runs.

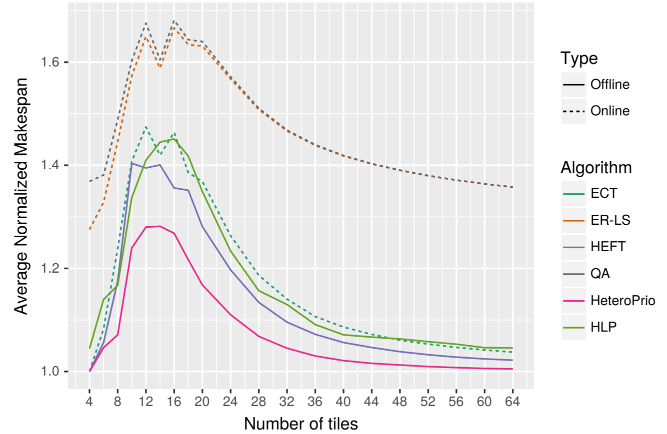


Fig. 9: Average results obtained on the Chameleon instances, over different instance types and number of CPUs.

VII. Conclusion

In this paper, we considered the problem of scheduling tasks with dependencies on two types of unrelated resources. This scheduling problem is very relevant to modern task-based runtime systems for hybrid CPU-GPU platforms, increasingly used by HPC applications. We analyzed an algorithm named HeteroPrio designed for independent tasks. We proposed HeteroPrioDep, an extension of this algorithm for task graphs. HeteroPrioDep schedules tasks on their favorable resources and relies on a recovery mechanism to cope with bad decisions of critical tasks. We presented a theoretical approximation guarantee of $2+\max(\frac{n}{m},\frac{m}{n})$ for this algorithm, with an almost matching lower bound of $1+\max(\frac{n}{m},\frac{m}{n})$. We also presented extensive experimental results which show that the practical performance of HeteroPrioDep is significantly better than previously proposed algorithms, with very low running time, for linear algebra kernels (Cholesky, QR and LU factorizations) and fork-join applications. For further study on HeteroPrio, we believe that the question of locality and data movement is important, and it would be very valuable to understand how to compromise between locality and resource affinity. We also plan to evaluate HeteroPrioDep for task graphs arising in molecular chemistry and machine learning domains.

References

[1] TOP 500 List. URL: https://www.top500.org. Accessed: 2019-10-12.

[2] P. Brucker and S. Knust. Complexity results for scheduling problems. Web document, URL: http://www2.informatik.uni-osnabrueck.de/knust/class/. Accessed: 2019-10-12.

[3] L.-C. Canon, L. Marchal, B. Simon, and F. Vivien, "Online scheduling of task graphs on hybrid platforms," in *Euro-Par 2018: Parallel Processing*, M. Aldinucci, L. Padovani, and M. Torquati, Eds. Cham: Springer International Publishing, 2018, pp. 192–204. [Online]. Available: https://doi.org/10.1007/978-3-319-96983-1_14

[4] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, vol. 23, pp. 187–198, Feb. 2011. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1631

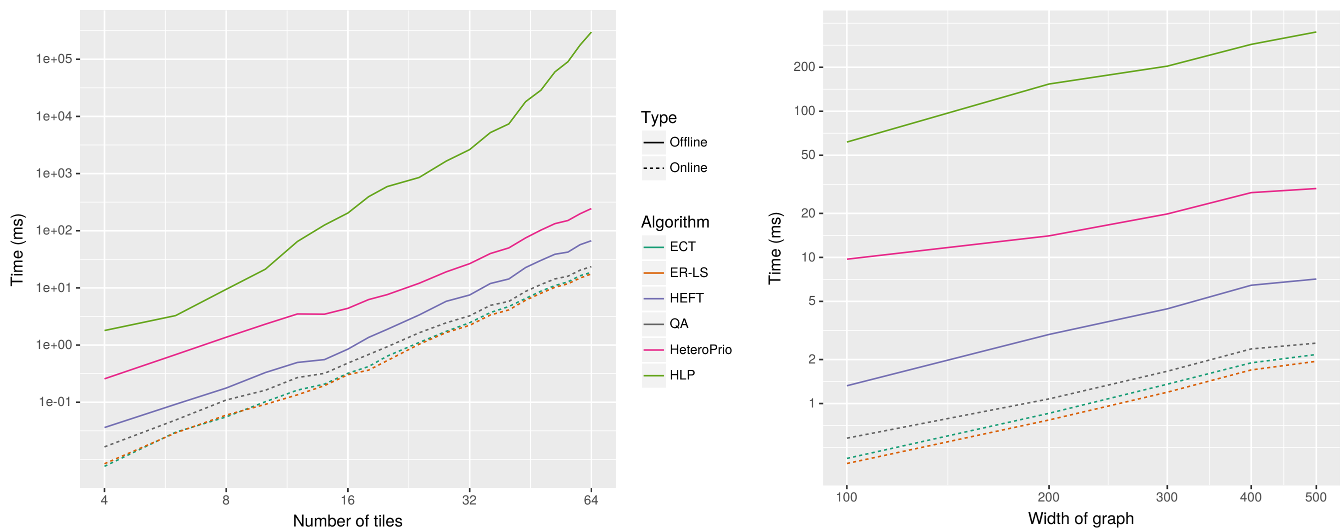


Fig. 10: Running times of all algorithms, averaged over instance types and number of CPUs. Left: Chameleon instances, right: fork-join task graphs.

[5] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, 2012, pp. 66:1–66:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389086

[6] A. YarKhan, J. Kurzak, and J. Dongarra, "QUARK Users' Guide: QUeueing And Runtime for Kernels," *University of Tennessee Innovative Computing Laboratory Technical Report*, no. ICL-UT-11-02, Feb 2011.

[7] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. Dongarra, "PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability," *Computing in Science and Engineering*, 2013. [Online]. Available: https://doi.org/10.1109/MCSE.2013.98

[8] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 2, pp. 173–193, 2011. [Online]. Available: https://doi.org/10.1142/S0129626411000151

[9] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi, "Task-based fmm for heterogeneous architectures," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 9, pp. 2608–2629, 2015. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3723

[10] E. Agullo, O. Beaumont, L. Eyraud-Dubois, and S. Kumar, "Are static schedules so bad? a case study on cholesky factorization," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 1021–1030. [Online]. Available: https://doi.org/10.1109/IPDPS.2016.90

[11] O. Beaumont, L. Eyraud-Dubois, and S. Kumar, "Approximation proofs of a fast and efficient list scheduling algorithm for task-based runtime systems on multicores and gpus," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2017, pp. 768–777. [Online]. Available: https://doi.org/10.1109/IPDPS.2017.71

[12] O. Beaumont, L. Eyraud-Dubois, and S. Kumar, "Fast approximation algorithms for task-based runtime systems," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 17, p. e4502, 2018, e4502 cpe.4502. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4502

[13] J. K. Lenstra, D. B. Shmoys, and É. Tardos, "Approximation algorithms for scheduling unrelated parallel machines," *Mathematical Programming*, vol. 46, no. 1, pp. 259–271, Jan 1990. [Online]. Available: https://doi.org/10.1007/BF01585745

[14] C. Imreh, "Scheduling problems on two sets of identical machines," *Computing*, vol. 70, no. 4, pp. 277–294, Aug 2003. [Online]. Available: https://doi.org/10.1007/s00607-003-0011-9

[15] L. Chen, D. Ye, and G. Zhang, "Online scheduling of mixed CPU-GPU jobs," *International Journal of Foundations of Computer Science*, vol. 25, no. 06, pp. 745–761, 2014. [Online]. Available: https://doi.org/10.1142/S0129054114500312

[16] R. Bleuse, S. Kedad-Sidhoum, F. Monna, G. Mounié, and D. Trystram, "Scheduling independent tasks on multi-cores with gpu accelerators," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 6, pp. 1625–1638, 2015. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3359

[17] O. Beaumont, T. Cojean, L. Eyraud-Dubois, A. Guermouche, and S. Kumar, "Scheduling of linear algebra kernels on multiple heterogeneous resources," in *2016 IEEE 23rd International Conference on High-Performance Computing (HiPC)*. Los Alamitos, CA, USA: IEEE Computer Society, dec 2016, pp. 321–330. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/HiPC.2016.045

[18] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969. [Online]. Available: http://www.jstor.org/stable/2099572

[19] H. Topcuouglu, S. Hariri, and M.-y. Wu, "Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 260–274, Mar. 2002. [Online]. Available: http://dx.doi.org/10.1109/71.993206

[20] S. Kedad-Sidhoum, F. Monna, and D. Trystram, "Scheduling tasks with precedence constraints on hybrid multi-core machines," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, May 2015, pp. 27–33. [Online]. Available: https://doi.org/10.1109/IPDPSW.2015.119

[21] M. Amaris, G. Lucarelli, C. Mommessin, and D. Trystram, "Generic algorithms for scheduling applications on heterogeneous platforms," *Concurrency and Computation: Practice and Experience*, 2018. [Online]. Available: https://doi.org/10.1002/cpe.4647

[22] L. Jaulmes, M. Casas, M. Moretó, E. Ayguadé, J. Labarta, and M. Valero, "Exploiting asynchrony from exact forward recovery for due in iterative solvers," in *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2015, pp. 1–12. [Online]. Available: https://doi.org/10.1145/2807591.2807599

[23] "Chameleon: A dense linear algebra software for heterogeneous architectures," https://project.inria.fr/chameleon, 2014, accessed: 2019-12-10.

[24] M. Amaris, G. Lucarelli, C. Mommessin, and D. Trystram, "Generic algorithms for scheduling applications on hybrid multi-core machines," in *Euro-Par 2017: Parallel Processing*, F. F. Rivera, T. F. Pena, and J. C. Cabaleiro, Eds. Cham: Springer International Publishing, 2017, pp. 220–231. [Online]. Available: https://doi.org/10.1007/978-3-319-64203-1_16