Low Complexity On-Line Scheduling Algorithm for Hybrid Multi-Core Machines

Clément Mommessin Giorgio Lucarelli

Univ. Grenoble Alpes, CNRS, Inria, LIG, F-38000 Grenoble, France
{clement.mommessin, giorgio.lucarelli}@imag.fr

Keywords : *on-line scheduling, parallel applications, precedence constraints, CPU, GPU.*

1 Introduction

In each new generation of supercomputers the number of computing units increases [11]. The appearance of new types of computing units, such as General Purpose Graphical Processing Units (GPUs), along with the Central Processing Units (CPUs) tends to complexify the architecture of such platforms. The efficient management of these available resources to execute parallel applications on distributed platforms is a challenging problem in the domain of High Performance Computing.

There exist in the literature a huge amount of works dealing with the problem of scheduling in hybrid machines composed of both CPUs and GPUs. However, most of these works focus on specific applications. We are interested in generic algorithms which can be applied for any parallel application. More specifically, we aim to design efficient and low complexity algorithms for scheduling parallel applications on hybrid machines. An application consists of different tasks which are subject to precedence relations. We assume that the processing time of each task in each type of processing units is known in advance by applying appropriate predicting models [1]. The on-line version of the problem is considered, where tasks arrive one by one in a list that respects the precedence constraints. An algorithm has to take irrevocable decision on the scheduling of a task at the time of its arrival. The objective is to minimize the *makespan* of the schedule, that is the maximum completion time over all tasks of the application.

Problem definition and notations. We consider a parallel application that has to be scheduled on m identical CPUs and k identical GPUs. Without loss of generality, we assume that $k \leq m$. The application is composed of a list $\mathcal{T}$ of non-preemptable sequential tasks. Each task T_j is characterized by two processing times depending on which type of processing unit it will be executed. We denote by $\overline{p_j}$ (resp. $\underline{p_j}$) the processing time of T_j on a CPU (resp. GPU) and by C_j its completion time on a given schedule. The relations of precedence between tasks are represented by a Directed Acyclic Graph $G = (V, E)$ whose nodes are the tasks of the application and arcs indicate relations of precedence. For each arc $(i, j) \in E$, the task T_j cannot start its execution before the completion of T_i. We denote by $\Gamma^-(T_j)$ the set of all predecessors of T_j. To create a schedule, tasks are considered one by one following the original order of the list $\mathcal{T}$. The optimization objective is to minimize the completion time of the last finishing task, i.e., minimize $C_{max} = \max_{T_j \in \mathcal{T}}\{C_j\}$.

Our contribution and organization of the paper. In this work we study on-line scheduling algorithms for hybrid machines that take into account precedence constraints. Section 2 reviews previous work related to the addressed problem. We present in Section 3 an algorithm which relies on two greedy rules in order to decide the allocation of a task before applying the standard List Scheduling policy [9]. We call this algorithm Enhanced Rules - List Scheduling

(ER-LS) and show that it achieves a competitive ratio of $\Theta(\sqrt{\frac{m}{k}})$, which can be considered as constant-factor since the ratio $\frac{m}{k}$ is bounded in practice. Note that the ratio of this algorithm is asymptotically tight. Section 4 presents an experimental study of ER-LS with two baseline algorithms. We present the benchmark composed of 6 parallel applications and study the performance of the algorithms in various machine configurations. Results showed that ER-LS outperformed the two baseline algorithms with an improvement of more than 16% on average. Finally, all studied algorithms presented an experimental competitive ratio much smaller than the theoretical competitive ratio of ER-LS. We conclude and discuss future work in Section 5.

2 Related Work

Since the emergence of hardware accelerators, algorithms have been designed for scheduling tasks on hybrid machines composed of identical processors (CPUs) and one or several accelerators (GPUs). Many studies in the literature concern specific applications while only a few propose generic methods that deal with precedence constraints.

The problem of scheduling on two types of resources is more complex than the problem of scheduling tasks on parallel identical machines, but it is easier than the problem on unrelated machines. Moreover, if all tasks are accelerated by the same factor in the GPU side, then the hybrid problem coincides with the problem of scheduling on uniformly-related parallel machines. In this sense, we can say that the former is more general than the latter one. However in our problem all tasks have only two different processing times, that makes it simpler.

For the off-line version of the addressed problem, Kedad-Sidhoum et al. proposed a 6-approximation algorithm [10] with an allocation and a scheduling phase. This is the first generic approach achieving a constant approximation factor. Moreover, Amaris et al. [3] showed that this ratio was tight. They also modified the second step that improves the performance in practice but keep the same approximation ratio.

On a more practical side, the Heterogeneous Earliest Finish Time (HEFT) [13] algorithm is well used in runtime systems such as StarPU [4]. HEFT is a heuristic for scheduling tasks on unrelated machine, i.e., where the processing time of a task may differ on every machine, taking into account precedence constraints and communication times. However, HEFT cannot provide any constant-factor guarantee on its approximation ratio [3, 5].

In on-line mode with a list, Graham's List Scheduling (LS) [9] is a $(2 - \frac{1}{m})$-competitive algorithm for the case of only one type of processing units and precedence constraints between tasks, that also works with unknown processing times. Assuming a variant of the unique games conjecture, Svensson [12] showed that it was $\mathcal{NP}$-hard to improve this competitive ratio.

On the other hand, Chen et al. [6] showed that LS could not have a competitive ratio smaller than m in the context of hybrid machine and proposed a 4-competitive algorithm for scheduling independent tasks.

No work is known dealing with the on-line case on hybrid machine in the presence of precedence constraints.

3 Algorithm and Theoretical Bounds

We detail in this section the Enhanced Rules - List Scheduling (ER-LS) algorithm and propose theoretical bounds for its competitive ratio.

The ER-LS algorithm combines two greedy rules, in a similar way as in the 4-competitive algorithm proposed by Chen et al. [6] when independent tasks are considered. These rules decide the allocation of a task on either a CPU or a GPU, taking into account the actual schedule and the processing times of the task. Once the allocation of a task is decided, a

classical List Scheduling algorithm is used to schedule the task as early as possible on the processor type decided by the rules, taking into account its precedence constraints.

We denote by τ_{gpu} the earliest time when at least one GPU is idle. Let also $R_{j,gpu} = \max\{\tau_{gpu}, \max_{T_i \in \Gamma^-(T_j)}\{C_i\}\}$ be the *ready time* of T_j for GPUs, i.e., the earliest time at which T_j can be executed on a GPU. The two rules are defined as follows:

Rule 1: If $\overline{p_j} \geq R_{j,gpu} + \underline{p_j}$ then assign T_j to the GPU side; Otherwise apply Rule 2.

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

3.1 Upper bound

In the following, we propose an analysis of a schedule produced by ER-LS to give an upper bound of its competitive ratio.

Theorem 1 *ER-LS is at most a* $(4\sqrt{\frac{m}{k}})$*-competitive algorithm.*

Proof : Let W_{CPU}, W_{GPU} and CP be the total load on all CPUs, the total load on all GPUs and the length of the critical path of a schedule produced by the algorithm, respectively.

Given a schedule produced by ER-LS of makespan C_{max}, we can partition the time interval $[0, C_{max}]$ into three possibly non-disjoint subsets as follows:

- I_{CP} : Contains the time slots where at least one CPU and one GPU is idle.
- I_{CPU} : Contains the time slots where all CPUs are busy.
- I_{GPU} : Contains the time slots where all GPUs are busy.

Note that $I_{CPU} \cap I_{GPU}$ can be non-empty. By the above definitions, for the makespan of the schedule produced by ER-LS we have:

$$C_{max} \leq |I_{CPU}| + |I_{GPU}| + |I_{CP}| \leq \frac{W_{CPU}}{m} + \frac{W_{GPU}}{k} + CP \tag{1}$$

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

We denote by $\mathcal{SA}_{cpu}$ (resp. $\mathcal{SA}_{gpu}$) the set containing the tasks placed on the CPU (resp. GPU) side in both a solution of the algorithm and the optimal solution, by $\mathcal{SB}_{gpu}$ the set containing tasks placed by Rule 1 on the GPU side in a solution of the algorithm but on the CPU side in the optimal solution, and by $\mathcal{SC}_{cpu}$ (resp. $\mathcal{SC}_{gpu}$) the set containing tasks placed by Rule 2 on the CPU (resp. GPU) side in a solution of the algorithm but on the GPU (resp. CPU) side in the optimal solution. We also denote by sa_{cpu}, sa_{gpu}, sb_{gpu}, sc_{cpu} and sc_{gpu} the sum of processing times of all tasks in the sets $\mathcal{SA}_{cpu}$, $\mathcal{SA}_{gpu}$, $\mathcal{SB}_{gpu}$, $\mathcal{SC}_{cpu}$ and $\mathcal{SC}_{gpu}$, respectively. Note that we use here the processing times according to the allocation of ER-LS.

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

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

By adding the three inequalities we have the following:

$$\frac{sb_{gpu}}{k} + \frac{sa_{cpu} + \sqrt{\frac{m}{k}}sc_{gpu}}{m} + \frac{sa_{gpu} + \sqrt{\frac{k}{m}}sc_{cpu}}{k} \leq 3C^*_{max} \tag{2}$$

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

$$\frac{sa_{cpu}}{m} + \frac{sc_{cpu}}{\sqrt{mk}} \geq \frac{sa_{cpu} + sc_{cpu}}{m} \geq \sqrt{\frac{k}{m}}\frac{sa_{cpu} + sc_{cpu}}{m} \tag{3}$$

and:

$$\frac{sa_{gpu} + sb_{gpu}}{k} + \frac{sc_{gpu}}{\sqrt{mk}} \geq \frac{sa_{gpu} + sb_{gpu}}{k} + \frac{sc_{gpu}}{k}\sqrt{\frac{k}{m}} \geq \sqrt{\frac{k}{m}}\frac{sa_{gpu} + sb_{gpu} + sc_{gpu}}{k} \tag{4}$$

Summing these two bounds we finally obtain:

$$\sqrt{\frac{k}{m}}(\frac{sa_{cpu} + sc_{cpu}}{m} + \frac{sa_{gpu} + sb_{gpu} + sc_{gpu}}{k}) \leq 3C^*_{max} \tag{5}$$

and thus:

$$\frac{W_{CPU}}{m} + \frac{W_{GPU}}{k} \leq 3\sqrt{\frac{m}{k}}C^*_{max} \tag{6}$$

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

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

$$sa^{CP}_{cpu} + sa^{CP}_{gpu} = sa^{CP^*}_{cpu} + sa^{CP^*}_{gpu} \tag{7}$$

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

By summing the previous inequalities for the critical path we get:

$$CP = sa^{CP}_{cpu} + sa^{CP}_{gpu} + sb^{CP}_{gpu} + sc^{CP}_{cpu} + sc^{CP}_{gpu} \tag{8}$$

$$\leq \sqrt{\frac{m}{k}}(sa^{CP^*}_{cpu} + sa^{CP^*}_{gpu} + sb^{CP^*}_{gpu} + sc^{CP^*}_{cpu} + sc^{CP^*}_{gpu}) \leq \sqrt{\frac{m}{k}}CP^* \tag{9}$$

Since $CP^* \leq C^*_{max}$, we have $CP \leq \sqrt{\frac{m}{k}}C^*_{max}$ and, combining this inequality with Equations (1) and (6), the theorem follows. □

3.2 Lower bound

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

Theorem 2 *There is an instance for which ER-LS achieves a competitive ratio of* $\Omega(\sqrt{\frac{m}{k}})$.

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

TAB. 1: Instance of tasks for which ER-LS achieves an approximation ratio of $\sqrt{\frac{m}{k}}$.

Proof : Consider a hybrid system with m CPUs and $k \leq m$ GPUs. The instance consists of $m + k$ tasks that are partitioned into 2 sets as shown in Table 1. The k tasks of type A are independent to each other and the m tasks of type B are subject to the following precedence constraints: $B_1 \prec B_2 \prec \cdots \prec B_m$.

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

The ER-LS algorithm will first place the k tasks of type A on a GPU according to Rule 1. The completion time of these tasks is $\sqrt{m}$. Then, since $\sqrt{m} \leq \sqrt{m} + \sqrt{k}$, the task B_1 will be placed on a CPU according to Rule 2, with completion time $\sqrt{m}$. The task B_2 will also be placed on a CPU according to Rule 2, starting at time $\sqrt{m}$ and completing at time $2\sqrt{m}$. With the same reasoning, each task B_i, $i \in \{1, m\}$ is placed on a CPU according to Rule 2 starting at time $(i-1)\sqrt{m}$ and completing at time $i\sqrt{m}$. Thus, the schedule produced by ER-LS for this instance has a makespan of $C_{max} = m\sqrt{m}$.

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

Hence, ER-LS achieves a competitive ratio $\sqrt{\frac{m}{k}}$ for this instance and the theorem holds. □

4 Experiments

In this section we compare the performance of ER-LS with two greedy algorithms by a simulation campaign with 6 different parallel applications[1].

4.1 Benchmark and environment

The benchmark is composed of 5 applications generated by *Chameleon* [8], a dense linear algebra software, and a more irregular application (*fork-join*) generated using *GGen* [7], a library for generating directed acyclic graphs.

The applications of *Chameleon*, named *getrf*, *posv*, *potrf*, *potri* and *potrs*, are composed of multiple sequential basic tasks of linear algebra. Different tilings of the matrices have been used, varying the number of sub-matrices denoted by *nb_blocks*, from 5 to 20, and their size denoted by *block_size*, from 64 to 960. The applications were executed with the runtime StarPU [4] on a machine with two Dual core Xeon E7 v2 with a total of 10 physical cores with hyper-threading of 3 GHz and 256 GB of RAM. The machine had 4 GPUs NVIDIA Tesla K20 with each 5 GB of memory and 200 GB/s of bandwidth.

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 our experiments, we used $p \in \{2, 5, 10\}$ and $width \in \{100, 200, 300, 400, 500\}$. The processing time of each task on CPU was computed using a Gaussian distribution with center p and

[1]The data set is available under Creative Commons Public Licence at `github.com/marcosamaris/heterogeneous-SWF`, last visited on Jan. 2018.

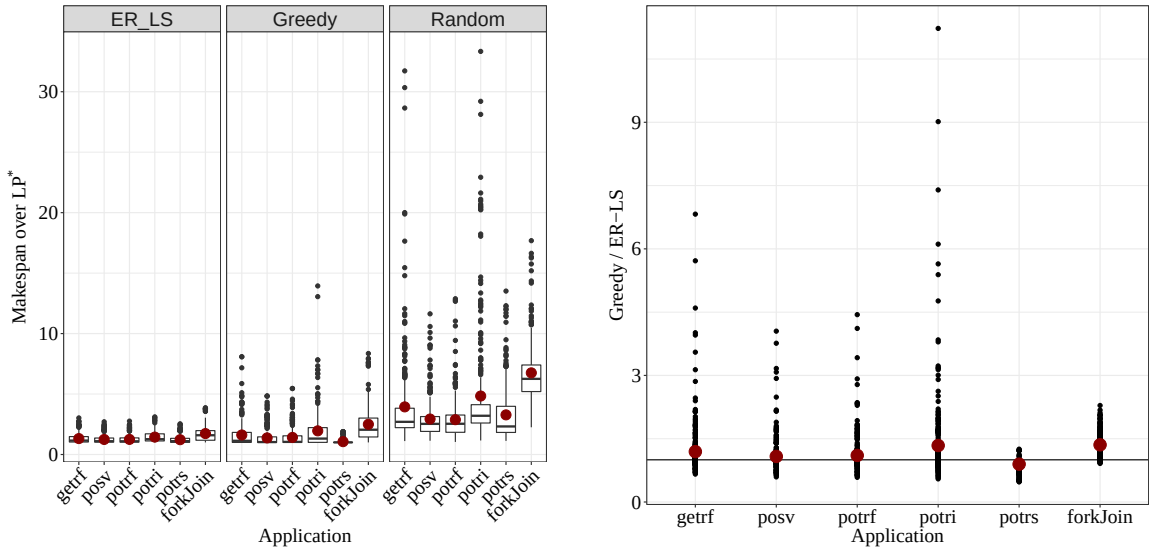

FIG. 1: Ratio of makespan over LP^* for each instance, grouped by application (left). Ratio between the makespans of Greedy and ER-LS for each instance, grouped by application (right).

standard deviation $\frac{p}{4}$. For the processing times on GPU, 5% of tasks within each phase have an acceleration factor in $[0.1, 0.5]$ while the other tasks have an acceleration factor in $[0.5, 50]$.

We compared the performance, in terms of makespan, of ER-LS with 2 baseline algorithms: *Greedy*, which allocates a task on the processor type which has the smallest processing time for that task; and *Random*, which randomly assigns a task to the CPU or GPU side. For these two algorithms, we used List Scheduling to schedule the tasks once the allocation has been made. The algorithms were implemented in Python (v. 2.8.6). Other policies were tested, such as Earliest Finish Time, and results are presented in a parallel work of the authors [2].

For the machine configurations, we determined different sets of pairs (Nb_CPUs, Nb_GPUs). Specifically, we used 16, 32, 64 and 128 CPUs with 2, 4, 8 and 16 GPUs for a total of 16 machine configurations. We executed the algorithms only once with each combination of application and machine configuration since all algorithms are deterministic, except Random. The running time of each algorithm took at most 5 seconds for the biggest instances of applications.

We also computed a lower bound, denoted by LP^*, of the optimal makespan for each combination of application and machine configuration by solving a linear program. This linear program, proposed by Kedad-Sidhoum et al. [10], is used to give an allocation of the tasks for the off-line version of the addressed problem. Since the allocation of tasks allows preemption and does not provide a schedule, the solution of the linear program gives a good lower bound on the optimal makespan.

4.2 Results

FIG. 1(left) compares the ratios between the makespan of each algorithm and LP^*. We observe that Random is greatly outperformed by the two other algorithms and presents instances with an approximation ratio larger than 30. We also note that ER-LS presents less outliers than the two baseline algorithms.

FIG. 1 (right) compares more specifically Greedy and ER-LS by showing the ratio between the makespans of the two algorithms. We can see that ER-LS outperforms Greedy on average, with a maximum for the potri application where ER-LS performs 11 times better than Greedy for a specific instance. In general, there is an improvement of between 8% and 36% on average for ER-LS depending on the application considered, except for *potrs* whose makespans are on

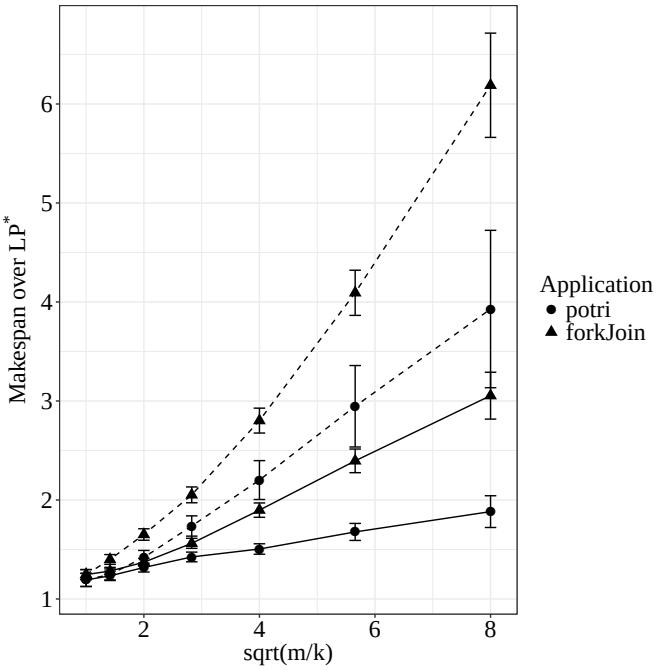

FIG. 2: Mean competitive ratio and standard error of ER-LS (plain) and Greedy (dashed) as a function of $\sqrt{\frac{m}{k}}$ (right).

average 10% greater than for Greedy.

These observations show that simple allocation rules taking into account the actual schedule can lead to smaller makespans than pure greedy allocations while keeping low-complexity scheduling policies, which is a desired feature in practice. Moreover, the extra computation of $R_{j,gpu}$ in Rule 1 of ER-LS is used afterwards by the List Scheduling algorithm and, thus, the running time of ER-LS is similar to the running times of the two baseline algorithms.

We also study the performance of the algorithms with respect to the theoretical upper bound of ER-LS given in Section 3.1. FIG. 2 shows the mean competitive ratios of ER-LS and Greedy along with their standard error as a function of $\sqrt{\frac{m}{k}}$ associated to each instance. To simplify the lecture, we discard the algorithm Random and only present the applications *potri* and *fork-join*, since other *Chameleon* applications showed similar results. We observe that the competitive ratios of the algorithms are smaller than $\sqrt{\frac{m}{k}}$ and far from the theoretical upper bound of $4\sqrt{\frac{m}{k}}$ for ER-LS.

5 Conclusion

We studied the problem of scheduling parallel applications with precedence constraints on hybrid machines composed of several identical CPUs and GPUs. We focused on designing a low-complexity algorithm for the on-line context where tasks arrive in order and the scheduler has to irrevocably make an allocation and scheduling decision at the arrival of each task.

We proposed Enhanced Rules - List Scheduling (ER-LS), the first on-line algorithm taking into account precedence relations between the tasks with hybrid machines, and showed that ER-LS achieves an asymptotically tight competitive ratio of $\Theta(\sqrt{\frac{m}{k}})$, which can be considered as constant-factor in practice.

The performance of ER-LS was evaluated in a simulation campaign with 6 different parallel applications. Experiments showed that ER-LS outperformed two baseline algorithms using a

pure greedy allocation while having a similar running time.

As a future work, we intend to refine the set of rules of ER-LS to better consider the schedule during the allocation decision, as well as introducing communication times between tasks linked by precedence.

Acknowledgments

We would like to thank Marcos Amaris, from the University of São Paulo, for providing us with the experimental benchmark. We also greatly thank Denis Trystram, from Inria Grenoble, for initiating the work and giving us numerous advice. This work is supported by the ANR Greco project.

References

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

[2] M. Amaris, G. Lucarelli, C. Mommessin, and D. Trystram. Generic algorithms for scheduling applications on heterogeneous multi-core platforms. ArXiv preprint 1711.06433, 2017.

[3] M. Amaris, G. Lucarelli, C. Mommessin, and D. Trystram. Generic algorithms for scheduling applications on hybrid multi-core machines. In *Euro-Par: 23rd International Conference on Parallel and Distributed Computing*, pages 220–231, Sept 2017.

[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*, 23(2):187–198, 2011.

[5] R. Bleuse, S. Kedad-Sidhoum, F. Monna, G. Mounié, and D. Trystram. Scheduling independent tasks on multi-cores with GPU accelerators. *Concurrency and Computation: Practice and Experience*, 27(6):1625–1638, 2015.

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

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

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

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

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

[11] TOP500 Supercomputer. `http://www.top500.org` (last visited on Jan. 2018).

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

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