# MulTreePrio: Scheduling task-based applications for heterogeneous computing systems

Hayfa Tayeb, Bérenger Bramas, Abdou Guermouche, Mathieu Faverge

**Compas'2022** : Parallélisme / Architecture/ Système
MIS/UPJV - Amiens France, 5-8 juillet 2022

# MulTreePrio: Scheduling task-based applications for heterogeneous computing systems

Hayfa Tayeb[1,3,4], Bérenger Bramas[1,4], Abdou Guermouche[2,3,4], and Mathieu Faverge[2,3,4]

[1]CAMUS Inria Nancy, ICube, University of Strasbourg, France
[2]HiePACS, Inria Bordeaux, LaBRI, Talence, France
[3]University of Bordeaux, Talence, France
[4]{first_name.last_name}@inria.fr

**Abstract**
Effective scheduling is crucial for task-based applications to achieve high performance in heterogeneous computing systems. These applications are usually represented by directed acyclic graphs (DAG). In this paper, we present a dynamic scheduling technique for DAGs intending to minimize the overall completion time of the parallelized applications. We introduce MulTreePrio, a novel scheduler based on a set of balanced trees data structure. The assignment of tasks to available resources is done according to priority scores per task for each type of processing unit. These scores are computed through heuristics built according to a set of rules that our scheduler should fulfil. We simulate the scheduling on three DAGs coming from numerical kernels with different configurations and we compare its behavior with both dynamic schedulers and static scheduling techniques based on the critical path. We show the efficiency of our scheduler with an average speedup of x2 with respect to the dynamic scheduler and x0,99 compared to the critical path-based scheduler. MulTreePrio is promising and in future works, it will be integrated into a task-based runtime system and tested in real-life scenarios.

**Keywords :** Scheduling, DAG, Task-based applications, Heterogeneous systems

## 1. Introduction

High-performance computing (HPC) relies on heterogeneous computing systems that come with an overall increased parallelism diversity such as multiprocessors and accelerators, e.g., graphical processing units (GPUs). HPC experts work tediously to narrow the gap between domain experts implementations and the use of heterogeneous systems. A range of research-driven projects has established diversified task-based support, employing various programming and runtime features [14]. The task-based programming model has shown great potential in various applications [1, 2, 9]. In this model, the developer defines atomic tasks with the dependencies between them. A directed acyclic graph (DAG) represents the application. The runtime, which is an intermediate software layer supporting this DAG execution, schedules tasks and data migrations efficiently on all available cores while reducing the waiting time between tasks. Therefore, the goal of DAG scheduling is to minimize the global completion time of the program, i.e. the makespan. Various runtime systems capable of handling heterogeneous workloads have emerged (Parsec [6], StarPU [4]). HeteroPrio [2] is a scheduler that is implemented in StarPU. It has been designed for heterogeneous machines and is used by several applications showing significant improvements [3, 13]. HeteroPrio is a semi-automatic scheduler where users must provide priorities

for the different types of tasks that exist in their applications. A fully automatic version of this scheduler that computes efficient priorities for HeteroPrio is proposed in [11]. HeteroPrio and its automatic version are cheap and effective. However, they show a limitation which is the priority assignment per type of task. Every application has a set of task types that will be used in different stages of the scheduling. Setting a priority per type could hide relevant information related to a given scheduling context. This also brings us to the limitation of the data structures that are used to manage the ready tasks in the scheduler which are tied to the strategy of priorities per type.
This study aims to address the limitations raised and therefore proposes a novel scheduler based on priority per task for a given processing unit. We define a data structure managing the ready tasks in the system per priority and per processing unit type. Our objective is to minimize the global completion time of task-based applications in heterogeneous environments thanks to good scheduling decisions. The major contributions of this paper can be summarized as follow:

- We present a novel dynamic scheduler for heterogeneous systems based on priority scores per task and per processing unit which are stored within a set of binary trees and managed by the internal mechanisms of the scheduler.
- We propose heuristics based on rules that we set out that should be fulfilled during the scheduling and evaluate the performance of the scheduler using emulated executions of DAGs on diverse configurations.

The paper is organized as follows: In Section 2 we briefly analyze some related work. In Section 3 we present our proposed scheduler. In Section 4 we describe the performance study with emulated task-based applications and simulate the scheduling on different configurations of heterogeneous systems. Finally, Section 5 concludes.

## 2. Related work

The problem of scheduling on heterogeneous computing systems has been proven NP-complete in general cases [8]. Numerous research works have proposed different scheduling algorithms. Thoman et al. [14] divide them into three categories, namely static, dynamic, and hybrid scheduling methods. The distribution of work at compile time is static scheduling, while the distribution of work at runtime is dynamic scheduling. In general, one cannot have global visibility on the entire DAG. This is why recent research has been focused on dynamic scheduling. Choi and al. [10] propose dynamic scheduling that relies on a history-based Estimated-Execution-Time (EET) for each task. The idea of this algorithm is to schedule each task on its fastest architecture. In some cases, the scheduler ignores this rule and executes a task on a slower processor (e.g. in the case of work starvation for a worker type).
HeteroPrio [2] is a scheduler designed for heterogeneous machines and implemented in StarPU. It has shown its efficiency when used on several applications [3, 13]. HeteroPrio relies on a priority assignment per type of task with respect to its performance on a processing unit (PU). Figure 1 shows that tasks are dispatched to buckets with respect to their priority. Each task is executed on the most prioritised available PU. However, this approach comes with a limitation. The different types of tasks are involved in various stages of the execution. Setting a priority per type hides relevant information related to a given scheduling scenario.

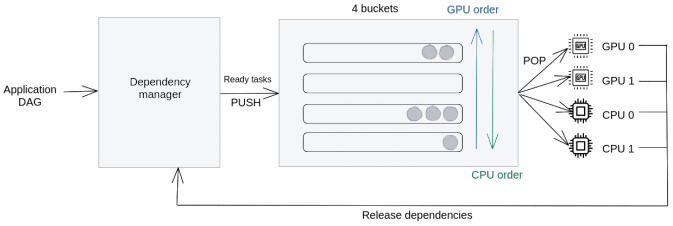



Figure 1: HeteroPrio scheduler overview

## 3. MulTreePrio scheduler

### 3.1. Context and notations

For a heterogeneous computing system, we are given a set of memory nodes denoted as $M$. In our study, we see the main RAM of a computing node as a single memory node despite the NUMA effects but the approach remains valid otherwise. $m \in M$ can be either the main RAM, a GPU-embedded memory or disk memory. $P$ is the set of processing units of all types which could be for example {CPU, GPU}. We note $P_m \subset P$ the subset of processing units tied to $m \in M$. Given this infrastructure illustrated in Figure 2, our scheduler will manage ready tasks. A ready task is a task for which all dependencies in the DAG are fulfilled. To store the tasks and the necessary information about each of them, we introduce a data structure based on binary balanced trees. We note $T$ the set of trees managed by our scheduler. $t_m \in T$ is a tree with ready tasks that can be computed by any processing unit in $P_m$. The runtime system has a set of workers denoted $W$ that consume tasks from the trees to execute them on the dedicated processing unit. Each worker $w \in W_m$ picks its task from $t_m \in T$ and executes them on any processing unit in $P_m \subset P$. From this description, we have $|T| = |M|$, and we have at least one worker per memory node and expect $|M| \leq |P| \leq |W|$.

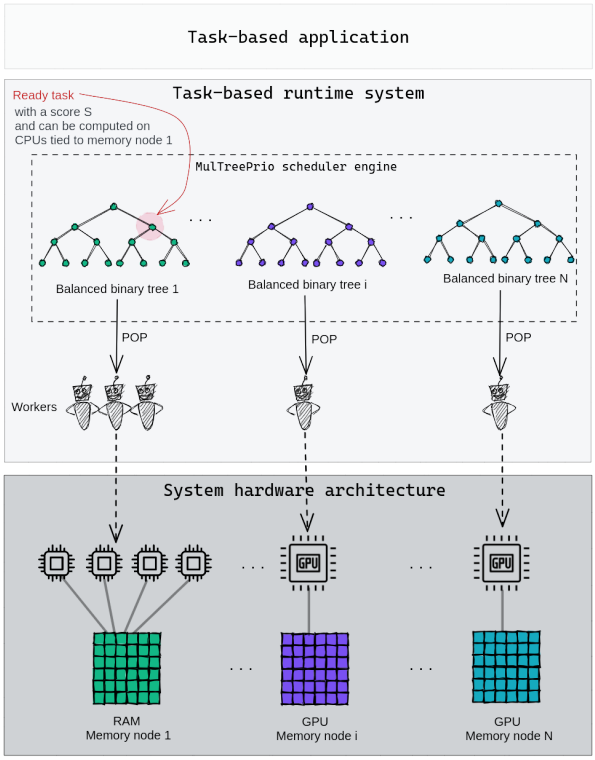



Figure 2: MulTreePrio scheduler overview

For a given newly ready task $\mathcal{T}$ and a given memory node $m$, the runtime system provides us with numerous information about the scheduling context. The task can be executed on a set of processing units tied to their respective memory nodes. By means of the data structure tracking the tasks into our scheduler, at any instant of the scheduling processing, we have the number of ready tasks present at that moment in the system that can be computed on the processing units $P_m \subseteq P$. It is equal to $|t_m|$ but those tasks could be duplicated on several trees if there is more than a memory node tied to the processing unit able to compute it. As the execution progresses, the DAG is dynamically constructed. We suppose that we can retrieve the set of tasks that will be released when $\mathcal{T}$ is computed, i.e., its successors, denoted $succ(\mathcal{T})$. The successors could target different processing units, therefore, we note $succ(\mathcal{T}, P_m)$ the subset of tasks that will be released for $P_m$. This does not take into account the dependencies that could exist with other tasks. Therefore, we use the same metric Normalized Out-Degree (NOD) as in [12]. This metric tells us if $\mathcal{T}$ is critical to release or not. Moreover, we take into account the workload of the workers in $W_m$, i.e. they are starving or not. We consider $\xi(\mathcal{T}, P_m)$ the execution time estimation for $\mathcal{T}$ computed by processing units $P_m$. This estimation could be provided by a performance model from the runtime system. We, thus, introduce the amount of work that will be released when $\mathcal{T}$ is computed denoted $Q_{succ(\mathcal{T},P_m)}$. It is the busy time per processing units type. We consider the total busy time released by $succ(\mathcal{T})$ as the busy time on the best processing units type, noted $Q_{succ(\mathcal{T})}$.

### 3.2. Heuristics

For a newly ready task $\mathcal{T}$, our objective is to compute $score(\mathcal{T}, m), \forall m \in M$ in order to push $\mathcal{T}$ in all the trees that are respectively targeting processing units $P_m$ which are able to compute the task. Based on the different scores of all the tasks in a given tree, the scheduler picks the most prioritized tasks. The score is defined as a combination of heuristics fulfilling a set of rules. We illustrate different situations that can be encountered during the scheduling and from there, try to define the expected behaviour for good scheduling decisions. We take into account mainly the degree of suitability of a task for a processing unit, i.e. is it good on that processor, the amount of work that will be released if a task will be executed and the load of the workers. In order to define the rules, we consider two tasks $\mathcal{T}_0$ and $\mathcal{T}_1$, and $p_0 = prio(\mathcal{T}_0, m)$ and $p_1 = prio(\mathcal{T}_1, m)$.

- *Rule 1*: If $\mathcal{T}_0$ releases more work than $\mathcal{T}_1$, we should have $p_0 \geq p_1$.
$$Q_{succ(\mathcal{T}_0)} \geq Q_{succ(\mathcal{T}_1)} \Rightarrow p_0 \geq p_1 \quad (1)$$

- *Rule 2*: For an equivalent amount of released work, if $\mathcal{T}_0$ has a larger number of released tasks than $\mathcal{T}_1$, we should have $p_0 \geq p_1$.
$$(Q_{succ(\mathcal{T}_0)} = Q_{succ(\mathcal{T}_1)}) \wedge (|succ(\mathcal{T}_0)| \geq |succ(\mathcal{T}_1)|) \Rightarrow p_0 \geq p_1 \quad (2)$$

- *Rule 3*: If $\mathcal{T}_0$ releases more work for starving workers than $\mathcal{T}_1$, we should have $p_0 \geq p_1$.
$$(\exists W_{m_i} \subseteq W(succ(\mathcal{T}_0))/starving(W_{m_i})) \wedge (Q_{succ(\mathcal{T}_0, P_{m_i})} \geq Q_{succ(\mathcal{T}_1, P_{m_i})}) \Rightarrow p_0 \geq p_1 \quad (3)$$

- *Rule 4*: If $\mathcal{T}_0$ has a better speedup than $\mathcal{T}_1$ on $P_m$, or is less worse on these workers, we should have $p_0 \geq p_1$.
$$\sigma(\mathcal{T}_0, P_m) \geq \sigma(\mathcal{T}_1, P_m) \Rightarrow p_0 \geq p_1 \quad (4)$$

- *Rule 5*: If $\mathcal{T}_0$ is expected to be longer than $\mathcal{T}_1$ on $P_m$, we should have $p_0 \geq p_1$.
$$\xi(\mathcal{T}_0, P_m) \geq \xi(\mathcal{T}_1, P_m) \Rightarrow p_0 \geq p_1 \quad (5)$$

- *Rule 6*: If $\mathcal{T}_0$ is more critical than $\mathcal{T}_1$ (in terms of dependencies), we should have $p_0 \geq p_1$.
$$NOD(\mathcal{T}_0) \geq NOD(\mathcal{T}_1) \Rightarrow p_0 \geq p_1 \quad (6)$$

- *Rule 7*: If $\mathcal{T}_1$ can be computed by workers that are starving and $\mathcal{T}_0$ cannot, we should have $p_0 \geq p_1$.
$$(\exists W_{m_i} \subseteq W(\mathcal{T}_1)/starving(W_{m_i})) \wedge (W_{m_i} \nsubseteq W(\mathcal{T}_0)) \Rightarrow p_0 \geq p_1 \quad (7)$$

Given the above rules, we propose different heuristics normalized between 0 and 1 (more detail in the appendix). For a given task $\mathcal{T}$ and a memory node $m$, there is a heuristic that satisfies the fixed rules. The priority is calculated as follow:

$$score(\mathcal{T}, m) = \frac{\sum_{i=1}^{|B|} B_i(\mathcal{T}, m) - \sum_{j=1}^{|\mathcal{P}|} \mathcal{P}_j(\mathcal{T}, m) + |\mathcal{P}|}{|B| + |\mathcal{P}|}, \quad (8)$$

with $B_i \in [0, 1], \forall i$ bonuses and $\mathcal{P}_j \in [0, 1], \forall j$ penalties.

### 3.3. Other heuristics

**Speedup factor:** During our study, we noticed that at the end of a scheduling or more generally when workers are starving, it becomes more difficult to find good scheduling decisions. The trees have fewer tasks and they could be the ones that were left last in the trees, possibly having the worst scores. We do not want to risk workers choosing slow implementations of tasks. It will inevitably extend the makespan. To be more selective, we present a heuristic that makes the scheduler decide if a processing unit type is not the best at executing this task, we only accept to let it compute the task if a factor of speedup is guaranteed. The proposed heuristic is as follow: if the remaining work for the workers of a certain processing unit is greater than the cost of a task, we accept computing the task on this processing unit, therefore, we avoid an idle time and gain on the total completion time of the application.

**Critical path:** Since we are in a simulated environment, we studied the profitability of adding a heuristic based on the critical path. In a recent work, Beamont et al. [5] present ALAP-schedule (As Late As Possible) on Cholesky factorization in a homogeneous model without communication costs. They consider the reversed DAG therefore tasks are scheduled according to their distance to the end of the critical path. In our work, we use a slightly different heuristic, which is the latest starting time without delay. To obtain this number, we first calculated the critical path with a static technique by traversing the graph on two phases, a forward pass for computing the earliest starting and finish time then a backward pass for computing the latest starting and finish time. We detected the critical tasks (i.e. their earliest and latest starting times are equal) that, if delayed, would lead to a longer makespan. Based on that information, we set the degree of criticality of a task as [1 - (latest starting time / total time of execution)]. This allows us to prioritize the critical tasks and compare the scheduler with its dynamic version and see if even with less information on the whole DAG, we can find good scheduling decisions.

## 4. Experiments

In order to evaluate the proposed scheduler, we simulate the execution of 3 different emulated applications within a task-based runtime system [7]. This simulation does not take into account data communication between nodes. We simulated Cholesky factorization which is a widespread matrix decomposition in linear algebra, and two numerical methods for the computation of long-ranged forces in the n-body problem (tree-based method and Fast multipole method). We simulated the scheduling of the DAGs and compared the results of Heteroprio scheduler, our scheduler MulTreePrio, a static version of our scheduler with a critical path heuristic and a static scheduler based only on the critical path heuristic.

For the simulation, we tested the scheduler on different configurations of heterogeneous systems. The Table 1 shows the 3 configurations used. We note that we are in simulation environment, thus, we tested different configurations (even ones that do not exist in real-life systems) in order to observe the behavior of the schedulers and see their limits. We present two different test cases on the 3 emulated applications (details in Table 2).

| Configurations | 1 | 2 | 3 |
|---|---|---|---|
| Number of CPUs | 16 | 18 | 20 |
| Number of GPUs | 4 | 10 | 50 |

Table 1: Configurations of the heterogeneous computing system in the simulation

We report in Figure 3 the summary of the results of our simulation on the 3 applications with the different test cases while varying the configurations of the number of CPUs and GPUs. Prioritizing costly tasks to be computed on the best PU is not possible for the CP-based scheduler, thus, its results are worse than the others. Here, we show the importance of the performance of tasks and the impact of this information in taking good scheduling decisions. The scheduling with MulTreePrio of the emulated Cholesky factorization is better than HeteroPrio. When we increase the resources of the heterogeneous system, MulTreePrio, records a makespan very close and sometimes equal to the length of the critical path. For the tree-based method, the dependencies in the DAG are huge. We can see in Figure 3 (b) that MulTreePrio is approximately 3,5 times faster than HeteroPrio. The priority system per task reaches its limit in this case because the costs of the tasks are very close except for one of them, thus, finding the right priorities is not easy or even profitable anymore. However, for the second test case, we present different costs for the task types, thus they can be prioritized more easily, even though they have the same relative speedup on GPU equal to 10. The results of MulTreePrio are better than HeteroPrio. We recall that HeteroPrio was designed in the context of optimizing the fast multipole method. Therefore, the results of the scheduling are more or less close and this implies that MulTreePrio is able to make good scheduling decisions. Given

the fact that the DAG has few dependencies, the heuristic of the critical path is less impacting. The DAG of the FMM is very detached consequently the length of the critical path with an infinity of resources is very short.

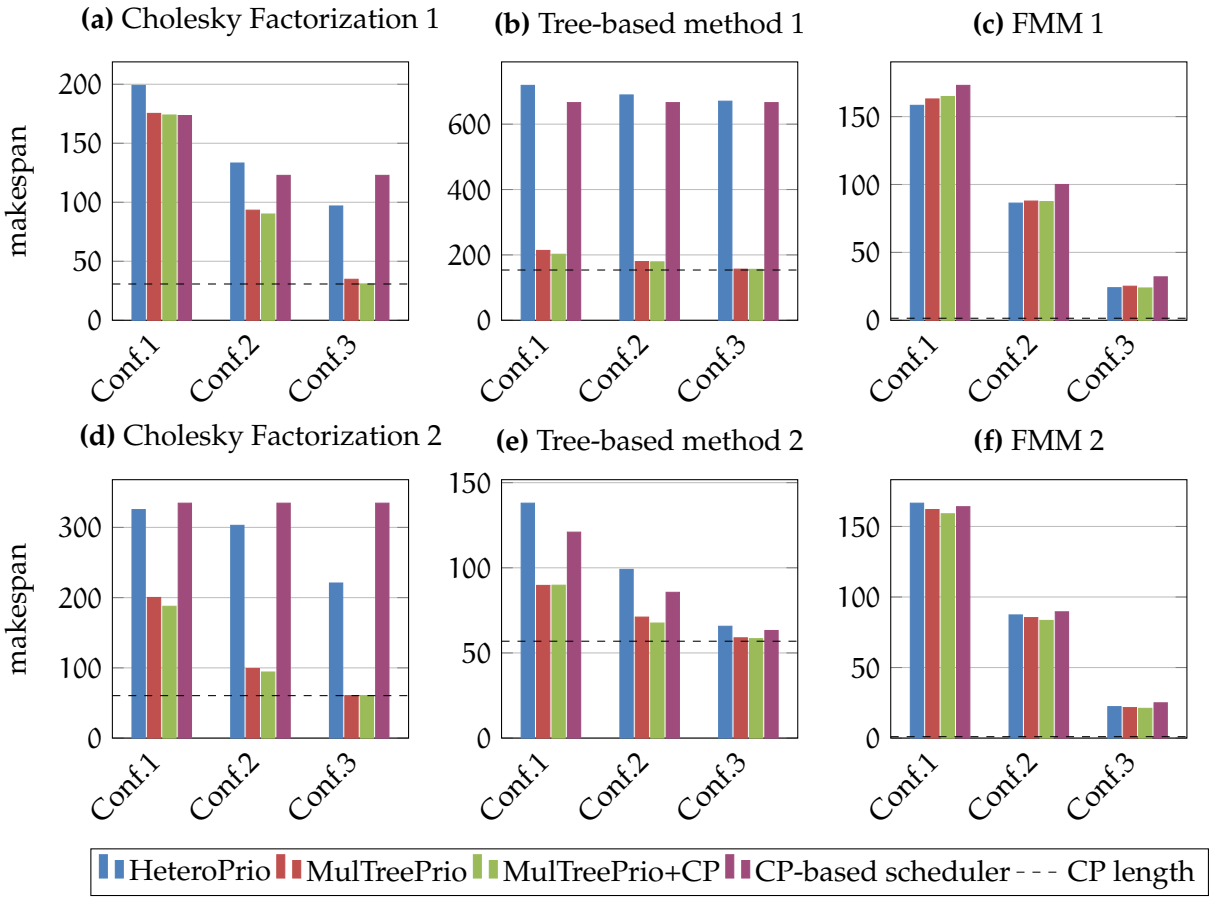



Figure 3: Comparison of the makespans of the 4 schedulers while varying the number of CPUs and GPUs according to the Table 1. The minimum achievable makespan is the CP length.

## 5. Conclusion

Dynamic scheduling of task-based applications on heterogeneous systems is an NP-complete problem to whom different algorithms were proposed. Despite all the efforts, having a minimized total makespan of an application is not always guaranteed. Among others, this comes down to the fact that the DAGs from an application to another varies considerably and this impact directly the scheduling decisions. We have shown that our scheduler makes overall good scheduling results thanks to its fast and efficient heuristics. The fact that these heuristics were based on a set of rules covering different issues that may be encountered while scheduling has enhanced the robustness of the scheduler in front of different DAGs. In the future work, we aim to integrate this strategy into a scheduler and benchmark real-life applications dynamic scheduling.

## Acknowledgements

This work is supported by the TEXTAROSSA project G.A. n.956831 part of the EuroHPC initiative.

**Bibliographie**

1. Agullo (E.), Bramas (B.), Coulaud (O.), Darve (E.), Messner (M.) and Takahashi (T.). – Task-based fmm for multicore architectures. *SIAM Journal on Scientific Computing*, 2014.
2. Agullo (E.), Bramas (B.), Coulaud (O.), Darve (E.), Messner (M.) and Takahashi (T.). – Task-based fmm for heterogeneous architectures. *Concurrency and Computation: Practice and Experience*, vol. 28, n9, 2016, pp. 2608–2629.
3. Agullo (E.), Buttari (A.), Guermouche (A.) and Lopez (F.). – Task-based multifrontal qr solver for gpu-accelerated multicore architectures. – In *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*, pp. 54–63, 2015.
4. Augonnet (C.), Thibault (S.), Namyst (R.) and Wacrenier (P.-A.). – StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, vol. 23, n2, 2011, pp. 187–198.
5. Beaumont (O.), Langou (J.), Quach (W.) and Shilova (A.). – A makespan lower bound for the tiled cholesky factorization based on alap schedule. *In: Malawski M., Rzadca K. (eds) Euro-Par 2020: Parallel Processing. Euro-Par 2020. Lecture Notes in Computer Science, vol 12247. Springer, Cham.*, 2020.
6. Bosilca (G.), Bouteiller (A.), Danalis (A.), Faverge (M.), Hérault (T.) and Dongarra (J. J.). – Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, vol. 15, n6, 2013, pp. 36–45.
7. Bramas (B.). – SPETABARU: A Task-based Runtime System with Speculative Execution Capability. – In *SIAM CSE 2019 - SIAM Conference on Computational Science and Engineering*, Spokane, United States, février 2019.
8. Brucker (P.) and Knust (S.). – *Complexity results for scheduling problems*, 2009. `http://www2.informatik.uni-osnabrueck.de/knust/class/`.
9. Carpaye (J. M. C.), Roman (J.) and Brenner (P.). – Design and analysis of a task-based parallelization over a runtime system of an explicit finite-volume CFD code with adaptive time stepping. *Journal of Computational Science*, 2018.
10. Choi (H.), Son (D.), Kang (S.), Kim (J.), Lee (H.-H.) and Kim (C.-H.). – An efficient scheduling scheme using estimated execution time for heterogeneous computing systems. *The Journal of Supercomputing*, vol. 65, 08 2013.
11. Flint (C.) and Bramas (B.). – Finding new heuristics for automated task prioritizing in heterogeneous computing, 2020. `https://hal.inria.fr/hal-02993015/document`.
12. Lin (H.), Li (M.-F.), Jia (C.-F.), Liu (J.-N.) and An (H.). – Degree-of-node task scheduling of fine-grained parallel programs on heterogeneous systems. *Journal of Computer Science and Technology*, vol. 34, n5, 2019, pp. 1096–1108.
13. Lopez (F.) and Duff (I.). – Task-Based Sparse Direct solver for Symmetric Indefinite Systems, 2018. 10th International Workshop on Parallel Matrix Algorithms and Applications (PMAA), mini-symposium on task-based programming for scientific computing.
14. Thoman (P.), Dichev (K.), Heller (T.), Iakymchuk (R.), Aguilar (X.), Hasanov (K.), Gschwandtner (P.), Lemarinier (P.), Markidis (S.), Jordan (H.), Fahringer (T.), Katrinis (K.), Laure (E.) and Nikolopoulos (D. S.). – A taxonomy of task-based parallel programming technologies for high-performance computing. *J. Supercomput.*, vol. 74, n4, apr 2018, p. 1422–1434.

**A. Metrics:**

The NOD is given by the following formula:

$$\mathrm{NOD}(\mathcal{T}) = \sum_{s_i \in \mathrm{succ}(\mathcal{T})} \frac{1}{|\mathrm{pred}(s_i)|} \tag{9}$$

The metrics relative the released amount of work when a task is computed are defined as follow:

$$\begin{aligned} Q_{\mathrm{succ}(\mathcal{T},P_m)} &= \sum_{s \in \mathrm{succ}(\mathcal{T})} \xi(s, P_m) \\ Q_{\mathrm{succ}(\mathcal{T})} &= \sum_{s \in \mathrm{succ}(\mathcal{T})} \min_{\forall m \in M_s} \xi(s, P_m) \end{aligned} \tag{10}$$

We compute the relative speedup of $\mathcal{T}$ computed by $P_m$. It is defined as follow:

$$\sigma(\mathcal{T}, P_m) = \frac{\xi_{worst}(\mathcal{T}, P_m/worst)}{\xi(\mathcal{T}, P_m)} \tag{11}$$

where $P_m/worst$ is the processing units type that computes $\mathcal{T}$ in the slowest execution time compared to other possible $P_m$. In the worst case, we have $\sigma(\mathcal{T}, P_m) = 1$, i.e. no speedup.

**B. Heuristics formulas:**

For a given task $\mathcal{T}$ and a memory node $m$, we propose heuristics that satisfy the fixed rules. The following entities are normalized between 0 and 1.

- To satisfy the rules 1 and 2, we consider the released amount of work normalized by the largest amount of work that has been released so far weighted by the number of successors.

$$B_1(\mathcal{T}, m) = 1 - e^{-\left[\frac{Q_{\mathrm{succ}(\mathcal{T})}}{Q_{highest}} \times |\mathrm{succ}(\mathcal{T})|\right]} \tag{12}$$

- To satisfy the rule 3, we consider the sum of the released amount of work for each starving workers weighted by the number of released tasks.

$$B_2(\mathcal{T}, m) = \frac{\sum_{m_i \in M} \left(1 - e^{-\frac{Q_{\mathrm{succ}(\mathcal{T},P_{m_i})}}{Q_{highest/P_{m_i}}} \times |\mathrm{succ}(\mathcal{T},P_{m_i})|}\right) \times starving(W_{m_i})}{|M|} \tag{13}$$

- To satisfy the rule 4, we consider the speedup $\sigma(\mathcal{T}, P_m) \geq 1$ and we normalize it as follow to underline if it is a really good speedup (very close to 1) or if it is average or with no speedup (equal to 0).

$$B_3(\mathcal{T}, m) = 1 - \frac{1}{\sigma(\mathcal{T}, P_m)} \tag{14}$$

- To satisfy the rule 5, we consider the time the task will take when executed on $P_m$ if it has a good speedup otherwise it is 0. Thus, we calculate the ratio:

$$B_4(\mathcal{T}, m) = \frac{\xi(\mathcal{T}, P_m)}{\xi_{longest/P_m}} \times \frac{\sigma(\mathcal{T}, P_m) - 1}{\sigma_{best}(\mathcal{T}) - 1} \tag{15}$$

- To satisfy the rule 6, we consider the criticality of the task based on the metric $\mathrm{NOD}(\mathcal{T})$ and we calculate the ratio:

$$B_5(\mathcal{T}, m) = \frac{\mathrm{NOD}(\mathcal{T})}{\mathrm{NOD}_{highest/P_m}} \tag{16}$$

- To satisfy the rule 7, we consider a penalty $\mathcal{P}$ for the current worker when there is other workers starving and have a relatively good speedup. $\mathcal{P}$ increases the more the speedup on

other starving workers increases and inversely.

$$\mathcal{P}_1 = \frac{\sum_{m_i \in M, m_i \neq m} \left[1 - \frac{1}{\sigma(\mathcal{T}, P_{m_i})}\right] \times starving(W_{m_i})}{|M| - 1} \tag{17}$$

**C. Simulation details:**

Each application has a set of task, i.e. task types. Each task is a heterogeneous one and thus has CPU cost and GPU cost. It is the execution time estimated for the task on each processing unit. For the simulation of our scheduler, we changed the costs of the tasks for all the applications. We fixed for the simulated DAGs the costs on CPU and GPU. For Heteroprio scheduler, we also need to have the priorities per type of task for each processing unit. It is a configuration that can be seen as the order of iteration on the workers (0 means first or more prioritized). We show in the Table 2 for each task the speedup on GPU over CPU to highlight the gain compared to other tasks. For example, for the first test case, for the Cholesky factorization, a GEMM task costs 0.7 on CPU and 0.1 on GPU, hence, the speedup of this task on GPU is x7. It is more prioritized to be executed on GPU, i.e. its GPU priority is equal to 0.

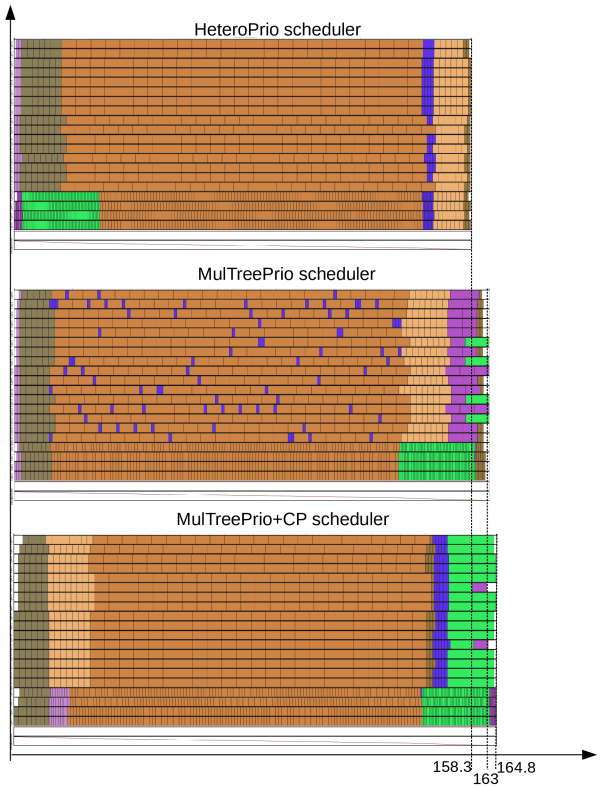
| | | First test case | | | | | Second test case | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | type of tasks | Costs | | Speedups | Priorities | | Costs | | Speedups | Priorities | |
| | | CPU | GPU | | CPU | GPU | CPU | GPU | | CPU | GPU |
| Cholesky factorization | GEMM | 0.7 | 0.1 | 7 | 3 | 0 | 1. | 1.1 | 0.9 | 0 | 3 |
| | SYRK | 1.3 | 0.3 | 4.3 | 2 | 1 | 5.5 | 0.8 | 6.8 | 1 | 2 |
| | POTRF | 1.2 | 0.35 | 3.4 | 1 | 2 | 2. | 0.2 | 10 | 2 | 1 |
| | TRSM | 1. | 0.3 | 3.3 | 0 | 3 | 3. | 0.05 | 60 | 3 | 0 |
| Tree based method | In-Cell | 1. | 0.3 | 3.3 | 0 | 3 | 2. | 0.2 | 10 | 2 | 1 |
| | P2C | 1.2 | 0.35 | 3.42 | 1 | 2 | 5. | 0.5 | 10 | 3 | 0 |
| | C2P | 1.3 | 0.3 | 4.3 | 2 | 1 | 0.11 | 1.1 | 10 | 1 | 3 |
| | N2N | 0.7 | 0.1 | 7 | 3 | 0 | 0.08 | 0.8 | 10 | 0 | 2 |
| Fast multipole method | P2M | 1. | 0.8 | 1.25 | 0 | 7 | 0.5 | 0.8 | 0.625 | 0 | 7 |
| | M2M | 2. | 1.5 | 1.3 | 1 | 6 | 3. | 0.5 | 6. | 4 | 3 |
| | M2L | 5. | 1. | 5 | 2 | 2 | 5. | 1. | 5. | 3 | 4 |
| | M2L' | 1. | 0.5 | 2 | 3 | 3 | 1. | 0.5 | 2. | 2 | 5 |
| | L2L | 2. | 1.5 | 1.3 | 4 | 4 | 3. | 0.5 | 6. | 5 | 2 |
| | L2P | 1. | 0.8 | 1.25 | 5 | 5 | 1. | 0.8 | 1.25 | 1 | 6 |
| | P2P | 8. | 0.5 | 16 | 6 | 0 | 4. | 0.5 | 8. | 6 | 1 |
| | P2P' | 5. | 0.3 | 16.6 | 7 | 1 | 2.5 | 0.3 | 8.3 | 7 | 0 |

Table 2: Task graph properties for the different test cases: the Cholesky factorization, a tree-based method, and the fast multipole method. The priority columns correspond to the Heteroprio configuration, which can be seen as the order of iteration on the workers (0 means first).

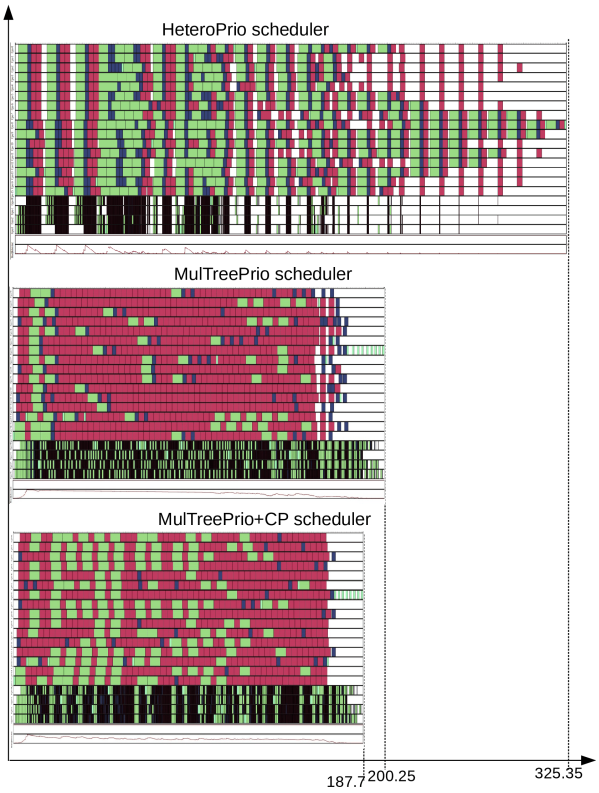
## D. Scheduling traces:

(a) Color legend for FMM tasks:
M2M, L2L, P2M, M2L, M2L', P2P, P2P'

(b) Color legend for Cholesky tasks:
GEMM, TRSM, SYRK, POTFR




(c) Trace of FMM test case 1




(d) Trace of Cholesky test case 2

Figure 4: Scheduling traces for the FMM 1st test case and Cholesky factorization 2nd test cases. The configuration is 16 CPUs and 4 GPUs (the last 4 rows). Comparison of HeteroPrio, MulTreePrio and MulTreePrio+CP a static version of our scheduler with a critical path heuristic.