Finding new heuristics for automated task prioritizing in heterogeneous computing

Clément Flint, Bérenger Bramas

Finding new heuristics for automated task prioritizing in heterogeneous computing

Clément FLINT, Bérenger BRAMAS

ICPS Team, ICube laboratory
University of Strasbourg
CAMUS Team, Inria Nancy

November 2020

Abstract

In many sciences, processing costly computations has become frequent and the execution time of an application is often viewed as a bottleneck. High-Performance Computing is a growing area of interest whose role is to reduce the execution time of these applications by optimizing their granularity, data accesses, data transfers, etc. During the late 2010 decade, Graphics Processing Units have started to become a popular solution for speeding up programs. For some operations, using GPUs instead of CPUs drastically speeds up the computation time thanks to their massive parallelization potential. There are, however, downsides to using GPUs along with CPUs. Firstly, it adds costly data transfers to make data available to the GPUs. Secondly, scheduling decisions are more complex and may, therefore, be less efficient. In this study, we improve Heteroprio, a hybrid CPU/GPU scheduler that works with user-defined CPU and GPU priorities. Our improved version now affects priorities automatically, based on heuristics that have been validated on theoretical executions.

1 Introduction

In this study, we focus on CPU/GPU hybrid systems, which are commonly used in High-Performance Computing.
The task scheduling problem in homogeneous architectures is well known, and there is a considerable amount of scientific literature treating it. In heterogeneous architectures, however, research is still vastly ongoing.

Heteroprio has been created in 2016, as an attempt to design an efficient scheduler for heterogeneous machines. It has been implemented in StarPU (a task-based execution engine on heterogeneous multicore architectures). Heteroprio schedules tasks based on user-defined CPU and GPU priorities.
The Heteroprio scheduler gave solid results on multiple applications, but at the cost of a high degree of required expertise to find proper priorities. Indeed, unlike most other StarPU's schedulers, Heteroprio requires the user to choose and fill task priorities. The scheduling efficiency will be highly impacted by these. A wrong choice of priorities can easily lead to a worse execution time than a naive FIFO scheduler.

This study has been conducted in the framework of this observation. Our goal is to demonstrate that it is possible to affect these priorities automatically, without hurting the scheduling performance.

Experiments presented in this report were carried out using the PlaFRIM experimental testbed, supported by Inria, CNRS (LABRI and IMB), Université de Bordeaux, Bordeaux INP and Conseil Régional d'Aquitaine (see https://www.plafrim.fr/).

2 Context

2.1 Background of the project

2.1.1 Related work

Our work falls within a particular concern that has come with the rise of heterogeneous programming. Working with a heterogeneous architecture adds complexity to the scheduling problem which is already hard in the homogeneous case (proven to be NP-complete in the general case [8]). There have been few attempts for creating efficient scheduling algorithms in heterogeneous environments:

- in 1996, Yu-Kwong Kwok et al. propose a static scheduling algorithm for allocating task graphs to fully connected multiprocessors [22]. This algorithm is called the Dynamic Critical-Path (DCP) scheduling algorithm and is based on the computation of a critical path
- in 2002, Haluk Topcuoglu et al. present the Heterogeneous Earliest-Finish-Time (HEFT) algorithm [20], which is an adaptation of the classical (homogeneous) Earliest-Finish-Time algorithm to heterogeneous applications that relies on the Critical-Path-On-Processor (CPOP) algorithm. It has become a very popular algorithm and is often implemented in execution engines
- in 2010, L. F. Bittencourt et al. present an improvement of the HEFT algorithm where the task prioritizing algorithm also takes into account the short-term cost of a scheduling decision [4]
- in 2013, Hong Jun Choi et al. describe a new dynamic scheduling algorithm that is based on a history-based Estimated-Execution-Time (EET) for each task [10]. The global idea of the algorithm is to schedule each task on its fastest architecture, as long as there is no specific reason to do otherwise (e.g. work starvation for a certain type of worker)
- finally, in 2014, Yuan Wen et al. [21] compute the speed-up of each architecture (CPU and GPU) and use this metric directly to compute the priority of the task. This tends to execute as soon as possible the tasks with the highest architecture difference.

A common problem with scheduling algorithms (heterogeneous or not) is the generated overhead. Indeed, performing calculations in real-time to find efficient scheduling can take more time than the scheduling saved time. That is why the state-of-the-art scheduling algorithms in both heterogeneous and homogeneous cases tend to be intentionally simplified.

2.1.2 Heteroprio

Heteroprio has been developed with these challenges in mind. It has been submitted in the context of Bérenger BRAMAS' Ph.D. thesis in 2016 [5] [1]. It is a scheduler that has been integrated into StarPU [3]. StarPU is a task-based execution engine. Its goal is to create an abstraction for programmers and offer them a C interface for submitting tasks. In this execution engine, tasks can have multiple implementations, on the same, or different architectures. This makes it particularly convenient for designing heterogeneous applications.

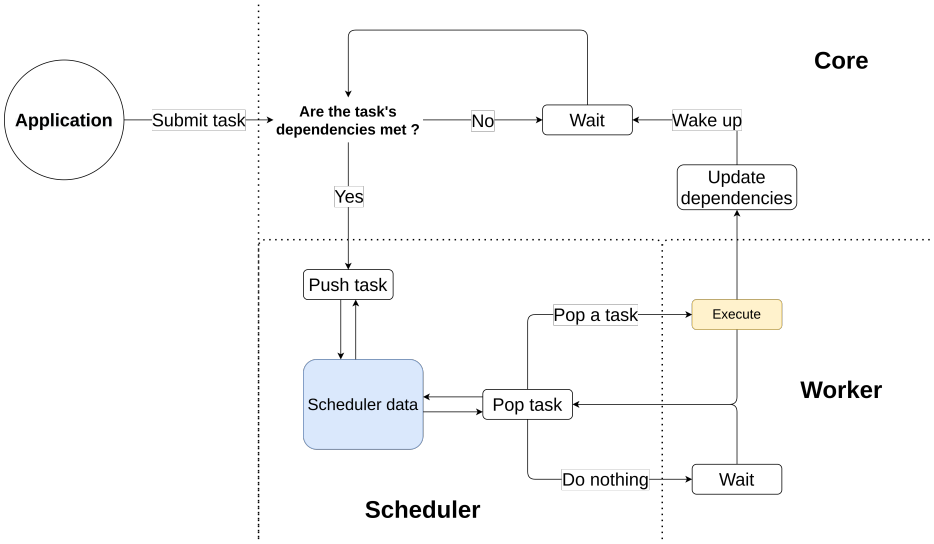

Figure 1: Schema of StarPU's internal functioning

StarPU has been designed so that the scheduler is a distinct component. A user can easily switch from one scheduler to another. Figure 1 shows how this component interacts with StarPU's modules.
In StarPU, schedulers rely mostly on two mechanisms: push_task and pop_task. push_task is called whenever a task becomes available (i.e. when all its dependencies are satisfied). pop_task is called when a worker is ready to execute a task. That can be because he just finished executing a task or because he has been idle for a certain amount of time and needs to check if he can now execute a new task. Usually, push_task will be used for keeping track of "ready" tasks, whereas pop_task will take the scheduling decision for each worker.

Heteroprio relies on an extremely simple mechanic. The scheduler stores a list of "buckets". A bucket is a FIFO queue of tasks. For each architecture (CPU, GPU, etc.), an indirect array is stored. This indirect array represents the priority of each task type on this architecture. When a task becomes available, it is pushed to its matching bucket. When a worker becomes available it retrieves the highest priority (based on the indirect array) non-empty bucket's next task. It is the user's role to define the indirect array for each architecture.
Figure 2 schematizes how workers select their tasks in Heteroprio. For simplicity purposes, CPU and GPU priorities are mirrored, but this is not necessarily the case. The two indirection arrays can be arranged in any order.

In 2019, an enhancement has been brought to Heteroprio [7], aiming to improve the task popping mechanism. The previous version treated all workers of the same type exactly equally (because of the FIFO queues). This can lead to problems in some cases.
Let us illustrate this with a problematic case: a sequential execution of a list of tasks. The execution is sequential because there is a single dependency between each task and its successor. This means that every task has one successor (except the last one) and one predecessor (except the first one).

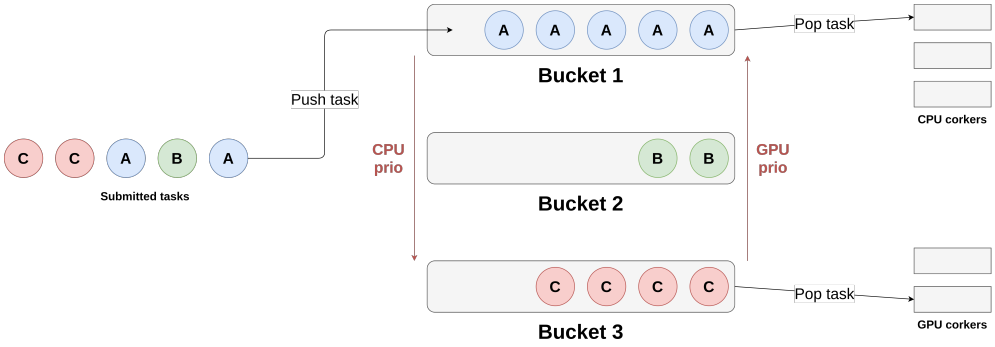

Figure 2: Schema of Heteroprio's principle

In this case, each task will be executed on a random worker (the first one that calls pop_task). If we have $N > 1$ workers then when a task has finished its execution on a given worker, its successor has only a $\frac{1}{N}$ probability of being executed on the same worker. That is, of course, problematic in practice because executing the successor on another worker is probably going to generate a slowdown due to the caused data transfer.
In Heteroprio's new version (which has been called LAHeteroprio), workers select their tasks not only depending on their position in the bucket's FIFO list but also depending on their affinity with the worker. The affinity is computed throughout multiple heuristics that the user can choose.

The great strength of Heteroprio is its simplicity, thanks to which its generated overhead is very limited. It also has a great heterogeneous scheduling potential, as long as the architecture's priorities are correctly set. The downside, however, is that the priorities have to be set by the user. The scheduler's efficiency will be highly correlated to the user's ability to set adequate priorities.

It is with this limitation in mind that this project has been issued. Fundamentally, indeed, priorities do not necessarily need to be set by a human.
The goal of this work is to find a way of setting efficient Heteroprio priorities automatically.

3 Problem formalization

Directed Acyclic Graphs (DAGs) are a common way of representing the dependencies between tasks in a program. The objective of the DAG scheduling problem is "to minimize the overall program finish-time by proper allocation of the tasks to the processors and arrangement of execution sequencing of the tasks" [14].

There are variations of this objective. Some research has been conducted towards reducing the Mean Finish Time (or mean time in system, or Mean Flow Time) which is the average finish time of all tasks [9], [15]. The MFT criterion has the advantage of tending to minimize the memory required to hold incomplete tasks. The overall finish-time, however, remains the most natural and widely used metric for evaluating scheduling performance.

3.1 Definitions

There are multiple ways of modeling a scheduling problem. We will describe one formalization that is the most widely used throughout the scientific documentation.

We represent an application with a DAG (Directed Acyclic Graph), $G = (V, E)$, where V is the set of nodes and E is the set of edges. Each node represents a task and each edge represents a dependency.
A dependency is a relation between two tasks (written $\rightarrow$). If $A \rightarrow B$ then B can not start its execution before A is finished. We can note that a graph that matches an application is necessarily acyclic. This is because the dependency relation is transitive (if $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$) and irreflexive ($\forall a, \neg(a \rightarrow a)$).
In practice, dependencies are typically deduced by the execution engine depending on the task submission order, data accesses, and their types (read, write, read-write, commute, etc.). Indeed, two workers can read at the same time but not write, therefore the access type will eventually impact the dependency structure.

In this formalization, we assume that we have q processors and v tasks. W is the computation cost matrix. The cost can be the estimated execution time as in our case, but it also can be any metric (e.g. energy consumption, [23]). This computation cost matrix is a $v \times q$ matrix, where $w_{i,j}$ is the estimated execution time of task n_i on processor p_j.

To take account of data transfer, we suppose we have the Data matrix which is a $v \times v$ matrix where $Data_{i,k}$ represent the amount of data required to be transferred from task n_i to n_k. The B matrix is of size $q \times q$ and $B_{i,k}$ is the estimated transfer rate from p_i to p_j.

Finally, the communication startup cost is represented by the L vector of size q. L_i is the communication startup cost of p_i.
We can now define the communication cost of the edge (i, k) on one scheduling:

$$c_{i,k} = L_m + \frac{Data_{i,k}}{B_{m,n}}$$

Where m and n represent, respectively, the chosen processor for n_i and n_k.
For presenting the objective function, we introduce two attributes: Actual Start Time, and Actual Finish Time (AST, and AFT).
$AST(n_i)$ and $AFT(n_i)$ represent, respectively, the actual start time and finish time of the n_i task. If we suppose that a task's actual execution time is constant and that it matches the W estimation matrix, then $AFT(n_i) = AST(n_i) + w_{i,j}$ (with p_j being the actual executing processor for task n_i). $AFT(n_i)$ depends on the scheduling decision.
Finally, the *schedule length* (or *makespan*) is defined as the finish time of the last task:

$$makespan = \max_{n_i \in V}(AFT(n_i))$$

We now have an objective function. A schedule is an assignment of tasks to processors. The best schedulings are the ones that minimize the makespan.

3.2 Heteroprio

Heteroprio adds multiple specificities compared to the general scheduling problem.

Applications often work with two types of processors, usually CPUs and GPUs. Hence, numerous state-of-the-art schedulers limit themselves to this situation. It is the case of Heteroprio, which only considers CPUs and GPUs. This is convenient since we do only need to handle two execution times (costs). We, therefore, choose to use a new notation for referencing costs. w_i^{arch} represent the cost of task n_i on architecture "$arch$" (CPU or GPU). Since we have only two types of architectures, we can use "$\overline{arch}$" for referencing the other architecture.

Heteroprio also has the specificity of grouping tasks by "type". The type of a task is an adjustable attribute that is chosen by the user. Usually, tasks are grouped depending on their codelets (their inner code). The idea behind this is to consider that each task of the same type should be treated equally.

This grouping corresponds to Heteroprio's inner mechanism of *buckets*. As explained in 2.1.2, each *bucket* is a FIFO list of tasks. If a new task of a certain type is pushed to the scheduler, it will necessarily be executed after the tasks that are already in the bucket.
This is a constraint on our scheduling flexibility, but also has the advantage of avoiding starvations. Starvation is a situation where processors lack work. This can be caused by an important task never being executed because other ones take the priority.

Since our goal is to automatically find priorities for Heteroprio, it is important to keep these specific features in mind. The goal of this internship is to find heuristics for automatizing Heteroprio and its functioning should, therefore, not be changed.

3.3 Example

For understanding the theoretical principle of Heteroprio, let us consider an example DAG (figure 3) and each task's associated cost (table 1).

Task \ Architecture	CPU	GPU
A	1s	2s
B	2s	1s
C	1s	1s

Table 1: Example execution times of tasks

We have three task types (A, B, and C). We suppose that within a type, all tasks have the same costs. Let us suppose that we have 2 CPU workers and 1 GPU worker available. For simplicity, we will assume that the workers are select their task in a predefined order: CPU-1 pops a task if there is one available, then GPU-1, and then CPU-2. In practice, the order in which tasks are popped is unpredictable, but there is a "prefetch" mechanism to avoid that a worker pops tasks when it is slower than

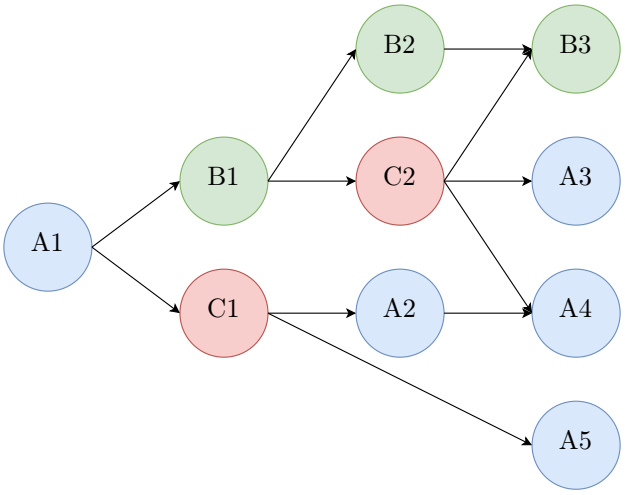

Figure 3: Example DAG

its "co-worker".

What priorities would be best suited for this problem? Ideally, we would want to execute the tasks on their favorite architecture. We would also want to minimize the idle time of workers. To do so, we want to release as much parallel work as possible. One way to help this is to give a high priority to tasks that have numerous successors.

We can make some immediate observations. "A" tasks seem to be better suited for CPU, who will execute them twice as fast, whereas "B" tasks seem better suited for GPU. The "C" tasks do not, at first glance, seem to have particular affinities. We can also note that whatever priorities we set, A1 will always be executed by CPU-1. Finally, if we go deep enough in the execution, we will see that the C2 is of great importance since it has 3 successors.

Let us test what happens under three different priorities. In table 2, we show three different test cases.

Case \ Architecture	CPU	GPU
1	B-C-A	A-C-B
2	A-C-B	B-C-A
3	C-A-B	B-C-A

Table 2: Example priorities

In case 1, B is the most prioritized task type for CPU, and A is the less prioritized. For the GPU, it is the opposite. For both processors, C is the median priority. In this case, we intentionally set bad priorities by promoting the slowest architecture for each task type.
For case 2 and case 3, we promoted the fastest architecture. The difference between the two is that case 2's priorities are mirrored compared to the ones of case 1,

whereas in case 3, we exchanged the C and A types in the CPU. The reason for the swapping in case 3 is to favor the execution of C2 which has a lot of successors.

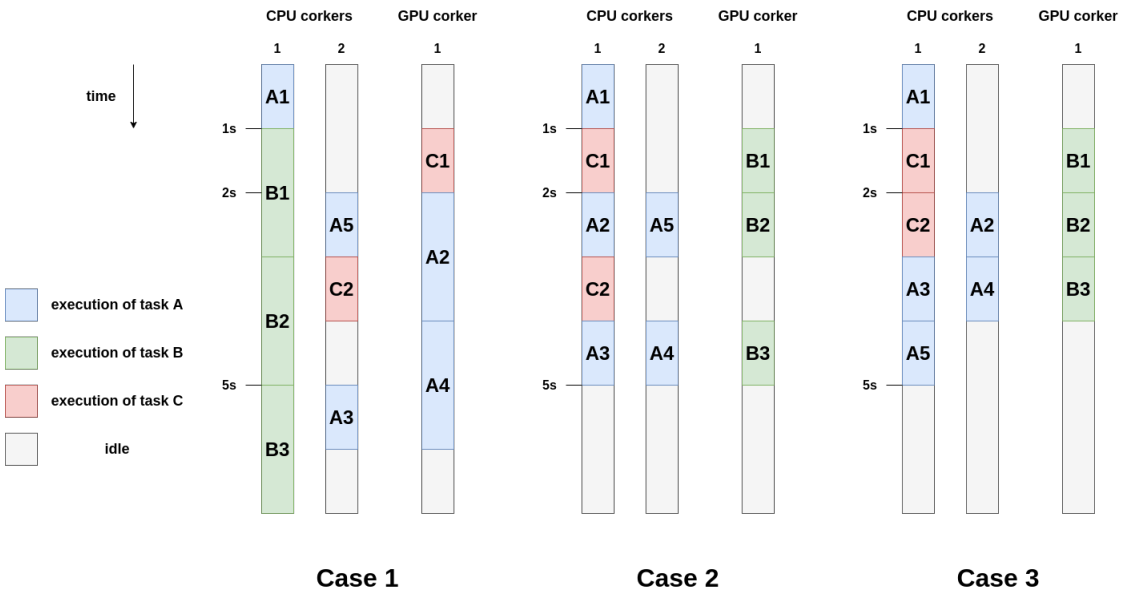

Figure 4: Example executions for the 3 cases

The three executions are schematized in figure 4.
Unsurprisingly, we find that the makespan of case 1 is slower (7s) than case 2 and case 3 (5s). In our simplified example, case 2 and case 3 are equivalent in terms of schedule length. In case 3, however, CPU-2 and GPU-1 are freed sooner (after 4s of execution) than in case 2, where they are working until 5s of execution. Case 3 could therefore potentially be better if we added more tasks to the graph, and shows the difficulty of finding heuristics automatically. Indeed, some tasks should be prioritized depending on their execution time, but others should be prioritized because they have particular importance in the execution graph (as C in our example).

3.4 Discussion

This theoretical model has notable issues that we would like to discuss in this section.

3.4.1 Combinatorial explosion

As we can intuitively see in 3.3, scheduling problems are combinatorial problems. It has been proven that DAG scheduling problems are NP-complete in the general case.

As often in combinatorial problems, the goal is to find approximate solutions without exploring all the possibilities. In Heteroprio's paradigm, we do not look at the structure of the graph in its entirety. Instead, we work at the task's level. In this aspect, our work differs from other schedulers such as the HEFT scheduler for example, which performs look-ahead operations to take its decision.

We, therefore, aim at inventing heuristics that rely only on task characteristics and, eventually, local graph analysis. We avoid involving information that implies analyzing the graph's structure. Doing so ensures that Heteroprio's overhead remains limited.

3.4.2 Execution time

We suppose that we know the execution time of each task.
In practice, however, we do not have any guarantee to know this information. In StarPU, the user has the choice of associating a "perfmodel" to each task type.

A perfmodel is a tool that gathers information about each task's execution and uses it to predict the next executions. The efficiency of the prediction depends on too many parameters to always be considered correct (although in most cases the predictions are accurate).

One problem is the type of regression the user chooses to perform. For a constant (or near-constant) execution time, he can choose a history-based perfmodel which will output the average of previous executions. But the execution time is often correlated with its data size and a more complex regression needs to be performed (the efficiency then depends on the relevance of the chosen regression).
In other cases, the execution times are simply too dispersed for the prediction to be close to the actual execution time.

In addition to this, in our theoretical model, we consider that each group of task types has the same execution times within itself (see figure 3 and table 1). This is an approximation that is convenient for finding heuristics but is rarely true in practice.

We postulate, however, that this lack of accuracy will not significantly impact the correctness of the heuristics.

3.4.3 DAG representation

Another problem with our model is that we consider our application to be a DAG.

A DAG is the most standard way of representing a task-based application. Indeed, from the most memory accesses types (READ, WRITE, READ-WRITE), we can always deduce a corresponding DAG. But some accesses can not be transcribed in terms of direct static dependencies.

For example, StarPU has a memory access type called STARPU_COMMUTE which represents an access that can be performed successively and in any order by multiple tasks but not at the same time. A simplistic example of this would be some different operations that require to increment a shared counter. Incrementing the counter can be done in any order, but not by two tasks at the same time.
This access mode has been used in mathematical applications, e.g. for an optimized discontinuous Galerkin solver [6]. The dependencies can only be deduced at execution time: if no task is commuting on the data, any task can take the memory node, and if one task is commuting, the memory node is blocked.

Thus, our theoretical model can lack information on some applications which use these relatively uncommon memory access types. In practice, in our heuristics, these

accesses are treated like write accesses.

We assume that these types of access are rare enough so that the scheduling performance is not significantly impacted.

3.4.4 Static vs. Dynamic scheduling

In static scheduling, the computation is performed before the execution whereas, in dynamic scheduling, the decisions are taken during the execution. There can be different degrees of static and dynamic scheduling, e.g. in [12] where a hybrid static/dynamic scheduling is performed for dense matrix factorization.
In practice, high-performance applications tend to lean towards dynamic scheduling, because their task number and dependency structure's complexity become too high.

This forces one to be cautious about scheduling results obtained in a theoretical model. Indeed, there is no clear equivalence between the static scheduling of a theoretical graph and the real-time scheduling of an application. The dynamic scheduler has a local view of the problem. It has clear information about the current state of the program (ready tasks, idle workers) but little information about the rest of the program. Its primary purpose is not to extract global characteristics from a graph.

Finally, in dynamic scheduling, the computation cost of the scheduling directly impacts the execution time. This additional cost is called the scheduler's overhead.

It is, therefore, crucial to minimize this overhead to release an efficient scheduler.

4 Research of heuristics

4.1 Spetabaru's simulator

Spetabaru is a task-based runtime system. Its main feature is that can handle speculative execution. That is to say, it can handle multiple states for data depending on the speculation paths. A simple C++ execution simulator is delivered with it.
This simulator works by using false clocks for simulating elapsed time. The user can choose the GPU and CPU worker number and task priorities. At the beginning of the execution, each clock is set to 0. Once a worker becomes available, it pops a task according to the priorities. It then adds the task's time to its clock and the task's successors are added to the ready tasks if all their dependencies are fulfilled.
When all the tasks have been treated, the simulator returns the total execution time. This execution time is the maximum of all the workers' clocks.

Using this simulator is very convenient for finding heuristics. It can be viewed like a black box where we input priorities and get an execution time as output. We, therefore, chose this tool as a base for elaborating our experimental protocol.

4.2 Proposed protocol

We wanted to set up a clear protocol for evaluating our heuristics. This protocol has two purposes. Firstly, for us, it is a significant help to have a clean methodology. It gives us confidence in our theoretical results and saves us time when our protocol

clearly shows that a heuristic is slow. Secondly, it will give strength to our heuristics if they perform as good in real applications as in the protocol.

4.2.1 Graph generation

To be able to evaluate our heuristics, we aim at generating a dataset of 32 fake applications that are as close as possible to real-life programs.

Directly using real-life applications would be possible but would have two notable downsides. Firstly, it would be tedious to get the graph of an application and transform it into a graph that is understandable by the simulator. Secondly, our dataset would likely lack diversity, as application designs presumably contain biases (e.g. three sequential tasks can be merged into a single task or two sequential tasks, this decision is entirely up to the developer). Lacking diversity would be problematic in terms of generalization potential. We, therefore, opted for a randomly generated dataset.

Han Lin et al. present a way of generating DAGs for heterogeneous scheduling evaluation purposes [10]. An implementation of this algorithm has been created by Frederic Suter [18]. It has 5 hyperparameters (n, fat, regularity, jump, and CCR).
This graph generating method was our first idea for generating our dataset, but it has several downsides. It does not handle task types. Therefore each task is treated individually. In a real application, however, there are usually different task types. Besides, there are often correlations between a task's dependencies and its type.
For example, an application can have a regular pattern where task A performs a matrix inversion and then 16 tasks of type B read the matrix. In this case, A tasks would systematically have 16 successors of type B, and conversely for B tasks. This kind of behavior is not represented at all with the presented DAG generation method.

We, therefore, chose to use a custom graph generation method. The exact functioning is complex, but we will present the main idea here. The tasks are created through a false execution, in such a way that each time a worker becomes available, a task appears. The appearing task is always of a type that has the fastest execution on the worker's architecture. The idea behind is that if all workers are always executing a task where they are the fastest, then the false execution is perfectly scheduled.
Thus, we know the best possible scheduling, its makespan, and have therefore a lower boundary for the graph's execution time.

To improve the resemblance of the generated DAGs to real applications, we introduce the concept of predecessor matrix. A predecessor matrix P is of size $v \times v$ and $P_{i,j}$ is the average number of predecessors of task type i that have the j type. Our graph generation method uses a predecessor matrix as input and adjusts the predecessors of newly created tasks so that they match the matrix's values. This lets us create random graphs that have correlations in their dependencies, as real-life applications.

Finally, we add an expected proportion for each task type. This expected proportion will guide the graph generator during its choice of task types. This feature lets us have control over the quantities of each task type. It is important to

have a mechanism that avoids having an equal repartition of all task types because it rarely occurs in real applications.

Our generator takes multiple parameters as input:

- a seed for random number generation
- the total number of tasks
- a list of task types, with their associated CPU and GPU cost and expected proportion
- number of CPU and GPU workers
- a predecessor matrix

We have generated a set of 32 randomized parameters. These parameters have been used to generate 32 graphs that form our dataset.

We can now use this dataset of false applications for evaluating a scheduling.

4.2.2 Finding control priorities

Finding control schedulings was central for us. Control schedulings have the advantage of giving us an anchor point during the process of finding heuristics. They can also help us understand why some scheduling paths are better than others. Finally, It is an additional argument for our work if we can compare our heuristics against solid control schedulings.

Our graph generation method is convenient because it instantly gives us the best possible scheduling. However, Heteroprio works in a specific way that discards most scheduling possibilities.
To convince ourselves of this fact, let us consider a graph of 32 tasks with no edges (no dependencies), same execution times, and two different types: A and B. If we have only one worker, there are $\binom{32}{1} = 32! \approx 2.63 \cdot 10^{35}$ scheduling possibilities. The scheduling decisions Heteroprio can take depend on its set priorities. Since we have CPU/GPU priorities and two task types, there are only 4 possibilities. Actually, in every situation, Heteroprio has always exactly $(t!)^2$ possible schedules (where t is the number of different task types). Numerous schedulings are, therefore, impossible with Heteroprio.

We can, therefore, not hope to systematically find priorities that perform as well as the perfect scheduling. With this in mind, it seems inopportune to compare a makespan that comes from Heteroprio's restrained paradigm with any other makespan. We, therefore, made an effort to find a proper way of finding relevant control makespans. These makespans have to result from a Heteroprio execution in the simulator.

As far as possible, our control executions should be as optimized (minimize the makespan) for each graph of the dataset (the method should be the same for each graph). Our first idea was to perform an exhausting search for each graph. I.e. all possible priorities are tested and the ones that minimize the makespan become the control priorities. Unfortunately, this was impossible because some graphs contained 8 different tasks, which lead to $(8!)^2 \approx 1.63 \cdot 10^9$ possible executions, which was too

much for our machines to execute.

We, therefore, opted for a compromise. Instead of exploring all of the possibilities, we use an iterative optimization algorithm. We begin with any CPU and GPU priorities. We perform multiple iterations, alternating between CPU and GPU. At every iteration, we try all possible permutations for the current architecture (CPU/GPU) and keep the best. If there is a tie, we randomly choose between the bests. We stop after 3 pairs (CPU-GPU) of iterations because on our dataset it seemed sufficient for finding near-perfect solutions.
We can verify that 3 pairs of iterations are enough by comparing the best priorities against the perfect scheduling, that we have thanks to our graph generating method. On some graphs, the found priorities are equivalent to the perfect scheduling in terms of makespan. On others, however, the found priorities are significantly slower than the perfect execution, but that is probably attributable to Heteroprio's functioning (which can discard important scheduling paths).

Our method for generating control priorities can be disputable. We have no guarantee that with enough iteration we will find the optimal solution. We know, however, that with enough iterations there is no priority permutation inside a single architecture that is better than the found solution. It also counter-intuitive to use a step-by-step descent on a sequence but we assume that close sequences (in terms of distance [11]) will have close execution times.

Now that we have control priorities we have an anchor point for evaluating our heuristics, as well as a set of strong priorities that can be analyzed to find new heuristics.

4.2.3 Discussions

There are some approximations in our protocol that we would like to discuss in this section.

The first that we would like to mention is that we falsely assume that each task type has a CPU and a GPU cost. Nonetheless, not all applications give both CPU and GPU implementations for all tasks. E.g. if the GPU is extremely slow on a particular operation, it is a justifiable choice not to write a GPU kernel.
In our dataset, we did not include any case where a task could only be executed on one processor type. This is an obvious bias, but taking these situations into account would significantly increase the problem's complexity. We, therefore, chose to keep it simple and assume that every task has a CPU and a GPU implementation.
In real executions, we found two workarounds for addressing this issue. We can either consider the cost of the missing architecture to be extremely high or consider it to be the same as the other architecture.
There are two different ideas with these two workarounds. In the first case we consider that since the task can not be executed on an architecture, it is as if it can be executed but at a tremendous cost. In the second case, we consider that a heuristic should not be biased by the cost of the missing architecture.

Another defect in our protocol is that we do not take into account data transfer times (mentioned in 3.1). We decided not to add a data transfer model in the simulator because it did not seem to have significant importance in our problem.

Indeed, data transfers occur during the execution and depend on which worker executed which task. In real executions, it should not be problematic, because Heteroprio has a feature (LAHeteroprio) for optimizing data transfers.

Finally, there is an inevitable bias in our dataset. We endeavored to mimic the behavior of real-life applications by adding affinity between task type dependencies. However, we use numerous hyperparameters for calibrating our graph generation. These hyperparameters are generated randomly, but the range of acceptable values has been set by us. Hence, we can only guarantee that our dataset has a good diversity, but not that its structure resembles any real-life application.

4.3 Pursued lines for finding heuristics

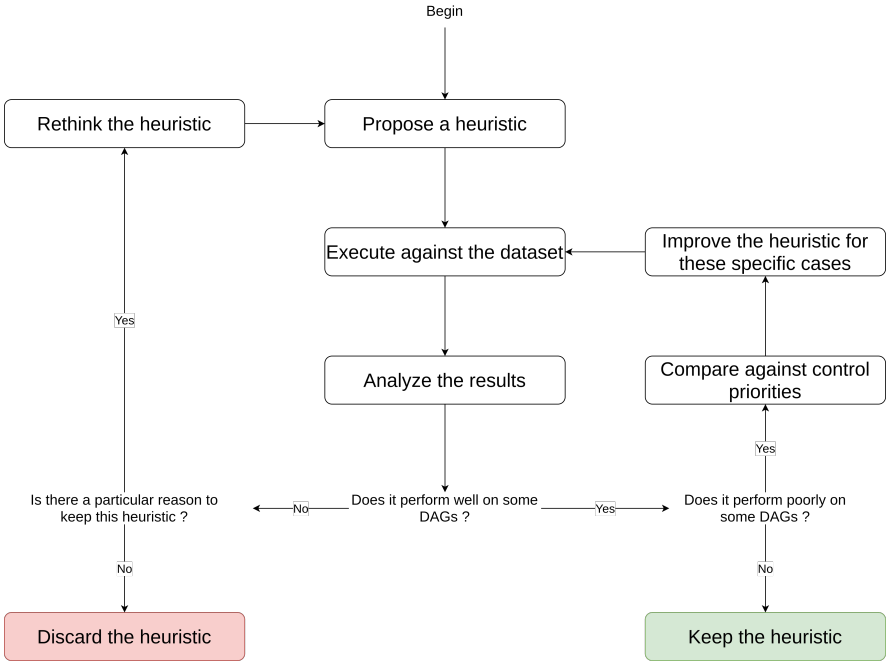

Figure 5: Schema of our proceeding method for finding heuristics

In this section, we will present different lines that we explored for finding new heuristics.
Thanks to our improvements to the simulator, we have an impartial method that gives us a heuristic's score. Indeed, if we want to test a heuristic, we run it on each DAG and take the average slowdown compared to the control priorities. The lower the slowdown, the better our heuristic is in theory.
In figure 5, we present a schema of our method for evaluating and improving our heuristics.

In the following sections, we will present the most interesting lines we have investigated.

Worker \ Task	A	B
CPU	100s	1s
GPU	130s	10s
relative difference	×1.3	×10
absolute difference	30s	9s

Table 3: Tasks costs and time differences

4.3.1 Relevant metrics

An early concern we had was to look for features that could be extracted on the local scale (without exploring all the DAG). In this section, we will present the different features we decided to use.

CPU-GPU difference The difference between CPU and GPU costs plays a central role in an efficient task priority generation. Indeed, we can have the qualitative reasoning of saying that if a task is fast on the CPU and slow on the GPU it should be executed on the CPU.
This cost difference can be expressed either with the relative or absolute cost difference, but the two metrics do not seem equivalent. To explain this, let us consider table 3 as an example.

We consider that a GPU is free and can execute tasks A and B. Two metrics are shown: the relative and the absolute difference. Depending on which metric we consider, we will make a different conclusion. The relative difference would suggest executing A is a better choice (it is reasonable to let the higher ×10 acceleration for a CPU). The absolute difference would suggest executing B is a better choice (because we would loose 9 seconds instead of 30 seconds).

We can not categorically say that one metric is better than the other. We tend to think that the absolute cost difference is the most suited metric because it showed better overall results when used in our heuristics.
It also seems to be a natural way to think about the cost of making the wrong decision. E.g. in our example in table 3, if a GPU worker decides to execute task A, the execution will lose exactly 30 seconds compared to if a CPU executes it. The relative cost is still an acceptable metric that is used in some of our heuristics.

We will use the following notations for expressing these two metrics:

$$diff_{arch}(v_i) = w_i^{\overline{arch}} - w_i^{arch}$$

$$rel\_diff_{arch}(v_i) = \frac{w_i^{\overline{arch}}}{w_i^{arch}}$$

NOD While assessing the state of the art on heterogeneous scheduling we discovered the concept of Normalized Out-Degree (NOD) [16]. The NOD is given by the formula:

$$NOD(v_i) = \sum_{v_j \in succ(v_i)} \frac{1}{ID(v_j))}$$

Where $ID(v_j))$ is the inner degree of task v_j (i.e. its number of predecessors). This quantity gives us key information. It can be viewed as "how much tasks will be released". In other words, it is a metric for measuring the parallelizing potential of a task.
It is implied, that releasing $\frac{1}{ID(v_j))}$ of a task v_j is as if it was partly released, at a proportion of $\frac{1}{ID(v_j))}$. This seems reasonable from a heuristical point of view. Releasing 2 tasks at a "ratio" of $\frac{1}{2}$ can be view as being equivalent to releasing 10 tasks at a ratio of $\frac{1}{10}$. We, therefore, chose to include this metric to some of our heuristics.

There are, however, downsides to using the NOD. It does not take into account the type of tasks that will be released. There are cases where this can be extremely important. For example, if we are lacking GPU jobs (starvation), what is relevant is not the number of released tasks, but the number of GPU released tasks (because we are lacking GPU work).

We, therefore, propose the Normalized Released Time metric.

Normalized Released Time We introduce the concept of Normalized Released Time (NRT). This metric, derived from the NOD aims at measuring how much CPU and GPU busy time will be released. We propose the following NRT definition:

$$NRT_{arch}(v_i) = \sum_{v_j \in succ(v_i)} \frac{P(exec(v_j, arch)) \cdot w_j^{arch}}{ID(v_j))}$$

where $P(exec(v_j, arch))$ is the probability that v_j is executed on architecture $arch$ and w_j^{arch} is the cost of v_j on $arch$.
Estimating the probability that a task is executed on one architecture before the execution may be impossible. However, we can measure the proportion execution of each task type during the execution and use this proportion as an approximation of the probability in our formula.

This formula has two upsides compared to the first NOD formula. First, it takes into account the cost of the released (or "in release process") successors. It is presumably better to release N tasks with a cost of 10 seconds, than N tasks of 1 second (it means our workers will have more work).
Secondly, we make a difference between CPU and GPU time. This difference is crucial in a heterogeneous system. Let us suppose, for example, that our CPU workers are in great abundance of tasks, but our GPU workers are idle 90% of the time. In this case, we would ideally want to favor tasks that release GPU work.

Having a NRT formula for both CPU and GPU gives information about where the released work is likely to be executed.

Useful Released Time We use our concept of Normalized Released Time to create a new metric: the Useful Released Time (URT). We propose the following definition:

$$URT(v_i) = NRT_{CPU}(v_i) \cdot IDLE(CPU) + NRT_{GPU}(v_i) \cdot IDLE(GPU)$$

where $IDLE(arch)$ is the idle proportion of $arch$ workers over all the execution.

The primary goal of this metric is to represent how much useful time will be released after the task has finished its execution. By "useful time", we mean work time that "has a great chance of giving work to idle workers".
To do this, we ponderate the Normalized Released Time (NRT) of an architecture with a quantity representing how needed the architecture is. The idle proportion of an architecture's worker seemed a natural quantity to use in this case. For example, if CPU idle proportion is 0% and GPU idle proportion is 100%, we will only take NRT_{GPU} for evaluating the Useful Released Time (i.e. the more GPU time is released, the more critical the task is), which does seem to be a reasonable choice.

By itself, this metric is not sufficient for affecting priorities to Heteroprio. Indeed, it only gives information about the successors of a task. It is, however, a useful indicator for evaluating the criticality of a task. The goal of the heuristics that use this metric will be to combine this quantity with the cost difference in a smooth way.

Discussion Since we suppose each task of the same type always has the same CPU and GPU costs, the CPU-GPU difference does not change within a type. The NOD, NRT, and URT, however, have no reason to be the same for each task of the same type. E.g. a task A has 3 successors ($NOD > 0$) in the middle of the DAG, but another task A at the end has 0 successors ($NOD = 0$). They can (and should), however, be correlated to the task type due to the predecessor matrix we input to the DAG generator.

We need to have a single value for these metrics on each task type. To do so, we chose to take their average value throughout the graph. This allows us to work with single quantities for each task type. This way of proceeding is, of course, disputable. The distribution should also be taken into account, in addition to the average.

Nevertheless, given Heteroprio's functioning, it is reasonable to assume that each task type has only one NOD, NRT, and URT. But it should be kept in mind, that there could be room for improvement on this specific point.

4.3.2 Priority generating method

The output of a priority generator should be two sequences of integers: one representing task type priorities on the CPU, the other on the GPU. Handling sequences of integers create difficulties for our problem.
Indeed, it is not straightforward what process is the best suited for transforming a set of real metrics (execution time, NOD, etc.) into a sequence of integers.
In this section, we will present two priority generating methods.

Iteration by criticality The first method is to affect the task types one by one in the output priority list. We will use figure 6 for explaining the functioning of this solution.

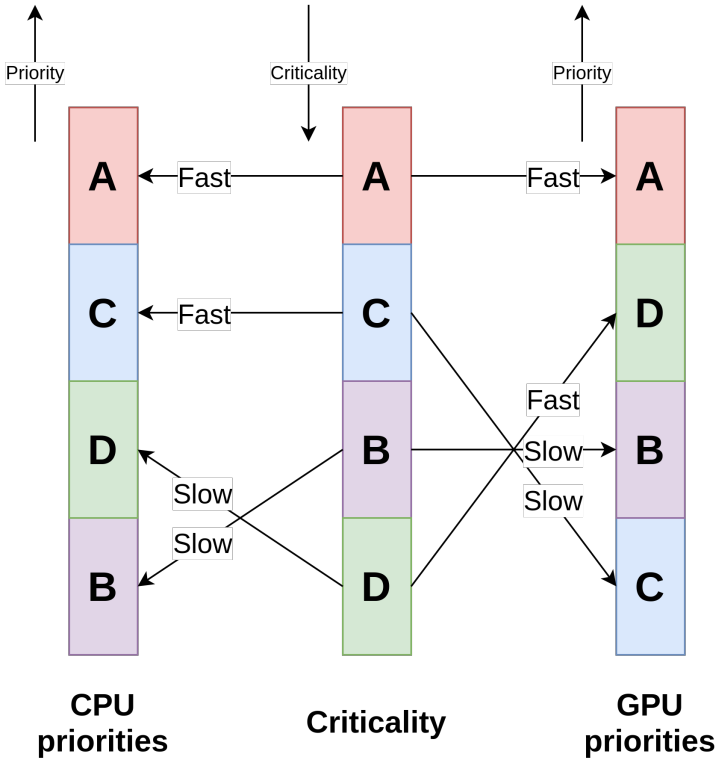

Figure 6: Schema of a prioritization by criticality

This method relies on two parameters: a criticality list and a slow/fast boolean for each pair (task type, architecture). The criticality list is iterated in descending order. Each task type is put either in the highest or the lowest remaining priority for each architecture.
For example, A can be placed in the first or fourth position in the CPU priorities. C can be placed on the second or the fourth (because the first place has already been taken by A). The decision is taken depending on the slow/fast boolean list.

We have found multiple downsides to this method. There are only two possibilities for the slow/fast boolean. This can make tasks jump from the highest to the lowest priority, while we would ideally want to be able to express nuances between "fast" and "slow". This is why we did not further explore this idea, though we kept one heuristic that is based on this concept.

Criticality per architecture The second idea that we have found for generating a priority list is to compute a score for each pair task type/architecture. The priority lists are then sorted so that each list is sorted by descending score. This method is schematized in figure 7.

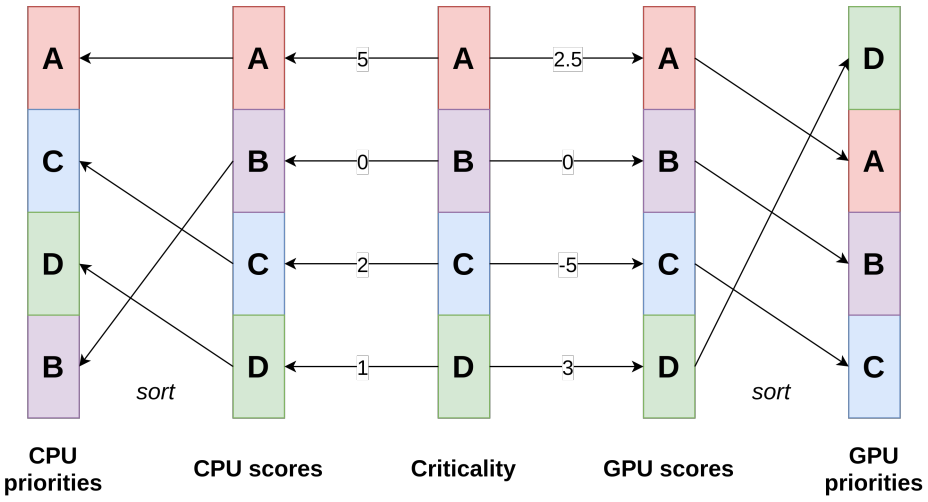

Figure 7: Schema of a prioritization by score ordering

Proceeding by score sorting seems more natural than the previous method. The score indicates how critical the tasks are on an architecture and the sort allows us to switch from continuous scores to discrete priorities.
Another upside is that this method allows any possible permutation choice for priorities, whereas the first one could only output a restricted subset of these.

Thanks to this method, we can now restrict our heuristics to a function that outputs a score for each task type and architecture. This is convenient compared to our first method that required both a score and a boolean.

4.3.3 Cost normalization

For prioritizing tasks, we have access to multiple information. Important features of a task are its CPU/GPU executions costs. Directly working with these costs can be problematic.

We can prove that scaling every task cost by a factor of α does not change the solution to the best scheduling problem. This means that the DAG scheduling problem is equivalent between two identical graphs when their associated costs differ by a ratio of α.

If we want our heuristics to be coherent in view of this result, they should output the same priorities if we multiply all the costs of a graph by α. One idea that springs to mind is to normalize the costs so that whatever scaling factor we have, the normalized costs are the same. Yet, normalizing a set of heterogeneous costs is not a trivial task. We have come with different normalizing methods.

The first idea is to normalize the costs so that the average task type cost is 1. We can express this trough the formula:

$$z_{i,j} = qt \cdot \frac{w_{i,j}}{\sum_{0 \leq i < t} \sum_{0 \leq j < q} t_{i,j}}$$

Where $t_{i,j}$ is the cost of task type i on processor j.
The merits of this normalization are disputable. Each task type has an equal weight in the sum. This can be problematic if the task type distribution is highly uneven. Let us consider 1 task A with a cost of 999s, 999 tasks B with a cost of 1s, and only one type of processor. Applying our formula would give a cost of $\frac{999}{500} \approx 2$ for A and a cost of $\frac{1}{500} = 0.002$ for B (without unit, because normalized). In this case, the average task would have a cost of $\frac{0.002 \times 999 + 2 \times 1}{1001} \approx 0.004$, which is far from the average "1" we get from task types. This is not necessarily a problem, but to avoid biasing our heuristics, we created a new normalization:

$$z_{i,j} = qv \cdot \frac{w_{i,j}}{\sum_{0 \leq i < v} \sum_{0 \leq j < q} w_{i,j}}$$

The difference with the first normalization is that the costs are now normalized over all the tasks, and not over task types. With this normalization, we have the guarantee that if only one processor type is available, the average cost of tasks is 1. Once we have multiple processor types, the average cost is 1 if the processor choice is equiprobable.

In a real execution, however, the tasks are not assigned with equiprobability. We would rather expect the tasks to be assigned to their best processor (the one with the lowest execution time).
In an attempt to improve this heuristic, we propose the following formula:

$$z_{i,j} = v \cdot \frac{w_{i,j}}{\sum_{0 \leq i < v} \min\limits_{0 \leq j < q} (w_{i,j})}$$

Instead of taking the average execution time for each architecture, we only take the best (the one that minimizes the cost). With this method, we have the guarantee that if each task are scheduled on their faster architecture, then the average normalized execution time of the task is 1.

The postulate of this method is certainly inexact. Tasks are not always scheduled on their best architecture. However, over the three normalization methods, this one seemed to produce the less biased results. It is resistant to inequal task type distribution and disparate CPU/GPU costs.

We, therefore, chose to systematically use this normalization method when applying our heuristics.

5 Found heuristics

In this section, we will present a non-exhaustive list of heuristics we have implemented. Some have been selected thanks to their efficiency in the simulator, whereas others due to their conceptual benefit.

5.1 Best NOD on best architecture

The NOD seems to be a powerful indicator of how critical a task is. We would ideally want the tasks that have the best NODs to have the highest priorities on at least one architecture. In these heuristics, the expected goal is to affect tasks with higher NODs to their favorite architecture.

Smoothened best architecture To meet this criterion, we implement a first method based on the first algorithm described in 4.3.2. We iterate through all the task types by descending NOD. For each type and architecture, we affect it:

- to the highest available priority on the architecture if it is the fastest one
- to the lowest available priority on the architecture if it is the slowest one
- to the highest available priority on the architecture if the cost is the same on CPU and GPU.

To handle the third case more properly, we can define a range in which we assume the costs are close enough to consider that they are equal. This avoids having a situation where a task is sent to the lowest available priority on one architecture because it is 1% slower (which would usually be excessive).

The benefit of this heuristic is that the most critical tasks are always scheduled on the highest priority of at least on architecture (assuming the NOD is a good indicator for evaluating the criticality). It also has the advantage of not requiring any external parameter.
Its downside, however, is its abruptness. Indeed, a task is either considered "fast" or "slow". Thus, as we increase the CPU/GPU cost difference, there will be a point in which a task moves from the highest to the lowest priority.

This is strange behavior, but this heuristic has actually performed better than all our other ones on two graphs of our dataset. We can, therefore, safely say that this heuristic is worthy of interest, even if it is not the most solid we have found. It is also, presumably, efficient in cases where cost differences are especially different.

Smoothened best architecture To lower the impact of brutally switching a task from highest to the lowest priority depending on if it is the fastest or the slowest on the architecture (regardless of how close the two costs are), we created a smoothened version of this algorithm, so that if the CPU cost and the GPU cost of a task are close, the highest cost still gets a chance to have a decent priority.
Unlike the first method, this heuristic is based on a score, whereas the first was based on an entire algorithm (as explained in 4.3.2). The score is computed as so:

$$score_{arch}(v_i) = \begin{cases} NOD(v_i) & \text{if } diff_{arch}(v_i) \geq 0 \\ NOD(v_i) \cdot (2e^{-\alpha \cdot diff_{arch}(v_i)^2} - 1) & \text{otherwise} \end{cases}$$

If the cost difference is adverse, the score will continuously decrease. The speed at which it decreased is controlled by the α parameter. As this parameter increases,

we tend towards the non-smoothened version of this algorithm.

Unfortunately, this smoothened method did not show any improvement in our dataset. The found priorities are often the same, and if we increase α, we get worse average execution times.

5.2 NOD-difference combination

One of the early ideas we have tried is to combine the NOD with the cost difference metric (see section 4.3.1). This is not straightforward, as we work on two independent metrics. Indeed, the NOD depends on a task's dependencies whereas the cost difference depends on the CPU/GPU costs of a task.

We can safely say that a high NOD should lead to a high priority on both architectures and that a high difference should lead to a high priority on the best architecture and a low priority on the other one.
The challenge is to find a score that combines these two metrics in a relevant way (which is measured against our dataset).
For example, let us assume that we have two tasks A and B. Task A has an average NOD of 2 and a cost difference of 1 in favor of the CPU architecture. Task B has an average NOD of 0.5 and a cost difference of 5 in favor of the CPU architecture.
The two tasks are obviously well suited to the CPU workers. Which one is the best suited is, however, not obvious. Favoring task B may optimize the processor's usage while favoring task A may decrease the idle time of processors (due to its high NOD which measures the released task number after an A task is completed).

This decision depends on the heuristic's computed score.

Linear combination An idea for combining NOD and time difference is to compute a score based on a linear combination of them:

$$\text{NOD\_LC}_{arch}(v_i) = \alpha \cdot NOD(v_i) + \beta \cdot diff_{arch}(v_i)$$

In this naive approach, the importance of the NOD and the cost difference is calibrated by the α and the β values. More specifically, it depends on the ratio $\frac{\alpha}{\beta}$, since we can always bring α to 1.

In our datatest, this heuristic performed the best when $\alpha \times 3 \approx \beta$.

This was one of our first heuristics and it quickly became obsolete.
To understand why doing a simple linear combination is problematic, let us imagine a task "A" that has a NOD of 100 and a diff of -10 ($\alpha = 1$ and $\beta = 3$). A NOD of 100 means that the task has 100 expected successor, while a diff of -10 indicates that the current architecture is 10 seconds slower than the other one.
In this case, even though the NOD is high, executing "A" would cost 10 seconds. "A" should, therefore, have a low priority on this architecture, whereas the score given by a linear combination would be high.

NOD-Time Combination (NTC) To improve the previous heuristic, we modify the formula as so:

$$\text{NTC}_{arch}(v_i) = diff_{arch}(v_i) + \beta \cdot NOD(v_i) \cdot e^{-\gamma \cdot (pos\_rel\_diff_{arch}(v_i)-1)^2}$$

With:

$$\text{pos\_rel\_diff}_{arch}(v_i) = \begin{cases} rel\_diff_{arch}(v_i) & \text{if } rel\_diff_{arch}(v_i) \geq 1 \\ \frac{1}{rel\_diff_{arch}(v_i)} & \text{otherwise} \end{cases}$$

In other words, $pos\_rel\_diff_{arch}(v_i)$ represents the relative the speed-up of the fastest architecture and is, therefore, always superior to 1.

This heuristic affects a score based on the diff and adds a NOD bonus based on how close the CPU and GPU costs are. This bonus is scaled with a β factor, while the selectivity of the bonus is controlled by the γ parameter.

The idea of this heuristic is to reduce the importance of a task's NOD value if its costs are too distant. This avoids setting a high score to tasks that are particularly bad on an architecture. This solves the issue raised in the linear combination paragraph.

NOD dot Released time (NODxR) In this heuristic, we take the successors' costs into account:

$$\text{NODxR}_{arch}(v_i) = \frac{NOD(v_i)}{w_i^{arch}} \cdot \sum_{v_j \in succ(v_i)} \min_{arch \in Q}(w_j^{arch})$$

For each successor, we add the lowest cost. The assumed implication is that a task will often be executed by its fastest architecture.

This heuristic is an attempt to maximize the released working time. The theoretical executions, however, are relatively slow. By analyzing the mistakes made in the choice of priorities, we have been able to create an improved version of this heuristic: NODxRD.

NOD dot Released time-Diff (NODxRD)

$$\text{NODxRD}_{arch}(v_i) = \frac{NOD(v_i)}{w_i^{arch}} \cdot (diff_{arch}(v_i) + \sum_{v_j \in succ(v_i)} \min_{arch \in Q}(w_j^{arch}))$$

In this improved version of NODxR, we add the architecture's cost difference to the successors' costs.
The idea behind this formula is that the released work time comes at a cost. If an architecture can release 5 seconds of work but is 6 seconds slower than the other one, the effort can be seen as not worth the gain.

In contrast to the previous heuristic, this one leads to effective scheduling in the simulator.

NOD per second We can consider that the NOD metric is not relevant by itself. If a task is extremely costly and has a NOD of n, it can be seen as less "parallelizing" than a light task with the same NOD value. We, therefore, created a heuristic which is based on NOD per second:

$$\text{NOD/s}_{arch}(v_i) = \frac{NOD(v_i)}{w_i^{arch}}$$

The score is equal to the average NOD per second of the task type. We can observe that for a given task, the slowest arch will have a lower priority than the fastest one because we divide by the architecture's cost.

This is a conceptually interesting heuristic, but its average execution time is not particularly valuable on our dataset, compared to other heuristics.

5.3 URT-difference combination

In this section, we will detail heuristics that depend on the URT metric.

Useful Released Time For this heuristic, we affect a score directly based on the Useful Released Time (URT):

$$\text{URT}(v_i) = NRT_{CPU}(v_i) \cdot IDLE(CPU) + NRT_{GPU}(v_i) \cdot IDLE(GPU)$$

where $IDLE(arch)$ is the idle proportion of $arch$ workers over all the execution. URT's aim is to ponderate the Normalized Released Time (NRT) of an architecture with a quantity representing how needed the architecture is.
For example, if CPU's idle proportion is 0% and GPU's one is 100%, we will only take NRT_{GPU} for evaluating the Useful Released Time (i.e. the more GPU time is released, the more critical the task is).

This heuristic gives similar priorities for CPU and GPU and is therefore incomplete. It is, however, a useful indicator for evaluating the criticality of a task. In other NRT-related heuristics, the goal is to combine this "criticality" (URT) score with other metrics such as cost difference to compute meaningful priorities.

URT-difference linear combination The first idea to prioritize differently between CPU and GPU URT scores is to simply add cost difference to the score. Which gives the following formula:

$$\text{URT\_LC}_{arch}(v_i) = \alpha \cdot URT(v_i) + \beta \cdot diff_{arch}(v_i)$$

In our dataset, we measured that the heuristic performed the best when α was about 10 times smaller than β. For performance measurements, we used α=1 and β=10.

URT dot difference For this heuristic, instead of summing the cost difference and the Useful Released Time, we multiply them.

$$\text{URTxD}_{arch}(v_i) = URT(v_i) \cdot diff_{arch}(v_i)$$

This method gives comparable results (to a linear combination) in our dataset but does not need any hyperparameter. It is, however, unnatural to multiply these two metrics as is.
Let us consider a case where CPU and GPU workers are never idle (i.e. $= URT(v_i) = 0$). We then have this score set to 0 for each task, which is definitely not correct (if CPU and GPU are always busy, that does not mean all priorities are equal).

URT dot difference, normalized To address this issue, we compute a score based on this new formula:

$$\text{URTxDn}_{arch}(v_i) = (1 + URT(v_i)) \cdot diff_{arch}(v_i)$$

With this new heuristic, the edge case where $URT(v_i) = 0$ now makes sense. In such a situation, only the cost difference will be taken into account to compute the score.

URT dot relative difference This heuristic is the same as the one presented in 5.3, but instead of the absolute cost difference, we take the relative cost difference between a CPU and a GPU execution.

$$\text{URTxRD}_{arch}(v_i) = URT(v_i) \cdot rel\_diff_{arch}(v_i)$$

This score also has the advantage not to require any additional parameter, but also produce counter-intuitive priorities as CPU and GPU idle times tend towards 0.

URT dot relative difference, normalized We apply the same idea as in URTxD. This gives the following formula:

$$\text{URTxRDn}_{arch}(v_i) = (1 + URT(v_i)) \cdot rel\_diff_{arch}(v_i)$$

Favouring tasks when an architecture is lacking job There are often cases where one architecture is nearly always busy (let us say a CPU), but the other (a GPU) has a significant idle time. In this case, it may be beneficial to slightly modify our URTxDn heuristic. If a task is approximately executed 50% of the time on a CPU and 50% on a GPU, penalizing this task on CPUs, favoring it on GPUs, or doing both may help to equally distribute work between worker types. We, therefore, propose 3 new heuristics:

$$\text{URTxDn\_f1}_{arch}(v_i) = (1+URT(v_i))\cdot diff_{arch}(v_i)+\alpha\cdot w_i^{arch}\cdot(IDLE(arch)-IDLE(\overline{arch}))$$

$$\text{URTxDn\_f2}_{arch}(v_i) = (1+URT(v_i))\cdot diff_{arch}(v_i)+\alpha\cdot w_i^{arch}\cdot reLU(IDLE(arch)-IDLE(\overline{arch}))$$

$$\text{URTxDn\_f3}_{arch}(v_i) = (1+URT(v_i))\cdot diff_{arch}(v_i)-\alpha\cdot w_i^{arch}\cdot reLU(IDLE(\overline{arch})-IDLE(arch))$$

where

$$reLU(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

In these 3 scores, we add a bonus or a penalty to the URT dot diff heuristic (URTxDn). It is weighted by an αmodifier. This bonus or penalty will become larger as CPU and GPU idle proportions stray from each other. We also multiply by the cost (w_j^{arch}), because we want to quantify the gained time on the "weakest" (most idle) architecture.

With URTxDn_f1, we apply a penalty to the busiest architecture and a bonus to the other. With URTxDn_f2, we only apply the bonus and with URTxDn_f3 we only apply the penalty.

In our dataset, URTxDn_f1 and URTxDn_f2 perform better than URTxDn_f3, but URTxDn_f3 was the unique fastest heuristic we had on data 25 and 27.

URT per second As for NOD, we can consider that the URT per second is a more significant metric than the URT alone. This leads to the following formula:

$$\text{URT/s}_{arch}(v_i) = \frac{URT(v_i)}{w_i^{arch}}$$

As for the NOD per second heuristic, we have measured relatively slow executions in our simulator.

5.4 Theoretical analysis

The results of our simulations are available in tables 4, 5 6, and 7 (given in the Appendix). They show the slowdown of each heuristic (compared to the control priorities). The best heuristics are marked in bold. The last row indicates the slowdown of the best heuristic, while the last column is the average of the row (heuristic).
In this section, we will make general observations from these results.

Firstly, we can notice that some slowdowns are smaller than 1. This means that some heuristics find better priorities than the control priorities.
This is an unanticipated result, as our control priorities should be close to the best possible ones. We have a guarantee that no permutation inside an architecture's priorities can improve the priorities. The only way to improve control priorities is to modify both the CPU and GPU priorities.
This is a welcome surprise. It proves that our heuristics can, in some cases, compete with our iterative optimization algorithm.

We can judge a heuristic's performance on the last column which is the average slowdown we measure on the entire dataset. A typical value for our heuristics is ≈ 1.1 which means a $+10\%$ execution time compared to the control priorities. Some heuristics perform poorly (e.g. URT, URTxRD, etc.).
We consider, however, that they are worthy of interest due to their conceptual benefits. For example, the NOD/s heuristic has an average slowdown of 33%, but it relies on a solid concept and does not need any hyperparameter to work. Moreover, it overcomes all its competitors on the 23^{th} DAG.
These average slowdowns are only indicators, as our goal is to perform efficient scheduling in a dynamic execution. We do not consider them to be an end in themselves.

It is also worth noting that if we run the best heuristic on each DAG, the average slowdown will be less than 2%. Hence, we can assume that if the heuristic is correctly chosen, the execution time will be close to the control execution.
If we look at it in more detail, we can see that the best slowdown is particularly high in some specific DAGs (0, 4, 6, 19, and 22). In these cases, we have not been able to infer what phenomenon makes the control priorities faster. We suppose that it could be due to a lucky combinatorial path. This path would be undetectable by our heuristics since they are not allowed to look at the graph's structure.

This theoretical study gives us confidence in our heuristics. We can now implement them in StarPU. We expect that our simulations are close enough to real executions so that our heuristics perform as well in both cases.

6 Performance on a real application

To test our heuristics in real-life, we modified the Heteroprio scheduler inside StarPU. In this new version, the user can control the behavior of Heteroprio through environment variables. He can, inter alia, use an automatic mode in which the priorities are automatically updated.
In this section, we will present our method for testing our heuristics on a real application named QR_mumps, and give our conclusions regarding our results.

6.1 Implementation in StarPU

A significant part of our work consisted of modifying the Heteroprio scheduler. The first raised questions concern the design of the new scheduler.
As is, Heteroprio can only work if the user provides a priority mapping (a priority list) at the initialization of the scheduler. It is problematic because if we implement an automatic mode, the first execution will be unable to infer proper priorities at the beginning of the program.

Our first modification is, therefore, the inclusion of a dynamic priority update. This improvement lets the possibility of changing the CPU and GPU priorities during the execution.
We have multiple options concerning when the update should be performed. Nevertheless, the update can only be performed in the push_task or pop_task function, as they are the only functions that are called throughout all the execution. We chose a simple solution: the priorities are updated the first time a task is pushed, and then every n^{th} pushed task (where n can be changed with an environment variable).

There is another problem in Heteroprio's native implementation. The application has to give a priority to each task it pushes. This priority corresponds to the target bucket of the task (which is generally the same within a task type).
To have an automatic scheduler, we have to eliminate this user interaction. To achieve this, we have to make assumptions on each task's target bucket. We have decided to consider that the name of the task will be the identifier of its target bucket.
This means that if two tasks have the same name, they will share the same bucket. In other words, we consider that each task inside a task type has the same name. Hence, we store an internal array to keep track of task names and be able to identify the target bucket of a task.

We also have to face the differences between our theoretical model and real-time execution. In our simulator, we know the exact costs of a task, whereas in a real execution we can only make a prediction.
Moreover, within a task type, the costs are not always the same. We, therefore, needed to find a way to approximate a CPU and a GPU cost for each task type.
We chose to estimate costs based on a history-model. For each pair architecture/task type, we store the current average and the sample size. When an entry is added, we can compute the new average thanks to the formula:

$$avg_{n+1} = \frac{1}{n+1} \cdot time + \frac{n}{n+1} \cdot avg_n$$

Where $time$ is the actual execution time of the task and avg_n is the stored average after n executions.

StarPU is designed to be an efficient task-based execution engine. That is why its dependency structure is complex.
However, our theoretical model assumes the application has the structure of a DAG (see section 3.4.3). To extract a graph from the application, we used a feature in StarPU, which allows the scheduler to access an approximation of the application's DAG. This approximation comes with a performance cost (that we consider to be negligible).
Once we have access to the graph, we can compute a task's NOD and NRT. For storing these two metrics' average, we used the same principle as for task type costs: for each metric and task type, the average and the sample size are stored. This lets us use these metrics as approximations for our heuristics.
To keep track of the acquired information, a data file is created at the end of the execution, and read at its beginning.

Finally, we can implement our heuristics. During the refresh process (every n^{th} time a task is pushed), a score is computed for each task and architecture. This score depends on the selected heuristic. The CPU and GPU indirection arrays (priorities) are then sorted by descending score.

We are now able to launch an execution in which the CPU and GPU priorities are automatically set.

6.2 QR-mumps

The multifrontal method is a method for factorizing sparse, symmetric linear systems [13]. QR-mumps is the adaptation of this method to the QR factorization of sparse matrices [2]. It has initially been designed for homogeneous (CPU) architectures.
It relies on StarPU runtime system for handling scheduling and data transfers. It has been extended to heterogeneous architectures in 2016 in the context of Florent LOPEZ's Ph.D. thesis [17], with the introduction of CUDA kernel for using GPU workers.
Of all the available schedulers available in StarPU, Heteroprio was the one that performed the best with QR-mumps.

We chose to use QR-mumps as a benchmark for two reasons.
First, it is an application that uses StarPU and Heteroprio. It gave solid results with the Heteroprio scheduler and we have, therefore, interest in showing that equivalent results can be achieved with an automatic scheduler.
Second, it is a lightweight application with little dependencies, which makes it easy for us to modify.

We first need to launch a fake execution, where our automated priorities work properly. To find these priorities, our scheduler needs to be able to estimate each task type's execution time. The only intended way to do so in StarPU is to manually set a perfmodel for each task type. Usually, optimized applications set

these perfmodels which are central for schedulers. QR-mumps, however, did not need to affect perfmodels because the non-automatic version of Heteroprio worked with hand-written priorities. We, therefore, modified QR-mumps' code so that every task type has a perfmodel.

Finally, we added options for the user. Now, the scheduler can be chosen through an environment variable. This will let us configure our executions and be able to compare our scheduler's performance against other ones.

6.3 Methodology

To assess the performance of our heuristics, we chose to run the application on PlaFRIM (Plateforme Fédérative pour la Recherche en Informatique et Mathématiques). PlaFRIM is a scientific instrument designed for experiments in applied mathematics and informatics.
PlaFRIM uses Slurm Workload Manager for managing job schedulings and resource allocations. Slurm is a popular job scheduler, with approximately 60% of TOP500 supercomputers using it.

QR-mumps needs a CUDA compatible GPU to use its CUDA kernels. There are 3 compatible GPU architectures available on the PlaFRIM cluster: p100, v100, and k40m. We will use all of them for executing our experiments.

QR-mumps is launched on the TF16 matrix [19] of the JGD_Forest dataset. We measure the factorization time in different scenarios.
We keep track of each execution time in CSV files. This lets us analyze the data distribution in addition to the average.
We use python and matplotlib for plotting our results.

6.4 Results

We have conducted different experiments, that we will present in this section.

6.4.1 Acquisition time

As explained in section 6.1, our scheduler works by acquiring data through executions. These data are stored internally and are necessary for the heuristics to be accurate.

On the first execution, we will have no information at all. Data will become more accurate as the execution advances, and as we relaunch executions. The more precise our metrics are, the more efficient we can expect our heuristics to be. We can, therefore, wonder how much QR-mumps executions are needed before Heteroprio is calibrated enough.

We, therefore, ran an experiment to measure this suspected effect. Each heuristic is run 16 consecutive times, while we measure each execution time.
Since we have 28 different heuristics, each n^{th} execution has a sample size of 28. If we are correct, we should see the makespan decrease until it starts to level off.

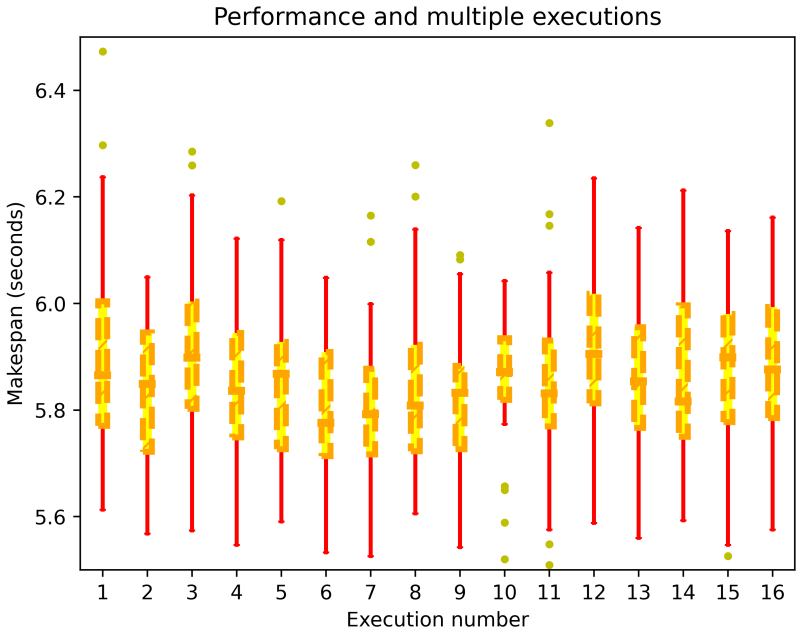

Figure 8: Execution times of the 28 heuristics on consecutive executions

Figure 8 shows the result of this experiment on a v100 architecture. A box plot represents the execution times of the 28 heuristics on the n^{th} consecutive execution.

It does not seem that the makespan diminishes over time. To ensure this, we have performed a linear regression. The found slope is approximately 0.001, which is positive. We can, therefore, conclude that the execution time does not improve with consecutive runs. It is, therefore, reasonable to assume that a single QR-mumps execution is sufficient for Heteroprio to gather enough data for performing efficient scheduling.

This test showed that even though each execution impacts the next executions, we can consider that each one is independent. It is also a promising result for our concept. It shows that automated priorities do not necessarily need several executions to be able to produce efficient scheduling.

6.4.2 Comparison with other schedulers

In this experiment, the goal is to compare the performance of different StarPU schedulers on QR-mumps. To do so, we launched a series of runs with different schedulers. We chose to consider 7 different StarPU scheduling strategies:

- the Eager scheduler uses a central task queue. All workers retrieve tasks from this queue concurrently
- the LWS (Locality Work Stealing) scheduler uses a queue per worker. Tasks are moved to the queue of the worker that released them. If a worker's queue is empty, it will try to steal tasks from other workers

- the DM (Deque Model) scheduler uses a HEFT-similar strategy. It tries to minimize the minimum finish time by using a look-ahead strategy
- the DMDA (Deque Model Data aware) is the same as the DM scheduler but takes into account data transfer costs.
- the DMDAS (Deque Model Data aware) acts as the DMDA scheduler but sorts the tasks by priority (user-defined). Its behavior is close to the HEFT scheduling strategy
- the Heteroprio strategy is the first version of Heteroprio, which was based on user-made priorities
- the AHP (Auto-HeteroPrio) strategy is our automated version of Heteroprio.

Eager and LWS are often considered to be slow schedulers. They are, however, interesting from a conceptual point of view and often used in real applications.

For this study, each scheduler has 32 measurements, except for the AHP which has 32×28 measurements (32 for each heuristic). Indeed, our 28 heuristics are represented in this dataset. The reason behind this choice is that we do not want to introduce a bias in our experiment by measuring only one heuristic of our choice.

We ran the experiment on the three architectures, but the p100 results will not shown because they are mostly similar to the v100 ones.

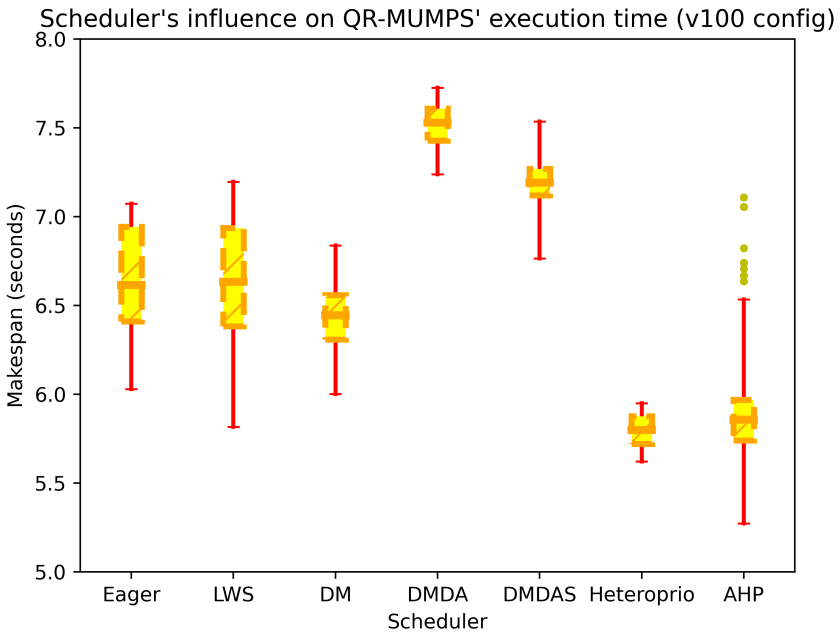

Figure 9: Execution times of schedulers on a v100 configuration

Figure 9 shows our results on the v100 configuration.
Eager, LWS, and DM share comparable decent results, with an average execution

time of around 6.5 seconds, whereas the DMDA and DMDAS scheduler seem particularly slow (more than 7 seconds on average). Heteroprio and AHP give comparable results, with less than 6 seconds on average. Heteroprio seems to be slightly, but significantly faster.

This experiment shows that Heteroprio does not fundamentally need to have static hand-written priorities to perform efficient scheduling: AHP follows the same principle, but with heuristic-based priorities, and gets comparable results. Since Heteroprio is not an automatic scheduler, we can conclude that on these executions of QR-mumps the best automatic scheduler is AHP.

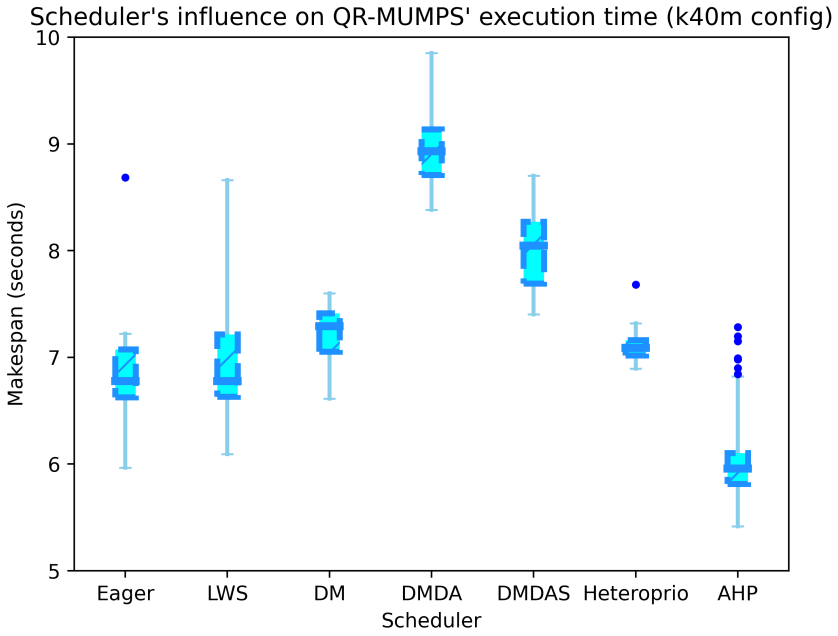

Figure 10: Execution times of schedulers on a k40m configuration

On figure 10, we can see the results on the k40m architecture.
There are similarities compared with the v100 configuration, but the main difference we can note is that the Heteroprio scheduler now gives comparable results to the Eager, LWS, and DMDA schedulers.
AHP, however, is still significantly faster than the other presented schedulers. Our interpretation of this change is that Heteroprio's static priorities are not well suited for the specific case of our k40m machine. AHP, on the other hand, can modify its priorities depending on the situation (tasks' execution time, workers' idle time, etc.).

This experiment on a k40m machine shows an example where automated dynamic priorities perform better than human static priorities. The question that remains is whether Heteroprio's slowdown is due to improper priorities, or to the fact that its priorities are static. In either case, this is a positive result for AHP.

6.4.3 Inter-heuristic comparison

Finally, we decided to compare the efficiency of different heuristics. For the sake of clarity, we will consider only 6 of our 28 heuristics.

In this experiment, we ran multiple executions of different heuristics on different configurations. The results are shown in figure 11.

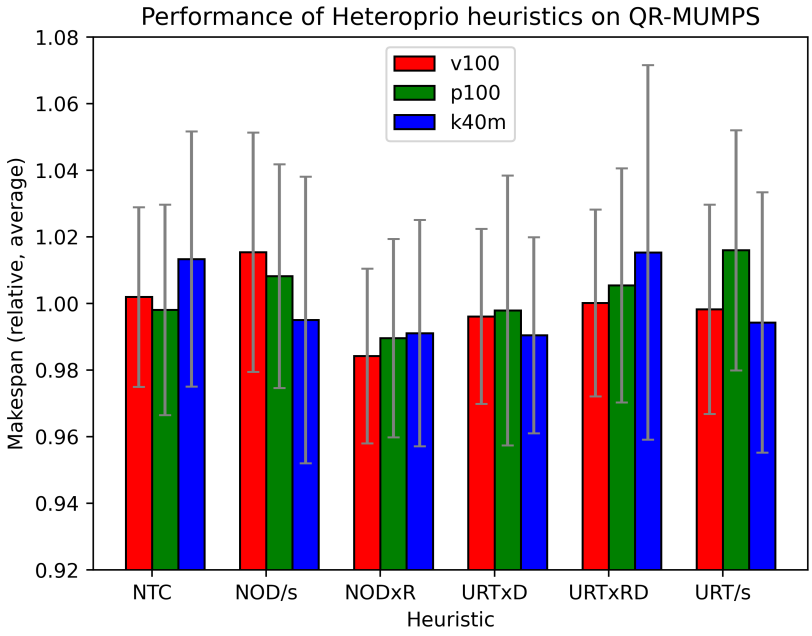

Figure 11: Execution times on different heuristics and configurations

To be able to compare the execution times between different architectures, we do not plot these times directly. Instead, we compute the relative time to the average heuristic execution time. Therefore, a heuristic that is below 1 on an architecture in this chart can be considered to be better than this architecture's average. The standard errors are indicated at the bar's edges.

We can see that, even though there seem to be clear differences between these 6 heuristics, all the executions times gravitate around the same value. This means that on one configuration, we can expect all the heuristics to be included within the same range ($\pm$ 2%).
The chosen heuristic, therefore, does not seem to have a considerable impact on the execution time. From a user's point of view, this is convenient, as it means he is not required to spend time fine-tuning his application through the choice of heuristic. It is also an argument in favor of our concept of automated priorities, as it suggests that all the efficient priorities are close in terms of execution time.

These observations are, naturally, restricted to the case of QR-mumps. Generalizing them (or not) to other applications will be the subject of future work. Yet, our

work has shown that automatizing Heteroprio with heuristic-based priorities could lead to valuable results on QR-mumps.

7 Conclusion

Our study has shown that Heteroprio priorities can be computed automatically during the execution. In the case of QR-mumps, computed priorities were as efficient as handmade priorities in the v100 and p100 configurations. In the k40m case, computed priorities perform even better than our control priorities.

We have designed 28 different heuristics with the hope that in a real-case scenario, a few would come out on top. Unfortunately, in our experiment, all heuristics seemed to be about as fast.

We intend to conduct further works to see if our observations can generalize to a wider set of real applications. We also hope to find applications in which the choice of heuristic is critical. Finding such applications will help us understand the influence of this choice.

data index	CPU number	GPU number	CPU/GPU	close CPU-GPU task proportion	far CPU-GPU task proportion	task with numerous predecessors proportion	average predecessor number	max predecessor number	task without successor proportion	task with numerous successors proportion	max successor number	average CPU-GPU diff
0	4	14	0.286	0.549	0.336	0.502	4.038	7	0.423	0.243	41	0.580
1	13	8	1.625	0.377	0.623	0.000	2.101	3	0.467	0.084	105	0.905
2	11	12	0.917	0.687	0.135	0.000	2.526	3	0.781	0.033	540	0.855
3	2	7	0.286	0.191	0.302	0.211	2.529	4	0.388	0.104	142	1.089
4	13	15	0.867	0.508	0.233	0.007	1.289	5	0.575	0.057	102	1.605
5	9	7	1.286	0.276	0.573	0.141	2.301	4	0.360	0.127	546	1.154
6	13	3	4.333	0.314	0.026	0.000	2.396	3	0.339	0.118	62	0.715
7	11	1	11.000	0.226	0.531	0.000	1.491	3	0.496	0.062	307	1.036
8	9	12	0.750	0.418	0.582	0.000	1.675	3	0.255	0.070	22	0.187
9	12	1	12.000	0.405	0.043	0.367	2.927	4	0.176	0.180	57	0.388
10	2	9	0.222	0.167	0.777	0.000	0.995	1	0.301	0.005	7	3.791
11	13	10	1.300	0.232	0.529	0.000	1.384	3	0.507	0.083	24	1.720
12	4	6	0.667	0.286	0.530	0.000	1.325	3	0.658	0.060	103	1.179
13	4	11	0.364	0.018	0.497	0.000	1.462	3	0.563	0.031	72	1.484
14	8	1	8.000	0.498	0.468	0.000	1.850	2	0.459	0.088	52	1.756
15	3	8	0.375	0.112	0.888	0.000	2.686	3	0.570	0.072	111	4.153
16	15	3	5.000	0.294	0.126	0.000	1.347	2	0.466	0.094	20	1.243
17	10	1	10.000	0.452	0.514	0.000	2.258	3	0.766	0.064	548	0.228
18	7	3	2.333	0.160	0.565	0.139	1.679	4	0.432	0.094	54	1.793
19	9	14	0.643	0.269	0.725	0.000	1.817	3	0.334	0.084	108	2.238
20	8	11	0.727	0.386	0.392	0.000	1.859	3	0.294	0.093	25	0.850
21	8	8	1.000	0.527	0.324	0.323	2.655	5	0.386	0.083	439	0.917
22	15	9	1.667	0.350	0.650	0.126	2.268	4	0.281	0.107	147	2.050
23	14	4	3.500	0.008	0.973	0.000	1.288	3	0.228	0.022	116	12.786
24	1	2	0.500	0.115	0.175	0.133	1.934	5	0.327	0.115	18	0.881
25	9	11	0.818	0.278	0.278	0.000	2.030	3	0.275	0.119	13	0.626
26	4	14	0.286	0.299	0.512	0.166	1.884	4	0.372	0.111	34	0.771
27	15	1	15.000	0.453	0.417	0.000	1.551	2	0.685	0.090	55	0.253
28	9	3	3.000	0.635	0.266	0.099	1.474	5	0.477	0.066	131	0.187
29	15	8	1.875	0.288	0.539	0.396	3.558	6	0.186	0.264	50	1.534
30	10	10	1.000	0.612	0.000	0.169	2.552	7	0.368	0.197	28	0.434
31	12	13	0.923	0.395	0.605	0.000	1.516	3	0.482	0.094	17	2.245

Table 4: Details of the dataset

data index / priority method	0	1	2	3	4	5	6	7	8	9	10
NTC	1.135	**1.001**	1.209	1.241	1.110	1.062	1.452	**1.019**	1.025	1.023	1.048
BEST_NODS_SCORE	1.336	1.034	1.108	1.032	1.101	1.246	1.342	1.588	1.007	1.027	**0.983**
BEST_NODS	1.356	1.034	1.108	1.032	1.101	1.246	1.342	1.588	1.007	1.027	**0.983**
URT	1.280	1.183	1.372	1.265	1.289	1.342	1.969	3.531	1.026	1.041	1.270
URT_LC	1.134	1.049	1.154	1.126	1.098	**0.987**	1.318	1.436	1.002	1.032	1.329
URT_LC_need	1.134	1.049	1.154	1.112	1.098	**0.987**	1.318	1.436	1.002	1.032	1.329
URTxD	**1.055**	**1.001**	1.083	1.174	1.121	1.016	1.405	1.671	**0.986**	1.019	1.329
URTxDn	**1.055**	**1.001**	1.220	1.214	1.094	1.016	1.405	1.059	1.022	1.033	1.329
URTxRD	1.285	1.283	1.379	1.267	1.242	1.242	1.927	3.581	1.133	1.033	1.571
URTxRDn	1.180	**1.001**	1.318	1.069	1.153	1.079	1.153	1.286	1.011	1.042	1.333
URTxD_f1	**1.055**	**1.001**	1.229	1.212	1.121	1.080	1.405	1.671	1.175	1.019	1.438
URTxD_f2	**1.055**	**1.001**	1.229	1.105	1.121	1.097	1.405	1.578	1.091	1.019	1.048
URTxD_f3	**1.055**	**1.001**	1.083	1.243	1.121	1.088	1.405	1.459	1.038	1.019	**0.983**
URTxD_L1	1.119	**1.001**	1.045	1.096	1.117	1.052	1.318	1.436	1.144	1.029	1.329
URTxD_L2	1.120	1.049	**1.031**	1.154	1.104	1.048	1.280	1.536	1.078	1.042	1.329
URTxD_L3	1.119	**1.001**	1.063	**1.032**	1.138	1.007	1.361	1.530	1.076	1.029	1.329
URTxD_L4	1.139	**1.001**	1.063	1.083	1.115	1.152	1.361	1.545	1.078	1.042	1.329
URTxD_L5	1.101	**1.001**	1.083	1.111	1.103	1.083	1.163	1.059	**0.986**	1.023	1.329
URTxD_L6	1.166	**1.001**	1.066	1.125	1.117	1.052	1.190	1.458	1.015	1.029	1.329
URTxD_L7	1.101	**1.001**	1.141	1.101	**1.091**	1.081	1.318	1.059	1.011	1.023	1.329
URT/s	1.249	1.133	1.402	1.169	1.352	1.262	1.971	3.154	1.052	1.041	1.270
URT/s_2	1.249	1.096	1.204	1.153	1.244	1.133	1.966	1.286	1.144	1.042	1.329
URT/s_diff	1.249	1.096	1.246	1.200	1.196	1.158	1.095	1.313	1.078	1.029	1.329
URTxRD	1.285	1.062	1.170	1.178	1.159	1.194	1.182	1.752	1.046	1.017	1.000
UTC	1.087	**1.001**	1.179	1.098	1.110	1.062	1.333	**1.019**	1.015	1.023	1.329
NOD/s	1.393	1.133	1.369	1.172	1.220	1.232	1.970	3.270	1.091	1.047	1.502
NODxR	1.408	1.220	1.453	1.231	1.264	1.214	1.659	2.989	1.055	**1.017**	1.235
NODxRD	1.285	1.062	1.264	1.208	1.119	1.165	**1.061**	1.056	1.017	**1.017**	1.000
best	**1.055**	**1.001**	**1.031**	**1.032**	**1.091**	**0.987**	**1.061**	**1.019**	**0.986**	**1.017**	**0.983**

Table 5: Slowdown of each method, part 1

data index / priority method	11	12	13	14	15	16	17	18	19	20	21
NTC	1.010	0.990	1.126	1.014	1.283	**1.020**	1.026	1.193	1.354	1.163	1.158
BEST_NODS_SCORE	1.058	1.034	1.042	1.014	**1.010**	**1.020**	1.118	1.530	1.272	1.293	1.162
BEST_NODS	1.058	1.034	1.042	1.003	**1.010**	**1.020**	1.118	1.530	1.272	1.293	1.162
URT	1.261	1.253	1.195	1.282	2.475	1.489	1.147	1.677	1.693	1.479	1.523
URT_LC	1.119	0.990	1.104	1.014	1.281	**1.020**	1.050	1.144	1.244	**1.000**	1.160
URT_LC_need	1.119	0.990	1.104	1.014	1.321	**1.020**	1.050	1.144	1.336	**1.000**	1.160
URTxD	1.119	0.990	1.130	1.003	1.283	1.113	1.023	1.193	**1.163**	**1.000**	1.303
URTxDn	1.119	0.990	1.126	1.003	1.283	**1.020**	1.023	1.193	**1.163**	**1.000**	1.303
URTxRD	1.284	1.314	1.195	1.285	2.103	1.489	1.121	1.642	1.790	1.361	1.634
URTxRDn	**0.965**	**0.985**	1.148	1.011	1.325	**1.020**	1.055	**1.040**	1.293	**1.000**	1.201
URTxD_f1	1.177	1.088	1.042	1.003	1.283	1.303	1.023	1.193	1.466	1.233	1.470
URTxD_f2	1.010	1.046	1.042	1.003	1.283	1.113	1.023	1.193	1.385	1.163	**1.141**
URTxD_f3	1.175	0.990	1.137	1.003	1.283	1.113	1.023	1.193	1.336	1.029	1.415
URTxD_L1	1.010	0.992	**1.026**	1.014	**1.010**	**1.020**	1.052	1.193	**1.163**	**1.000**	1.156
URTxD_L2	1.010	1.009	1.126	**1.003**	**1.010**	1.297	1.019	1.054	1.487	**1.000**	1.251
URTxD_L3	1.010	1.035	1.069	1.014	**1.010**	1.297	1.058	**1.040**	**1.163**	1.268	1.156
URTxD_L4	1.010	1.004	1.049	1.014	**1.010**	1.297	1.058	1.459	1.385	1.455	1.424
URTxD_L5	1.119	0.990	1.042	1.014	**1.010**	**1.020**	1.046	1.137	1.224	**1.000**	1.160
URTxD_L6	1.010	0.992	**1.026**	1.014	**1.010**	**1.020**	1.058	1.193	**1.163**	**1.000**	1.160
URTxD_L7	1.119	0.990	**1.026**	1.014	**1.010**	**1.020**	1.046	1.137	1.224	**1.000**	1.160
URT/s	1.153	1.230	1.038	1.274	1.742	1.495	1.073	1.245	1.509	1.248	1.483
URT/s_2	1.060	0.990	1.131	**1.003**	1.325	1.297	1.046	1.054	1.292	1.060	1.465
URT/s_diff	1.188	1.004	1.116	1.014	1.325	1.297	**1.019**	1.054	1.483	1.203	1.437
URTxRD	1.128	1.038	1.183	1.034	1.321	**1.020**	1.050	1.366	1.224	1.254	1.474
UTC	1.119	0.990	1.126	1.014	1.283	**1.020**	1.026	1.193	1.354	**1.000**	1.158
NOD/s	1.114	1.122	1.183	1.272	1.291	1.705	1.098	1.636	1.413	1.535	1.474
NODxR	1.259	1.047	1.175	1.282	1.861	1.716	1.097	1.472	1.492	1.248	1.477
NODxRD	1.160	1.047	1.183	1.034	1.281	**1.020**	1.050	1.304	1.279	1.254	1.351
best	**0.965**	**0.985**	**1.026**	**1.003**	**1.010**	**1.020**	**1.019**	**1.040**	**1.163**	**1.000**	**1.141**

Table 6: Slowdown of each method, part 2

data index / priority method	22	23	24	25	26	27	28	29	30	31	avg
NTC	**1.065**	1.154	1.043	1.055	1.070	1.019	1.034	**1.009**	0.992	1.118	1.101
BEST_NODS_SCORE	1.143	1.108	1.038	1.042	1.207	1.011	1.295	1.084	0.960	**1.009**	1.133
BEST_NODS	1.143	1.108	1.038	1.042	1.207	1.011	1.295	1.084	0.960	**1.009**	1.133
URT	1.324	1.261	1.213	1.193	1.286	1.201	1.274	1.281	1.147	1.178	1.419
URT_LC	1.112	1.176	1.013	1.042	1.070	1.011	1.034	1.130	1.014	1.118	1.110
URT_LC_need	1.112	1.176	1.013	1.042	1.083	1.011	1.034	1.130	1.014	1.118	1.114
URTxD	1.078	0.999	1.035	1.042	**1.037**	1.015	1.026	1.084	**0.934**	1.118	1.111
URTxDn	1.124	0.999	1.035	1.042	1.075	1.011	1.034	1.084	**0.934**	1.118	1.098
URTxRD	1.269	1.133	1.249	1.107	1.153	1.204	1.332	1.180	1.024	1.231	1.408
URTxRDn	1.165	1.002	1.067	1.042	1.045	1.003	**1.009**	1.098	1.038	1.091	1.101
URTxD_f1	1.111	0.999	1.070	0.991	1.102	1.021	1.106	1.084	**0.934**	1.118	1.163
URTxD_f2	1.078	0.999	**0.997**	1.027	**1.037**	1.246	1.073	1.084	**0.934**	1.118	1.117
URTxD_f3	1.111	0.999	1.080	**0.968**	**1.037**	**1.001**	1.134	1.084	**0.934**	1.118	1.114
URTxD_L1	1.134	0.999	1.007	1.042	1.124	1.028	1.034	1.092	1.014	1.106	1.091
URTxD_L2	1.118	1.126	1.055	1.068	1.063	1.028	1.018	1.082	1.014	1.106	1.116
URTxD_L3	1.134	0.999	1.020	1.042	1.076	1.013	1.114	1.083	1.014	1.106	1.106
URTxD_L4	1.118	1.030	1.038	1.068	1.060	1.028	1.082	1.083	1.014	1.106	1.147
URTxD_L5	1.134	1.035	1.007	1.042	1.091	1.011	1.026	1.083	0.993	1.106	**1.073**
URTxD_L6	1.118	1.035	1.008	1.042	1.082	1.019	1.034	1.089	1.014	1.106	1.086
URTxD_L7	1.134	0.999	1.000	1.042	1.091	1.011	1.034	1.113	0.993	1.106	1.079
URT/s	1.190	1.175	1.136	1.119	1.093	1.188	1.115	1.127	1.075	1.451	1.319
URT/s_2	1.197	1.126	1.174	1.096	1.069	1.003	**1.009**	1.115	1.075	1.451	1.184
URT/s_diff	1.197	1.126	1.065	1.096	1.074	1.013	1.028	1.100	0.998	1.451	1.165
URTxRD	1.143	1.002	1.191	1.037	1.075	1.002	1.065	1.159	1.075	1.118	1.156
UTC	1.071	1.176	1.043	1.042	1.039	1.011	1.034	1.063	**0.934**	1.118	1.096
NOD/s	1.190	**0.953**	1.208	1.119	1.073	1.188	1.094	1.127	0.998	1.451	1.333
NODxR	1.514	1.066	1.226	1.045	1.070	1.185	1.101	1.164	1.051	1.091	1.325
NODxRD	1.197	1.126	1.154	1.037	1.084	1.002	1.077	1.159	1.051	1.118	1.132
best	**1.065**	**0.953**	**0.997**	**0.968**	**1.037**	**1.001**	**1.009**	**1.009**	**0.934**	**1.009**	**1.019**

Table 7: Slowdown of each method, part 3

References

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

[2] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, and Florent Lopez. Multifrontal QR Factorization for Multicore Architectures over Runtime Systems. In 19th International Conference Euro-Par (EuroPar 2013), volume 8097, pages pp. 521–532, Aachen, Germany, August 2013.

[3] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience, 23(2):187–198, 2011.

[4] L. F. Bittencourt, R. Sakellariou, and E. R. M. Madeira. Dag scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm. In 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, pages 27–34, 2010.

[5] Berenger Bramas. Optimization and parallelization of the boundary element method for the wave equation
Theses, Université de Bordeaux, February 2016.

[6] Bérenger Bramas, Philippe Helluy, Laura Mendoza, and Bruno Weber. Optimization of a discontinuous Galerkin solver with OpenCL and StarPU. International Journal on Finite Volumes, 15(1):1–19, January 2020.

[7] Bérenger Bramas. Impact study of data locality on task-based applications through the heteroprio scheduler. PeerJ Computer Science, 5:e190, May 2019.

[8] P. Brucker and S. Knust. Complexity results for scheduling problems. `http://www2.informatik.uni-osnabrueck.de/knust/class/`.

[9] J. Bruno, E. G. Coffman, and R. Sethi. Scheduling independent tasks to reduce mean finishing time. Commun. ACM, 17(7):382–387, July 1974.

[10] Hong Choi, Dong Son, Seung Kang, Jongmyon Kim, Hsien-Hsin Lee, and Cheol-Hong Kim. An efficient scheduling scheme using estimated execution time for heterogeneous computing systems. The Journal of Supercomputing, 65, 08 2013.

[11] John D. Cook. Permutation distance. `https://www.johndcook.com/blog/2019/11/13/permutation-distance/`.

[12] Simplice Donfack, Laura Grigori, William D. Gropp, and Vivek Kale. Hybrid static/dynamic scheduling for already optimized dense matrix factorization. Research Report RR-7775, INRIA, October 2011.

[13] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear. ACM Trans. Math. Softw., 9(3):302–325, September 1983.

[14] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Comput. Surv., 31(4):406–471, December 1999.

[15] Joseph Y.-T. Leung and Gilbert H. Young. Minimizing schedule length subject to minimum flow time. SIAM J. Comput., 18(2):314–326, April 1989.

[16] Han Lin, Ming-Fan Li, Cheng-Fan Jia, Jun-Nan Liu, and Hong An. Degree-of-node task scheduling of fine-grained parallel programs on heterogeneous systems. Journal of Computer Science and Technology, 34(5):1096–1108, 2019.

[17] Florent Lopez. Task-based multifrontal QR solver for heterogeneous architectures. Theses, Université Paul Sabatier - Toulouse III, December 2015.

[18] Frederic Suter. Dag generation program. `https://github.com/frs69wq/daggen`.

[19] Nicolas Thiery. Matrix: JGD_Forest/TF16. `https://www.cise.ufl.edu/research/sparse/matrices/JGD_Forest/TF16.html`.

[20] H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE Transactions on Parallel and Distributed Systems, 13(3):260–274, 2002.

[21] Y. Wen, Z. Wang, and M. F. P. O'Boyle. Smart multi-task scheduling for opencl programs on cpu/gpu heterogeneous platforms. In 2014 21st International Conference on High Performance Computing (HiPC), pages 1–10, 2014.

[22] Yu-Kwong Kwok and I. Ahmad. Dynamic critical-path scheduling: an effective technique for allocating task graphs to multiprocessors. IEEE Transactions on Parallel and Distributed Systems, 7(5):506–521, 1996.

[23] J. Zhou, T. Wei, M. Chen, J. Yan, X. S. Hu, and Y. Ma. Thermal-aware task scheduling for energy minimization in heterogeneous real-time mpsoc systems. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 35(8):1269–1282, 2016.

List of Figures

List of Tables