Dynamic Tasks Scheduling with Multiple Priorities on Heterogeneous Computing Systems

Hayfa Tayeb, Bérenger Bramas, Mathieu Faverge, Abdou Guermouche

Dynamic Tasks Scheduling with Multiple Priorities on Heterogeneous Computing Systems

Hayfa Tayeb
Bordeaux University
Inria, ICube Lab
Bordeaux, France
hayfa.tayeb@inria.fr

Bérenger Bramas
Strasbourg University
Inria, ICube Lab
Strasbourg, France
berenger.bramas@inria.fr

Mathieu Faverge
Bordeaux University
Bordeaux INP, CNRS, Inria
Bordeaux, France
mathieu.faverge@inria.fr

Abdou Guermouche
Bordeaux University
Inria, LaBRI Lab
Bordeaux, France
abdou.guermouche@labri.fr

***Abstract*—The efficient utilization of heterogeneous computing systems is crucial for scientists and industrial organizations to execute computationally intensive applications. Task-based programming has emerged as an effective approach for harnessing the processing power of these systems. However, effective scheduling of task-based applications is critical for achieving high performance. Typically, these applications are represented as directed acyclic graphs (DAGs), which can be optimized through careful scheduling to minimize execution time and maximize resource utilization. In this paper, we introduce MultiPrio, a dynamic task scheduler that aims to minimize the overall completion time of parallelized task-based applications. The goal is to find a trade-off between resource affinity, task criticality, and workload balancing on the resources. To this end, we compute scores for each task and manage the available tasks in the system with a data structure based on a set of priority queues. Tasks are assigned to available resources according to these scores, which are dynamically computed by heuristics based on task affinity and criticality. We also consider workload balancing across resources and data locality awareness. To evaluate the scheduler, we study the performance of dense and sparse linear algebra task-based applications and task-based FMM application using the StarPU runtime system on heterogeneous nodes. Our scheduler shows interesting results compared to other state-of-the-art schedulers in StarPU for regular applications, and excels at optimizing irregular workloads, improving performance by up to 31%.**

***Index Terms*—Priority-based scheduling, Runtime system, heterogeneous computing systems**

I. Introduction

High-performance computing relies on heterogeneous computing systems that come with an overall increased parallelism diversity such as multiprocessors and accelerators, e.g., graphical processing units (GPUs). HPC experts work tediously to narrow the gap between domain experts' implementations and the use of heterogeneous systems. A range of research-driven projects has established diversified task-based support, employing various programming and runtime features [1].

The task-based programming model has shown great potential in various applications [2]–[4]. In this model, the developer defines atomic tasks with the dependencies between them. A directed acyclic graph (DAG) represents the application. The runtime, which is an intermediate software layer supporting this DAG execution, schedules tasks and data migrations efficiently on all available cores while reducing the waiting time between tasks. Therefore, the goal of DAG scheduling is to minimize the global completion time of the program, i.e. the makespan. Various runtime systems capable of handling heterogeneous workloads have emerged (PaRSEC [5], StarPU [6]). These runtime systems serve as an overlay on which task-based applications are executed. The StarPU runtime system employs a Sequential Task Flow (STF) representation of applications, where computations are defined as tasks, each with input and output data and access modes. The application developer provides different implementations for each task to enable their execution on either CPU or GPU. In the STF model, the runtime system automatically builds the DAG by relying on data access modes and a sequential submission order. It infers task dependencies based on their data requirements and uses this information to schedule tasks on the appropriate processing unit while organizing data transfers between memory nodes.

To achieve high performance on heterogeneous systems, efficient task scheduling is essential. The research community has proposed multiple scheduler families, each aimed at tackling specific scheduling challenges. HeteroPrio [3] is a scheduler designed for heterogeneous machines in the context of task-based Fast Multipole Method (FMM) implementation. It is used by several applications showing significant improvements [7, 8]. HeteroPrio is a semi-automatic scheduler where users must provide priorities for the different types of tasks that exist in their applications. A fully automatic version of this scheduler that computes efficient priorities for HeteroPrio is proposed in [9]. HeteroPrio and its automatic version are cheap and effective. However, they show a limitation which is the priority assignment per type of task. Every application has a set of task types that will be used in different stages of the scheduling. Setting a priority per type could hide relevant information related to a given scheduling context. This also brings us to the limitation of the data structures that are used to manage the ready tasks in the scheduler which are tied to the strategy of priorities per type.

This study aims to address the limitations raised and therefore proposes a novel scheduler based on priority per task for a given processing unit. We define a data structure managing the ready tasks in the system per priority and per processing unit type. Our objective is to minimize the global completion time of task-based applications in heterogeneous environments

thanks to good scheduling decisions. The major contributions of this paper can be summarized as follow:

- We present a novel automatic dynamic scheduler for heterogeneous systems that balances between task affinity and criticality while taking into account data locality and resource workload.
- We evaluate the performance of our scheduler relative to existing schedulers in StarPU, in real-life scenarios, and show that it efficiently schedules irregular applications without requiring user expertise, while being competitive with highly tuned schedulers on more regular workloads.

The paper is organized as follows: In Section 2 we briefly analyze some related works. In Section 3 we present an overview of the model of the proposed scheduler: context and its building blocks. Afterward, we instantiate the model in the context of StarPU as a task-based runtime system and we explain the underlying mechanisms that orchestrate scheduling. In the following section, we present the heuristics and their impact. In Section 6 we describe the performance study with real-life task-based applications and show the scheduling results on different configurations of heterogeneous systems using StarPU. Section 7 discusses challenges and future work, and finally we conclude.

II. Related work

Runtime systems have a significant role in supporting program execution in heterogeneous parallel and distributed computer systems. Our work is focused on runtime systems managing dynamically the execution of task-based applications, i.e. the distribution of work is at runtime. Several works are proposed, we cite StarPU [6], OmpSs [10], XKaapi [11], PaRSEC [5] and more recently IRIS [12] that improves portability across a wide range of diverse heterogeneous architectures with negligible overhead. From a task scheduling perspective, most of these task-based runtime systems rely on dynamic strategies for task scheduling. These dynamic scheduling heuristics can be classified into two families.

Resource-centric schedulers aim to maximize resource utilization by allocating tasks to processors based on resource availability and workload. When a resource is getting close to the idle state, the scheduler selects a task for this resource. A famous heuristic is work stealing (ws) [13, 14] where the worker steals a task from the most loaded worker. An improved version takes into account data locality (lws) such that the scheduler steals a task from neighbor workers. This scheduling policy is proposed in StarPU, XKaapi and IRIS.

Task-centric schedulers focus on improving task execution time and reducing task waiting time. Unlike resource-centric schedulers, task-centric ones take decisions when a task is ready to be executed. A common heuristic is Heterogeneous Earliest-Finish-Time (HEFT) [15]. H. Choi et al. [16] proposed a dynamic scheduling algorithm that uses a history-based Estimated-Execution-Time (EET) for each task. The algorithm aims to schedule each task on its fastest architecture but may deviate from this rule in cases of work starvation for a worker type. K. Chronaki et al. [17] proposed a criticality-aware task scheduler (CATS) that dynamically assigns critical tasks to fast cores in a heterogeneous multi-core. This scheduling policy consistently improved performance compared to the dynamic implementation of HEFT. In StarPU, the dequeue model (dm) scheduler family [18] (also called heft-tm-pr) considers task execution performance models to implement a scheduling strategy similar to HEFT. The scheduler estimates and selects the best expected completion time of the task on each processing unit. This evaluation is based on the measured execution times of previously scheduled tasks. The Dmda (dequeue model data-aware) scheduler (also called heft-tmdp-pr) goes a step further by also considering the time required to transfer data to the processing unit. The Dmdas (dequeue model data aware sorted) variant sorts the queues based on the task priority values specified by the application expert. For the tasks with the highest priorities, it will give preference to those whose data buffers are already available on the target device, which makes it more sensitive to the data locality. Dmdas uses user priorities, while our scheduler relies exclusively on heuristics, without requiring user knowledge of the DAG. In the upcoming experimental evaluation, we will only compare our scheduler to Dmdas, which is representative of the family of dynamic heft-based schedulers. In each case, we will specify whether or not the priorities in each application were set by the user. If it's not the case, Dmdas will simply act as if all tasks have equal priorities and will push the tasks into the queue in the order they become ready. DARTS [19], a novel dynamic GPU task allocation strategy based on data selection and a customized eviction policy, was shown to outperform existing strategies in StarPU. Although, this scheduler schedules on mono-resource computing system.

Some task-based execution systems may use a combination of both policies to achieve a balance between resource utilization and task execution efficiency. We can find affinity-based schedulers that assign tasks to resources based on their affinity, i.e. the task's preference for a specific resource. One example is HeteroPrio [3] scheduler that uses different priorities for the different processing units. It relies on a priority assignment per type of task according to its performance on a processing unit (PU). Tasks are dispatched to buckets according to their priorities. Each task is executed on the most prioritized available processing unit. An improved version of HeteroPrio is presented in [20]. It takes into account data locality during task distribution. The main principle is to use different task lists for the different memory nodes and to investigate how to evaluate the locality between the tasks and the different memory nodes without looking at the task dependencies. HeteroPrio is a semi-automatic scheduler that asks users to specify priorities for different types of tasks in their applications. The scheduler is considered to be cost-effective and efficient in its scheduling. However, this approach comes with a limitation. The different types of tasks are involved in various stages of the execution. Setting a priority per type hides relevant information related to a given scheduling scenario. Therefore, we propose a priority per task to address this. This paper presents a novel scheduler that exploits the strengths of the two scheduler families. Our

proposal is based on a set of priority queues based on a binary heap data structure. On the one hand, we use heuristics to assign two scores to each task, first based on task-resource affinity and second based on criticality, and thereby sort the tasks using the scores. On the other hand, we introduce an effective workload distribution across heterogeneous resources using heuristics and an eviction mechanism.

III. Overview of the scheduler

We begin with a comprehensive overview of our proposed scheduler, providing insight into the context of the computing system and the data structures used, without reference to a specific runtime system. In the following section, we proceed with the implementation of the outlined model within StarPU, delving into the specifics of the implementation.

A. Context and notations

A heterogeneous computing system consists of different types of processing units, such as CPUs and GPUs. We denote $\mathcal{A}$ the set of architecture types, i.e., the types of processing units, and $\mathcal{P}$ the set of processing units of the heterogeneous computing system. The system includes a set of memory nodes denoted as $\mathcal{M}$. In our study, we see the main RAM of a computing node as a single memory node despite the NUMA effects but otherwise the approach remains valid. m can be either the main RAM, a GPU-embedded memory or disk memory. We note $\mathcal{P}_m \subset \mathcal{P}$ the subset of processing units tied to $m \in \mathcal{M}$. The notations are shown in table I.

Notation	Description
$\mathcal{A}$	set of architecture types
$\mathcal{P}$	set of processing units
$\mathcal{M}$	set of memory nodes
$\mathcal{P}_m$	subset of processing units tied to $m \in \mathcal{M}$
$\mathcal{P}_a$	subset of processing units of type $a \in \mathcal{A}$
$\mathcal{T}$	set of tasks representing the application
t	a task from the DAG of the application
$\lambda^-(t)$	set of all direct predecessors of t
$\lambda^+(t)$	set of all direct successors of t
$\delta(t,a)$	estimated execution time of t on $a \in \mathcal{A}$

TABLE I: List of notations used in the paper.

A task-based application is represented by a DAG denoted $G = (V, E)$ where the vertices V correspond to the set of tasks. The edge $e \in E$ connecting a pair of vertices corresponds to the precedence relationship between two tasks. In any feasible schedule, for each edge $(t_i, t_j) \in E$, t_j cannot start its execution before the completion of t_i. In this case, t_i is considered as a predecessor of t_j and $\lambda^-(t_j)$ denotes the set of all direct predecessors of t_j. Similarly, t_j is a successor of t_i and $\lambda^+(t_i)$ denotes the set of all direct successors of t_i.

In our context, the runtime system infers the DAG automatically by taking into account the data dependencies between tasks. The construction of the DAG is at runtime, i.e. the whole DAG cannot be known in advance. A task t is seen as *ready* when all its predecessors $\lambda^-(t)$ have finished executing, i.e. a ready task is a task for which all dependencies in the DAG are released. The task can have multiple implementations, i.e., it can be executed on different $\mathcal{A}$. We consider $\delta(t,a)$ the execution time estimation for t computed by processing units $\mathcal{P}_a$. This estimation could be provided by a history-based performance model from the runtime system [21, 22]. The role of the scheduler is to assign ready tasks to the appropriate processing units. We denote W the set of workers. A worker is a software component that is responsible for executing computational tasks on a specific processing unit. We have at least one worker per memory node and expect $|\mathcal{M}| \leq |\mathcal{P}| \leq |W|$. The scheduler intervenes at two key moments at runtime: (i) when a new task is ready meaning its dependencies are released and (ii) when a processing unit is idle and asks for a task for execution. All decisions about distribution, load balancing, etc. are internal to the scheduler.

B. General idea of the MultiPrio scheduler

Each scheduling policy requires generally a data structure to store ready tasks from the moment they become ready until a worker can execute them. We propose a set of priority queues implemented as binary max-heap data structures denoted $\mathcal{H}$ managed by our scheduler.

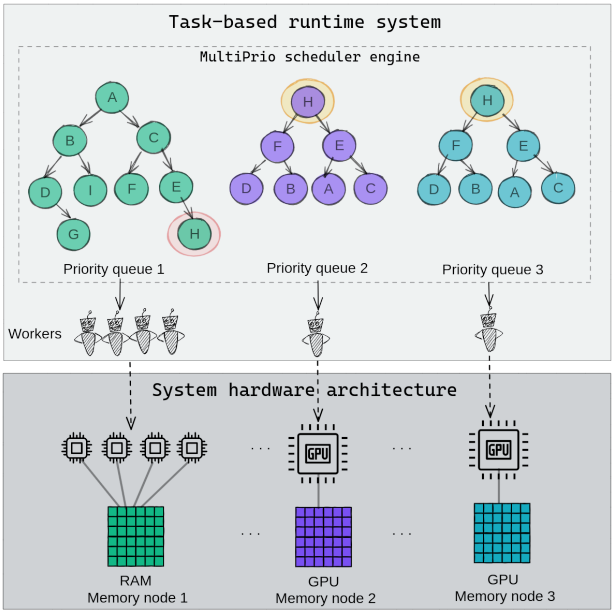


Fig. 1: MultiPrio scheduler example with 2 GPUs and 1 CPU. The tasks in the priority queues are ready for execution and are sorted according to their scores, which can be different depending on the type of processing unit. For example, the ready task H has the highest score on the GPU (and is stored at the top of the queue) and a lower score on the CPU.

For each $m \in \mathcal{M}$, we create a binary heap $h_m \in \mathcal{H}$, i.e., $|\mathcal{H}| = |\mathcal{M}|$. The number of memory nodes in a heterogeneous computing system remains small, which makes the number of binary heaps reasonable without creating a huge overhead in our scheduling policy. The scheduler inserts each ready task t in all respective $h_m \in \mathcal{H}$ such that t can be executed on $\mathcal{P}_m$. Tasks are then duplicated in the heaps.

We show an example of the scheduler data structures in Figure 1. The workers W consume tasks from the priority queues to execute them on the dedicated processing unit. Each worker $w \in W_m$ picks its task from $h_m \in \mathcal{H}$ and executes them on any processing unit in $\mathcal{P}_m \subset \mathcal{P}$. In the example, the first GPU worker that turns idle pulls the task "H" which is the most prioritized. This same task has a low probability of being picked up by a CPU worker, since it is at the bottom of the corresponding priority queue.

Tasks are sorted in the priority queues by scores. When a new task t becomes ready, the scheduler calculates scores for each $m \in \mathcal{M}$ and $a \in \mathcal{A}$ using various heuristics. These heuristics include the acceleration of t on each $\mathcal{P}_a$ and a heuristic indicating if t is critical to release based on the state of the DAG. We describe the proposed heuristics for computing task scores in section V. The scheduler then inserts t into the appropriate priority queues, and its score determines its priority compared to other tasks. When a worker asks for a task, the scheduler attempts to minimize data movement between memory nodes due to its cost. Finally, we propose an eviction mechanism that ensures that when efficient workers are available, less efficient workers don't take their tasks. This approach helps to balance tasks effectively among available resources based on their affinities. More details are given in section V.

IV. Scheduler implementation

A. Scheduling in StarPU

Scheduling in StarPU is an essential part of its task-based programming model. Tasks can have different implementations specific to a type of processing, such as CPU or GPU. This means that the performance of a task can vary significantly depending on the processing unit it is assigned to. StarPU has scheduling policies to determine which task to assign where and when, based on factors like task characteristics and resource affinities. Custom scheduling policies can be defined through the following key operations:

- PUSH operation, when a task becomes ready to execute, i.e. it does not wait for certain data dependencies. Schedulers making decisions at this level are task-centric.
- POP operation, when a worker is idle and asks for a task to execute. Schedulers making decisions at this level are resource-centric.

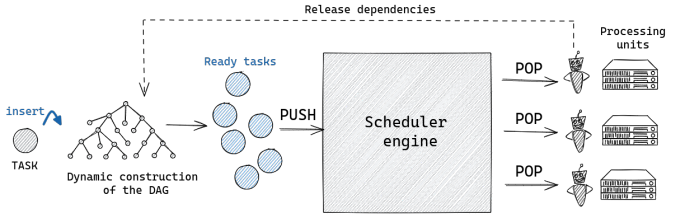


Fig. 2: Dynamic scheduling overview in StarPU.

The scheduling policies require a data structure to store the tasks between the time they become available and the time a worker can perform them. Typically, the scheduler engine contains at least one queue of tasks for this purpose. In the following, we describe our proposed scheduler, which intervenes in both PUSH and POP operations, implemented in the StarPU runtime system.

B. Scheduler mechanisms

To manage the ready tasks between the PUSH and the POP operations, we use a set of binary max-heaps managed by the scheduler. For each memory node $m \in \mathcal{M}$, we have a binary max-heap data structure representing a priority queue such that in each node we have the ready task and its two scores computed by the heuristics presented later (Subsections V-A and V-B). Each binary max-heap root is the task with the highest score. Using this data structure, we can determine the number of tasks ready at any point during the scheduling process on each processing unit p_m tied to a memory node $m \in \mathcal{M}$. If a task can be executed by multiple processing unit types or the processing unit type is tied to multiple memory nodes, it may be duplicated across several priority queues. When a worker picks a task from a given priority queue, any duplicate tasks in other priority queues will remain there. Instead, when workers try to select these duplicates, they will recognize that they have already been processed and remove them. For the implementation of the binary max-heap, we define the insert and delete mechanisms. The scheduler acts on PUSH and POP operations detailed in the following.

Algorithm 1 PUSH operation in the scheduler engine

```
enabled_archs ← {0}
for m = 1 to M do
  a ← get_memory_node_arch_type(m)
  if can_exec(t, a) and get_worker_count(a) > 0 then
    enabled_archs[m] ← 1
    gains[m] ← get_gain_score_normalized(t, a)
    prios[m] ← get_prio_score_normalized(t)
    ready_tasks_count[m]+ = 1
    if normalized_speedup(t, a) == 1 then
      best_remaining_work[m]+ = δ(t, a)
    end if
  end if
end for
heaps_insert(heaps, t, enabled_archs, gains, prios)
```

The PUSH operation in our scheduler engine is presented in the algorithm 1. We indicate the binary heaps on which the task can be executed. For each memory node, if there is an implementation of the task on the respective processing unit type $\mathcal{P}_m$ of architecture type a, we compute the scores. We use two scores to sort the inserted tasks in each binary heap. We first sort the tasks using the gain heuristic detailed in Subsection V-A. If two tasks have equal scores, we then sort them using the criticality heuristic (Subsection V-B). We insert the task into its respective binary heap, or binary heaps if there are multiple implementations or resources.

Algorithm 2 POP operation in the scheduler engine

$w \leftarrow worker\_get\_id()$
$w\_a \leftarrow get\_worker\_arch\_type(w)$
$w\_m \leftarrow get\_worker\_worker\_node(w)$
$task\_found \leftarrow 0$
$nb\_tries \leftarrow 0$
while $!task\_found$ **and** $nb\_tries \leq MAX\_TRIES$ **do**
  $t_{prio} \leftarrow get\_most\_local\_prio\_task(heaps, w\_m)$
  if $pop\_condition(t_{prio}, w\_a) == 1$ **then**
    $task\_found \leftarrow 1$
    $ready\_tasks\_count[w\_m] - = 1$
    $best\_remain\_work[m] - = \delta(t_{prio}, w\_a)$
    $heaps\_pop\_and\_update(heaps, t_{prio}, w\_a)$
  else
    $heaps\_evict\_task(heaps, t_{prio}, w\_a)$
  end if
  $nb\_tries + = 1$
end while

The POP operation in our scheduler engine is described in Algorithm 2. When a worker is idle and requests a task, the scheduler prioritizes the locally available task with the highest data priority. It then evaluates whether the worker is a suitable match based on the $pop\_condition$. This heuristic helps the scheduler decide whether or not it is prudent to pop a task from a binary heap for a particular processor (see details in subsection V-D). If the condition is met, the scheduler pops the task and updates the relevant data. Otherwise, it evicts the task from the current queue and continues trying to pop the next prioritized task until it either succeeds or reaches a maximum number of attempts.

V. Proposed heuristics

Our goal is to compute the different $score(t, a)$ for a newly ready task t, to push t into all binary heaps targeting processing units $\mathcal{P}_m$ of architecture type a that can execute the task. All values are normalized between 0 and 1.

A. Gain heuristic

To compute $score\_gain(t, a)$, we take into consideration the processing units available to perform a task. If there is only one type of processing unit that can perform the task, the gain heuristic is set to 1. If the processing unit type is the fastest of all available processing units, the gain heuristic is calculated using the execution time on the second fastest processing unit and the highest gain. Otherwise, the gain heuristic is calculated using the execution time on the fastest processing unit and the highest gain. By incorporating this gain heuristic into our scheduling algorithm, we can improve the overall efficiency of the system by selecting the best processing type for each task. The gain heuristic that we propose in our scheduling algorithm is defined by the following formula:

$$gain(t,a) = \begin{cases} 1 & |\mathcal{A}| = 1 \\ \frac{(\delta(t,a_{2nd}) - \delta(t,a)) + |hd(a)|}{2 * |hd(a)|} & a \text{ is the fastest} \\ \frac{(\delta(t,a_{1st}) - \delta(t,a)) + |hd(a)|}{2 * |hd(a)|} & \text{else} \end{cases} \quad (1)$$

where $hd(a)$ is the highest execution time difference recorded so far on the processing unit type a.

We show an example in Table II with three tasks t_A, t_B and t_C and two processing unit types a_1 and a_2. Using the gain heuristic formula(1), we calculate the heuristic scores for each task on each architecture type. The calculation of the gain heuristic gives us the order of priority in each heap. In the binary heap relative to a_1, we find that t_A is the most prioritized then t_B then t_C with a lower score. However, in the binary heap relative to a_2, C is the most prioritized then B then A. This example illustrates how the gain heuristic optimizes task prioritization for efficient resource allocation in our scheduling algorithm, ultimately taking full advantage of all available heterogeneous resources to optimize the overall performance of the application.

	t_A	t_B	t_C
$\delta(t, a_1)$	1ms	5ms	20ms
$\delta(t, a_2)$	20ms	10ms	10ms
$gain(t, a_1)$	1	0.631	0.236
$gain(t, a_2)$	0	0.368	0.763

TABLE II: Example of the gain heuristic calculation with 3 tasks and 2 architecture types. Here, $hd(a_1) = hd(a_2) = 19$.

B. Tasks criticality

Computing task priorities in a DAG is a well-studied but challenging problem. The complexity arises when using dynamic scheduling because we do not have the complete DAG in advance, making it difficult to anticipate upcoming tasks. The scheduling decisions must be made on the fly at runtime. Therefore, we focus on lightweight strategies that work on a partial view of the DAG. The challenge is to design heuristics that are both highly effective and operationally efficient, while avoiding excessive complexity. Therefore, to compute $score\_criticality(t, a)$, we use the same metric Normalized Out-Degree (NOD) as in [23]. We suppose that we can retrieve the set of tasks that will be released when t is computed, i.e., its successors, denoted $\lambda^+(t)$. The successors could target different processing units, therefore, we note $\lambda^+(t, \mathcal{P}_m)$ the subset of tasks that will be released for $\mathcal{P}_m$. The same logic applies to the predecessors denoted $\lambda^-(t, \mathcal{P}_m)$. We consider the criticality of the task based on the metric $NOD(t)$ and we calculate the ratio:

$$NOD(t) = \sum_{s_i \in \lambda^+(t, \mathcal{P}_m)} \frac{1}{|\lambda^-(s_i, \mathcal{P}_m)|} \quad (2)$$

In Figure 3, we present an illustrative DAG that represents task dependencies. In particular, this example represents the

situation during dynamic scheduling when tasks are marked as ready. In the example, tasks 2 and 3 are ready to be executed and are marked in yellow. To make informed scheduling decisions, we apply the NOD heuristic to determine which of these ready tasks should have a higher priority. Using the formula above, we get NOD(T2)=2.5 and NOD(T3)=1. Task 2 has a higher priority according to this heuristic. In fact, it has more successors that will be released, which will create more workload and improve the parallelism.

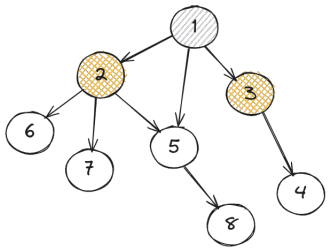


Fig. 3: Example of a DAG. Two released tasks are marked as ready to execute (yellow nodes), and the NOD heuristic is applied to determine the most prioritized task.

C. Data locality

Data locality is an important consideration when scheduling tasks on heterogeneous computing nodes. To minimize data transfers between nodes and increase efficiency, our scheduling algorithm takes data locality into account when selecting tasks in the POP operation. In fact, the scheduler pops the most local task among the first n tasks in the heap, where "local" refers to tasks that physically have the data they need to process on the memory node. We only consider tasks with scores close to the highest priority task, where the score difference is within a certain threshold (denoted as ϵ). We use the LS_SDH^2 [20] heuristic, which stands for Locality Strategy - Sum of Data Hosted.

$$LS\_SDH^2(m,t) = \left(\sum_{d \in D^R_{t,m}} d.size \right) + \left(\sum_{d \in D^W_{t,m}} d.size^2 \right) \tag{3}$$

Here, $D_{t,m}$ is the set of data used by task t that resides on memory node m. $D^R_{t,m}$ and $D^W_{t,m}$ are the sets of data used by t that resides on m and is accessed in read and write mode, respectively. The LS_SDH^2 is the score obtained by summing the amount of data already on a node, with each data write counted in a quadratic manner. This approach optimizes task execution on heterogeneous compute nodes by minimizing data transfer requirements. Selecting tasks based on their proximity to the required data increases efficiency and speeds up task processing, reducing the time and bandwidth required to transfer data between memory nodes.

D. Eviction mechanism

In our proposed scheduler, we incorporate an eviction mechanism along the $pop\_condition$ to improve workload distribution and resource utilization efficiency. In particular, we identified this challenge when the system is close to a starvation state, i.e., there are fewer ready tasks available. This can happen towards the end of an application's execution, or when there are unfulfilled dependencies in the DAG that result in fewer ready tasks. When this phenomenon occurs, there is a risk that workers with slower implementations will pop the task, and potentially increase the makespan. To mitigate this, we have implemented the $pop\_condition$ algorithm in our scheduler.

When a worker attempts to pop a task, the scheduler examines the $pop\_condition$. First, if the current worker is considered the best, i.e., with the fastest estimated execution time, the task is immediately assigned. Otherwise, the scheduler compares the $best\_remaining\_work$ to the task estimated execution time on the current worker. The $best\_remaining\_work$ is maintained and updated as described in the PUSH and POP algorithms. If $best\_remaining\_work$ is larger, the condition is favorable and the task is assigned to the current worker. In fact, in cases where the best worker is sufficiently busy, we allow the task to go to a slower worker to maintain progress in the DAG, thereby improving overall performance. If this is not favorable, our eviction mechanism will remove the task from the current queue and another worker will attempt to execute it. To summarize, the $pop\_condition$ is valid when the best worker asks for the task or another worker satisfies the $best\_remaining\_work$ condition. This dynamic assignment strategy optimizes task distribution and resource utilization throughout the scheduling process.

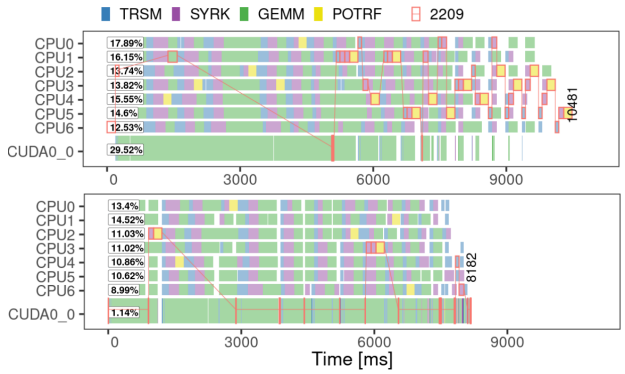


Fig. 4: Simulated scheduling traces with StarPU over SimGrid for Cholesky factorization of a 960x20 matrix on 1 GPU and 6 CPUs. MultiPrio scheduler traces with (bottom) and without eviction mechanism (top). The figure highlights the practical critical path (the tasks with a red border), the percentage of idle time per resource (left) and the makespan (right).

We study the impact when using this heuristic. We simulate the execution on a node with one GPU and 16 CPUs using StarPU over SimGrid, a versatile simulator of distributed systems [24, 25]. We show, in Figure 4, using StarVZ [26], the benefit of using the eviction mechanism in our scheduler by comparing the execution traces. MultiPrio, with the eviction mechanism, makes significantly better decisions at the end of

execution, reducing the percentage of GPU idle time from 29% to 1%.

VI. Experimental evaluation

To evaluate the MultiPrio scheduler and compare its potential with the existing state-of-the-art schedulers in StarPU, we perform the execution of three types of task-based applications. First, a **regular** application, CHAMELEON[1] which is a dense linear algebra library. Second, two **irregular** applications, TBFMM[2] which is a task-based FMM and QR_MUMPS[3] which is a sparse direct linear solver. The experiments are conducted on two platforms, which are presented below.

- **Intel-V100**: The architecture is composed of 2 Intel Xeon Gold 6142 of 16 cores each running at 2.6GHz, 384 GB of memory, and 2 Nvidia V100 (16 GB).
- **AMD-A100**: The architecture is composed of 2 AMD Zen3 EPYC 7513 of 32 cores each running at 2.6GHz, 512 GB of memory, and 2 Nvidia A100 (40 GB).

Both platforms have GCC v10.2.0, NVCC v11.4.120 and the Intel MKL library. We use all available computing resources of both platforms. We empirically set the hyperparameters of the data locality heuristic as follows: $n = 10$ and $\epsilon = 0.8$.

We evaluate the performance of our scheduler through a comparative analysis using two different state-of-the-art schedulers: Dmdas and HeteroPrio. The Dmdas scheduler exploits task priorities provided by user knowledge and relies on a heft-based heuristic for resource allocation, while the automated HeteroPrio scheduler uses priorities per task type according to architecture affinity. The former is a task-centric scheduler, while the latter is an affinity-based scheduler. Notably, in the context of StarPU, there is the LWS scheduler, which is categorized under resource-centric schedulers. However, we excluded it from our comparison because it is not optimized to take full advantage of GPU accelerators. It treats CPUs and GPUs as identical resources, ignoring the heterogeneous potential of the system.

A. Dense linear algebra

We conduct a performance evaluation of widely used dense linear algebra kernels: `potrf`, `getrf`, and `geqrf`. Chameleon is a regular application where each routine will have the same form of the DAG independent of the data. Thus, it provides user priorities for these routines, optimized by experts offline, and uses a fine-tuned task submission strategy. In this application, task criticality is the primary factor that affects performance, followed by resource management (affinity and data locality). In our analysis, we compare the performance of the schedulers, considering the impact of different tile sizes on both task granularity and overall system performance. For the AMD-A100, we set the tile sizes to 960, 1920, and 3840, and for the Intel-V100, we consider tile sizes of 640, 1280, and 2560. For each combination of tile size and scheduler, we run experiments over different matrix sizes and select the best performing configuration to get a fair view of the performance of the routines. We conduct a performance evaluation of the widely used Cholesky factorization (`potrf`). We perform an identical experiment with the dense LU decomposition without pivoting (`getrf`). It has similarities with the Cholesky decomposition and the same diamond-shaped DAG structure. However, due to its non-symmetric nature, LU has a larger workload and induces more memory transfers. Finally, we evaluate the dense QR factorization routine (`geqrf`).

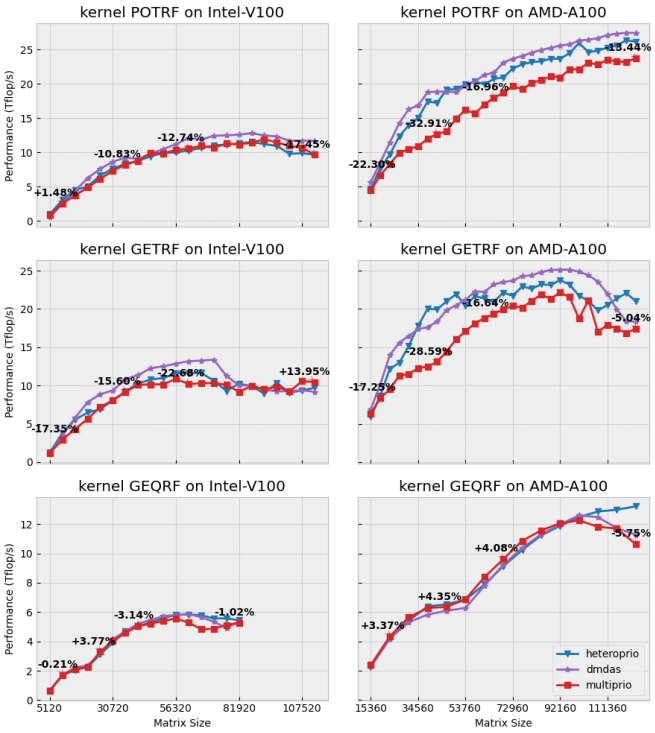


Fig. 5: Performance of CHAMELEON dense kernels on Intel-V100 and AMD-A100, both with 2 GPUs, on various matrix sizes showing MultiPrio gains/losses over Dmdas.

In the Figure 5, our overall performance remains comparable to other schedulers, with variations depending on different scenarios. While there are cases where we are less efficient, there are also cases where our scheduler outperforms, achieving almost a 14% performance gain over Dmdas on the `getrf` kernel for matrices larger than 100k on Intel-V100. This gain is attributed to data transfer issues encountered by Dmdas, likely related to GPU memory limits or conflicts between prefetching and memory eviction mechanisms [27]. For the `potrf` and `getrf` kernels, and especially on AMD-A100, we observe a larger performance gap. In fact, criticality plays a key role in scheduling such DAGs. Although the NOD heuristic is dynamic and fully automated, it provides less precise results than user-predefined priorities, which explains the effectiveness of Dmdas. In addition, for such intensive workloads, we suspect that the execution process is particularly affected by

[1]https://gitlab.inria.fr/solverstack/chameleon
[2]https://gitlab.inria.fr/bramas/tbfmm
[3]https://gitlab.com/qr_mumps/qr_mumps

data transfer times, and optimizing these is critical for high-performance GPUs. Our scheduler mapping decision is made at the POP operation, as opposed to Dmdas, which makes the mapping at the PUSH operation and thus can request data prefetching in advance. For the `geqrf` kernel, scheduler performance is generally competitive. MultiPrio shows slightly improved performance, particularly noticeable on AMD-A100 for matrix sizes between 60k and 90k, achieving a performance gain of about 4%.

B. Fast Multipole Method

FMM is an important pairwise particle interaction algorithm widely used in astrophysical simulations, molecular dynamics, and more. Task-based FMM, combined with dynamic scheduling, allows efficient handling of diverse particle distributions on multi-core and heterogeneous platforms. In our study, we compare the performance of the schedulers on Intel-V100 and AMD-A100 while varying the GPU streams for optimization. In this application, the priorities of the tasks are not set by the user. In the Figure 6, MultiPrio stands out for achieving the shortest makespan. In fact, the DAG of the TBFMM is very disconnected, so the critical path with infinite resources is very short. To be efficient, the scheduler should be able to achieve perfect workload balancing among the heterogeneous resources and take resource affinities into careful consideration. The results show that this is less favorable for Dmdas because it is task-centric. HeteroPrio uses priorities per task type for each processor type. This gives a good distribution. However, MultiPrio manages to be more efficient thanks to the gain scores per task, providing more accurate priorities.

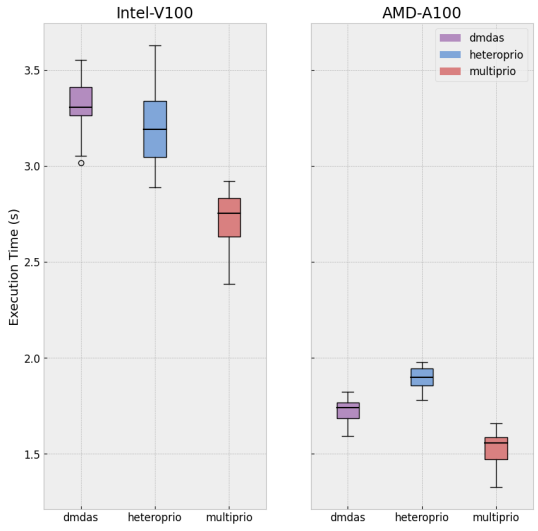


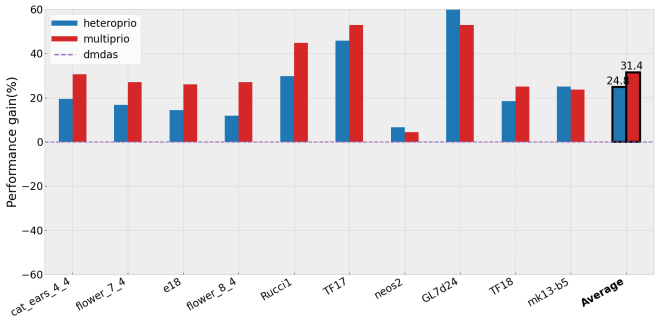
Fig. 6: Comparison of execution time of TBFMM on Intel-V100 and AMD-A100 both with 2 GPUs (10^6 particles, tree height 6).

C. Sparse linear algebra

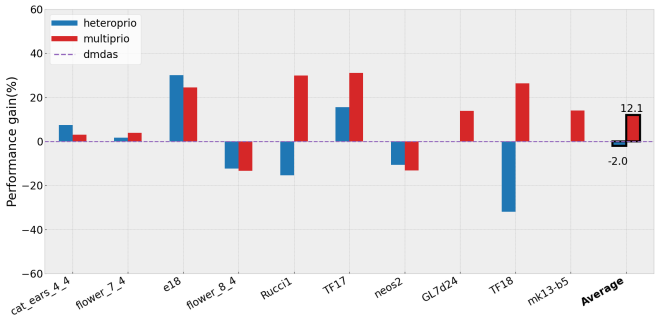
We extend our experimental evaluation to a more irregular application QR_MUMPS. We evaluate QR multi-frontal factorization on sparse matrices, which introduces highly irregular workloads, including tasks of different granularities and characteristics, while exhibiting variable memory consumption. Typically, schedulers face greater challenges in optimizing such scenarios. In Figure 8, we present the results of the sparse QR factorization using QR_MUMPS on the two platforms using the ordering library METIS [28]. We use four streams on each GPU. The performance ratio of each scheduler is represented in comparison to the Dmdas scheduler, which is used as a reference. Higher ratios indicate better results, i.e. shorter overall completion times. Our experiment use a set of sparse matrices, which are detailed in Table 7. In this application, the fine-grained priorities of the tasks are not set by the user.

Matrix name	Rows	Cols	Nonzeros	op.count (Gflop)
cat_ears_4_4	19020	44448	132888	236
flower_7_4	27693	67593	202218	889
e18	24617	38602	156466	1439
flower_8_4	55081	125361	375266	3072
Rucci1	1977885	109900	7791168	5527
TF17	38132	48630	586218	15787
neos2	132568	134128	685087	31018
GL7d24	21074	105054	593892	26825
TF18	95368	123867	1597545	229042
mk13-b5	135135	270270	810810	352413

Fig. 7: Matrices used for QR_MUMPS sorted by Gflops count.



(a) Performance on Intel-V100 platform.



(b) Performance on AMD-A100 platform.

Fig. 8: Performance of QR factorization (ordering METIS) on Intel-V100 and AMD-A100 both with 2 GPUs, relative to the Dmdas scheduler. Matrices sorted by Gflops count.

MultiPrio outperforms the other schedulers for most ma-

trices on Intel-V100, with an average performance gain of 31% over Dmdas. However, the results on AMD-A100 show some variation. In fact, it is a more heterogeneous platform because it has twice as many CPUs as Intel-V100, but each CPU is 2x slower, and the GPUs are much faster. Therefore, assigning a large critical task to a CPU instead of a GPU would result in a longer execution time, ultimately impacting overall performance. As a result, scheduling irregular workloads is more difficult on AMD-A100, which explains the variance in overall scheduler performance. MultiPrio achieves an average performance gain of about 12% over Dmdas and up to 20% for the larger matrices that provide a more suitable load for leveraging the heterogeneous resources of this platform.

VII. Discussion and future work

In this section, we discuss the various challenges faced by each scheduler, based on our extensive experimental study involving several different DAGs. It's crucial to recognize the strengths of the Dmdas scheduler, which has been highly tuned for scheduling dense linear algebraic routines for the last decade, especially with Cholesky factorization. However, Dmdas has weaknesses in over-prioritizing accelerators over CPUs, resulting in under-utilization of CPUs in a heterogeneous environment. It generally undervalues data locality because it considers choosing the best worker for a task in terms of end-of-execution time to be more important than using the data available on a node, resulting in huge amounts of data being transferred. Despite Dmdas's well-established strengths in dense routines, MultiPrio offers advantages in scenarios where applications have more irregular tasks. In the context of sparse factorization, particularly the multi-frontal QR factorization, our results are far more promising. This is mainly attributed to the increased parallelism in the DAG. In fact, Agullo et al. [29] highlight that they propose a front partitioning strategy that effectively utilize both CPU and GPU resources. It optimizes parallelism in the DAG while efficiently utilizing GPUs with appropriately sized tasks. By employing a different scheduling strategy, our scheduler better distributes tasks across different resources, leading to improved efficiency in most cases (demonstrated with the sparse QR and TBFMM cases). Therefore, the choice between Dmdas and MultiPrio should be based on the specific characteristics of the workload of the parallelized application.

Nevertheless, there is still room for improvement. Potential performance issues in MultiPrio can arise from the task assignment at the POP, which affects the effectiveness of the prefetching strategy. This issue is particularly prevalent on AMD-A100, which is characterized by high-speed GPUs capable of rapidly executing a large number of tasks. As a result, ensuring timely data access becomes paramount. Our proposed strategy emphasizes prioritizing data locality at the POP, favoring the selection of tasks with the most readily available data among those with the highest priority. This methodology aims to reduce the reliance on prefetching mechanisms and improve the optimization of data transfers. Further refinement of the data locality strategy could be an area of improvement to close the gap between our scheduler and others that have benefited from prefetching strategies on AMD-A100 alike platforms. Another aspect that can affect the performance is underestimating the remaining work of the best worker and instead deciding to let a CPU worker perform a task, such as a matrix-matrix multiplication that is 20x slower on CPU, can significantly affect the makespan. A future improvement could be to refine the estimation of the remaining work for the best workers at runtime to better enlighten the scheduler's decisions. Our scheduler is designed with the specific goal of optimizing the use of all heterogeneous resources, as opposed to traditional techniques that primarily maximize accelerator usage. The novelty of our scheduler lies in its ability to automatically and efficiently schedule irregular applications without requiring user expertise, while being competitive with highly tuned schedulers on a more regular workload.

As part of our future work, we aim to extend this to incorporate energy efficiency heuristics to take advantage of the CPUs and re-balance the workload between them and the accelerators without compromising overall performance. The fact that our scheduler has proven its efficiency in scheduling tasks across heterogeneous resources will help in achieving this delicate balance. Another interesting research direction is the scheduling of applications with hierarchical tasks. Indeed, these tasks submit subgraphs at runtime. They have recently been included in StarPU [30]. This strategy exposes different task sizes in the DAG, providing a sufficient amount of large-granularity tasks to efficiently utilize GPUs, along with fine-granularity tasks to take advantage of CPUs and thus unlock more parallelism. Such scenarios are similar to QR_MUMPS, and that's why we expect better results than Dmdas when scheduling hierarchical tasks.

VIII. Conclusion

In summary, effective utilization of heterogeneous computing systems for resource-intensive applications depends on efficient scheduling. We present MultiPrio, an automatic dynamic task scheduler with the specific goal of optimizing the use of all heterogeneous resources on a compute node, as opposed to traditional techniques that primarily maximize accelerator utilization. Our approach uses binary heap data structures. Each task is assigned a pair of scores, first an affinity score using the gain heuristic, and then a criticality score. The scores are used to sort tasks without immediately deciding which worker is responsible for executing them. This flexibility allows slower workers to take over tasks when faster workers are too busy, under certain conditions related to the estimated remaining workload. In addition, we incorporate a data locality heuristic to minimize data transfers between resources, thereby improving overall performance. MultiPrio has proven to be efficient for distributing irregular workloads across heterogeneous resources. For example, on the QR_MUMPS application, we observed performance improvements of up to 40% over the state-of-the-art Dmdas scheduler. Our work has highlighted the strengths of MultiPrio while identifying areas

for potential improvement. With ongoing research and development, the MultiPrio scheduler holds promise for further advances in task-based programming and dynamic scheduling in heterogeneous computing systems.

ACKNOWLEDGMENTS

This work is supported by the TEXTAROSSA project G.A. n.956831, as part of the EuroHPC initiative. Experiments presented in this paper 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).

REFERENCES

[1] P. Thoman, K. Dichev, T. Heller, R. Iakymchuk, X. Aguilar, K. Hasanov, P. Gschwandtner, P. Lemarinier, S. Markidis, H. Jordan, T. Fahringer, K. Katrinis, E. Laure, and D. S. Nikolopoulos, "A taxonomy of task-based parallel programming technologies for high-performance computing," *J. Supercomput.*, vol. 74, no. 4, p. 1422–1434, apr 2018. [Online]. Available: https://doi.org/10.1007/s11227-018-2238-4

[2] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi, "Task-based fmm for multicore architectures," *SIAM Journal on Scientific Computing*, 2014.

[3] ——, "Task-based fmm for heterogeneous architectures," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 9, pp. 2608–2629, 2016. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3723

[4] J. M. C. Carpaye, J. Roman, and P. Brenner, "Design and analysis of a task-based parallelization over a runtime system of an explicit finite-volume CFD code with adaptive time stepping," *Journal of Computational Science*, 2018.

[5] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. J. Dongarra, "Parsec: Exploiting heterogeneity to enhance scalability," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 36–45, 2013.

[6] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011. [Online]. Available: https://hal.inria.fr/inria-00550877

[7] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez, "Task-based multifrontal qr solver for gpu-accelerated multicore architectures," in *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*, 2015, pp. 54–63.

[8] F. Lopez and I. Duff, "Task-Based Sparse Direct solver for Symmetric Indefinite Systems," 2018, 10th International Workshop on Parallel Matrix Algorithms and Applications (PMAA), mini-symposium on task-based programming for scientific computing.

[9] C. Flint, L. Paillat, and B. Bramas, "Automated prioritizing heuristics for parallel task graph scheduling in heterogeneous computing," *PeerJ Computer Science*, vol. 8, p. e969, 2022.

[10] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguadé, and J. Labarta, "Productive programming of gpu clusters with ompss," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, 2012, pp. 557–568.

[11] T. Gautier, J. V. Lima, N. Maillard, and B. Raffin, "Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, 2013, pp. 1299–1308.

[12] J. Kim, S. Lee, B. Johnston, and J. S. Vetter, "Iris: A portable runtime system exploiting multiple heterogeneous programming systems," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, 2021, pp. 1–8.

[13] J. V. Lima, T. Gautier, N. Maillard, and V. Danjean, "Exploiting concurrent gpu operations for efficient work stealing on multi-gpus," in *2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing*, 2012, pp. 75–82.

[14] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguadé, and J. Labarta, "Productive programming of gpu clusters with ompss," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, 2012, pp. 557–568.

[15] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, 2002.

[16] H. Choi, D. Son, S. Kang, J. Kim, H.-H. Lee, and C.-H. Kim, "An efficient scheduling scheme using estimated execution time for heterogeneous computing systems," *The Journal of Supercomputing*, vol. 65, 08 2013.

[17] K. Chronaki, A. Rico, R. M. Badia, E. Ayguadé, J. Labarta, and M. Valero, "Criticality-aware dynamic task scheduling for heterogeneous architectures," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 329–338. [Online]. Available: https://doi.org/10.1145/2751205.2751235

[18] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst, "Data-aware task scheduling on multi-accelerator based platforms," in *2010 IEEE 16th International Conference on Parallel and Distributed Systems*, 2010, pp. 291–298.

[19] M. Gonthier, L. Marchal, and S. Thibault, "Memory-aware scheduling of tasks sharing data on multiple gpus with dynamic runtime systems," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2022, pp. 694–704.

[20] B. Bramas, "Impact study of data locality on task-based applications through the Heteroprio scheduler," *PeerJ Computer Science*, vol. 5, p. e190, May 2019. [Online]. Available: https://hal.inria.fr/hal-02120736

[21] C. Augonnet, S. Thibault, and R. Namyst, "Automatic calibration of performance models on heterogeneous multicore architectures," in *Euro-Par 2009 – Parallel Processing Workshops*, H.-X. Lin, M. Alexander, M. Forsell, A. Knüpfer, R. Prodan, L. Sousa, and A. Streit, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 56–65.

[22] E. Agullo, B. Bramas, O. Coulaud, L. Stanisic, and S. Thibault, "Modeling Irregular Kernels of Task-based codes: Illustration with the Fast Multipole Method," INRIA Bordeaux, Research Report RR-9036, Feb. 2017. [Online]. Available: https://inria.hal.science/hal-01474556

[23] H. Lin, M.-F. Li, C.-F. Jia, J.-N. Liu, and H. An, "Degree-of-node task scheduling of fine-grained parallel programs on heterogeneous systems," *Journal of Computer Science and Technology*, vol. 34, no. 5, pp. 1096–1108, 2019. [Online]. Available: https://doi.org/10.1007/s11390-019-1962-4

[24] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, "Versatile, scalable, and accurate simulation of distributed applications and platforms," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2899–2917, Jun. 2014. [Online]. Available: http://hal.inria.fr/hal-01017319

[25] L. Stanisic, S. Thibault, A. Legrand, B. Videau, and J.-F. Méhaut, "Faithful performance prediction of a dynamic task-based runtime system for heterogeneous multi-core architectures," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 16, pp. 4075–4090, 2015. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3555

[26] V. Garcia Pinto, L. Mello Schnorr, L. Stanisic, A. Legrand, S. Thibault, and V. Danjean, "A visual performance analysis framework for task-based parallel applications running on hybrid clusters," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 18, p. e4472, 2018, e4472 cpe.4472. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4472

[27] L. Stanisic, S. Thibault, A. Legrand, B. Videau, and J.-F. Méhaut, "Modeling and simulation of a dynamic task-based runtime system for heterogeneous multi-core architectures," in *Euro-Par 2014 Parallel Processing*, F. Silva, I. Dutra, and V. Santos Costa, Eds. Cham: Springer International Publishing, 2014, pp. 50–62.

[28] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998. [Online]. Available: https://doi.org/10.1137/S1064827595287997

[29] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez, "Task-based multifrontal qr solver for gpu-accelerated multicore architectures," in *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*, 2015, pp. 54–63.

[30] G. Lucas, "On the Use of Hierarchical Task for Heterogeneous Architectures," Theses, Université de Bordeaux, Oct. 2023. [Online]. Available: https://theses.hal.science/tel-04316145