RSCHED: An effective heterogeneous resources management for simultaneous execution of task-based applications

Etienne Ndamlabin, Bérenger Bramas

RSCHED: An effective heterogeneous resources management for simultaneous execution of task-based applications

Etienne Ndamlabin, Bérenger Bramas
Inria Nancy – Grand Est, CAMUS Team, Villers-lès-Nancy, France
ICPS Team, ICube, Illkirch, France

May 6, 2024

Abstract

Modern parallel architectures have heterogeneous processors and complex memory hierarchies, offering up to billion-way parallelism at multiple hierarchical levels. Their exploitation of HPC applications greatly boosts scientific discoveries and advances, but it is still hard to utilize them maximally, engendering pro rata high energy consumption. The task-based programming model has demonstrated promising potential in developing scientific applications on modern high-performance platforms. In this work, a new framework for managing the concurrent execution of task-based applications, RSCHED, is introduced. RSCHED aims at minimizing the overall makespan while executing a set of applications, and maximizing resource utilization. We implemented our proposal on StarPU and evaluated it on real applications. RSCHED demonstrated the potential to speed up the overall makespan of ran applications compared to consecutive execution with an average factor of 10x and the potential to increase resource utilization.

Keywords: Heterogeneous resource management, Scheduling, Task-based applications, Gradient descent, StarPU.

Introduction

High-performance computing (HPC) is crucial to making discoveries and advances in several scientific domains (astrophysics, climatology, epidemiology, biology, geology, etc.). HPC offers the ability to perform complex calculations and massive data processing at very high speed by aggregating the power of several thousand processing units, called supercomputers. Supercomputers rely on a complex, heterogeneous, and hierarchical hardware organization. The largest supercomputers are mostly composed of central processing units (CPUs) and

graphical processing units (GPUs) [1], or even Field Programmable Gate Arrays (FPGAs). As parallel systems, they can process several jobs at the same time by scheduling their execution on the available resources.

In the current HPC paradigm, there are schedulers at multiple levels, which all have the same aim: distributing the workload over hardware resources. At the higher level, the batch-scheduler, like Slurm [2], manages the hardware resources of an entire supercomputer by deciding the order of execution of the jobs submitted by the users. Submitted jobs are treated by the batch scheduler as black boxes. This is advantageous because the batch scheduler can run applications implemented with any technology, giving freedom to the programmers. However, this approach might lead to resource wastage and increasing energy consumption. Such a situation can happen when an application suboptimally uses the allocated resources, when an adjustment of resources is required during different phases of execution, or when the resource manager cannot adapt the resources to the workload of newly submitted jobs. Moreover, executing one job after the other is most likely counterproductive, since HPC applications are often composed of interdependent executing kernels, and therefore cannot fully use resources due to their precedence constraints. In the current study, we aim to improve the batch scheduler by using a dynamic resource allocation strategy. Our objective is to improve the executions at the scale of the supercomputer, i.e., to reduce the overall makespan of an application set, and not to focus on a single application only. In addition, our work is tied to the task-based method, as we consider that each application is composed of tasks and that some of them can be executed on CPU, GPU or both.

In this paper, we present a resource manager for heterogeneous environments considering task-based model applications called RSCHED, aiming at optimizing their usage. In RSCHED, we propose strategies for resource distribution between concurrent task-based applications while orchestrating their execution. We have implemented our proposal within the StarPU [1] and analyzed the performance of our proposal via diverse experiments. Our contributions can be summarized as follows:

- We propose a framework for managing the execution of concurrent task-based applications
- We propose strategies for dynamically distributing resources between concurrent task-based applications
- We propose a new model of Gradient Descent in a (three-dimensional) discrete space
- We propose a strategy to automatically create and configure a scheduling context for a given task-based application
- We present performance analysis and results showing the effectiveness of our proposals

[1] `https://www.top500.org/`
[2] `https://www.schedmd.com/`

The remaining sections of the paper are organized as follows. In Section 1, we introduce the notion of task-based application and present the state-of-the-art concerning the scheduling of task-based application under StarPU. Then, in Section 2, we present our proposed resource management for simultaneous execution of task-based applications. Finally in Section 3, we evaluate the performance of our proposals.

1 Background

1.1 Task-based application

Several strategies to parallelize applications on heterogeneous computing nodes aim at maximizing resource usage. The task-based model has demonstrated high potential in various fields [2, 3, 4]. This method allows obtaining hardware-independent algorithm descriptions while developing efficient HPC applications. The level of abstraction and encapsulation relieves the users by shifting the complexity to the runtime systems, where researchers can invest the effort to create generic and efficient optimization solutions. The HPC community has at its disposal highly documented and maintained runtime systems supporting the task-based model, such as Parsec [5], and StarPU [1] a runtime system library developed at Inria Bordeaux.

Among other runtime systems supporting the task-based model, StarPU has a plus in that it has a component – *hypervisor* – allowing concurrent execution of task-based applications with minimal interference [6]. StarPU hypervisor provides confined execution environments – *scheduling contexts* – which can be used to partition computing resources. StarPU scheduling contexts can be dynamically resized and linked to a well-designed scheduler to optimize the allocation of computing resources among concurrent task-based applications/libraries. A scheduler can chosen for each application via its linked scheduling context. Since task-based applications have varied types and structures, a scheduler can be effective only for some applications, or a specific type of machine architecture [7, 8, 9, 10, 11]. StarPU also proposes basic strategies for resizing scheduling contexts and a platform for implementing additional custom ones. However, there is no means to launch several applications or dynamically distribute the workers among them.

Different task-based frameworks have been used to develop efficient HPC applications, such as the Lattice-Boltzmann method [12, 13], the fast-multipole method (FMM) [2, 3, 14], N-body simulations [15], linear algebra solvers [16, 17, 18], H-matrix solvers [19], the particle-in-cell method [20], the polar decomposition method [21], seismic imaging [22, 23, 24], Galerkin solver [25], to mention a few. This demonstrates that at least at a moderate scale and when used by experts, the existing task-based runtime systems can be efficient for various classes of algorithms.

1.1.1 Task-based parallelization

The task-based method divides an application into interdependent sections, called tasks. The dependencies between the tasks ensure valid parallel executions and task execution orders without race conditions. This can be likened to a graph, where the nodes represent the tasks and the edges represent the dependencies. We consider a task-based application as a Directed Acyclic Graph (DAG) $G(V, E)$ where $V = \{t_1, t_2, ..., t_n\}$ is the set of nodes and $E = \{e_{i,j} = (t_i, t_j) | 1 \leq i, j \leq n, i \neq j\}$ the set of edges representing the existing data dependencies between tasks. An edge $(t_i, t_j) \in E$ if there is a precedence constraint between t_i and $t_j \in V$, such that t_j can be executed only after the task t_i is over, and the data made available.

A task t_i is a computational element executable on one or (potentially) several types of hardware and incorporates different interchangeable kernels, each targeting a specific architecture. For instance, a matrix-matrix multiplication task in linear algebra could be either a call to cuBLAS and executed on a GPU, or a call to Intel MKL and executed on a CPU, but both kernels return equivalent results.

1.2 Task scheduling and related work

The scheduling problem on heterogeneous computing systems has been proven NP-complete [26], whether in static or dynamic situations. Dealing with the first situation requires prediction models which are not always accurate, and a knowledge of the complete view of the task graph [27], which need expensive analysis mechanisms and incur significant overhead. The latter one is the most used [28, 29, 30, 24, 31, 21, 3] and has demonstrated its ability to deliver high performance with reduced overhead.

The two main steps of a scheduler are task selection or prioritization, and resource selection. This action can be static or dynamic according to the scheduling situation.

Due to the evolution of computing architectures, task scheduling in heterogeneous computing is an aged but hot topic. Various strategies to schedule tasks have been proposed, and the most famous and widely reused scheduler is Heterogeneous Earliest Finish Time (HEFT) [32]. In HEFT, tasks are prioritized using a heuristic based on a prediction of the processing length of the tasks and the data transfer time between them. Whereas, resource selection is based on a heuristic that determines the resource providing the best finish time for the tasks according to the scheduling decision of previous tasks. Several variances of this approach with more advanced ranking and resource selection models have been proposed [33, 34, 35, 36]. These schedulers have in common the limitations of static schedulers previously argued, and therefore rely on greedy algorithms. In a changing environment, re-prioritizing the tasks could be necessary, which can add more overhead. A larger spectrum of task schedulers can be found in the literature [37, 38].

1.2.1 Task Scheduling in StarPU

Our scheduling process follows the terminology of StarPU. In StarPU, the user first splits the problem into smaller computational tasks. Afterwards, the tasks are implemented into codelets, which are simple C functions. One task can be implemented differently into several codelets according to the targeted hardware, allowing the user to harness special accelerators, such as vectorial CPU cores or OpenCL devices. In StarPU terminology, these devices are called workers. For each task, the user also has to describe precisely the input data, in read mode, and the output data, in write or read/write mode. StarPU considers that a scheduler has an entry point where the ready tasks are pushed, and it provides a request method where workers pop the tasks to execute, as depicted in Figure 2.

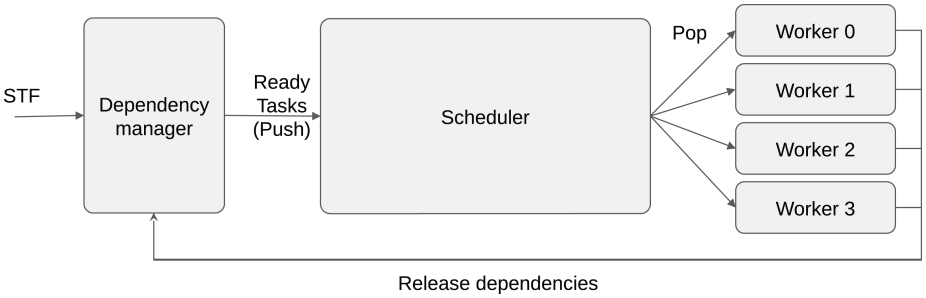


Figure 1: **Schematic view of task-based runtime system organization.** A program can be described using the sequential task flow (STF) model and converted into tasks/dependencies by the RS. When dependencies are released, the newly ready tasks are pushed into the scheduler. When a worker is idle, it calls the pop function of the scheduler to request a task to execute.

In StarPU, both pop/push methods are directly called by the workers that either release the dependencies or ask for a task. Consequently, assigning a task to a given worker means returning this task when the worker calls the pop method. During the execution of a StarPU program, it is possible to choose among several schedulers. The DMDA (deque model data-aware) scheduler is one of the most famous and sophisticated. It uses a HEFT-like strategy and tries to minimize the makespan by using a look-ahead strategy and data transfer costs. Another effective StarPU scheduler is Heteroprio [39, 3], a semi-automatic scheduler designed for heterogeneous machines where users must provide task priorities. A fully automatic version of the Heteroprio scheduler that computes efficient priorities is proposed [7]. Another extension of Heteroprio is the MulTreePrio [8] scheduler based on a set of balanced trees data structure, in which assignment of tasks to available resources is done according to priority scores per task for each type of processing unit. MulTreePrio makes overall good scheduling results thanks to its fast and efficient heuristics, despite the considerable variety of DAG structures from one application to another.

In all of the above, one task-based application is considered for execution/scheduling.

1.2.2 Scheduling concurrent Task-based applications

In general, there is contention for the usage of resources in an HPC environment. Each user's application requests a number of processing resources (CPU/GPU for instance), an entire node or part of it. However, in both cases, it is up to the user to determine the number and type of resources, therefore this might lead in most cases to resource wastage.

The problem presented here has a completely distinct parallelization and resources management approaches in cloud computing and Big-data frameworks, such as Spark[3] or Apache Hadoop[4]. In cloud computing, the scheduler orchestrates different executions from different types of applications, such as Big data programs, over given hardware resources. It has a view on the different operations that compose the executions, hence it can schedule and interleave them finely. However, there is a gap between the programming model and resource management.

In the context of HPC, there is not yet a dynamic solution for concurrent job execution on the same resource as it is done in a cloud environment. In Slurm for instance, there is the notion of Job Array which consists of submitting and managing collections of similar jobs such that they may run in parallel with different input parameters or data on a node. However, it is the responsibility of the programmer to orchestrate the execution of the tasks over the resources. The same problem is encountered when several jobs are submitted separately, but now at the level of the scheduler.

The RECIPE project [40, 41] attempts to control more efficiently the resources without bridging the gap with the applications, which will end as being oriented to cloud computing instead of HPC.

2 Presentation of RSCHED

In this section, we present our concurrent execution approach, named RSCHED. Let us consider that we have n concurrent users' applications requesting resources for execution in a node with p workers (nb_cpus the number of CPUS and nb_gpus the number of GPUS). The main objective of RSCHED is to minimize the overall makespan for all the n applications. A secondary objective of RSCHED is maximizing resource utilization, which is a well-known energy consumption reduction approach.

In this work, we consider that all the n applications have been submitted before the distribution. Before the execution of an application, a task scheduler is associated with a separate context linked with it. The required resource distribution can be done before any task is pushed, or after all tasks have been

[3] https://spark.apache.org/
[4] https://hadoop.apache.org/

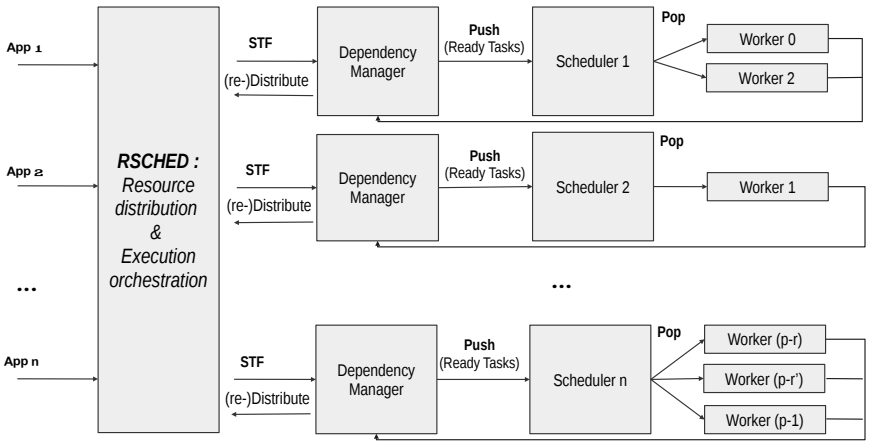


Figure 2: **Schematic view runtime system organization with n concurrent task-based applications and p workers.**

pushed and no task executed. In the beginning, the lack of information on the applications makes it difficult to process an advanced distribution strategy. That is why in the first case, a naive strategy can be used to distribute the workers between the contexts, and afterward a more advanced distribution strategy. In the second case, we can have sufficient information on the applications to process an advanced distribution strategy before no task is executed.

A compromise between having all information before starting and starting sooner is to process an advanced distribution at an arbitrary time after the tasks have started to be pushed and executed. That time could be for instance when any first application completes its execution, or after all tasks have been pushed. The execution of tasks starts as soon as possible, and afterward, we re-distribute workers to the rest of the applications to balance the load and therefore minimize the overall makespan. When distributing the resource, it is possible that two contexts share a worker, but not without a performance penalty, due to context switches.

2.1 RSCHED API

To use the API of RSCHED for resource distribution among some task-based applications, there is information to provide, that are to their performances on the targeted hardware. We assume in this work that we have two types of workers, CPUs and GPUs. For each application, the required information is the following.

- CPU_W: total (sequential) CPU workload of the graph on the targeted CPU.

- GPU_W: total (sequential) GPU workload of the graph on the targeted GPU.
- CPU_{PW}: total (sequential) pure CPU workload of the graph on the targeted CPU.
- GPU_{PW}: total (sequential) pure GPU workload of the graph on the targeted GPU.

By pure CPU (resp. GPU) workload, we mean the workload of tasks that can only be executed by a CPU (resp. GPU). Providing those pieces of information implies knowing (an approximation of) the processing time of each task in the application per type of worker. There are several means of obtaining such information, like Machine Learning, or history-based performance models as it exists in StarPU.

The constraints over this information are given by the Rule 1.

Rule 1 *Given the required information as defined above, either the total workload is equal to the pure one for all the types of workers, or strictly greater than the pure one for all (see the systems 1 and 2).*

$$CPU_W \geq CPU_{PW} \ \textbf{\textit{AND}} \ GPU_W \geq GPU_{PW} \tag{1}$$

$$\begin{array}{l}(CPU_W = CPU_{PW} \ \textbf{\textit{AND}} \ GPU_W = GPU_{PW}) \ \textbf{\textit{OR}} \\ (CPU_W > CPU_{PW} \ \textbf{\textit{AND}} \ GPU_W > GPU_{PW})\end{array} \tag{2}$$

The proof of system 1 is obvious. For system 2, there are two cases to have equality: all the tasks are either pure CPU or pure GPU. In both cases, the other type of worker will have zero workload.

2.1.1 RSCHED resource distribution

Given the n applications with the required information described in the last section, a distribution strategy should produce the following information for each graph.

- L_{CPUS}: list of CPUs assigned.
- L_{GPUS}: list of GPUs assigned.
- $\#CPUS$: number of distinct CPUs assigned ($\#CPUS = |L_{CPUS}|$).
- $\#GPUS$: number of distinct GPUs assigned ($\#GPUS = |L_{GPUS}|$).
- $CPUS_{PR}$: The power rate of the assigned CPUs ($0 \leq CPUS_{PR} \leq \#CPUS$).
- $GPUS_{PR}$: The power rate of the assigned GPUs ($0 \leq GPUS_{PR} \leq \#GPUS$).

Rule 2 *Each application must receive at least one worker, and applications can share all the workers.*

$$\begin{array}{l}(\#CPUS + \#GPUS > 0.0)\ \boldsymbol{AND} \\ (0.0 \leq \sum_{0 \leq r \leq n} \#CPUS_r \leq n \times nb\_cpus)\ \boldsymbol{AND} \\ (0.0 \leq \sum_{0 \leq r \leq n} \#GPUS_r \leq n \times nb\_gpus)\end{array} \tag{3}$$

Rule 3 *Each application must receive at least one worker per type of pure workload.*

$$\begin{array}{l}(CPU_{PW} = 0.0\ \boldsymbol{OR}\ (CPU_{PW} \neq 0.0\ \boldsymbol{AND}\ \#CPUS > 0.0))\ \boldsymbol{AND} \\ (GPU_{PW} = 0.0\ \boldsymbol{OR}\ (GPU_{PW} \neq 0.0\ \boldsymbol{AND}\ \#GPUS > 0.0))\end{array} \tag{4}$$

An estimation of the makespan of each application is used as a building block of our strategies, given a set of workers (CPUs/GPUs) assigned to the applications. We proposed an estimation called "Ideal Makespan", and more details are given in Appendix (see Algorithm 1). The following metrics are used in the evaluation of our "Ideal Makespan".

When an application has a pure workload for a given type of worker, there is a minimum length constraint over its makespan. We define it as equations 5 and 6.

$$tgpu_{minM} = \frac{GPU_{PW} \times coef\_par\_eff^{\#GPUS+\#CPUS-1}}{\#GPUS} \tag{5}$$

$$tcpu_{minM} = \frac{CPU_{PW} \times coef\_par\_eff^{\#GPUS+\#CPUS-1}}{\#CPUS} \tag{6}$$

In the case no CPUs (resp. GPUs) are assigned unto the application, $tcpu_{minM}$ (resp. $tgpu_{minM}$) is equal to zero.

The general formulation of our makespan estimation is given by equation 7.

$$ideal\_makespan = MAX(tcpu_{minM}, tgpu_{minM}) + \frac{cpu\_rem\_wl}{\#CPUS + \#GPUS \times \frac{cpu\_rem\_wl}{gpu\_rem\_wl}} \tag{7}$$

Where cpu_rem_wl (resp. gpu_rem_wl) is the exceeding CPU (resp. GPU) workload compared to the gap between $tgpu_{minM}$ and $tgpu_{minM}$, and the CPU/GPU (resp. GPU/CPU) speedup.

In this work, we present four distribution strategies: LpSolve, MinMaxWL (Min-Max Workload balancing), DSR-CLUS (Dedicated plus Shared Resource with Clustering) and DSR-GD (Dedicated plus Shared Resource with Gradient Descent).

LpSolve In this strategy, we rely on the linear programming model presented in a previous study [3]. Originally, this model was employed to compute an ideal makespan (a theoretical lower bound) for tasks executed on heterogeneous architectures. The model is given by:

$$\begin{cases} Objective\,function : min(T) \\ \sum\limits_{\omega\,in\,\Omega} \alpha_1^{\omega}\, t_1^{\omega} = t_1 \leq T \\ \sum\limits_{\omega\,in\,\Omega} \alpha_2^{\omega}\, t_2^{\omega} = t_2 \leq T \\ \qquad\qquad ... \\ \sum\limits_{\omega\,in\,\Omega} \alpha_P^{\omega}\, t_P^{\omega} = t_P \leq T \end{cases} \tag{8}$$

$$\begin{cases} \sum\limits_{p=1}^{P} \alpha_p^1 = 1 \\ \sum\limits_{p=1}^{P} \alpha_p^2 = 1 \\ \qquad ... \\ \sum\limits_{p=1}^{P} \alpha_p^{|\Omega|} = 1 \end{cases} \tag{9}$$

Here, P denotes the number of processing units and $|\Omega|$ is the total number of tasks. The coefficient α_p^{ω} indicates the proportion of task ω processed by unit p, and t_p^{ω} represents the time taken to complete task ω on unit p, given that this duration varies based on the type of the processing unit. Accordingly, the first system determines the computation duration for each unit, with T being the longest duration. The second part ensures that each task is computed at 100%.

While this model provides an upper bound for ideal performance, it doesn't account for the dependencies between the task order and consider that tasks can be divided among various processing units.

In our adaptation, we assume that a single application consists of three tasks: one for exclusive CPU work, another for exclusive GPU work, and the last one for work that can be executed on either CPU or GPU. Given this characterization, the aforementioned LP model remains applicable.

However, our focus isn't on the ideal makespan T, but on the α coefficients as we aim to find the most efficient way to allocate the application across processing units. Although the LP provides an optimal distribution for an ideal system, it also guides us on the proportion of each application that should be allotted to each processing unit type. However, this distribution might be inefficient in real-world scenarios, leading to a task being fragmented across all units or a skewed allocation, like 99% on one unit and 1% on another, which may not always be practical. Consequently, it's essential to transform these coefficients into practical distribution values to derive a feasible scheduling strategy.

To achieve this, we use a two-step method. First, we sum the distribution coefficients per unit type to determine the fraction of each application designated for every processing unit type. For instance, if the LP solution suggests distributing an application as 0.1 and 0.4 across two CPUs, and 0.2, 0.2, and 0.1 across three GPUs, we infer it should be equally split (0.5 for CPU and 0.5 for GPU). Subsequently, we decide on the application distribution based on these values. In the next step, we compute the processing time for each unit

type by multiplying the number of a given unit type with T. For instance, with two CPUs and a makespan of 10s, we have 20s of total CPU time to distribute. Each application is then assumed to use a fraction of this time proportional to its distribution coefficient. Our greedy algorithm identifies the application with the highest use proportion and allocates it to the processing unit with the least utilization, continuing until every application has been entirely mapped to processing units.

DSR (Dedicated plus Shared Resource) strategies For illustrations, let us consider for instance that we have 3 applications, 4 CPUs, and 2 GPUs. The DSR strategies proceed in two steps to distribute the workers among the applications:

The first step consists in assigning dedicated (unshared) workers (GPUs or CPUs) to each application, proportionally to their GPU/CPU workload compared to the sum of all the applications' workloads. Supposed the proportions of CPU workloads (*cpu_pwl*) are 0.31, 0.56, and 0.13, the numbers of dedicated CPUs (given by $\lfloor nb\_cpus \times cpu\_pwl \rfloor$) will be respectively $\lfloor 4 \times 0.31 \rfloor = \lfloor 1.24 \rfloor = 1$, $\lfloor 4 \times 0.56 \rfloor = \lfloor 2.24 \rfloor = 2$, and $\lfloor 4 \times 0.13 \rfloor = \lfloor 0.52 \rfloor = 0$. In that case, the number of remaining CPUs is 1. The same is similarly done for GPUs. In the case of hybrid workloads, and if the standard deviation between the GPU/CPU speedups is above a certain threshold, we proceed to the barter which consists of exchanging GPU against CPUs to accelerate the most GPU-optimized applications.

The second step involves sharing the remaining workers to the applications, using a given technique. Here proposed two DSR strategies, based on two different techniques for workers sharing, the gradient descent and the clustering: **DRS-CLUS** (Dedicated plus Shared Resource with Clustering) and **DSR-GD** (Dedicated plus Shared Resource with Gradient Descent). The sharing process is done based on the remaining *cpu_pwl* (here in the case of CPUs, we have: 0.24, 0.24, and 0.52).

As for the **DRS-CLUS** strategy, resource sharing is done as follows. The number of clusters is equal to the number of remaining workers. The workers/applications mapping is done to balance the load over the workers as much as possible.

In the **DSR-GD** strategy, the second step is done as follows (see Algorithm 2 in Appendix, from lines 18 to 24). The remaining workers are shared with the applications using the gradient descent (GD). Our GD strategy is modelled as follows.

Given the CPUs and GPUs ids, {0,1,2,3} and {0,1} respectively, our research space is modelled as a three-dimensional discrete space (X, Y, Z):

- X: the numbers of assigned CPUs ($\#CPUS$) per graph. Each application can have from zero to the number of CPUs (Ex. {1,2,1}, the first app has 1 CPU, the second one 2 CPUs, and the third one has 1 CPU)

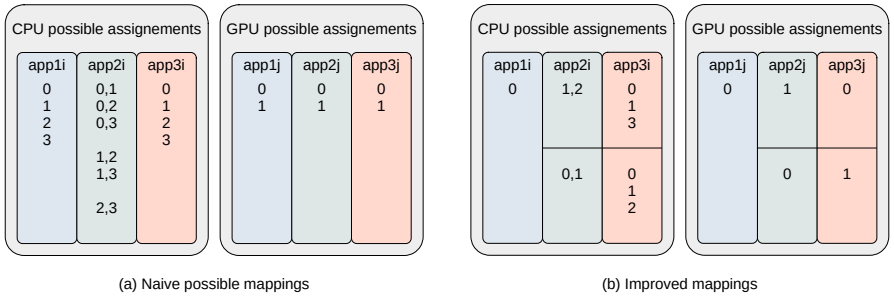


Figure 3: Determination of possible mappings with 3 applications, 4 CPUs, and 2 GPUs, and given that X={1,2,1} and Y={1,1,1}. We have $Z = (app1i \times app2i \times app3i) \times (app1j \times app2j \times app3j)$. While the naive version leads to $((4 \times 6 \times 4) \times (2 \times 2 \times 2) = 96 \times 8 =)$ **768** possible mappings, the improved one gives $((1 \times 2 \times 3) \times (1 \times 2 \times 2) = 6 \times 4 =)$ **24** possible mappings. The improved version has fewer duplicates in terms of obtained makespan.

- Y: the numbers of assigned GPUs ($\#GPUS$) per graph. Each application can have from zero to the number of GPUs (Ex. {1,1,1}, the first app has 1 GPU, the second one 1 GPU, and the third one has 1 GPU)
- Z: the possible (graph-to-gpu/cpu) mappings given X and Y. An example of mapping related to the Y and Y ones above is $\{\{0\},\{0\}\}$, $\{\{1,2\},\{0\}\}$, $\{\{3\},\{1\}\}$. In this example, the first app has the CPU id=0 and the GPU id=0, app 2 has the CPU ids=1,2 and the GPU id=0 and app 3 has the CPU id=3 and the GPU id=1. We can see that app 1 and app 2 share one GPU together.

Iteration in the axis is done as follows:

- the index in X and Y can be seen as permutations with repetition of a possible number of assigned workers in nb_graphs positions:
  - $X_i \in$ {{0,0,0}, {0,0,1},..., {0,0,4}, {0,1,0},...,{4,4,4}}. By looking carefully, we can realize that each X_i is likened to "i" in base (nb_cpus+1)
  - $Y_j \in$ {{0,0,0}, {0,0,1}, {0,0,2}, {0,1,0},...,{2,2,2}}. In like manner, Yi is likened to "j" in base ($nb\_gpus + 1$)
- Z_k (or Zijk): the k-th possible mapping, given X_i and Y_j. One naive way to get the Z_k's values is to generate the different 2-uplets, of the possible CPUs assignment given X_i and the possible GPUs assignment given Y_j. For our example, Figure 3(b) illustrates the determination of possible mappings and how to find the Z_k.

This model presents the whole research space. However, during our research process, the search space is circumscribed by setting for each application the

minimum number of workers (CPUs and GPUs) obtained in step one, and the maximum by adding the number of workers not yet assigned. Thus, we reduce the search space and speed up the search.

Our gradient function is evaluated as a symmetric linear interpolation [42]. Our search process follows the pattern direction [43], first, we search towards the X direction, then the Y direction, and finally the Z direction. To ensure process time scaling, the learning rates for the different axes are $tau_x = 0.001 \times (\lfloor nb\_graphs/\sqrt{nb\_cpus} \rfloor + 1)$, $tau_y = 0.001 \times (\lfloor nb\_graphs/\sqrt{nb\_gpus} \rfloor + 1)$ and $tau_z = 0.0005 \times (\lfloor nb\_graphs/\sqrt{nb\_cpus + nb\_gpus} \rfloor + 1)$.

MinMaxWL (Min-Max Workload balancing) The MinMaxWL algorithm is a load-balancing strategy that distributes the workers to the applications by minimizing the maximum ideal makespan. The strategy is depicted in Algorithm 3 (in Appendix), and has four main steps.

First of all, it assigns one worker to each application having a pure workload according to the type of worker (from lines 3 to 13). While trying to assign a worker to the current application, if there are no remaining workers of the type, the application shares one with the under-loaded application related to that type. A backpropagation is employed in the case of sharing to ensure load-balancing when less-loaded applications are treated after more-loaded ones.

The second step is to ensure all the applications have at least one worker, by assigning a worker to applications without a pure workload (from lines 15 to 28). Since those applications are hybrid, the type of worker to assign is the fastest on the application. A similar sharing process is also employed, but this time the sharing is made on the worker, leading to the smallest makespan at the point.

Finally, while there are remaining workers (per type of worker), it assigns a worker to the application that will minimize the maximum ideal makespan among all the applications.

2.1.2 Distribution options

If the distribution of resources to the applications is accurate, the applications will end almost at the same time, and so will the workers. Otherwise, some resources could be idle for a long, while remaining applications may need them. To deal with that situation, a redistribution of the resources might be necessary. One crucial aspect to consider for it is the condition of resizing, the when.

The condition of resizing we employed is the following:

- An application just ended, and there remain applications to run.
- There is a significant standard deviation between the advancement rate of the applications.

The estimation process time (workload) for a CPU or GPU may differ from the effective processing time during the execution. For instance, let us suppose an application with a CPU workload of 100s, and that has been assigned 10 CPU

workers. Suppose 5s after executions start, there is a need for redistribution, and it remains at an overall 20s processing time for the workload. We would have expected having executed $5 \times 10 = 50$s for the application, whereas we have $100 - 20 = 80$s. The advancement rate in this case is therefore equal to $80/50 = 1.6$; which means the application is running faster than expected. Now we know that possibly the application may end in $(20/1.6)/10 = 1.2$s instead of $20/10 = 2$s.

Before the redistribution, we adjust the workload of each application according to its advancement rate. We have proposed two resource redistribution options.

One Distribution: This is the default behavior, where the distribution is done once and for all.

Multiple Distributions: Here we do the distribution as initially, but considering the adjusted workloads of the remaining applications.

Inherit released workers: Here we distribute the released workers (by the just-ended application) to the remaining ones, with high privilege to those that were delaying.

2.2 RSCHED Implementation in StarPU

StarPU offers a platform to dynamically construct, delete, and modify Scheduling Contexts, which are used to execute several parallel kernels in an isolated way and without interference. This allows the users to assign workers to the contexts, at their creation time, or resize them during program execution. However, this is subject to the knowledge of the number of workers needed for each scheduling context. StarPU proposes online performance tools to monitor the execution of tasks, to make execution time estimations.

2.2.1 Multiple task-based applications

There are several applications implemented in StarPU. However, there is no mechanism to launch or orchestrate the execution of concurrent applications. For the sake of simplicity, instead of using several different applications, we have exploited the implementation of Cholesky factorization to have several independent applications. The Cholesky application in StarPU is implemented with a performance model for each codelet. We have added a parameter to specify the number of applications to create, and for each application, we gave the possibility to specify the size and the number of blocks via environment variables.

2.2.2 Context creation and Workload determination

For each application, a separate context is created and a task scheduler is associated with it. In this work, we have chosen to use DMDA as a scheduler for all the applications. DMDA relies on a historical performance model to be able to estimate in advance the duration of a codelet on each kind of processing unit. Using the StarPU historical performance model, we have been able to compute the different workloads of the concurrent applications.

3 Performance Study

3.1 Experiments setup

3.1.1 Hardware

We have carried out our experiments on five configurations with different GPU models described as follows.

- **A100**: composed of two 32-core AMD Zen3 EPYC 7513 @ 2.60 GHz, and 2 NVIDIA A100 (40GB). We use 30 CPU cores and 16 CUDA streams per GPU;
- **Quadro**: composed of 2 Icosa-core Cascade Lake Intel Xeon Gold 5218R CPU @ 2.10 GHz, and 2 NVIDIA Quadro RTX8000 (48GB). We use 30 CPU cores and 16 CUDA streams per GPU;
- **K40M**: composed of 2 Dodeca-cores Haswell Intel Xeon E5-2680 v3 2.5 GHz, and 4 K40m GPUs (12GB). We use 20 CPU cores and 8 CUDA streams per GPU;

We have configured StarPU as follows. For each configuration, we set the environment variables STARPU_NCPU to the number of CPU cores, STARPU_NCUDA to the number of GPU, and STARPU_NWORKER_PER_CUDA to the number of CUDA streams. Therefore, for all the configurations, we have more GPU workers than CPU ones.

3.1.2 Task-based applications

We have used the implementation of Cholesky factorization in StarPU and declined into twelve different configurations as follows.

app_0: Matrix of size 3.200 and 5 blocks	app_6: Matrix of size 19.200 and 20 blocks
app_1: Matrix of size 3.200 and 10 blocks	app_7: Matrix of size 19.200 and 30 blocks
app_2: Matrix of size 6.400 and 10 blocks	app_8: Matrix of size 25.600 and 40 blocks
app_3: Matrix of size 6.400 and 20 blocks	app_9: Matrix of size 25.600 and 80 blocks
app_4: Matrix of size 9.600 and 10 blocks	app_{10}: Matrix of size 76.800 and 80 blocks
app_5: Matrix of size 9.600 and 30 blocks	app_{11}: Matrix of size 76.800 and 120 blocks

3.1.3 Software configuration

For each application, we have made different affinities related to the types of compute units (CPU or GPU). By default, all the tasks of the Cholesky application have two codelets, one for CPU and one for GPU. Overall, we have used the five following affinities:

- Default (*affinity0*): each task has one CPU codelet and GPU codelet,
- Only CPU (*affinity1*): all the tasks have only a CPU codelet,
- Only GPU (*affinity2*): all the tasks have only a GPU codelet,

To analyze the influence of the number of concurrent applications, and of the workload, we have made experiments with 3, 6, and 12 concurrent applications. To be in accord with a realistic scenario, we have made a shuffle of the list of applications in each experiment. Then we took consecutive applications to form the groups. For instance, in the case of 3 applications, we have executed concurrently the applications at the first, second third positions, then the fourth, fifth, and sixth positions, and so on.

3.2 Metrics

In our experiments, we have compared our four distribution strategies against the concurrent execution using a unique context with all the workers (DMDA_CONC), and against sequential execution (ie. one application after the another) using a unique context with all the workers (DMDA_SEQ).

As metrics, we have considered the speedup, the data transfer, and the resource utilization efficiency (RUE). We also compare the distribution processing time of our strategies.

Definition 1 *We define the RUE as the ability to maximize the utilization of the resource, that is, using the adequate number and types of resources for the execution of each application. The RUE is given by Equation 10, which is the product of the resource utilization and the efficiency [44].*

$$RUE = \frac{\sum_{p \in USED\_WK}\{processing\_time\ of\ worker\ p\}}{\sum_{p \in USED\_WK}\{total\_active\_time\ of\ worker\ p\}} \times \frac{speedup}{|USED\_WK|} \tag{10}$$

$USED\_WK$ is the list of distinct workers (CPU or GPU) used for the execution of the applications, whether concurrently or sequentially (ie. in DMDA_SEQ). We normalized the RUE such that the values lie between 0 and 1.

3.3 Experiments results and analysis

3.3.1 Default experiments

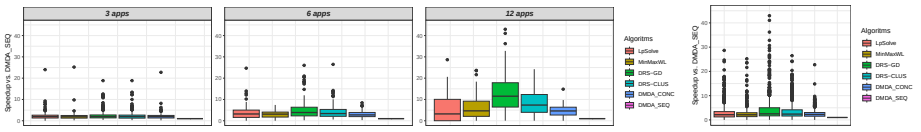
We first present the performance of our strategies (LpSolve, MinMaxWL, DSR-GD, and DSR-CLUS), and of DMDA_CONC, against DMDA_SEQ, in terms of Speedup, then in terms of data transfer, and finally in terms of RUE.

Speedup: The big picture of the speedup realized by the different strategies compared to DMDA_SEQ is presented in Figure 4. For all the configurations and affinities, the LpSolve, MinMaxWL, DSR-GD, DSR-CLUS, and DMDA_CONC can significantly accelerate the execution DMDA_SEQ (Figure 4b). Moreover, our strategies (except MinMaxWL) perform better even than DMDA_CONC with the increase in the number of applications. We observe in this study an outperformance over DMDA_CONC in more than 50% of cases for LpSolve, and more than 75% of cases for DSR-GD and DSR-CLUS (Figure 4b).

DSR-GD and DSR-CLUS reach an acceleration of 40× compared to DMDA_SEQ. However, DSR-GD outperforms DSR-CLUS in more than 50% of situations, with an increase of applications. This means that conceptually, the Gradient Descent performs better than the clustering since the two strategies have the same building block. We observe that the speedup of the strategies increases with the number of concurrent applications (Figure 4a). The study of the variation of the speedup according to the workload and of the GPU/CPU acceleration (see Figure 5 and Figure 6) reveals that DSR-GD and DSR-CLUS perform better when the percentage of the standard deviation of GPU/CPU acceleration among the application increases. This is explained by the employed bartering technique that gives GPU in preference to more accelerated applications in exchange for CPU to others.

The study of the variation of the speedup according to the number of applications over the different hardware configurations (Figure 7) reveals that the strategies perform better on recent architectures (Quadro and A100) which have more accelerated GPU than on older ones (K40M). More specifically, DSR-GD and DSR-CLUS perform better than the other strategies, due to the same reasons as previously.

Globally, the DSR-GD realizes better speedup and in more of the situations than the others, then DSR-CLUS followed by LpSolve.



(a) Speedup obtained for the 3, 6, and 12 concurrent applications.

(b) Speedup summary

Figure 4: **Speedup of LpSolve, MinMaxWL, DSR-GD, DSR-CLUS, DMDA_CONC against DMDA_SEQ for all the affinities (affinity0, affinity1, affinity2) and all the hardware configurations (K40M, Quadro, and A100).** (a) for the 3, 6, and 12 concurrent applications, (b) summary of all the experiments.

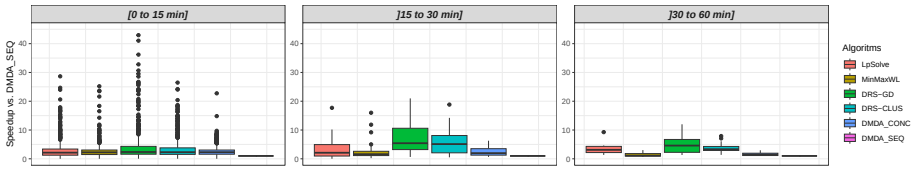


Figure 5: **Speedup of LpSolve, MinMaxWL, DSR-GD, DSR-CLUS, DMDA_CONC against DMDA_SEQ according to the average workload in minute (organized in three ranges), on K40M, Quadro or A100 configurations.**

Data transfer: The total amount of memory transfer obtained with the different strategies are provided in Figure 9. All the strategies for concurrent execution used in this study (LpSolve, MinMaxWL, DSR-GD, DSR-CLUS, and DMDA_CONC) significantly reduce the total memory transfer compared to the sequential execution (Figure 9b), DSR-CLUS been the best one.

Resource utilization efficiency: The Normalized RUE obtained with the different strategies are provided in Figure 10. We observe in this study that the concurrent execution of applications leads a more effective resource usage than the sequential one. The DSR-GD and DSR-CLUS strategies are more efficient in terms of resource utilization than the other (Figure 10b), DSR-GD been the best one as the number of applications increases. Executing the applications sequentially one after the other leads to more resource wastage, which is known as a major cause of energy consumption in data centers [9, 10]. Moreover, we observe that the improvement of our strategies (DSR-GD, DSR-CLUS, and LpSolve) in terms of RUE correspond with the situations in which they are speeding up compared to the sequential execution (Figure 10a $\equiv$ Figure 4a). Therefore, succeeding in speeding up the sequential execution of tasks-based applications using our strategies might also help to reduce energy consumption, thanks to the effectiveness of the scheduler used.

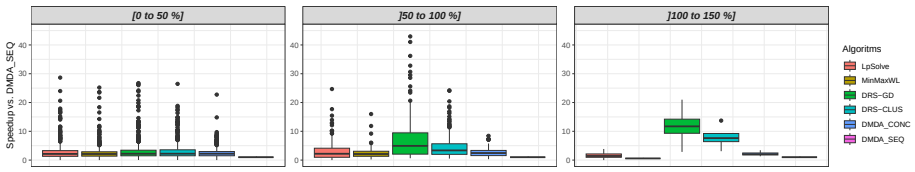


Figure 6: **Speedup of LpSolve, MinMaxWL, DSR-GD, DSR-CLUS, DMDA_CONC against DMDA_SEQ according to the standard deviation percentage of GPU/CPU speedup between the applications (organized in three ranges), on K40M, Quadro or A100 configurations.**

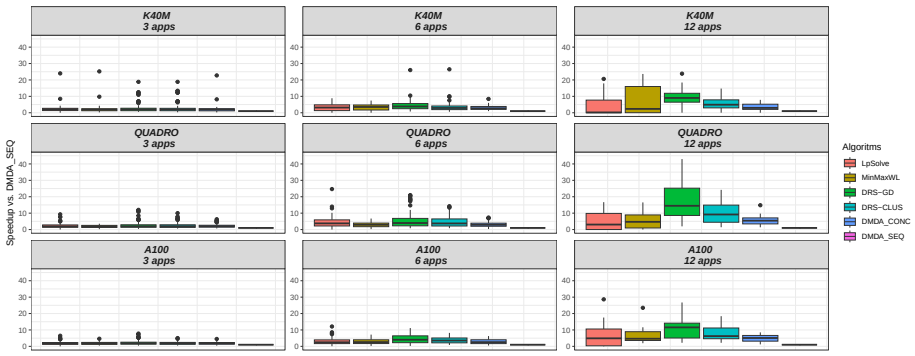


Figure 7: **Speedup of LpSolve, MinMaxWL, DSR-GD, DSR-CLUS, DMDA_CONC against DMDA_SEQ for the 3, 6, and 12 concurrent applications and including all the three architecture affinities, on K40M, Quadro or A100 configurations.**

3.3.2 Distribution processing time and options

Distribution processing time: In a dynamic situation where applications arrive continuously (as we will study in the future), the decision processing has to be fast. Figure 11 presents the evolution of processing time for each of our proposed strategies according to the acceleration of GPU compared to the CPU, and the number of executed applications.

The MinMaxWL and DSR-CLUS strategies are faster than LpSolve and DSR-GD which are meta-heuristics. However, the processing times of all the strategies are relatively small to expect good behavior even in a dynamic environment. Moreover, even though we add the decision process time to the overall makespan, we will still have almost the same results as presented above.

Furthermore, we realize that DSR-GD scale better than LpSolve given their processing time (Figure 11) and speedup compared to DMDA_SEQ (Figures 4, 5, 6, 7, 8). This performance of DSR-GD is due in part to the choice of learning

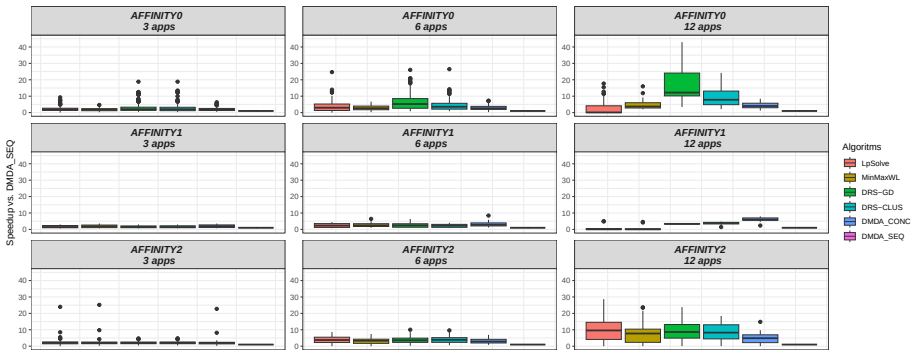
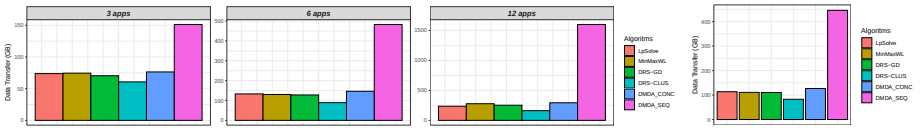


Figure 8: **Speedup of LpSolve, MinMaxWL, DSR-GD, DSR-CLUS, DMDA_CONC against DMDA_SEQ for the 3, 6, and 12 concurrent applications and each affinity (AFFINITY0, AFFINITY1, AFFINITY2), on K40M, Quadro or A100 configurations.**



(a) Memory transfer obtained for the 3, 6, and 12 concurrent applications.

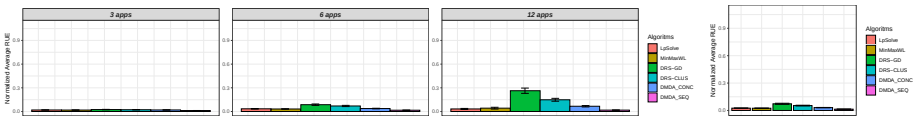
(b) Memory transfer summary

Figure 9: **Memory transfer of LpSolve, MinMaxWL, DSR-GD, DSR-CLUS, DMDA_CONC against DMDA_SEQ for all the affinities (affinity0, affinity1, affinity2) and all the hardware configurations (K40M, Quadro, and A100).** (a) for the 3, 6, and 12 concurrent applications, (b) summary of all the experiments.

rate that helped speedily converge towards the optimal solution no matter the number of applications and resources, and also due to the bartering technique employed (see Section 2.1.1).

Distribution options: We carried out a study on the effectiveness of the re-distribution options ("Multiple Distributions", and "Inherit released workers") presented in Section 2.1.2 comparatively to the default one ("One Distribution") when combined with each of our four strategies (LpSolve, MinMaxWL, DSR-GD, DSR-CLUS). Figure 12 presents the makespan obtained in each case, which reveals that the effectiveness of the re-distribution options ("Multiple Distributions", and "Inherit released workers") depends on the strategies and the configuration.

The combination DSR-GD/"Inherit released workers" always produces a gain for all the configurations. For the other cases, the combination strat-



(a) Average RUE obtained for the 3, 6, and 12 concurrent applications.

(b) Average RUE summary

Figure 10: **Average RUE of LpSolve, MinMaxWL, DSR-GD, DSR-CLUS, DMDA_CONC against DMDA_SEQ for all the affinities (affinity0, affinity1, affinity2) and all the arch configurations (K40M, Quadro, and A100).** (a) for the 3, 6, and 12 concurrent applications, (b) summary of all the experiments.

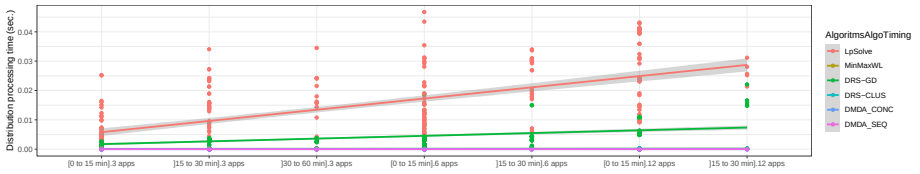


Figure 11: **Distribution processing time of LpSolve, MinMaxWL, DSR-GD, and DSR-CLUS according to the standard deviation percentage of GPU/CPU speedup between the applications (organized in groups), and the number of concurrent applications, on K40M, Quadro or A100 configurations.**

egy/option leads to a gain only for some configurations. We notify significant improvement in some cases, proving that there is hope for improving the results obtained above by using resource redistribution. However, it is imperative to do more investigations on configuration and strategy sensitivity to achieve this.

4 Conclusions

As computing resources are getting more complex and powerful, there is little doubt that we need methods to reduce the waste from the users' choices, bad application optimization, or heterogeneous workloads during executions. This is where the task-based model grants more opportunities by exposing a dynamic degree of parallelism with execution environments able to use this information in the most constructive and thus efficient way. To minimize the overall makespan and maximize resource utilization, while executing several task-based applications, we introduce RSCHED, a two-level resources management framework that allows (i) dynamic resource distribution for concurrent execution of task-based applications, and (ii) dedicated task scheduling for each application. We proposed strategies for resource distribution and implemented our proposal on the StarPU runtime system, proposing schedulers on which we rely for the second

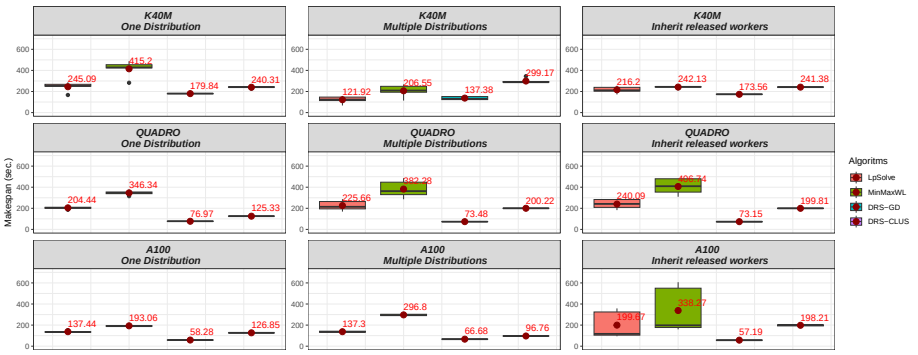


Figure 12: **Analysis of distribution options for LpSolve, MinMaxWL, DSR-GD, DSR-CLUS with 12 concurrent applications on K40M, Quadro or A100 configurations.** ***The averages makespan are displayed in red.***

level. We evaluated our proposal using real applications based on the StarPU implementation of Cholesky factorization. RSCHED demonstrated the potential to speed up the overall makespan compared to consecutive execution with an average factor of 10x, and a factor of 5x when compared against the concurrent execution without resource distribution using DMDA. RSCHED also demonstrated the potential to increase the rate of resource utilization as the number of applications increases.

In our future work, we would like to consider different applications (instead of just Cholesky), exploit multiple nodes, and improve RSCHED decisions by analysing the structures of task graphs.

Acknowledgments

Has been made anonymous.

References

[1] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. In *Euro-Par 2009 Parallel Processing: 15th International Euro-Par Conference, Delft, The Netherlands, August 25-28, 2009. Proceedings 15*, pages 863–874. Springer, 2009.

[2] Emmanuel Agullo, Bérenger Bramas, Olivier Coulaud, Eric Darve, Matthias Messner, and Toru Takahashi. Task-based fmm for multicore architectures. *SIAM Journal on Scientific Computing*, 36(1):C66–C93, 2014.

[3] 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):2608–2629, 2016.

[4] Jean Marie Couteyen Carpaye, Jean Roman, and Pierre 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*, 28:439–454, 2018.

[5] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013.

[6] Andra Hugo, Abdou Guermouche, Pierre-André Wacrenier, and Raymond Namyst. Composing multiple starpu applications over heterogeneous machines: a supervised approach. *The International journal of high performance computing applications*, 28(3):285–300, 2014.

[7] Clément Flint, Ludovic Paillat, and Bérenger Bramas. Automated prioritizing heuristics for parallel task graph scheduling in heterogeneous computing. *PeerJ Computer Science*, 8:e969, 2022.

[8] Hayfa Tayeb, Bérenger Bramas, Abdou Guermouche, and Mathieu Faverge. Multreeprio: Scheduling task-based applications for heterogeneous computing systems. In *COMPAS 2022-Conférence francophone d'informatique en Parallélisme, Architecture et Système*, 2022.

[9] Jean Etienne Ndamlabin Mboula, Vivient Corneille Kamla, Muhammad Hafizhuddin Hilman, and Clémentin Tayou Djamegni. Energy-efficient workflow scheduling based on workflow structures under deadline and budget constraints in the cloud. *arXiv preprint arXiv:2201.05429*, 2022.

[10] Jean Etienne Ndamlabin Mboula, Vivient Corneille Kamla, and Clémentin Tayou Djamegni. Dynamic provisioning with structure inspired selection and limitation of vms based cost-time efficient workflow scheduling in the cloud. *Cluster Computing*, pages 1–25, 2021.

[11] Jean Etienne Ndamlabin Mboula, Vivient Corneille Kamla, and Clémentin Tayou Djamegni. Cost-time trade-off efficient workflow scheduling in cloud. *Simulation Modelling Practice and Theory*, page 102107, 2020.

[12] Enrico Calore and Sebastiano Fabio Schifano. Porting a lattice boltzmann simulation to fpgas using ompss. In *Parallel Computing: Technology Trends*, pages 701–710. IOS Press, 2020.

[13] João VF Lima, Gabriel Freytag, Vinicius Garcia Pinto, Claudio Schepke, and Philippe OA Navaux. A dynamic task-based d3q19 lattice-boltzmann

method for heterogeneous architectures. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 108–115. IEEE, 2019.

[14] Mustafa AbdulJabbar, Rio Yokota, and David Keyes. Asynchronous execution of the fast multipole method using charm++. *arXiv preprint arXiv:1405.7487*, 2014.

[15] Miquel Pericàs, Xavier Martorell, and Yoav Etsion. Implementation of a hierarchical n-body simulator using the ompss programming model. In *Proceedings of the 1st Workshop on Irregular Applications: Architectures and Algorithms*, pages 23–30, 2011.

[16] Emmanuel Agullo, Luc Giraud, and Stojce Nakov. Task-based sparse hybrid linear solver for distributed memory heterogeneous architectures. In *Euro-Par 2016: Parallel Processing Workshops: Euro-Par 2016 International Workshops, Grenoble, France, August 24-26, 2016, Revised Selected Papers 22*, pages 83–95. Springer, 2017.

[17] Xavier Lacoste, Mathieu Faverge, George Bosilca, Pierre Ramet, and Samuel Thibault. Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, pages 29–38. IEEE, 2014.

[18] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, and Florent Lopez. Multifrontal qr factorization for multicore architectures over runtime systems. In *Euro-Par 2013 Parallel Processing: 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings 19*, pages 521–532. Springer, 2013.

[19] Rocío Carratalá-Sáez, Mathieu Faverge, Grégoire Pichon, Guillaume Sylvand, and Enrique S Quintana-Ortí. Tiled algorithms for efficient task-parallel -matrix solvers. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 757–766. IEEE, 2020.

[20] Rodrigo Arias Mallo. Particle-in-cell plasma simulation with ompss-2. Master's thesis, Universitat Politècnica de Catalunya, 2019.

[21] Dalal Sukkari, Hatem Ltaief, Mathieu Faverge, and David Keyes. Asynchronous task-based polar decomposition on single node manycore architectures. *IEEE Transactions on parallel and distributed systems*, 29(2):312–323, 2017.

[22] Lionel Boillot, George Bosilca, Emmanuel Agullo, and Henri Calandra. Task-based programming for seismic imaging: Preliminary results. In *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS)*, pages 1259–1266. IEEE, 2014.

[23] Víctor Martínez, David Michéa, Fabrice Dupros, Olivier Aumage, Samuel Thibault, Hideo Aochi, and Philippe OA Navaux. Towards seismic wave modeling on heterogeneous many-core architectures using task-based runtime system. In *2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 1–8. IEEE, 2015.

[24] Salli Moustafa, Wilfried Kirschenmann, Fabrice Dupros, and Hideo Aochi. Task-based programming on emerging parallel architectures for finite-differences seismic numerical kernel. In *Euro-Par 2018: Parallel Processing: 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings 24*, pages 764–777. Springer, 2018.

[25] 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, 2020.

[26] Peter Brucker and Sigrid Knust. Complexity results for scheduling problems, 2009.

[27] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-based scheduling: applying constraint programming to scheduling problems*, volume 39. Springer Science & Business Media, 2001.

[28] Mary Lai O Salvana, Sameh Abdulah, Huang Huang, Hatem Ltaief, Ying Sun, Marc G Genton, and David E Keyes. High performance multivariate geospatial statistics on manycore systems. *IEEE Transactions on Parallel and Distributed Systems*, 32(11):2719–2733, 2021.

[29] Qinglei Cao, Yu Pei, Kadir Akbudak, Aleksandr Mikhalev, George Bosilca, Hatem Ltaief, David Keyes, and Jack Dongarra. Extreme-scale task-based cholesky factorization toward climate and weather prediction applications. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, pages 1–11, 2020.

[30] Kadir Akbudak, Hatem Ltaief, Aleksandr Mikhalev, Ali Charara, Aniello Esposito, and David Keyes. Exploiting data sparsity for large-scale matrix computations. In *European Conference on Parallel Processing*, pages 721–734. Springer, 2018.

[31] Jean Marie Couteyen Carpaye, Jean Roman, and Pierre 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*, 28:439–454, 2018.

[32] Haluk Topcuoglu, Salim 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.

[33] Yuming Xu, Kenli Li, Jingtong Hu, and Keqin Li. A genetic algorithm for task scheduling on heterogeneous computing systems using multiple priority queues. *Information Sciences*, 270:255–287, 2014.

[34] Karan R Shetti, Suhaib A Fahmy, and Timo Bretschneider. Optimization of the heft algorithm for a cpu-gpu environment. In *2013 International conference on parallel and distributed computing, applications and technologies*, pages 212–218. IEEE, 2013.

[35] Hong Jun Choi, Dong Oh Son, Seung Gu Kang, Jong Myon Kim, Hsien-Hsin Lee, and Cheol Hong Kim. An efficient scheduling scheme using estimated execution time for heterogeneous computing systems. *The Journal of Supercomputing*, 65:886–902, 2013.

[36] Minhaj Ahmad Khan. Scheduling for heterogeneous systems using constrained critical paths. *Parallel Computing*, 38(4-5):175–193, 2012.

[37] Olivier Beaumont, Louis-Claude Canon, Lionel Eyraud-Dubois, Giorgio Lucarelli, Loris Marchal, Clément Mommessin, Bertrand Simon, and Denis Trystram. Scheduling on two types of resources: a survey. *ACM Computing Surveys (CSUR)*, 53(3):1–36, 2020.

[38] Ashish Kumar Maurya and Anil Kumar Tripathi. On benchmarking task scheduling algorithms for heterogeneous computing systems. *The Journal of Supercomputing*, 74(7):3039–3070, 2018.

[39] Bérenger Bramas. *Optimization and parallelization of the boundary element method for the wave equation in time domain*. PhD thesis, Bordeaux, 2016.

[40] William Fornaciari, Giovanni Agosta, David Atienza, Carlo Brandolese, Leila Cammoun, Luca Cremona, Alessandro Cilardo, Albert Farres, José Flich, Carles Hernandez, et al. Reliable power and time-constraints-aware predictive management of heterogeneous exascale systems. In *Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 187–194, 2018.

[41] Giovanni Agosta, William Fornaciari, David Atienza, Ramon Canal, Alessandro Cilardo, José Flich Cardo, Carles Hernandez Luz, Michal Kulczewski, Giuseppe Massari, Rafael Tornero Gavilá, et al. The recipe approach to challenges in deeply heterogeneous high performance systems. *Microprocessors and Microsystems*, 77:103185, 2020.

[42] Albert S Berahas, Liyuan Cao, Krzysztof Choromanski, and Katya Scheinberg. Linear interpolation gives better gradients than gaussian smoothing in derivative-free optimization. *arXiv preprint arXiv:1905.13043*, 2019.

[43] Steven C Chapra. *Numerical methods for engineers*. Mcgraw-hill, 2010.

[44] Hamid Arabnejad and Jorge G Barbosa. List scheduling algorithm for heterogeneous systems by an optimistic cost table. *IEEE transactions on parallel and distributed systems*, 25(3):682–694, 2013.

Appendix

Algorithm 1: Ideal makespan Algorithm

```
1  function ideal_makespan(graph_info G, double coef_par_eff)
2      nb_workers = G.#GPUS + G.#CPUS;
3      if (G.#GPUS == 0.0) || (G.#CPUS == 0.0) then
4          if (G.CPUS_PR == 0.0) then
5              m = G.GPU_W / G.#GPUS;
6          if (G.GPUS_PR == 0.0) then
7              m = G.CPU_W / G.#CPUS;
8          return m × coef_par_eff ^(nb_workers−1);
9      Compute tgpu_minM and tcpu_minM using equations 5 and 6;
10     if (G.GPU_W == G.GPU_PW) || (G.CPU_W==G.CPU_PW) then
11         return MAX(tgpu_minM, tcpu_minM);
12     gpu_rem_wl = compute the remaining GPU workload;
13     cpu_rem_wl = compute the remaining CPU workload;
14     if (gpu_rem_wl == 0.0) || (cpu_rem_wl == 0.0) then
15         return MAX(tgpu_minM, tcpu_minM) + gpu_rem_wl /
            G.#GPUS + cpu_rem_wl / G.#CPUS;
16     Compute ideal_makespan using equation 7;
17     return ideal_makespan;
18
```

Algorithm 2: DSR-GD (Dedicated plus Shared Resource with Gradient Descent)

```
1  //-->Dedicated workers per app;
2  rem_gpus = nb_gpus;
3  rem_cpus = nb_cpus;
4  foreach graph G in graphs do
5      gpus_dedicated = floor(G.GPU_W / SUM(GPU_W)) × nb_gpus;
6      cpus_dedicated = floor(G.CPU_W / SUM(CPU_W)) × nb_cpus;
7      while (gpus_dedicated) do
8          Assign the (rem_gpus)^-th GPU to graph G;
9          gpus_dedicated − −;
10         rem_gpus − −;
11     while (cpus_dedicated) do
12         Assign the (rem_cpus)^-th CPU to graph G;
13         cpus_dedicated − −;
14         rem_cpus − −;
15 if (Stdd_speedup > SPEEDUP_STDD_LIMIT) ∥ Stdd_idealm >
     MAKESPAN_STDD_LIMIT)) then
16     bartering();
17 //-->Shared workers between apps Using Gradient Descent;
18 Configure GD axis (X, Y) using rem_cpus and rem_gpus;
19 foreach axis in X, Y, Z do
20     while (No convergence) do
21         Use the best indexes from previous axes;
22         Fix the value of the following axis;
23         Search the best index in the axis using Gradient
            Descent with Min-Max ideal_makespan as objective
            function;
24         keep track of the best solution;
```

Algorithm 3: MinMaxWL (Min-Max Workload balancing)

```
1  //-->Ensure each graph has at least one worker;
2  //----->Graphs with pure workload;
3  foreach graph G in graphs do
4      if (G.GPU_PW != 0.0) then
5          if (Remaining GPUs) then
6              Assign one GPU to G;
7          else
8              Share one GPU with the under-loaded pure gpu graph,
                with backpropagation;
9      if (G.CPU_PW != 0.0) then
10         if (Remaining CPUs) then
11             Assign one CPU to G;
12         else
13             Share one CPU with the under-loaded pure cpu graph,
                with backpropagation;
14 //----->Graphs without pure workload;
15 foreach graph G in graphs do
16     if (G.CPU_PW == 0.0 && G.CPU_PW == 0.0) then
17         if (G.CPU_W > G.GPU_W) then
18             // GPU is faster;
19             if (Remaining GPUs) then
20                 Assign one GPU to G;
21             else
22                 Share one GPU with the under-loaded graph, with
                    backpropagation;
23         else
24             // CPU is faster;
25             if (Remaining CPUs) then
26                 Assign one CPU to G;
27             else
28                 Share one CPU with the under-loaded graph, with
                    backpropagation;
29 //-->Dist.  Remaining Workers:  Load-balancing using
    Min-Max;
30 while (Remaining GPUs Workers) do
31     Assign one GPU to graph leading to Min-Max
        ideal_makespan;
32 while (Remaining CPUs Workers) do
33     Assign one CPU to graph leading to Min-Max
        ideal_makespan;
```