Optimizing Parallel Heterogeneous System Efficiency: Dynamic Task Graph Adaptation with Recursive Tasks

Nathalie Furmento, Abdou Guermouche, Gwenolé Lucas, Thomas Morin, Samuel Thibault, Pierre-André Wacrenier

Graphical Abstract

Optimizing Parallel Heterogeneous System Efficiency: Dynamic Task Graph Adaptation with Recursive Tasks

Nathalie Furmento, Abdou Guermouche, Gwenolé Lucas, Thomas Morin, Samuel Thibault, Pierre-André Wacrenier

Highlights

Optimizing Parallel Heterogeneous System Efficiency: Dynamic Task Graph Adaptation with Recursive Tasks

Nathalie Furmento, Abdou Guermouche, Gwenolé Lucas, Thomas Morin, Samuel Thibault, Pierre-André Wacrenier

- A new mechanism determines before execution whether a task should be subdivided.
- An algorithm, based on linear programming, allows automatic task granularity adaptation.
- Experimental evaluation shows this approach matches or surpasses state-of-the-art libraries.
- The genericity of the solution opens an easy apply to more linear algebra applications.

Optimizing Parallel Heterogeneous System Efficiency: Dynamic Task Graph Adaptation with Recursive Tasks

Nathalie Furmento, Abdou Guermouche, Gwenolé Lucas, Thomas Morin, Samuel Thibault, Pierre-André Wacrenier

CNRS, Inria, LaBRI, Université de Bordeaux, Bordeaux, France

Abstract

Task-based programming models are currently an ample trend to leverage heterogeneous parallel systems in a productive way (OpenACC, Kokkos, Legion, OmpSs, `PaRSEC`, `StarPU`, XKaapi, ...). Among these models, the Sequential Task Flow (STF) model is widely embraced (`PaRSEC`'s DTD, OmpSs, `StarPU`) since it allows to express task graphs naturally through a sequential-looking submission of tasks, and tasks dependencies are inferred automatically. However, STF is limited to task graphs with task sizes that are fixed at submission, posing a challenge in determining the optimal task granularity. Notably, in heterogeneous systems, the optimal task size varies across different processing units, so a single task size would not fit all units. `StarPU` 's recursive tasks allow graphs with several task granularities by turning some tasks into sub-graphs dynamically at runtime. The decision to transform these tasks into sub-graphs is decided by a `StarPU` component called the Splitter. After deciding to transform some tasks, classical scheduling approaches are used, making this component generic, and orthogonal to the scheduler. In this paper, we propose a new policy for the Splitter, which is designed for heterogeneous platforms, that relies on linear programming aimed at minimising execution time and maximising resource utilization. This results in a dynamic well-balanced set comprising both small tasks to fill multiple CPU cores, and large tasks for efficient execution on accelerators like GPU devices. We then present an experimental evaluation showing that just-in-time adaptations of the task graph lead to improved performance across various dense linear algebra algorithms.

Keywords: Multicore, Accelerator, Task-based programming, Granularity, STF, Runtime System

1. Introduction

The increasing demand for computational power has driven the widespread adoption of heterogeneous architectures. However, to fully exploit these complex systems, programmers must efficiently distribute workloads across different processing units and manage data movement between them. To address these challenges, task-based programming models have emerged as a popular solution in domains such as data analytics [1], machine learning [2], and scientific computing. As a result, several Runtime Systems (RTS) have been developed to facilitate efficient task execution [3, 4, 5, 6, 7].

These models allow programmers to structure an application's workload by breaking it into self-contained tasks, with well-defined data dependencies. The tasks are organized into a Directed Acyclic Graph (DAG), which the RTS leverages to schedule and execute tasks efficiently.

Task granularity plays a critical role in optimizing performance, as it directly influences parallelism and resource utilization. CPUs and GPUs have different computational strengths: CPUs are optimized for fine-grained, multithreaded tasks, while GPUs perform best with coarse-grained tasks due to their massively parallel architecture. Consequently, tuning task granularity is a key challenge.

Various approaches have been proposed to address this issue, aiming to strike an optimal balance in task size [8, 9]. However, these methods have often fallen short, particularly for applications with irregular workloads. To overcome this limitation, some applications have adopted recursive task submission [10, 11], enabling task granularity to be adjusted at submission time. The success of these application-specific strategies has led to RTS implementations supporting recursive tasks, such as `PaRSEC` [12], `StarPU` [13] or StarSs [14]. These tasks can be transformed into sub-graphs at runtime, allowing for dynamic adaptation of parallelism during the execution. However, existing approaches typically rely on application-driven decision heuristics that do not account for real-time factors such as worker workload and data placement. By incorporating runtime information, it becomes possible to make more informed decisions, optimizing the execution of applications where task execution times are predictable. This is particularly beneficial

for both regular applications and certain irregular workloads, such as sparse linear algebra [10] and the Fast Multipole Method [15].

In this work, we introduce an RTS-driven task-splitting approach that automatically adapts task granularity based on RTS-based criteria, eliminating the need for programmer intervention. Our approach features an oracle that dynamically adjusts task granularity by deciding whether a task should be split, ensuring that computing resources are efficiently utilized. This work extends the dynamic task splitting mechanism introduced in [16] to heterogenous environments.

To our knowledge, this is the first fully automated RTS-driven approach for task granularity adaptation. Specifically, we:

- Present a component, called Splitter, that determines whether a task should be split.
- Introduce a linear programming-based heuristic to optimize task division on heterogeneous platforms based on the RTS state.
- Conduct an experimental evaluation on various dense linear algebra operations, demonstrating that our automatic approach achieves performance comparable to, or even exceeding, state-of-the-art solutions, including the `PaRSEC` library and `Chameleon`/`StarPU` [17] statically-tuned methods.

The remainder of this paper is organized as follows. Section 2 presents the mechanisms implemented in a runtime system such as `StarPU`, with a particular focus on the management and scheduling of recursive tasks on heterogeneous architectures. Section 3 introduces our methodology for task splitting on homogeneous platforms. Section 4 details our proposed heuristic for dynamic task splitting on heterogeneous platforms. Section 5 presents experimental results. Section 6 discusses related works, followed by conclusions and future directions in Section 7.

2. Background

The increasing complexity of modern computing systems, which consist of different types of processing units, has made Runtime Systems (RTS) essential for programming heterogeneous architectures in a portable and efficient manner.

```
T0(a)
T1(a, b)
T2(a, c)
```

```
submit(T0, a:RW)
submit(T1, a:R, b:RW)
submit(T2, a:R, c:RW)
wait_tasks_end()
```

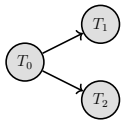


Figure 1: A sequential code (left), the corresponding STF code (center), and the corresponding DAG (right).

Task-based RTS have been widely adopted to exploit complex architectures. Examples of such RTS are CILK [18], OpenMP, Intel TBB [19] (for multicore architectures), APC [20], Charm++ [21], HPX [22], XKAAPI [7], Legion [3], PaRSEC [4], `StarPU` [5] or StarSs [6] (for heterogeneous architectures).

Applications using these RTS provide a high-level description as a set of self-contained tasks, each with clearly defined data inputs and outputs. Dependencies between tasks are defined at a high-level construct a graph of tasks at runtime, which is then scheduled during execution.

A commonly used task-based paradigm is the Sequential Task Flow (STF) model (e.g., OpenMP, `StarPU` [5], or StarSs [23]), which provides a straightforward representation of the application based on task data accesses. Alternatively, other RTS, such as PaRSEC, use the Parameterized Task Graph (PTG) model [24], where the task graph is expressed in a high-level algebraic form and unrolled at runtime.

2.1. The Sequential Task Flow Model

The STF model utilizes a non-blocking task insertion mechanism, enabling immediate control return to the caller while deferring the task's execution. When inserting a task, the caller must specify the data it will use along with their access modes. Based on the task submission order and the specified access modes, dependencies can be automatically determined, enabling the automatic construction of the task DAG [25].

Figure 1 shows an example of a sequential algorithm calling three functions T_0, T_1 and T_2. The resulting STF code creates three tasks using the *submit* function which takes several parameters, including the computation function to be performed, the data to be accessed by this function, and the corresponding access mode (Read, Write, or Read/Write). It is important to note that task completion must be awaited, as the submission mechanism is non-blocking.

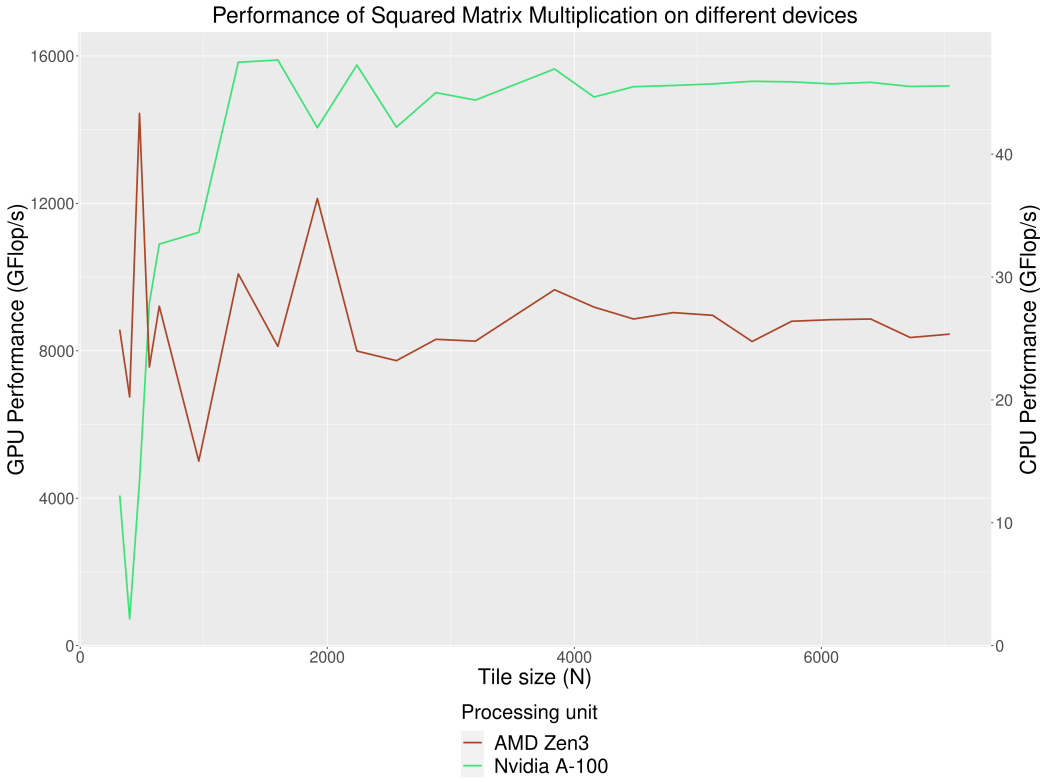


Figure 2: Performance of matrix multiplication of one tile with varying size. Nvidia A100 in green, AMD Zen3 in red.

2.2. Granularity Challenges within Runtime Systems

In the context of homogeneous processing units, performance optimization requires a careful evaluation of granularity. Larger granularity tends to improve kernel efficiency while reducing the load on the RTS. However, for a given problem size, decreasing task granularity can increase parallelism, maximizing the utilization of processing units during execution. Thus, selecting the appropriate granularity involves balancing efficiency and parallelism.

Additionally, efficiently utilizing different types of processing units can be challenging. GPUs are optimized for coarse-grained parallelism, whereas CPUs require fine-grained parallelism to maximize core utilization. A one-size-fits-all approach inevitably forces a trade-off between maximizing GPU efficiency and fully leveraging CPU cores.

Figure 2 shows the performance of matrix multiplication for a single tile

on an Nvidia A100 GPU and an AMD Zen3 core, with different tile sizes. The results demonstrate the importance of using small tiles on CPUs, as peak performance is achieved with a square tile of size 480. In addition, these small tiles have the advantage of creating more parallelism, allowing all cores to be fully utilized.

However, the use of small tiles on GPUs results in degraded performance compared to larger tiles, as GPUs cannot be used optimally with such fine-grained tasks (peak performance obtained with a square tile of size 1920, 2240 or 3840). Therefore, using both types of processing units requires the adoption of different granularity to efficiently utilize the entire platform.

To this end, `StarPU` introduces recursive tasks (see Section 2.4) with automatic data management in a heterogeneous context which aims at having tasks at different granularities.

2.3. The `StarPU` *Runtime System*

`StarPU` is a library designed to efficiently harness the power of heterogeneous architectures. It provides a portable interface based on the STF paradigm. The goal of `StarPU` is to provide a complete API to take full advantage of heterogeneous architectures.

Listing 1 provides a simple example of tasks submitted with `StarPU`. One must first register the data (line 23), here a vector. `StarPU` provides a variety of data interfaces, including vectors, matrices and sparse matrices and allows to define custom data interfaces. Since `StarPU` manages data transfers, this step is essential to inform `StarPU` about the data's structure. The data is then declared as potentially partitioned to leverage parallelism (line 25). This allows tasks to be submitted to the sub-parts of the vector (lines 16-17). Finally, we must wait for the completion of the submitted tasks (line 29).

Listing 1: Code corresponding to the submission of a vector scaling with StarPU.

```
float V[256];
starpu_handle vec, svec[PARTS];

void vector_scal(starpu_handle svec)
{
  /* StarPU provides helpers to retrieve
     information from opaque view*/
  float *V = starpu_get_vector(svec);
  for (int i=0; i < starpu_get_vector_length(svec); i++)
    V[i] *= 3;
}

void submit_tasks(starpu_handle *handles)
{
 /* one task submission by subpart */
 for (int i=0; i < PARTS; i++)
  starpu_task_insert(&vector_scal, STARPU_RW, svec[i], 0);
}

int main()
{
 /* registering data inside StarPU */
 starpu_vector_data_register(&vec, V, 256, sizeof(*V));
 /* declaring data subparts */
 starpu_data_partition_plan(vec, PARTS, svec);
 /* task submission */
 submit_tasks(svec);
 /* wait for task end */
 starpu_task_wait_for_all();
 /* clean */
 starpu_vector_data_unregister(V);
 return 0;
}
```

From there, StarPU orchestrates the execution of the tasks, using a task scheduling policy. This allows StarPU to automatically decide the task placement and manage the data transfers, without requiring any placement hints from the application.

The StarPU scheduling policy DMDA [26] relies on task performance modeling, which is achieved through automatic calibration, that can include averaging completion times or using regression models (linear or non-linear) to predict performance based on task size, matrix rank, etc.

To achieve this, each time a task is executed by a processing unit, a description of the task is registered and saved to disk upon StarPU shutdown. The description depends on the information needed by both the scheduler

and the application. For dense linear algebra, it includes task data size and completion time, while for sparse linear algebra, the matrix rank is also included.

This approach has been shown to be effective in sparse linear algebra (e.g., QR-mumps) [27] and Fast Multipole Method (FMM) [15]. Moreover, `StarPU` supports data that may have different representations between CPUs and GPUs (e.g. structure of arrays vs. array of structures). The programmer simply needs to provide `StarPU` with the appropriate conversion functions during the initialisation phase. For instance, for CPU/CUDA transfer, it requires `ram_to_cuda` and `cuda_to_ram` functions. To enable data partitioning that respects these specific representations, device-specific partitioning methods should also be provided. Naturally, enforcing a conversion introduces additional delay compared to direct device-to-device transfer, due to the invocation of these conversion functions.

While some initial runs are required to calibrate the models, this is already a standard practice when testing applications on the target platform prior to full-scale execution. The performance models generated through this method provide reliable runtime predictions, and small inaccuracies in the models have minimal impact on performance. In fact, dynamic scheduling heuristics are robust to minor variations, with noticeable performance degradation only occurring due to significant disruptions [28].

Moreover, this approach is highly portable across different GPUs (e.g., A100 vs. H100, or even mixed setups) since the performance model is automatically tuned for each architecture.

As mentioned earlier in Section 2.2, the increasing complexity and heterogeneity of computing platforms make it more difficult to efficiently utilize the entire system. To address these challenges, `StarPU` provides several extension, including Recursive Tasks, which will be the focus of the following subsection. It also supports parallel workers [29], where CPU cores are grouped according to hardware topology, allowing a task to be executed in parallel using another runtime system, such as OpenMP. This extension will be discussed in Section 6.3.

2.4. Recursive Tasks Management within `StarPU`

Recursive tasks have recently been introduced in `StarPU` [13]. This extension of the STF model enables to define a data hierarchy and submit tasks that can be transformed into DAGs at runtime.

We define:

- A *Regular Task* is a task having no recursive implementation, corresponding to traditional STF tasks.
- A *Recursive Task* is a task with a recursive implementation, allowing it to become a DAG at runtime.
- A *Split Task* is a recursive task that is transformed into a sub-DAG, or has been designated to become one.

To ensure sequential consistency, the sub-tasks of a split task can only use sub-data with more restrictive data accesses. For a split task that performs a Read-Write access to data A and a Read-only access to data B, sub-tasks can access subparts of data A in Read mode, Write mode, or Read-Write mode, but can only access subparts of data B in Read mode.

One key feature of `StarPU` is its automatic data management system, which eliminates false synchronization in a heterogeneous context. `StarPU`'s recursive tasks are included in this mechanism and are thus automatically managed by `StarPU`. Recursive tasks introduce additional costs. We distinguish between the overhead caused by the submission and data management process, which will be discussed in Section 2.4.3, and the overhead resulting from the decision process to split a task or not, as discussed in Section 5.

2.4.1. Writing a Code Using `StarPU`*'s Recursive Tasks*

Listing 2: Code corresponding to the submission of a vector scaling with `StarPU`'s recursive tasks.

```
float V[256];
struct rec_arg {
    starpu_handle *h;
    struct rec_arg *subparts[PARTS];
};
starpu_handle vec, svec[PARTS], ssvec[PARTS*PARTS];
starpu_handle *handles[3]={&vec, svec, ssvec};

struct rec_arg parent_arg;

int submit_tasks(struct rec_arg *rec_args, int n)
{
 for (int i = 0; i<n; i++)
  starpu_task_insert(&vector_scal,
   STARPU_GEN_DAG, &gen_dag, rec_args[i],
   STARPU_RW, rec_args[i]->h, 0);
```

```
17 }

19 void gen_dag(struct rec_arg *arg)
   {
21   submit_tasks(arg->subparts, PARTS);
   }
23
   int main()
25 {
    starpu_vector_data_register(&vec, V, 256, sizeof(*V));
27  starpu_data_partition_plan(vec, PARTS, svec);
    for (int i = 0; i<PARTS; i++)
29   starpu_data_partition_plan(svec[i], PARTS, ssvec+i*PARTS);
    /* Initializing parent_arg hierarchy */
31  ...
    submit_tasks(&parent_arg, 1);
33  ...
   }
```

To utilize recursive tasks, one must:

1. Describe how to recurse on data pieces. It is essential to ensure that sub-tasks are submitted only on data sub-pieces belonging to the task. One must explicitly describe to `StarPU` how to partition the data into subparts. This is done using `starpu_data_partition_plan()`
2. Define, for each type of task, a function that submits all the sub-tasks of the recursive form of the task. This function takes as parameter the data pieces.
3. Submit tasks at the highest granularity. `StarPU` will decide to potentially transform tasks into sub-DAGs thanks to the user-defined function described above.

Listing 2 shows a vector scaling simple example using recursive tasks. Comparing this example to Listing 1 exhibits the differences between regular tasks and recursive tasks. In this example, the differences between both approaches remain limited, since `submit_tasks` already provided a task-graph implementation of the tasks. More generally, to use recursive tasks, a programmer needs to provide a task-graph version of each task. In task-based dense linear algebra libraries for instance, most tasks already have a task-graph implementation, because these tasks are exposed in the library API and thus already deserved a parallel implementation. This may not

always be the case for some complex tasks which are not exposed in the API and thus require adding a task-graph implementation.

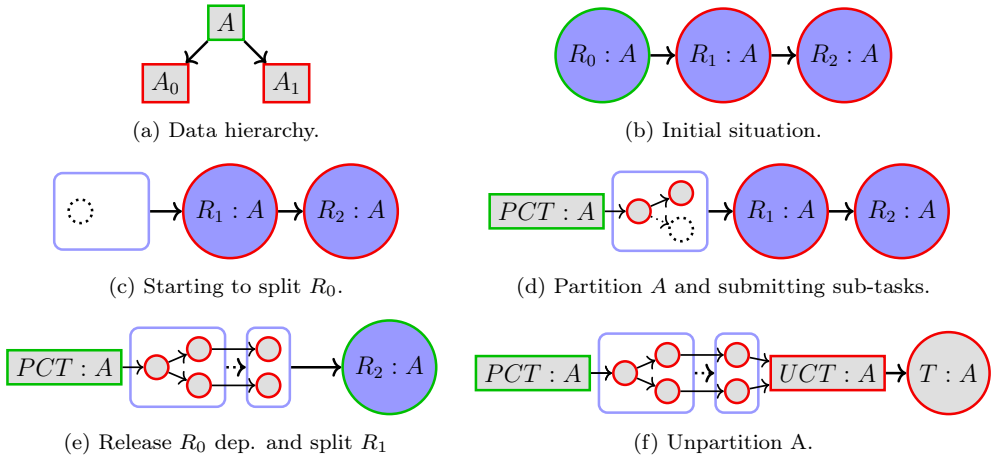
More specifically, the initialization of the process (lines 26-35) is similar to the non-recursive version. In fact, we start by registering the vector (the level 0, line 26), and then partitioning it (into level 1, line 27). The vector's subparts are also partitioned (into level 2, line 29) since recursive tasks will end up running on these sub-subparts.

Then, the submission process is different. Without recursive tasks, tasks were submitted directly at the execution granularity. With recursive tasks, programmers only have to submit the task at the highest granularity (here on the handle `vec`). `StarPU` will then iterate through all granularities, determining whether the task can be split, here by simply checking if the current data can be partitioned. If so, it will submit the sub-tasks using the `gen_dag()` function, which, in turn, calls the `submit_tasks()` function to handle the task submission. It is possible to force a task to stay regular by setting a `STARPU_IS_REC` flag to 0 in the `starpu_task_insert()` call. This task submission function (line 11) is very similar to the one presented in Listing 1: it expresses parallelism through a for loop that submits tasks. The difference is that it passes the address of the `gen_dag` function to allow recursion, and it uses a user-defined structure to determine which part of the vector to submit tasks for. Letting the application find the subparts of the data (here done with the `parent_arg` structure) allows more choices, especially in the context of matrices, where the division can either be column-based or row-based, or even both. As explained in Section 2.4.2, `StarPU`'s data manager is then responsible for ensuring coherency between the different data levels, since tasks can be executed on different levels, and different partitioning.

2.4.2. Ensuring Coherency Using `StarPU`*'s Data Manager Mechanism*

In order to ensure data coherency when using recursive tasks, `StarPU`'s data manager registers for each piece of data two fields. The first is the *active* field, and the second is the *partitioned* field. When a data is *active* but not *partitioned*, it is directly usable. If it is *active* and *partitioned*, it means that sub-data of this data are *active*, therefore the data cannot be used directly, it should be *unpartitioned* first. If the data is not *active*, it means a parent of this data is *active* and must be *partitioned* before this data can be used.

Figure 3 illustrates the processing of a split task by `StarPU`'s data manager. The state of each task (i.e. node in the graph) is described by its border: 1) a ready task has a **green** border (all dependencies are met), 2) a



(a) Data hierarchy.

(b) Initial situation.

(c) Starting to split R_0.

(d) Partition A and submitting sub-tasks.

(e) Release R_0 dep. and split R_1

(f) Unpartition A.

Figure 3: Recursive task graph processing by `StarPU`'s data manager. Splitting task R_0 means first submitting a partition coherency task that provides synchronization between data A and sub-data A_0 and A_1, then submitting the sub-tasks, then (if R_1 is not split) submitting an unpartition coherency task to synchronize back.

not-ready task has a **red** border (some dependencies are unsatisfied), 3) an already executed task has a **black** border. We show three tasks with write access to data A (parent of two data A_0 and A_1, Figure 3a), the three tasks R_0, R_1 and R_2 are recursive. R_0 and R_1 will be split, while R_2 will stay regular[1]. In the initial stage, data A, the parent of its data hierarchy, is *active* and not *partitioned* as no task has been submitted. The three recursive tasks are submitted, as illustrated in Figure 3b. Since R_0, R_1 and R_2 use data A that is *active*, `StarPU`'s data manager has nothing to do to allow this submission.

Figure 3c illustrates that, as R_0 has no incoming dependencies, it is ready, and as a result `StarPU` splits it. Upon submission of the first sub-task of R_0, which uses data A_0 and A_1 the data manager examines the status of A, as it is the direct parent of these sub-data. As described before, data A is *active* but not *partitioned*. To allow the use of A_0 and A_1 within the sub-DAG of R_0, it is therefore necessary to *partition* data A. `StarPU` will automatically submit a *Partition Coherency Task* (PCT) (Figure 3d). This task has the incoming dependencies of R_0 related to A, and all tasks submitted on subparts of

[1]We remind that a recursive task is a task with a recursive implementation, that can be split (and become a sub-DAG), or become a regular task.

A are dependent on it. This will also set A in a *partitioned* state, and A_0 and A_1 in a *active* state. The remaining sub-tasks of R_0 may then be submitted, as A_0 and A_1 are *active*. Subsequently, upon completion of the submission of the sub-DAG of R_0, the dependency from R_0 to R_1 is released, thus rendering R_1 ready, in parallel with the execution of sub-tasks of R_0. To ensure coherency, it would have been possible to insert a synchronization barrier after the sub-DAG of R_0. However, although such a barrier would enforce correctness, it would be detrimental to performance, as it would prevent the pipelined execution of consecutive sub-DAGs. Instead, `StarPU` ensures coherency through a different mechanism. Thus, since R_1 is a *split* task, the submission process can start. Given that A is *partitioned*, and A_0 and A_1 are already *active* (as a result of the splitting of R_0), the submission of the sub-DAG does not require the creation of an additional *Partition Coherency Task*. This results in fine-grained dependencies between the two sub-DAGs (similarly to OmpSs's *weak* dependencies), rather than a single coarse-grain dependency (Figure 3e). As the submission of R_1's sub-DAG only takes place after the submission of R_0's sub-DAG, dependencies between tasks belonging to different sub-DAGs are correctly handled by the sequential consistency mechanism, at the sub-data level. Overall, while coarse-grain dependencies guide the submission process, they are later substituted by fine-grain dependencies that reflect the true dataflow between computational tasks.

Following the submission of R_1's sub-DAG, the coarse-grained dependency from R_1 to R_2 is released, making R_2 ready for execution. R_2 remains a regular task (denoted as T) that accesses the entire data A. Since A is currently *partitioned*, the data manager must ensure coherency before executing T by submitting an *Unpartition Coherency Task* (UCT) (see Figure 3f). This task removes the *partitioned* state of A, as well as the *active* states of A_0 and A_1.

As seen in this simple example, `StarPU` automatically manages all data coherency, without requiring extra calls from the application. This is achieved through the submission of *Partition Coherency Tasks* and *Unpartition Coherency tasks*, which are strictly synchronization tasks. These tasks are solely executed on CPU cores, ensuring that all necessary data pieces are brought back to the memory node where they were originally registered. The general algorithm ensuring the correctness of a recursive task graph is described exhaustively in the initial article introducing `StarPU` recursive tasks [13].

It can also be observed that a recursive task that ultimately stays regular

(such as R_2 in the previous example) may need to wait for dependencies releases in two situations. First, there are the *regular dependencies*, which occur when tasks access the same data, such as the dependency from R_0 to R_1, and R_2 to R_2 due to accesses to A. Second, there are the *inter-level dependencies* introduced by the coherency tasks.

2.4.3. Overhead introduced by Recursive Tasks

[13] provides an analysis of the overhead induced by task submission and data management. In an intentionally extreme scenario, each task is submitted twice: the top-level task is split into a sub-DAG that contains only one task, identical to the original one. In this case the total submission time (comprising the data management process) becomes approximatively three times higher. This increase results from the submission of twice as many tasks by the application, the addition of two extra tasks (one *Partition Coherency Task* and one *Unpartition Coherency Task*) for each top-level piece of data by the data manager, and also an overhead on the data manager due to the recursive sequential consistency mechanism. To evaluate the impact, the total submission time is divided by the number of regular tasks (in this case, only the lowest-level tasks). Under this condition, the submission time per regular task increases from 2.4 μs to 7.5 μs. However, this case remains theoretical and does not reflect the usual practice of recursion, which aims at splitting a task into several smaller sub-tasks. Thus, when a task is split into a sub-DAG of a few dozens tasks (e.g. 27 tasks corresponding to dividing a GEMM task into a sub-DAG on $3 * 3$ sub-tiles), the submission time per fine-grained task is close to the one without recursion, with only about a 10% overhead on the total submission time (from 2.4 to 2.6 μs per computational task). One effective way to reduce the submission overheads is to record recursive task graphs and replay them when needed as proposed in [30, 31, 32, 33].

3. The Splitter, a Decision Tool for Turning Tasks into Sub-DAGs

The recursive task management described in the previous section only provides the mechanism for managing several task granularities. The remaining challenge is to determine the optimal granularity for each task automatically, instead of letting this decision to programmers. In a preliminary work [16], we introduced a new component, called the *Splitter*, within the

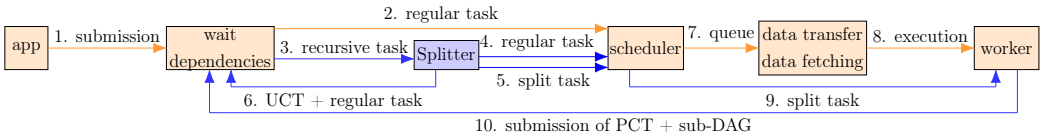


Figure 4: Adding the Splitter in the existing architecture. Existing components and paths are in orange, the added ones are in blue.

`StarPU` scheduling framework. The Splitter receives a recursive task as input, and determines whether or not the task needs to be split.

The Splitter's fundamental objective is to select the suitable granularity for a task, without being involved in the selection of the processing units that will execute it. However, it is important to note that the granularity selection process can have a significant impact on scheduling.

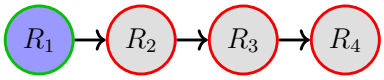
First, splitting a task along the critical path enables parallelism, reducing its overall duration. If a task T on the critical path has a duration of X and is parallelized across N processing units, the remaining path length from this task to completion decreases by $X - \frac{X}{N}$. The impact on scheduling can be significant. If the reduction in path duration causes a shift in the critical path – where another path in the graph becomes longer – all tasks on this new critical path must be prioritized. This shift can fundamentally alter scheduling decisions, requiring a complete reassessment of task execution order.

Secondly, creating parallelism by splitting a task increases the number of scheduling decisions available. This allows the scheduler to make more informed choices and efficiently fill gaps in the schedule by utilizing smaller tasks.

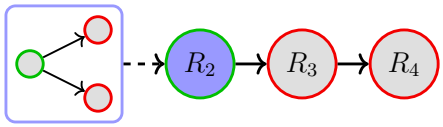
To decide whether a recursive task has to be split, the Splitter can leverage runtime information from the application, such as the availability of parallelism and task priorities, as well as insights provided by `StarPU`, including the performance models of the computation kernels at various granularities, and the current workload of the workers. We present, in the next section, the integration of the Splitter into `StarPU` to enable timely task splitting decisions, and discuss its efficient utilization on homogeneous machines to achieve high performance.

3.1. Optimizing Task Splitting Decisions: A Just-in-time Approach

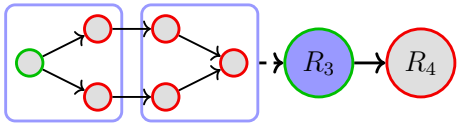
We must consider how to integrate the Splitter into a task-based RTS like `StarPU`. This requires addressing two key questions. The first concerns *which*



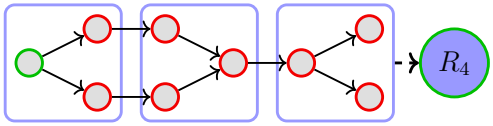
(a) Initial situation. R_1 is ready, and decided to be split. Start R_1 sub-DAG submission.



(b) Sub-DAG submitted. Release dependency. R_2 is ready, and split. Start R_2 sub-DAG submission.



(c) Sub-DAG submitted. Release dependency. R_3 is ready, and split. Start R_3 sub-DAG submission.



(d) Release dependency. R_4 is ready. Splitting it is decided before any execution of any real task.

Figure 5: Premature release of coarse-grained task dependency with recursive tasks, that leads to premature decision.

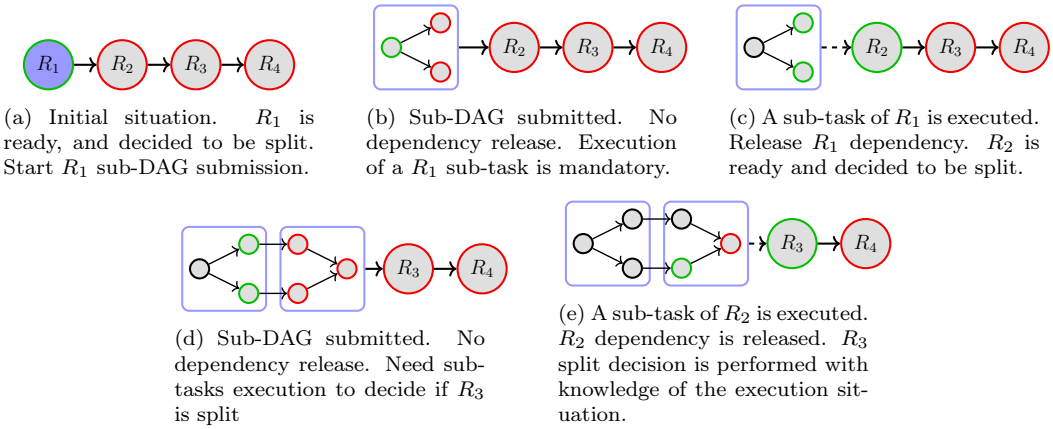
tasks should be split, a decision that depends on the information available in the RTS at the time the decision is to be taken. This naturally leads to the second question: *when* tasks should be split.

Task-splitting decisions can be made at various stages of the RTS workflow, including the submission stage. While splitting at submission minimizes overhead, it has two drawbacks: (1) it does not significantly reduce the number of tasks stored in the system, and (2) the decision to split a task could be made far in advance of its execution, lacking comprehensive knowledge of the machine's future state.

Alternatively, the decision could be postponed later in the execution process when the machine's state is better known. However, unlike regular tasks, executing split tasks does not require data transfers. To prevent redundant transfers, it is therefore advisable to decide on task splitting before the data prefetching stage.

Hence, a more promising approach is to delay the decision-making process until a recursive task has fulfilled all its regular dependencies. This enables a more informed decision, taking into account the machine's state closer to execution.

However, this precaution alone may not be entirely sufficient to ensure optimal decision-making. Let us consider a sequence of recursive tasks denoted as $R_1 \rightarrow R_2 \ldots \rightarrow R_k$. Let us assume that all tasks preceding R_k are split, as illustrated in Figure 5. The decision to split R_k can be made before the execution of the sub-tasks generated by R_1, as the regular dependencies of a split task are released when the submission of the sub-tasks is over.



(a) Initial situation. R_1 is ready, and decided to be split. Start R_1 sub-DAG submission.

(b) Sub-DAG submitted. No dependency release. Execution of a R_1 sub-task is mandatory.

(c) A sub-task of R_1 is executed. Release R_1 dependency. R_2 is ready and decided to be split.

(d) Sub-DAG submitted. No dependency release. Need sub-tasks execution to decide if R_3 is split

(e) A sub-task of R_2 is executed. R_2 dependency is released. R_3 split decision is performed with knowledge of the execution situation.

Figure 6: Solving problematic release of Figure 5 by delaying dependency release until some sub-task execution.

Without additional safeguards, the splitting decision might occur well before the computation progresses significantly.

Therefore, in order to harmonize the task splitting decision with the computation progress, we have developed a technique to postpone the release of outgoing regular dependencies of a split task. Instead of releasing these outgoing dependencies immediately after the submission of the sub-DAG, the release is deferred until the execution of one of its sub-tasks (see Figure 6). Thus, the sub-DAG submission of a sequence of split recursive tasks will occur gradually: a recursive task R_2 following a split task R_1 will remain blocked until the execution of a R_1 sub-task as depicted in Figures 6b and 6c.

We summarize the discussion about the possible positions of the Splitter in Table 1.

As shown in Figure 4, we position the Splitter at the initial stage of the scheduler. The handling of a recursive task is as follows: upon submission by the application (1), the recursive task is given to the Splitter (3) once its regular dependencies are satisfied. The Splitter then makes its decision. If the task requires splitting, it is assigned directly to the scheduler (5), which then places it in the execution queue of a worker (9).

The processing of this split task results in the submission of a sub-DAG (10), and a *Partition Coherency Task* if required by the data manager. If the task is not split, the recursive task is transformed into a regular task, and given to the data manager. In the scenario where its data is not split, the regular task is simply executed (4). Alternatively, if its data is processed

Stage	Runtime Information	Parallel Submission	Avoid Redundant Transfers
Submission	✗	✗	✓
Dependency completion	✗	✓	✓
Deferred release	✓	✓	✓
Execution	✓	✓	✗

Table 1: Summary of advantages and drawbacks of different Splitter positions within the task workflow.

by sub-tasks at a lower recursive level, an *Unpartition Coherency task* (6) is automatically submitted to maintain sequential consistency.

3.2. The Homogeneous CPU Case: Trade-Off Between Efficiency and Parallelism

The role of the Splitter is to decide which task should be split to optimize the overall system efficiency. The Splitter has to find a balance between task efficiency and parallelism: coarse-grain parallelism leads to more efficient but less parallel tasks, while fine-grain parallelism allows to use many processing units. As an initial strategy to address this question in a homogeneous CPU-only context, two different simple criteria are combined:

- The *splitting efficiency* is assessed by comparing a task's sequential execution time to the total execution time of its sub-tasks. A task is not split if this ratio falls below 50%, as the resulting inefficiency would outweigh the benefits of parallelism. Computing such ratios is easily done by `StarPU`: as described in Section 2.3, each time a task is executed by a worker, `StarPU` records the task execution time in its *performance model* [34]. Thus, both the sequential and the cumulative parallel execution times can be computed using the performance model.

- The *idleness* of the cores. To ensure balanced workload distribution across all cores, it is crucial to maximize resource utilization. Since all cores have the same processing power, tasks should be divided efficiently to keep them fully occupied. However, splitting tasks is counterproductive when all cores are already fully utilized. Therefore, we choose to split a task only when the number of tasks in the RTS falls

below three times the number of processing units, while ensuring this does not violate the previously established efficiency criterion.

3.2.1. Experimental Results

The benefits of this simple splitting strategy are illustrated by conducting a performance analysis of the tiled dense Cholesky factorization (POTRF) on a homogeneous system from the Chameleon dense linear library [17] [2]. The platform has two Intel Xeon 6240 CPU @ 2.60GHz, each with 18 cores. Three tile sizes are used: a coarse (1120), a medium (560) and a fine one (280). We use `StarPU`[3] with the locality-aware work stealing scheduler (*lws*).

This scheduler creates one queue by worker, and when a task is ready, it is pushed on the worker queue that released its last dependency. A worker pops tasks from its own queue while it is not empty, or otherwise it steals tasks from other workers, following the hardware hierarchy. We used this low-overhead scheduler as it obtains the best performance on homogeneous platform. [4]. The code was compiled using GCC 11.4. Each experiment was executed five times, and we report the average results across these runs.

Figure 7 compares the non-recursive version for these three tile sizes, with different recursive versions and state-of-the-art implementations, namely a Cholesky factorization on top of OmpSs-2 [14] [5] and the vendor implementation available in the Intel Math Kernel Library [6] [7]. For the sake of readability, we do not illustrate an OpenMP standard version as the results were similar to the OmpSs-2 version. A Peak Performance limit is also provided, which corresponds to the peak performance obtained using a general matrix multiplication of Chameleon on this platform, for the coarse granularity. The non-recursive curves show that: (1) using larger tile sizes alone is not interesting, since the efficiency for these processing units is not improved, (2) using finer tile sizes is not interesting either, as it decreases performance too much, while (3) using medium size seems promising.

[2]The version is a slighty modified version of Chameleon 1.2, commit `96355dd8acbbadfd2046ad72a8029085e5aba877`.

[3]The version is a slightly modified version of `StarPU` 1.4, commit `690ca6fb47ca825024f60613bc3ea99a89562ce8`.

[4]This scheduler is however not efficient in a heterogeneous platform.

[5]version 2024.05

[6]The memory interleave policy is set to all.

[7]version 2022.0.2

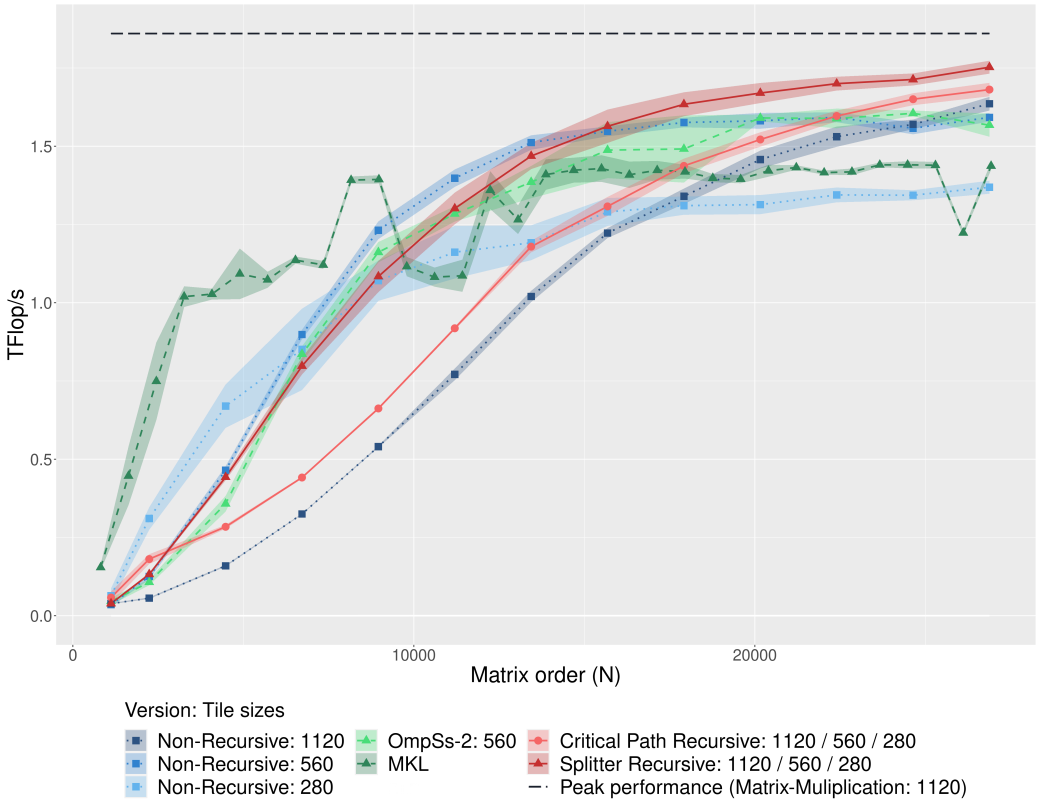


Figure 7: Performance of different version of Cholesky Factorization on a homogeneous platform.

The recursive curves show the performance obtained with several recursive approaches:

1. The *critical path* variant where we split a task if it writes to a data on the diagonal or the sub-diagonal. These tasks are considered to be on the critical path [35].
2. The *Splitter* variant where we split a task according to the dynamic criteria explained in the previous section.

We observe that our criteria allow a speed-up of approximately 10% (coarse tile) compared to the best non-recursive version for the largest matrix sizes, and a speed-up of 5% over the critical path recursive version.

For small matrices, the MKL version is the fastest, as it allows small overhead and fine-tuned parallelism compared to the non-recursive versions.

The Splitter version is generally a good trade-off, as it is close to the best non-recursive version at each point of the execution, and makes the execution really adaptive to the context.

Compared to the other state-of-the-art versions, it is notable that, for a given tile size, the OmpSs-2 version obtains similar results to the corresponding non-recursive `StarPU` version. For the sake of readability, the only displayed version for OmpSs-2 is for tile size 560, as it achieves the best asymptotic performance starting from a matrix size of 11200. The Splitter version also asymptotically outperforms the MKL version by approximately 20%.

Therefore, in a homogeneous context for dense Cholesky factorization, a simple splitting policy already easily improves performance by 10% compared to the non-recursive version, which demonstrates the benefits of splitting tasks to improve the performance. The next section aims to create a more general splitting policy, for a heterogeneous context.

4. Granularity Steering for Heterogeneous Platforms

In this section, we introduce an oracle to dynamically and automatically adjust task granularity. This is achieved by deciding whether a task should be split in heterogeneous environments, while allowing programmers to specify the set of granularities that can be used.

This work generalizes the contribution presented in the previous section, which was targeting homogeneous architectures. In the homogeneous context, a simple criterion, such as the number of tasks allocated per core, was sufficient to achieve near-optimal performance. Having an heterogeneous setup increases the difficulty, often requiring CPU cores to remain idle to fully utilize the computational power of GPUs.

HEFT-like scheduling strategies [36] are specifically designed to address this challenge. For each task, the algorithm selects the resource (CPU core, GPU, etc.) that is expected to complete it the earliest, taking into account both computation time and data transfer costs. In most of the cases, >this allows to distribute the workload effectively by considering the capabilities of each resource. However they do not have theoretical guarantees that would prevent too bad executions on some instances [37].

Kernel	$\frac{1\ gpu}{1\ core}$	$\frac{1\ gpu}{MKL\ 8\ cores}$; (efficiency)	$\frac{1\ gpu}{MKL\ 64\ cores}$ (eff.)	$\frac{1\ gpu}{StarPU\ 64\ cores}$ (eff.)
`GEMM`	380	61.8 (76%)	10.5 (57%)	9.4 (64%)
`TRSM`	307	37.8 (102%)	7.66 (63%)	6.7 (72%)
`SYRK`	343	45.4 (94%)	16.2 (33%)	11.7 (46%)

Table 2: Comparison of performance ratio for AMD Zen3 cores and Nvidia A100 GPUs on different kernels, and parallel CPU implementation efficiency. All performance correspond to one operation on squared matrices of size 3840. The tile granularity depends on the target device: squared tile of size 3840 for GPU, one CPU core, and several MKL cores; best parallelization for multiple `StarPU` cores (480 for `SYRK` and `GEMM`, 240 for `TRSM`). Similar results are observed on this platform between MKL and AOCL. The provided parallel CPU efficiency is compared to a single CPU core.

4.1. An Illustrative Example

In order to obtain a deeper understanding of the necessity for diverse granularities in the heterogeneous context, we can rely on a simple example to motivate our approach. Consider a scenario in which a set of independent `GEMM` computations is performed. As illustrated in the first column of Table 2, the Nvidia A100 GPU outperforms an AMD Zen3 core by a factor of 380 when executing `GEMM` kernels. Consequently, when 2 A100 GPUs and 64 Zen3 cores are employed, `StarPU`'s `DMDA` scheduling strategy will assign tasks to CPU cores only when more than 760 `GEMM` computations of size 3840 can be executed in parallel, as this strategy assigns a task to the worker that is able to complete it earliest. The execution of such a substantial number of tiled `GEMM` constitutes a matrix multiplication on a dense square matrix of size greater than 100000, resulting in a data footprint that exceeds 80 gigabytes.

Consequently, CPUs are rarely employed in the context of such large GPUs, with only a single granularity. Furthermore, when a task lacks a GPU implementation and must be executed on a CPU core, GPU idleness frequently occurs. This is because, to maintain GPU occupancy, it is needed to have a few hundreds tasks available.

The aggregation of CPUs can be a viable option, facilitated by the utilization of parallel workers. In such an architecture, it is generally advantageous to aggregate CPU cores by L3 cache to leverage cache effects. This scenario is illustrated by the second column of Table 2, which presents the ratio between the speed of a task for tile size 3840 executed on a GPU, and its speed on such a parallel worker with 8 cores (an L3 cache), as well as the efficiency of the parallel 8-core worker compared to one core. The parallel worker here

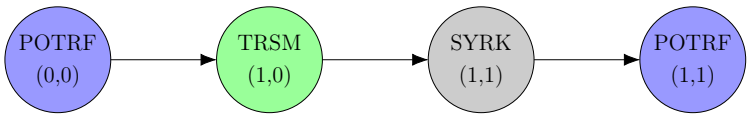


Figure 8: Sub-DAG corresponding to the split of one `POTRF` on a tile subdivided on 2×2 sub-tiles. Each node represents a sub-task. For each node, the first line indicates the task type, and the second line shows the coordinates of the sub-tile being written.

calls a multithreaded implementation of the operation provided by Intel MKL (AMD AOCL obtains equivalent behavior). The `DMDA` scheduling strategy, consequently, assigns tasks to CPU cores when the number of `GEMM` computations of size 3840 exceeds 124 (61.8×2). It represents a dense square matrix of size larger than 45000, with a data footprint of approximately 14 gigabytes.

Using recursive tasks that enables granularity adjustment, will optimize CPU and GPU utilization. The same table shows that by splitting `GEMM` tasks into sub-tasks with CPU-adapted granularity, which can be executed across multiple cores instead of just one, the overall ratio decreases from 123.6 to 18.8 (9.4×2). This indicates that the CPU cores could be engaged in the computation as soon as 20 `GEMM` computations are present in the system: 19 `GEMM` will be processed on the two GPUs, and the 20^{th} `GEMM` will be split into enough tasks for all the 64 CPU cores. Therefore, it can be concluded that a splitting ratio of 1 task for every 20 `GEMM` computations would be an effective policy, bridging the performance gap between CPU and GPU, and significantly enhancing the efficiency of the `DMDA` scheduling strategy.

However, the situation becomes more complex when considering diverse types of tasks: a simple ratio is insufficient to determine whether a task should be split. In Table 2, `GEMM` tasks demonstrate a higher speedup factor in GPU compared to `TRSM` tasks. Consequently, when the Splitter is confronted with the choice between splitting a `TRSM` or a `GEMM` task, prioritizing the splitting of `TRSM` tasks is advisable, as their efficiency on GPU is lower. This highlights the importance of considering all available tasks when computing a workload distribution.

When a `POTRF` task with a square tile of size 3840 is transformed into a sub-DAG on sub-tiles of size 1920, a chain of task is created, beginning by one `POTRF`, followed by one `TRSM`, one `SYRK`, and concluded with one `POTRF`, as represented in Figure 8.

It is noteworthy that executing solely the `POTRF` sub-tasks on CPUs, while

concurrently executing the `SYRK` and `TRSM` tasks on GPUs results in a reduction of the CPU completion time for the large `POTRF` task. Furthermore, the efficiency of the GPU-executed tasks is increased compared to the GPU execution of the large task, since these `SYRK` and `TRSM` tasks are more efficient than `POTRF` tasks. Thus, the execution of sub-tasks at a finer granularity on CPUs can be accomplished without inducing GPU idleness. This represents the main difference with parallel workers as relying on recursive tasks allows the execution of sub-tasks on all resources, including GPU devices.

4.2. A Generic Linear Program for Splitting Ratio Computations

To address this complexity, our starting point, inspired by Steady-State Scheduling [38], is to assume that the workload evolves progressively. We sample the workload over a period, compute an optimized distribution for this sample, and then apply this ideal distribution repeatedly. To adapt to the progressive evolution of the workload, we periodically re-sample the workload, i.e. we observe the present to take decisions on the near future.

This workload is obtained by maintaining in the RTS the set of available tasks, which includes the tasks that are either ready or currently being executed by a processing unit.

Our objective is thus to design a heuristic for identifying the optimal ratio of tasks of each type and level to be split within the set of ready tasks (which serves as our sample). The Splitter will then decide whether to split a task according to the calculated ratios, thus reaching the balance between efficiency and parallelism. The ratios are adjusted each time we re-sample the workload, thus guiding the Splitter along the evolution of the workload.

We have chosen to use a Linear Program (LP) to determine these ratios, although a handmade solution could be used instead. This choice is justified for several reasons. First, an LP allows us to formulate the problem in a structured manner, with well-defined objective functions and constraints, providing a formal mathematical representation of the task-splitting problem. Secondly the LP offers a proven near-optimal solution given the constraints, and, thirdly, its implementation is straightforward, thanks to highly optimized existing libraries.

For performance reasons, we simplify the model: it is not solved as an integer programming problem, and sub-task dependencies are not explicitly considered but are modeled by variables. As a result, the LP provides approximated ratios.

In the following LP, each granularity is associated with a specific level of recursion. It is therefore assumed that, for a task type t and level l, each task has the same granularity. We denote the execution time of task t at level l on a processing unit of type u as $Ex^u_{t,l}$. This time corresponds to the computational time of the task as estimated by `StarPU`'s performance model, plus the overhead incurred by the RTS in handling the task [8]. This last part is important to prevent the LP from deciding to split into tasks of completion time less than a few microseconds.

The generic linear program is presented in Figure 9. The objective of the LP is to minimize the global execution time for a given set of task $\mathcal{T}$ across a given set of processing units $\mathcal{R}$, R^u. The given set of task $\mathcal{T}$ and the number of tasks of each type $N^{tot}_{t,l}$ for each task type t and level l corresponds to all the tasks that are currently ready. This number is obtained by looking at the RTS state when launching the LP.

The minimization of this global execution time is performed through the utilization of a variable exT which encodes the global execution time. Furthermore, for each type of PU u, level l and task type t one variable $Ne^t_{l,u}$ is used to encode the number of tasks of type t and level l that the LP executes on a PU of type u. Similarly, for each level l and task type t, one variable Ns^t_l is used to encode the number of task of type t and level l that should be split.

(1) Task number splitting: The first constraint ensures consistency between the number of tasks that are split and the resulting number of sub-tasks. When a task of type p at level l (with $l < \mathcal{L}$) is split, it generates $nsub^t_{p,l}$ sub-tasks of type t, which are submitted at level $l+1$. Therefore, for each task type t and level l, the total number of tasks of this type - either executed on a PU ($Ne^t_{l,u}$) or split further (Ns^t_l) - must be at least equal to:

- The current number of tasks of type t already present at level l ($N^{tot}_{t,l}$),
- Plus the number of sub-tasks of type t generated from the splitting of parent tasks at level $l-1$.

This ensures that every time a task is split, its sub-tasks are properly accounted for, being either executed on a PU or split (if possible).

[8] Manually set to 5 μs

Automatically inferred parameters

$\mathcal{T}$		Set of task types.
$\mathcal{R}$		Set of processing units types.
$N_{t,l}^{tot}$	$t \in \mathcal{T}$	Number of task not split of type t at level l.
R^u	$u \in \mathcal{R}$	Number of PU of type u.
$Ex_{t,l}^u$	$t \in \mathcal{T}, l \leq \mathcal{L}, u \in \mathcal{R}$	Execution time of task t of level l on PU of type u
$nsub_{p,l}^t$	$p, t \in \mathcal{T}, l < \mathcal{L}$	Number of task t of level $l+1$ submitted when splitting p at level l

Programmer-specified parameters

$\mathcal{L}$		Maximum level of recursion.
$MinN_u$	$u \in \mathcal{R}$	Minimal wanted number of tasks on each PU of type u

Variables

exT		Total execution time.
Ns_l^t	$t \in \mathcal{T}, l < \mathcal{L}$	Number of split task of type t and level l.
$Ne_{l,u}^t$	$t \in \mathcal{T}, l \leq \mathcal{L}, u \in \mathcal{R}$	Number of task of type t and level l executed on PU u

Minimize

$$exT$$

Subject to

Task number splitting.

$$\sum_{u \in \mathcal{R}} Ne_{l,u}^t + Ns_l^t - \sum_{p \,\in\, parent(t)} nsub_{p,l-1}^t \cdot Ns_{l-1}^p \geq N_{t,l}^{tot} \qquad t \in \mathcal{T}, 0 < l \leq \mathcal{L} \qquad (1)$$

No last−level splitting.

$$Ns_{\mathcal{L}}^t = 0 \qquad \forall\, t \in \mathcal{T} \qquad (2)$$

Completion time when executing tasks.

$$\sum_{\substack{t \in \mathcal{T} \\ 0 \leq l \leq \mathcal{L}}} Ne_{l,u}^t \cdot Ex_{t,l}^u - R^u \cdot exT \leq 0 \qquad u \in \mathcal{R} \qquad (3)$$

Minimal number of tasks on PU type.

$$\sum_{\substack{t \in \mathcal{T} \\ 0 \leq l \leq \mathcal{L}}} Ne_{l,u}^t - R^u \cdot MinN_u \leq 0 \qquad u \in \mathcal{R} \qquad (4)$$

Figure 9: Linear program used by the Splitter to efficiently determines the ratio to split for each type of task.

(2) No last-level splitting: The purpose of the 2^{nd} equation is to prevent the splitting of a task at the finest level.

(3) Completion time when executing tasks:

The third constraint ensures consistency between the amount of executed tasks and the available computing time for each resource type. Specifically, for each processing unit type u, the total workload - calculated as the sum of the number of non-split tasks $Ne^t_{l,u}$ multiplied by their execution time $Ex^u_{t,l}$ - must not exceed the total available computing time for this resource. The total available computing time is obtained by multiplying the execution time exT by the number of resources of type u, R^u.

(4) Minimal number of tasks on PU type: Since the linear program is relaxed (i.e., not solved as an integer program, to reduce the cost), it would assign fractional tasks to a resource—leading to unrealistic execution plans where a single large task is partially assigned to a resource without being split. This will be discussed in Section 4.3.1. To avoid this, the 4^{th} constraint ensures that each processing unit type is assigned a minimal number of tasks, dependant on the number of processing units, R^u. This number is multiplied by a parameter $MinN_u$.

It is important to note that this linear program does not consider memory transfers, which can be crucial for achieving performance. This is because these transfers are taken into account by the scheduler, which is invoked afterwards.

After resolution, the LP provides a number Ns^t_l for each type of task t and level l, that are used by `StarPU` to determine which task should be split or not. To obtain more usable numbers, these numbers are turned into ratios. We note $S^l_t = \frac{Ns^t_l}{N^{tot}_{t,l}}$ the ratio of tasks of type t at level l that should be split.

4.3. Discussion

In the previous section, we have presented a modeling of the splitting problem using linear programming. However, this modeling is not entirely sufficient to consider all the difficulty of the problem. The major drawbacks are related to the consideration of task dependencies, and the manner of applying this splitting.

In this section, we discuss these limitations and we present solutions to improve the linear program and its usability. To illustrate this discussion, we present the parameters of an example that will be used below:

- $\mathcal{T} = \{t\}$. The set of task is only constituted of one type of task.

- $\mathcal{L} = 2$. This task exists at two levels of recursion, the coarse and the fine granularity.
- $\mathcal{R} = \{CPU, GPU\}$. The example platform is constituted of two types of Processing Units, CPUs, and GPUs.
- $N_{t,0}^{tot} = 12$, $N_{t,1}^{tot} = 0$. There are only 12 tasks of type t, all at level 0.
- $R^{CPU} = 20$, $R^{gpu} = 2$. The example platform has 20 CPU cores and 2 GPUs.
- $Ex_{t,0}^{CPU} = 20$ ms, $Ex_{t,1}^{CPU} = 4$. Executing t on a CPU core takes 20 ms at level 0 and 4 ms at level 1.
- $Ex_{t,0}^{GPU} = 2.5$ ms, $Ex_{t,1}^{GPU} = 0.5$. Executing t on a GPU takes 2.5 ms at level 0 and 0.5 ms at level 1.
- $nsub_{t,0}^{t} = 11$. Splitting task t at level 0 will creates 11 sub-tasks of type t. The created sub-DAG is represented in Figure 10.
- $MinN_{CPU} = 2$, $MinN_{GPU} = 4$. It is required to have at least 2 tasks per CPU core and 4 tasks per GPU.

4.3.1. Why we Need to Enforce a Minimal Number of Tasks

From the LP perspective, the completion time of a large task would ideally be equal to the sum of the sequential sub-tasks' execution times divided by the number of processing units. However, due to the dependencies among the sub-tasks within a given sub-DAG, its parallelization cannot be considered embarrassingly parallel. For instance, as shown in Figure 10, the execution of this sub-DAG remains partially sequential,independently of the number of processing units. Even worse, the 3^{rd} constraint of the LP assumes that all the work (and not simply the recursive work) is embarrassingly parallel.

Based on the configuration described above, we illustrate the consequences of a modeling approach that considers only the first three constraints when scheduling 12 recursive tasks at level 0. In this case, the 3^{rd} constraint can be satisfied by assigning 4 tasks to the GPU and the remaining 8 to the CPUs, without requiring any task splitting:

- $Ne_{0,GPU}^{t} = 4$
- $Ne_{0,CPU}^{t} = 8$

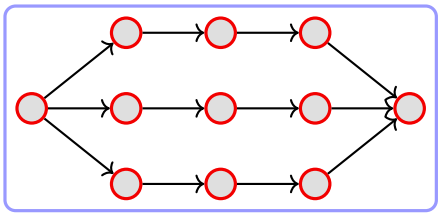
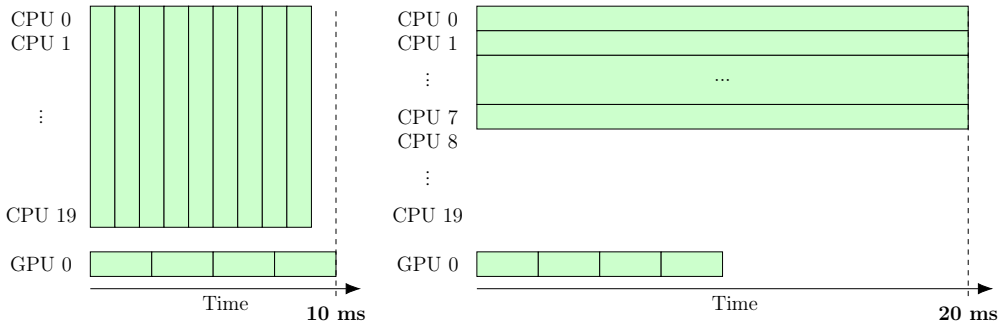


Figure 10: Sub-DAG of a task that results in a non-perfect parallelization.



(a) Graphical representation of a valid repartition of 12 tasks on 20 CPU cores and 1 GPU according to 3^{rd} constraint of the linear program of Figure 9

(b) Graphical representation of the repartition presented in Figure 11a when considering a real execution of this repartition. The 3^{rd} constraint assume the parallelization is perfect, while it is false: the real completion time is postponed. The modeling is not sufficient and should be enhanced.

Figure 11: Problematic example due to the consideration of perfect parallelization with the 3^{rd} constraint of the LP of Figure 9 with 12 identical tasks.

- $Ns_0^t = Ns_1^t = Ne_{1,CPU}^t = Ne_{1,GPU}^t = 0$
- $exT = 10$

For GPUs, the 3^{rd} constraint is satisfied, i.e. $2.5 \times 4 - 10 \leq 0$, and for CPU cores, the constraint is also satisfied, i.e. $8 \times 20 - 20 \times 10 \leq 0$. It is represented in Figure 11a. However, for CPUs, the minimal completion time of these 8 tasks is 20 milliseconds, since each task is processed by one CPU core. Therefore, the minimal completion time is the minimal completion time of one task, which is 20 milliseconds, as shown in Figure 11b.

This limitation has prompted us to introduce a minimal number of tasks for each type of PU, as outlined in the 4^{th} constraint. This constraint enforces a minimal number of tasks on a given type of PU. Since there is a minimal number of tasks, there is a minimal parallelism on each type of PU.

Note that another way to address this problem could be to consider each processing unit (PU) separately, rather than each type of PU. However, this approach would require additional constraints and variables, depending on the number of PUs, which would make the LP non viable due to its increased complexity.

4.3.2. How much the 4^{th} Constraint Takes Sub-DAG Idleness into Account

Coming back to the provided example, it is imperative to allocate a minimum of 40 tasks to CPUs (as $MinN_{cpu} = 2$), which enforces the splitting of tasks.

On the example, splitting one of the 12 tasks of type t at level 0 results in the creation of the sub-DAG, as depicted in Figure 10. The sub-DAG is constituted of 11 tasks of type t at level 1. In order to satisfy the 3^{rd} and the 4^{th} constraints, it is possible to devote 7 regular tasks to GPUs, and 5 split tasks to CPUs, with $exT = 11$. The 3^{rd} constraint is thus satisfied for GPUs, given by $2.5 \times 7 - 2 \times 11 \leq 0$. For the CPUs, it is also satisfied: $5 \times (11 \times 4) - 20 \times 11 \leq 0$. However, the minimal overall completion time for a split task executed on CPU cores correspond at least to the length of a path on the sub-DAG, that is constituted of 5 sub-tasks, with a duration of $5 \times 4 = 20$ milliseconds on CPUs, which is higher than exT. The 4^{th} constraint permits a more accurate representation, but is not perfect.

4.3.3. Improving Consideration of Sub-DAG Idleness

The LP does not take into account dependencies that are inherent to task splitting. Nevertheless, considering these dependencies directly inside the LP is not a viable option for the sake of performance. Consequently, we add to the Linear Program a manually-tuned parameter, termed $Idle_u$ for each type of PU u. This parameter represents the idleness that has been induced by task splitting on PU u, and the more the PU is supposed to need split tasks, the lower is the parameter. Therefore, the 3^{rd} equation in Figure 9 becomes :

$$\sum_{\substack{t \in \mathcal{T} \\ 0 \leq l \leq \mathcal{L}}} Ne^t_{l,u} \cdot Ex^u_{t,l} - R^u \cdot \mathbf{Idle_u} \cdot exT \leq 0 \qquad\qquad u \in \mathcal{R} \qquad (3)$$

It is mandatory to set $Idle_u \neq 0$, as the inverse renders the LP unsatisfactory. The 3^{rd} constraint becomes $\sum_{\substack{t \in \mathcal{T} \\ 0 \leq l \leq \mathcal{L}}} Ne^t_{l,u} \cdot Ex^u_{t,l} \leq 0$, implying that

$Ne^t_{l,cpu} = 0$ for each task t and level l. However, the 4^{th} constraint enforces the existence of a certain number of tasks devoted to u, meaning that it exist t and l with $Ne^t_{l,u} > 0$.

Despite the apparent oversimplification of the modeling, we will see in Section 5.1 that the selection of an adapted value has a positive impact on performance. In the forthcoming sections, we should employ this slightly modified version of Figure 9.

4.3.4. Careful Use of Ratios for Effective Task Splitting

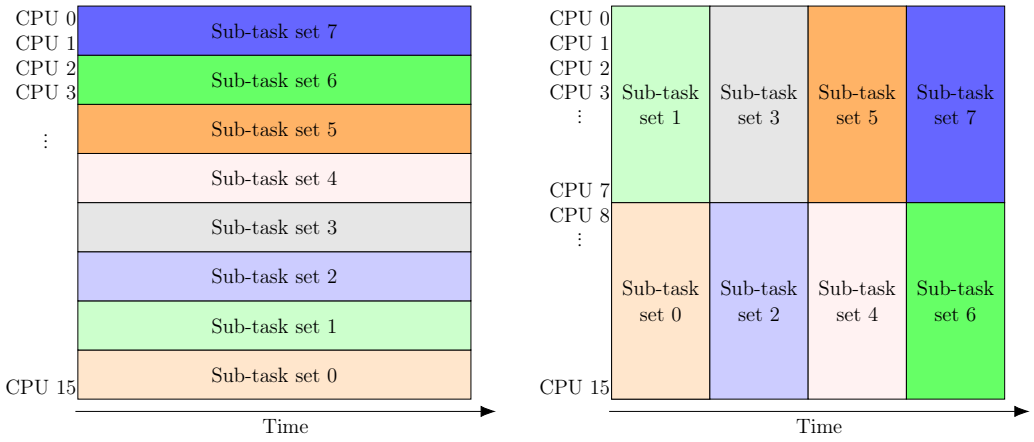
Once the splitting ratios are obtained by the LP, it is the responsibility of the Splitter to apply them. When applying these ratios, it is important to favor task completion time.

There are two typical ways to use ratios: immediate splitting and progressive splitting. In the immediate splitting approach, the Splitter divides all the required tasks at once. In the progressive splitting approach, the Splitter gradually divides tasks, splitting a few tasks at a time leading to a gradual completion of tasks and therefore a progressive release of dependencies.

For instance, let us suppose that there are 32 tasks, and that the splitting ratio for CPU usage for these tasks is 0.25, and splitting a task will create parallelism than can utilize the half of the CPU cores. Consequently, the Splitter is required to divide 8 tasks among the 32. It is possible to either split directly 8 tasks at the outset, as represented in Figure 12a, or attempt a gradual process: first splitting two tasks to utilize all the CPUs, and after their completion, split again two tasks, and so on, as represented in Figure 12b.

From the perspective of the linear program ratios, these two approaches are essentially equivalent, as the same number of tasks will eventually be split. However, from the application's perspective, there is a significant difference: progressive splitting leads to a staggered conclusion of the tasks, thereby facilitating a gradual release of dependencies. This is advantageous in terms of reducing the completion time for a given task, while the immediate approach results in a simultaneous execution of the tasks by the CPUs, with all tasks reaching completion at the same time. Therefore, the gradual approach should be implemented.

To implement this approach, the Splitter maintains a number of ready tasks for each type u of processing units, noted as Nr_u. When a regular task becomes ready, the Splitter looks at the $Ne^t_{l,u}$ variables from the LP to determine which type of processing units u the task should be planned for.



(a) Representation of the execution of 8 sub-task sets with immediate splitting methodology. Each set will be executed on one eighth of the cores for the whole execution.

(b) Representation of the execution of 8 sub-task sets with progressive splitting methodology. Each set will be executed on half of the CPU cores, and when a set is finished, a new recursive task is split.

Figure 12: Comparative scheme between immediate and progressive splitting methodology.

It can then update Nr_u.

Then, when the Splitter is about to split a task planned for a type of processing units u, if Nr_u is already higher than a threshold Th_u, the task will actually not be split.

As $MinN_u$ sets a minimal number of tasks for the processing units of type u in the LP, we set Th_u to $MinN_u \cdot R^u$.

4.4. Programmer-specified Inputs

The programmer-specified inputs for this LP are:

- The maximum recursion level $\mathcal{L}$. This parameter is determined at application submission time. It corresponds to the level where:
  - the task does not have a `gen_dag` function,
  - the task is forced not be recursive (`STARPU_IS_REC` was explicitly set to 0),
  - or the current data does not have any declared sub-data.

  This parameter is therefore determined automatically, without any further intervention from the programmers.

- The minimal number of tasks for a type of PU u, $MinN_u$. It needs to be at least 1 to ensure parallelism. Indeed, the discussion in Section 4.3.1 shows that otherwise, the modeling is unsufficient. To maintain an efficient task pipeline on GPUs, $MinN_{gpu}$ is set to 4. We set $MinN_{cpu} = 2$ to ensure a higher degree of parallelism [9].

In the next section, we present an experimental evaluation of our approach.

5. Experimental Validation

In this section, we evaluate the performance of our solution on a heterogeneous system through an experimental campaign using various dense linear algebra operations from the `Chameleon` library [17]. Our evaluation leverages `Chameleon`'s existing recursive task support, as introduced in [13], without any modifications to the code.

The experimental platform comprises two NVIDIA A100 GPUs, each with 40 GB of memory, and two AMD Zen3 EPYC 7513 CPUs, each featuring 32 cores and running at 2.6 GHz. We employed a modified version of StarPU, based on version 1.4 [10], with the *deque model data-aware ready* (`DMDAR`) scheduler. This scheduling algorithm is a variant of the `DMDA` policy [11] that favours the execution of tasks that have already their data ready for use. We use this variant because it was the most efficient in terms of performance for our evaluation. The code was compiled using GCC 11.4 and CUDA 11.7. Additionally, the GLPK linear programming solver was used to solve the associated Linear Program. Each experiment was executed five times, and we report the average results across these runs.

The experiments were conducted on a variety of naturally recursive kernels, including Cholesky factorization (`POTRF`), LU factorization without pivoting (`GETRF_nopiv`), symmetric matrix inversion using a pre-factored matrix (`POTRI`), symmetric linear system solution (`POSV`), and a sequence of operations combining `POTRF` and `POTRI`, referred to as `POINV`. These graphs

[9] An experimental study has shown us that for large computation, the attributed values have not an important impact on performance, provided that the values are higher than 1.

[10] commit `96355dd8acbbadfd2046ad72a8029085e5aba877`

[11] The `DMDA` policy is an HEFT-like scheduling strategy

have notable difference in terms of graph and dependencies. While `POTRF` is a factorization combining four different task types, it has the advantage of creating an important parallelism with a memory usage that is limited. On the other hand, the `GETRF_nopiv` graph is similar to the `POTRF` graph, but with much more parallelism and an increased memory usage. Obtaining performance on this graph is tedious because of the potential created memory transfers if no care is taken. The `POSV` graph combines a Cholesky factorization with a triangular solve, and can therefore be seen as an extension of the `POTRF` graph. The `POTRI` graph has a lot of anti-dependencies (Write-After-Read dependencies), unlike other graphs, which confers to the task graph a really different structure. The `POINV` graph being a composition of a Cholesky factorization, a triangular matrix inversion and a matrix product (enclenched into each other through fine-grain dependencies), it results in a complex graph involving 7 different task types.

We compare the non-recursive versions of these kernels with their splitter-recursive counterparts, highlighting the benefits of the latter. In the recursive versions, we use the notation $x_0/x_1/.../x_k$, where the initial matrix is partitioned into tiles size x_0, and splitting tasks with tile size x_i generates subtasks with tile size x_{i+1}. A detailed analysis is provided for both Cholesky and LU factorizations, including a comparison with state-of-the-art methods.

Before presenting the experimental results, we first introduce the manually-tuned parameters used in our study.

5.1. Determining the Best Parameters for the Linear Program

From the users, the LP only requires the following manually-tuned parameters:

- $MinN_u$: the minimum number of tasks required for workers of type u.
- The sampling frequency.
- $\mathcal{L}$: the maximum level of recursion for a task.

Additionally, as discussed in Section 4.3.3 accounting for idleness caused by task splitting is crucial. To address this, we introduce a new parameter, $Idle_u$, which requires experimental calibration to optimize performance.

All other parameters are automatically determined using `StarPU`'s performance models, which are calibrated with a few initial application runs. We

have already determined a suitable value for $MinN_u$, based on the observation that allocating a higher number of tasks to GPUs improves performance. The parameter $\mathcal{L}$ is directly given by the submission process, as explained earlier.

We begin by determining the sampling frequency. To simplify the process, we consider only one type of application. Specifically, we conduct multiple Cholesky factorizations on a matrix size 92160, each with different sampling frequencies.

After performing a set of experiments, we determined to solve the LP at 50-tasks intervals (This accounts only tasks at the top level). This choice is based on our observation that solving the LP occupies a CPU core for an amount of time lasting from 100 microseconds to 2 milliseconds during experiments. Additionally, the set of available tasks evolves gradually rather than undergoing drastic changes between intervals. Notably, minor adjustments to this interval (e.g varying it between 25 and 100) do not significantly impact performance.

Next, we determine a suitable value for $Idle_u$.

Regarding $Idle_{gpu}$, it is important to note that this parameter accounts only for idleness caused by task splitting. In our experiments, we assume that the largest granularity (x_0) is the most suitable for GPUs, minimizing task splitting. Therefore, we set $Idle_{gpu}$ to 1.

Figure 13 compares the performance of recursive Cholesky Factorization with fixed granularities and different values of $Idle_{cpu}$. Our results show that setting $Idle_{cpu}$ to 0.8 yields the best performance, providing an average 5% improvement compared to setting $Idle_{cpu}$ to 1.

Based on these results, we will use a sampling frequency of 50 tasks, $Idle_{gpu} = 1$, and $Idle_{cpu} = 0.8$ in the following sections.

5.2. Dense Linear Algebra Results

Figure 14 presents a comparison of the performance of recursive (solid curve) and non-recursive (dotted curves) versions of different dense linear algebra kernels. The tile sizes were selected based on the performance models and following an analysis of the best configuration for each type of execution.

Across all cases, regardless of matrix size or kernel type, recursive tasks consistently outperform non-recursive tasks, even for large matrix sizes. The algorithm enhances performance without requiring per-application tuning, demonstrating its versatility across different applications and levels of parallelism. For matrices size less than 23040, the performance gains are par-

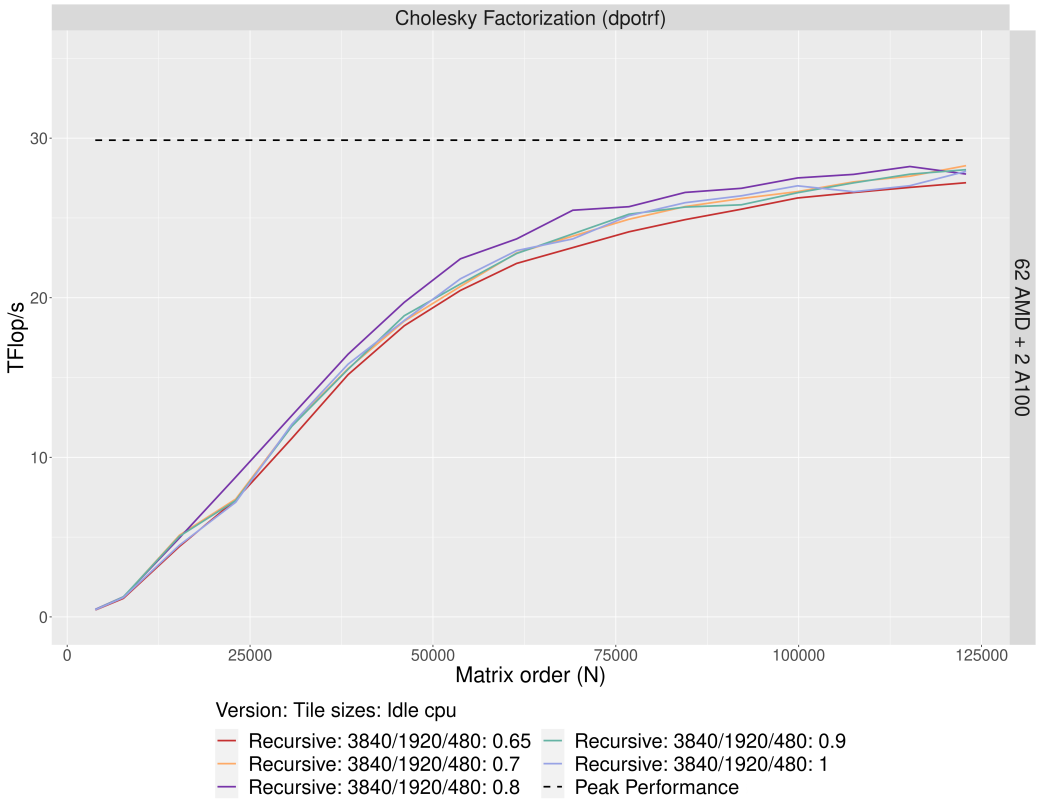


Figure 13: Performance comparison of Cholesky factorization with dynamic recursive tasks for different values of $Idle_{cpu}$. Peak Performance is computed by taking the peak performance for each type of processing units for a `GEMM` task, multiplying by the number of PU of this type, and summing the results.

ticularly significant. The recursive version consistently surpasses the best non-recursive version, achieving up to 100% for `GETRF_nopiv` and 20% for `POINV`. This is due to an enhanced utilization of the processing units. For such small matrices, the primary performance bottleneck arises from insufficient parallelism, as there are not enough tasks to fully utilize all the PUs, particularly the GPUs. Recursive tasks help alleviate this issue by enhancing mid-grained parallelism, ensuring that tasks remain efficient for GPUs while also enabling coarse-grained parallelism when sufficient workload exists to utilize all GPUs effectively. As a result, recursive tasks lead to higher GPU utilization compared to coarse-grained non-recursive methods and improve GPU efficiency compared to mid-grained non-recursive approaches. Addi-

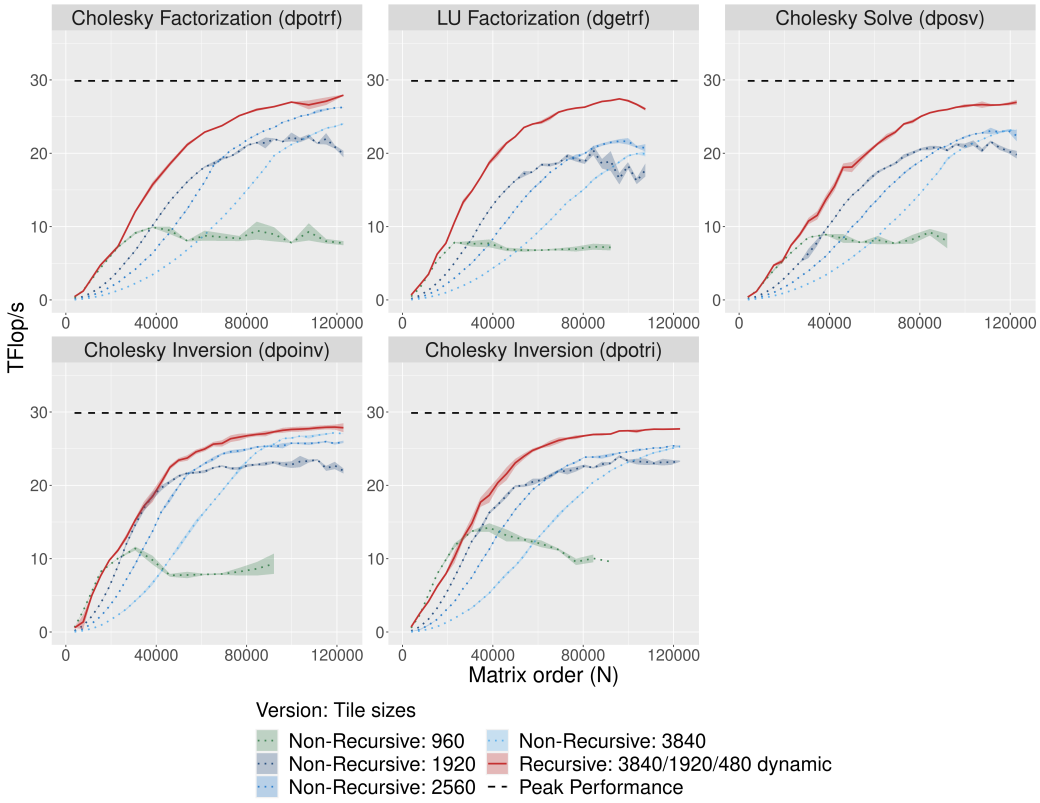


Figure 14: Performance comparison for `POTRF` (up-left), `GETRF_nopiv` (up-center), `POSV` (up-right), `POINV` (bottom-left) and `POTRI` (bottom-right) kernels between non-recursive (dotted curves) and a splitter-recursive (solid curve) versions.

tionally, certain operations, such as `POTRF` and `GETRF_nopiv`, lack GPU implementations for specific tasks on the critical path, namely the `POTRF` and `GETRF_nopiv` kernels. By splitting these tasks, recursive execution accelerates the critical path processing, allowing portion of these operations to be offloaded to GPUs. This approach maintains optimal GPU efficiency without forcing a small tile size for other tasks, thereby balancing performance and resource utilization.

For larger problems (*e.g.* 107520), performance improvements are observed across various operations: 5% for `POINV`, 10% for `POSV` and `POTRF`, and 30% for `GETRF_nopiv`. These gains stem from the ability of recursive tasks to employ more efficient tasks without significantly delaying the critical path.

Interestingly, the non-recursive version with a tile size 3840 is generally slower than the non-recursive version with a tile size 1920, except for large matrices. This is counter-intuitive, as the GPU kernel performance is typically higher for a tile size 3840 than 1920. This discrepancy suggests that using larger tiles postpones critical path tasks, leading to GPU idle periods and reduced overall efficiency.

In contrast, the recursive version with tile sizes 3840 / 1920 / 480 shows improved performance for large matrix sizes. This suggests that the critical path is not as significantly delayed as in the non-recursive case. Since recursive tasks enable better parallelization and acceleration, tasks on the critical path become less impactful, thanks to the increased parallelism. As a result, large tile sizes can be effectively utilized on GPUs, leading to faster execution.

We acknowledge that our evaluation focuses on a subset of dense linear algebra kernels, which does not reflect the full range of scientific workloads. However, the selected kernels exhibits different types of task graphs, parallelism pattern and dependence schemes, and our results demonstrate consistent performance improvements accross them.

5.3. Dense Cholesky Factorization: an Exhaustive Study Case

Figure 15 presents a comparative experimental analysis of the Cholesky factorization between the better splitter-recursive version (solid red curve) and various state-of-the-art implementations of the Cholesky factorization:

- A static recursive version that splits tasks among all the diagonal (solid brown curve, referred to as static recursive). In the case of the Cholesky factorization, the kernel type employed for tasks on the diagonal is either the `POTRF` or the `SYRK` kernel. As the `POTRF` kernels cannot be executed on GPUs, they represents a bottleneck in the execution process and are therefore considered to be on the critical path.

- A non-recursive version utilizing parallel workers from `StarPU` [29] (dotted-dashed light-blue curve).

- The hierarchical version from `PaRSEC` [12] (dashed light-green curve). To the best of our knowledge, it is the only RTS that allows a heterogeneous recursivity for dense linear algebra, with the dplasma library.

- A non-recursive version (dotted navy-blue curve).

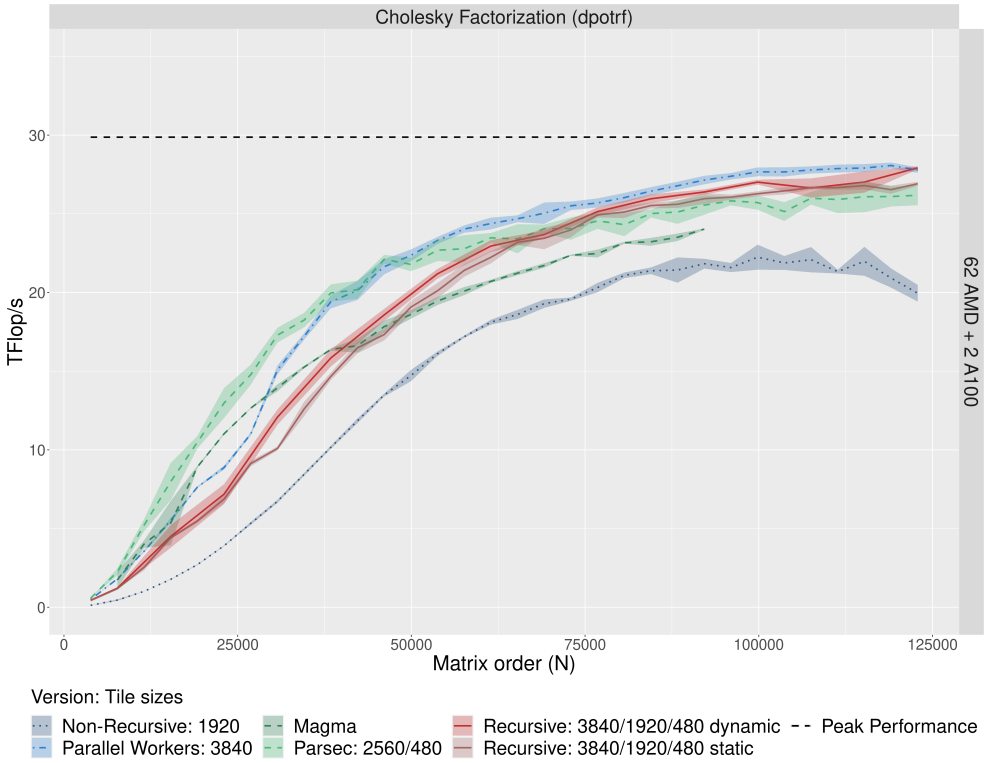


Figure 15: Performance of Cholesky factorization with dynamic recursive tasks (solid curve), static recursive tasks (long-dashed curve), parallel workers (dashed curve), `PaRSEC` (dash-dotted curve) and non-recursive (dotted curve).

- A non-recursive version from the `MAGMA` [39] dense linear algebra library (dotted green curve). As it requires the whole matrix to fit on one GPU, `MAGMA` was considered for matrix sizes lower than 92160.

It can be observed that the splitter-recursive version is competitive with all other state-of-the-art versions. This version exhibits asymptotic performance characteristics that are approximately equivalent to those observed in the parallel-workers version. This demonstrates how the Splitter can efficiently optimize the usage of the PUs.

In terms of peak performance, a speed up of 4% is observed with respect to the critical-path recursive version, and 6% with respect to `PaRSEC`. The enhanced performance in comparison with the critical-path version can be attributed to the overestimation of the number of split tasks in the static case. In contrast to the static version, which splits 9% of the tasks, the

dynamic Splitter splits 4% of the large granularity tasks. The overspliting of tasks results in a reduction of their overall efficiency. In comparison to `PaRSEC`, the improvement can be attributed to a optimized utilization of the GPUs. The best configuration for `PaRSEC` is obtained with tiles size 2560. However, the kernel performance for this tile size is inferior to the tile size 3840. Consequently, utilizing the latter tile size, as in the splitter-recursive version, results in a most efficient utilization of the GPUs.

For small matrix sizes, the dynamic recursive version remains competitive, though its performance is slightly inferior to the other versions. Specifically, it under-performs compared to the `PaRSEC` version, as `PaRSEC` allows the execution of `POTRF` tasks on GPUs, a feature not supported in `Chameleon`. Additionally, since the largest granularity used by `PaRSEC` is smaller than that of the splitter-recursive variant, `PaRSEC` can exploit greater parallelism for smaller matrices. This results in better GPU utilization and improved overall performance.

When using the largest granularity, the mean solving time for the linear program is 0.3 milliseconds. In comparison, a matrix-multiplication task for squared tiles size 3840, that is the most common type of task during the execution, takes 7.2 milliseconds to be executed on a Nvidia A-100 GPU. Also, the splitter-recursive Cholesky factorization for matrix size 49920 takes a mean-time of 1.986 seconds to execute the few thousands tasks. For matrix size 92160, it takes a mean of 9.7132 seconds, and for matrix size 122880, it takes a mean of 21.981 seconds. These numbers shows that the overhead introduced by the LP has a limited impact: it occupies one CPU core during less than a millisecond, each time several tasks of several milliseconds are executed, during an execution of several seconds. A detailed analysis of the induced overhead will be discussed in Section 5.5

To obtain a better understanding of the performance improvements with recursive tasks compared to non-recursive version, we present in Figure 16 an execution trace of a Cholesky Factorization for a large matrix (92160) with dynamic Splitter (3840/1920/480), compared to an execution trace representing the same Cholesky Factorization without recursive tasks (1920). Also, Table 3 compiles different statistics for the execution corresponding to the execution trace.

It is therefore notable that the Splitter has determined that 4% of the tasks at level 0, and 24% of the tasks at level 1 should be turned into sub-DAGs. Furthermore, 100% of the tasks at level 0, 90% of the tasks at level 1, and 17% of the tasks at level 2 were executed on GPUs. The discrepancy in

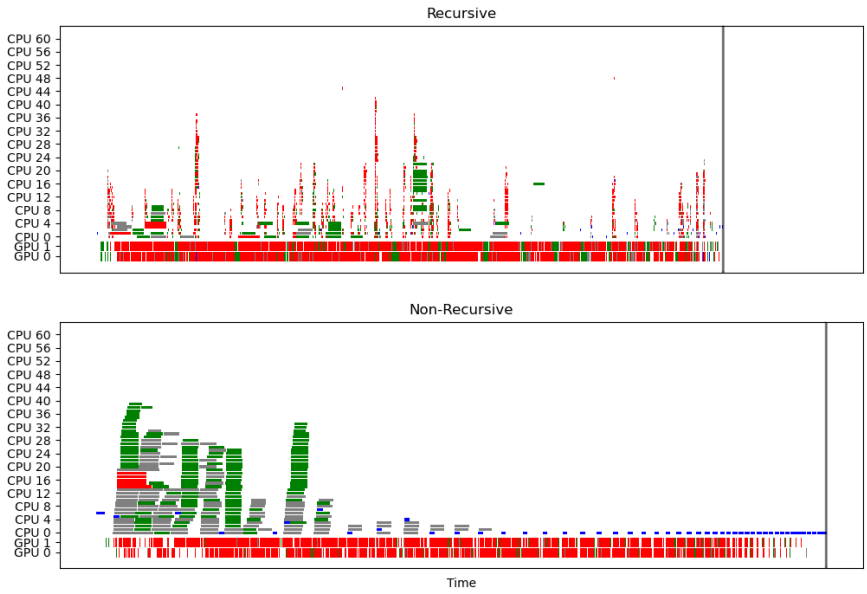


Figure 16: Comparison of execution trace for a Cholesky Factorization (92160) with recursive tasks (3840/1920/480, up) and non-recursive tasks (1920, down) and recursive tasks. `POTRF` tasks are in blue, `SYRK` tasks in gray, `TRSM` tasks in green and `GEMM` tasks in red. End of execution is represented by black vertical line.

performance between the non-recursive and recursive versions is attributable to a difference in worker idleness. In the non-recursive version, CPUs are active for 10% of the execution time, while GPUs remain idle for 28% of the time. By using recursive tasks, GPU idleness is reduced to just 8%, while CPUs idleness increases to 95%. The GPU idleness in the non-recursive case is primarily due to data fetching at the start of the execution and a lack of tasks for GPUs at the end (corresponding to the blue tasks). In contrast, in the recursive case, this lack of task is mitigated. The `POTRF` tasks, which must be executed on CPUs in the non-recursive case, are split in the recursive case, creating additional tasks that can be offloaded to GPUs. As a result, the remaining `POTRF` are executed more quickly. The performance improvement can therefore be attributed to two key factors:

- GPU efficiency: GPUs are able to execute high-efficiency tasks without compromising CPU performance.

	Recursive	Non-Recursive
Part split level 0	4 %	-
Part split level 1	24 %	-
Part executed on GPU level 0	100 %	98 %
Part executed on CPU level 0	0 %	2 %
Part executed on GPU level 1	90 %	-
Part executed on CPU level 1	10%	-
Part executed on GPU level 2	17 %	-
Part executed on CPU level 2	83 %	-
General occupancy of GPU	92 %	72 %
General occupancy of CPU	5 %	10 %

Table 3: Comparison of several statistics for a Cholesky Factorization (92160) with recursive tasks (3840/1920/480, up) and non-recursive tasks (1920, down) and recursive tasks. The executions correspond to the ones presented in Figure 16

- Critical path speedup: using recursive tasks reduces GPU idle time leading to a speedup of the critical path

However, these numbers also highlight the potential for improving CPU utilization to further enhance performance.

5.4. Dense LU Factorization: an Exhaustive Study Case

Figure 17 illustrates the performance of the LU factorization without pivoting (`GETRF_nopiv`) for varying matrix sizes and versions.

The recursive-splitter version (solid red curve) is generally the fastest one, offering performance equivalent to the recursive-static version (solid brown curve), and a 30% enhancement in comparison to the non-recursive version (dotted navy-blue curve). The parallel-workers version (dash-dotted light -blue curve) is, in this case, not a competitive option, as the MKL does not provide a `GETRF_nopiv` kernel with a parallel implementation. The `PaRSEC` version of the LU factorization (dashed light-green curve) is not a competitive option, given that the hierarchical version is not available for the LU factorization. Furthermore, it was not feasible to execute the LU factorization with `PaRSEC` for matrix size exceeding 99840, due to the excessive execution time. This illustrates the value of automating the task splitting process of

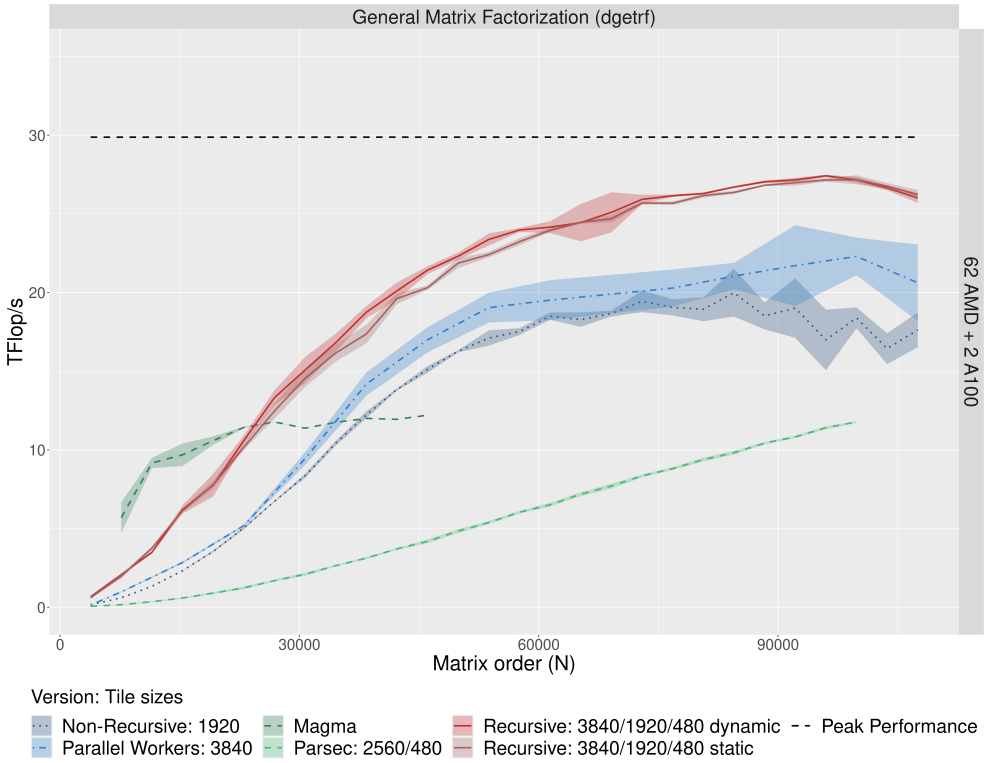


Figure 17: Performance of LU factorization without pivoting with dynamic recursive tasks, static recursive tasks, non-recursive tasks, parallel workers, Magma and PaRSEC.

our approach, rather than requiring manual specification for each application as used in PaRSEC. After defining a recursive expression for each LU factorization kernel (which is straightforward given that this kernel is typically already expressed recursively), we can simply apply the same algorithm used for the Cholesky factorization without any modification.

A notable decline in performance is observed when handling large matrix sizes with recursive tasks in LU factorization. This occurs when the total data volume exceeds the combined capacity of the two GPUs. This issue stems from the scheduler. DMDAR is subject to a conflict between data prefetching and eviction. Once the GPU is filled, DMDAR prioritizes executing tasks that utilize the current data set, thereby halting the prefetching of additional data. If no task can be executed with the available data, eviction occurs. However, already-scheduled tasks are not being reconsidered for scheduling, even if a change could improve performance. Since DMDAR lacks a global view

of all tasks and data, it cannot effectively balance prefetching and eviction. As a result, the volume of data transfers increases significantly. For matrices smaller than 80 GB (with each GPU having 40 GB of memory), the data transfer to matrix size ratio remains around 6, corresponding to a maximum matrix size 92160. As the matrix size increases, this ratio rises exponentially, reaching 10 (respectively 18) for a matrix size 90 (respectively 120) GB, which represents a matrix size 107520 (respectively 122880). A memory-constraint-aware scheduler such as `DARTS` [40] would help resolve this issue, highlighting the advantage of separating the Splitter from the scheduler.

5.5. Evaluating the cost of the Splitter

In previous experiments, we presented applications with a number of different task types ranging from 4 to 7, and 2 resource types. In this section, we would like to evaluate if the approach remains relevant, by analyzing the additional costs introduced by the Splitter component inside `StarPU`. Previous work [13] has already quantified the cost of using recursive tasks - showing about a 10% overhead on task submission and data management in a scenario similar to ours.

However, the Splitter can induce two additional types of overhead.

1. The cost of the splitting decision. For each recursive task, a function is invoked to check the current splitting ratios, as determined by the Linear Program, and to decide whether the recursive task should be split.
2. The cost of computing the splitting ratios. These ratios are periodically recomputed by solving the described Linear Program, which occupies one CPU core during the resolution.

For the splitting decision cost, we conducted an experiment comparing a fully-recursive version with a fully-recursive without splitting version. In the first case, each task is submitted at coarse granularity (3840∗3840)), and then split to reach computational granularity ($1920x1920$). In the second case, each task is directly submitted at computational granularity ($1920 * 1920$), and can be split into smaller tasks ($480 * 480$) but is not, with a splitter invocation. We compare both to a baseline version without splitting, where each task is already at computational granularity and cannot be split. We observed that the fully-recursive without splitting version showed no performance difference compared to the non-recursive baseline. In contrast, the

fully-recursive version with active splitting exhibited a performance decrease of about 2 %, regardless of the problem size.

5.5.1. Overhead from the Linear Program

The time it takes to solve the LP depends mainly on three factors: the number of types of task, the number of types of resource and the number of recursion levels. Importantly, the total number of tasks of the same type or the number of resources does not increase the LP's complexity. These only change the bounds of the constraints, not the number of constraints.

To measure how these factors affect solving time, we created LP instances based on a real execution. Missing values (e.g. the execution time of a task that does not exist) were generated by mixing random numbers with real parameters to keep things realistic. For instance, this methods keeps the rule that, for a given resource and level, task completion times of two different tasks do not differ by more than a factor 10. For each configuration, we ran 650 random solves, and averaged the time, shown in Figure 18.

In most cases, the LP solution time is under 10 milliseconds - faster than a typical StarPU task execution. Solving time grows noticeably when the number of task types goes over 48 and the number of level exceeds the number of resource types. For example, with 4 different types of resources and 6 recursion levels, increasing the number of task types from 32 to 64 multiplies the solving time by 3.8. But in real platforms, there are usually no more than three types of resources, and typically one granularity level for each resource type is sufficient. In such configuration, the solving time is usually a few milliseconds.

Some other points help limit the impact even more:

- The LP runs on a single CPU core, so it does not slow down the whole system.
- Even when it takes a long time, it does not block other workers because in that case the Splitter continues using the previous ratios.
- LP calls happen based on top-level tasks, not on the splitting scheme, so they do not happen too often even if splitting ratios are high.

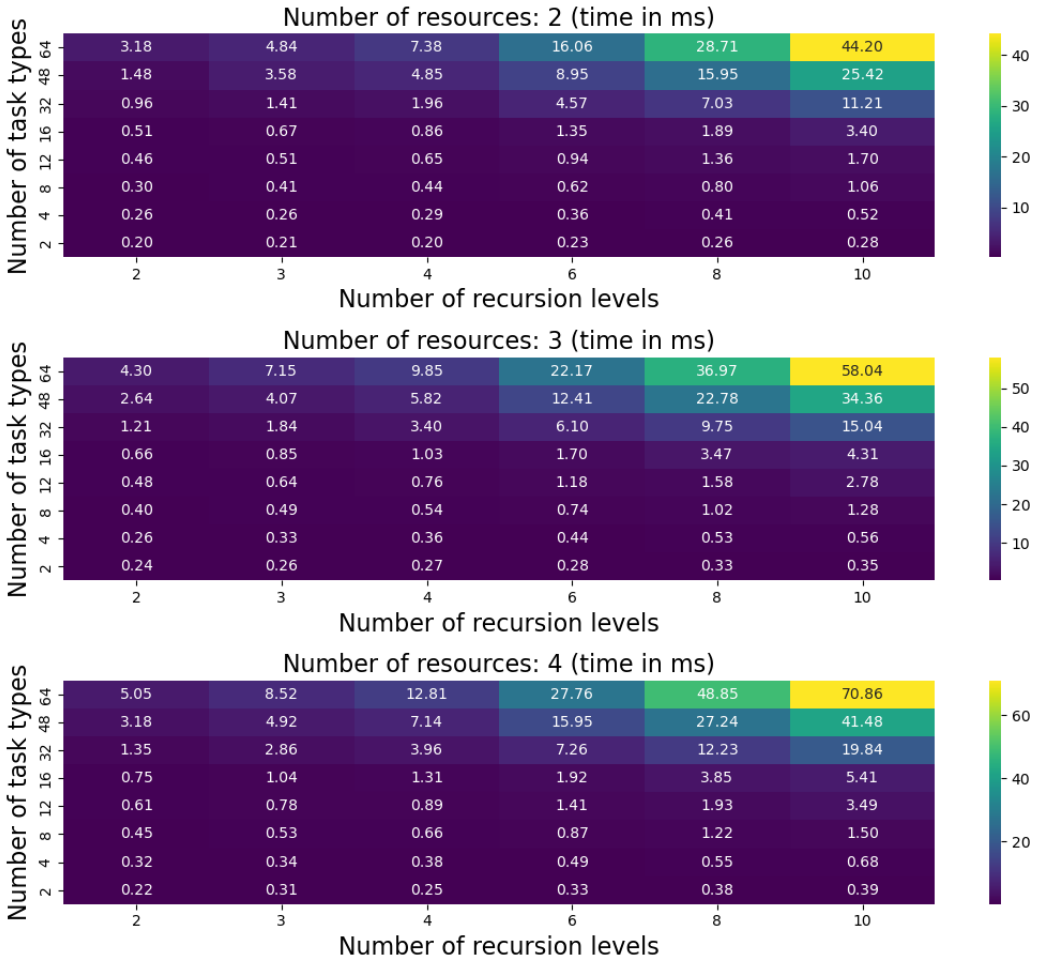


Figure 18: Completion time of the Linear Program for different number of task types, levels and resource types.

6. Related Work

6.1. Scheduling heuristics for Runtime Systems

Scheduling strategies in this context can either be application-driven like in Legion [3], resource-centric or task-centric. Resource-centric schedulers take decision whenever a resource is getting idle. Examples of such strategies are work-stealing or priority-based approaches. They are generally more adapted to homogeneous systems even if some contributions like Hetero-Prio [15, 41] try to target heterogeneous systems. Resource-centric schedulers take decision when a task has all its dependencies fulfilled. They are often inspired by the Heterogeneous Earliest Finish Time (HEFT) algorithm [36], a widely-used reference for task scheduling in such contexts. Most heterogeneous runtime systems IRIS [42], OmpSs [43], StarPU [28] and XKaapi [44] can leverage performance models to implement HEFT-like scheduling strategies. Importantly, these runtimes are not tied to a specific scheduling policy, the scheduler is implemented as a modular and interchangeable component. A theoretical and experimental analysis of various scheduling strategies on heterogeneous platforms with two resource types is provided in [37].

6.2. Granularity Challenges within Runtime Systems for homogeneous platforms

Selecting the appropriate granularity involves balancing efficiency and parallelism amount. To support this trade-off in practice, automatic granularity control mechanisms help mitigate the overhead associated with fine-grained task creation. The proposed strategies either submit tasks at the finest level and try to coarsen parallelism to achieve the trade-off, or submit tasks at the coarsest grain and split tasks progressively to reach the target amount of parallelism. For coarsening approaches, the key idea is to let programmers expose as much parallelism as possible, while compilers or runtimes decide how to coarsen tasks to reduce overhead and optimize performance. In parallel programs with balanced for-loops, the optimal grain size can often be determined analytically [45]. For irregular or unbalanced loops, work-stealing techniques such as Lazy Binary Splitting [46] have shown to be effective. On the other hand, in divide-and-conquer paradigms, granularity control typically relies on cut-off strategies, where recursive task creation is halted once sub-problems become sufficiently small or when the available parallelism is deemed adequate. Such strategies have been successfully implemented in OpenMP [47, 48], leading to the introduction of standard

constructs like the `if`, `final`, and `mergeable` clauses, which aim to reduce tasking overhead and improve granularity control. These approaches have been further refined in several RTS designed for homogeneous systems, including MassiveThreads [49], an extension of OpenMP [50], an extension of Cilk [51] and HPX [52] which incorporate static analysis, runtime-based or cost-model-based decisions. However, as noted in [53], OpenMP still lacks mechanisms to effectively prioritize the critical path – features that are partially available in OMPSs [43] for instance. With respect to these efforts, our work follows the divide-and-conquer approach while targeting heterogeneous systems where the memory coherency needs to be guaranteed by the RTS.

6.3. Granularity Challenges within Runtime Systems for heterogeneous platforms

Concerning the granularity problem faced when exploiting heterogeneous systems equipped with accelerator devices like GPUs, a proposal has been made to allow the RTS to execute a task in a multithreaded manner on a parallel worker, which can be defined as an aggregation of multiple CPU cores, as demonstrated in `StarPU` [29]. Aggregating CPU cores into a parallel worker makes them competitive with GPUs for coarse-grained tasks. With this approach, the RTS only sees one task granularity, a large one that will either occupy a GPU, or a parallel worker. To execute a task on a parallel worker, another runtime, such as Intel MKL, is called to execute the task in parallel.

Nevertheless, this approach has its drawbacks. Since parallel workers are statically defined, CPU cores may be underutilized, particularly when a task lacks sufficient parallelism.

`PaRSEC` proposes an alternative approach, by enabling the use of hierarchical DAGs to achieve high performance on hybrid distributed systems [12]. When a large-grain task is scheduled on a CPU, `PaRSEC` decomposes the workload into finer-grained parallelism. This approach is similar to the previous method, except that the RTS directly manages the different granularities.

Recent extensions of the Legion runtime explore a complementary strategy: task fusion, performed via just-in-time compilation. This technique merges tasks that operate on shared data regions in order to increase granularity and reduce runtime overhead, thereby improving overall execution efficiency [54].

6.4. Just-in-Time Task Graph Adaptation

When the application workload is irregular, meaning the amount of work varies throughout execution, recursive parallelism and just-in-time DAG adaptation naturally address this irregularity problem. When the overall workload is small, breaking it into smaller tasks is effective, as the efficiency loss from fragmentation is compensated by an increased use of processing units, reducing the overall completion time. Conversely, as the workload increases, using larger tasks improves efficiency without compromising resource utilization. Thus, expressing tasks recursively allows the execution to dynamically prioritize either efficiency or parallelism as needed.

Some contributions, mostly at the programmer level, without any specific support for recursive tasks from the RTS, have considered the use of recursive tasks for irregular applications. For example, in the context of sparse QR factorization [10], the recursive expression of the DAG has been efficiently used either to control the memory footprint of the application [55] or to exploit different granularities to efficiently utilize GPU devices [56]. This has also been the case when considering $\mathcal{H}$-matrix computation which can naturally be expressed recursively [11]. Moreover, the highly irregular nature of the computation when using $\mathcal{H}$-matrices requires online adaptation of the DAG in order to feed all computational resources.

To adapt to the dynamic nature of the graph, RTS can provide features that allow tasks to transparently become sub-graphs at execution time. Therefore, turning a task into a sub-graph will allow to occupy all the CPU cores with the sub-tasks while the GPU core will execute the large tasks. StarSs [14] introduces weak dependencies to establish fine-grained relationships between tasks and sub-graphs, and has been successfully applied to $\mathcal{H}$-LU factorizations [23]. However, it lacks support for GPU acceleration in the context of weak dependencies.

6.5. Discussion

We discuss how our approach can generalize to other runtime systems. Our work relies on four key features, namely recursive tasks and runtime splitting decision, performance models, and advanced data management. In order to extend our work to runtime systems other than `StarPU`, one need to adapt each of the features as sketched below.

Recursive tasks. Our work relies on recursive tasks, or nested parallelism, which must be supported by the RTS. This is already the case in IRIS, Legion,

PaRSEC and StarSs. Nested parallelism is also available in the OpenMP standard, with sub-DAGs submission within tasks, and nested loops.

Runtime decision to turn a task into sub-DAG. The key point is that the splitting decision should be made by the runtime. As we have seen, such mechanisms have been explored in several runtime systems in homogeneous settings, including OpenMP [47], HPX [52] as well as extensions of extensions of OpenMP[50] and Cilk[51].

Completion time model. The LP model we present requires estimated completion times per task and processing unit, or at least speedup ratios. Since this requirement exists for each runtime that apply HEFT-like strategies, such information is already recorded on IRIS, OmpSs and XKaapi.

Data management. To apply our approach, the runtime system must provide advanced support for data partitioning and tracking. Among other existing RTS, only Legion offers such capabilities natively, through its *logical region* abstraction.

7. Conclusion

In this paper, we proposed to dynamically and automatically choose suitable granularities of tasks, by using recursive tasks, which can be split into a sub-DAG of tasks at runtime. We introduced a heuristic for determining which tasks should be split into sub-DAGs, according to different criteria, such as the amount of parallelism available in the Runtime System, the efficiency of the tasks, and the task resource affinity. This is done through the online solution of a linear program which computes task splitting ratios. We then apply these obtained ratios during execution. This produces adequate sets of tasks of different granularities. A typical result is that most tasks are kept at a large granularity for efficient execution on GPUs, some tasks are split into sub-DAGs to increase available parallelism, and some of these are split even further to load CPUs carefully.

On different dense linear algebra operations, our automatic approach shows a performance improvement up to 30% over the asymptotic performance, compared to single-granularity approach (widely used in dense linear algebra libraries). Furthermore, this is on par with state-of-the-art programmer-driven multi-granularity approaches, in the cases where those

were implemented. Our approach is however fully automatic, allowing programmers to obtain performance in heterogeneous achitectures in a portable way.

7.1. Future Work

Our ongoing efforts aim firstly to study synergy between recursive tasks and parallel workers [29]. These two methodologies are two complementary approaches for reaching the same goal. On the one hand, the recursive task approach decreases task granularity to make it possible to execute a task on slow processing units. On the other hand, the parallel worker approach increases worker size to lower the heterogeneity of the platform. Studying the benefits of a combination of these two approaches, for example with an architecture presenting several types of GPUs, is interesting.

Furthermore, extending the recursive tasks to the distributed memory case may help mitigate STF overhead by allowing automatic task pruning, and also facilitating load-balancing on different nodes, thanks to a global hierarchical view. Additionally, the recursive expression of the code can be generated by polyhedral compilation techniques, which can automatically gather tasks. This would allow at the compilation stage, to automatically choose the suitable sub-DAG structure and generate the code that submits it, similarly to automatic tiling generation [57].

Finally, it would also be of interest to extend our study to more irregular applications, such as sparse linear algebra. Since they expose very different tile sizes instead of a finite set of tile sizes, this would require extending the Linear Program to use algebraic performance models for tasks instead of discrete models, and abstract the notions of levels.

7.2. Acknowledgments

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] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, I. Stoica, Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing, in: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, NSDI'12, USENIX Association, USA, 2012, p. 2.

[2] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, A. Lerer, Automatic differentiation in pytorch, in: NIPS 2017 Workshop on Autodiff, 2017.
URL `https://openreview.net/forum?id=BJJsrmfCZ`

[3] M. Bauer, S. Treichler, E. Slaughter, A. Aiken, Legion: Expressing locality and independence with logical regions, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2012.

[4] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, J. J. Dongarra, PaRSEC: Exploiting Heterogeneity to Enhance Scalability, Computing in Science & Engineering 15 (6) (2013) 36–45.

[5] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures, in: Euro-Par Parallel Processing, 2009.

[6] J. M. Perez, R. M. Badia, J. Labarta, A dependency-aware task-based programming environment for multi-core architectures, in: 2008 IEEE International Conference on Cluster Computing, 2008.

[7] T. Gautier, J. V. Lima, N. Maillard, 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. `doi:10.1109/IPDPS.2013.66`.

[8] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, S. Tomov, A Hybridization Methodology for High-Performance Linear Algebra Software for GPUs, in: W. mei W. Hwu (Ed.), GPU Computing Gems Jade Edition, Applications of GPU Computing Series, Morgan Kaufmann, Boston, 2012.

[9] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, J. Dongarra, Flexible development of dense linear algebra algorithms on massively parallel architectures with dplasma, in: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, 2011.

[10] E. Agullo, A. Buttari, A. Guermouche, F. Lopez, Implementing multifrontal sparse solvers for multicore architectures with Sequential Task Flow runtime systems, ACM Transactions on Mathematical Software (Jul. 2016).

[11] B. Lizé, Résolution directe rapide pour les éléments finis de frontière en électromagnétisme et acoustique : $\mathcal{H}$-Matrices. Parallélisme et applications industrielles, Theses, Université Paris-Nord - Paris XIII (Jun. 2014).

[12] W. Wu, A. Bouteiller, G. Bosilca, M. Faverge, J. Dongarra, Hierarchical DAG scheduling for Hybrid Distributed Systems, in: IPDPS, IEEE, IEEE, India, 2015.

[13] M. Faverge, N. Furmento, A. Guermouche, G. Lucas, R. Namyst, S. Thibault, P.-A. Wacrenier, Programming Heterogeneous Architectures Using Hierarchical Tasks, Concurrency and Computation: Practice and Experience 35 (25) (2023).

[14] J. M. Perez, V. Beltran, J. Labarta, E. Ayguadé, Improving the Integration of Task Nesting and Dependencies in OpenMP, in: IPDPS, 2017.

[15] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, T. Takahashi, Task-based fmm for heterogeneous architectures, Concurrency and Computation: Practice and Experience 28 (9) (2016) 2608–2629. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.3723, doi:https://doi.org/10.1002/cpe.3723.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3723

[16] N. Furmento, A. Guermouche, G. Lucas, T. Morin, S. Thibault, P.-A. Wacrenier, Optimizing parallel system efficiency: Dynamic task graph

adaptation with recursive tasks, in: P. Diehl, J. Schuchart, P. Valero-Lara, G. Bosilca (Eds.), Asynchronous Many-Task Systems and Applications, Springer Nature Switzerland, Cham, 2024, pp. 166–172.

[17] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, S. Tomov, Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs, in: W. mei W. Hwu (Ed.), GPU Computing Gems, Vol. 2, Morgan Kaufmann, 2010.

[18] M. Frigo, C. E. Leiserson, K. H. Randall, The implementation of the Cilk-5 multithreaded language, in: Conference on Programming Language Design and Implementation, PLDI '98, 1998.

[19] J. Reinders, Intel threading building blocks - outfitting C++ for multi-core processor parallelism., O'Reilly, 2007.

[20] T. D. Hartley, E. Saule, Ümit V. Çatalyürek, Improving performance of adaptive component-based dataflow middleware, Parallel Computing 38 (6) (2012) 289–309.

[21] K. H. Tsoi, A. H. Tse, P. Pietzuch, W. Luk, Programming framework for clusters with heterogeneous accelerators, SIGARCH Comput. Archit. News 38 (4) (2011) 53–59.

[22] H. Kaiser, P. Diehl, A. S. Lemoine, B. A. Lelbach, P. Amini, A. Berge, J. Biddiscombe, S. R. Brandt, N. Gupta, T. Heller, K. Huck, Z. Khatami, A. Kheirkhahan, A. Reverdell, S. Shirzad, M. Simberg, B. Wagle, W. Wei, T. Zhang, HPX - The C++ Standard Library for Parallelism and Concurrency, Journal of Open Source Software 5 (53) (2020) 2352.

[23] R. Carratalá-Sáez, S. Christophersen, J. I. Aliaga, V. Beltran, S. Börm, E. S. Quintana-Ortí, Exploiting nested task-parallelism in the h-lu factorization, Journal of Computational Science 33 (2019) 20–33.

[24] M. Cosnard, M. Loi, Automatic task graph generation techniques, in: System Sciences, 1995. Proceedings of the Twenty-Eighth Hawaii International Conference on, Vol. 2, IEEE, 1995, pp. 113–122.

[25] R. Allen, K. Kennedy, Optimizing Compilers for Modern Architectures: A Dependence-Based Approach, MORGAN KAUFMANN PUBL Incorporated, 2001.

[26] C. Augonnet, J. Clet-Ortega, S. Thibault, R. Namyst, Data-Aware Task Scheduling on Multi-Accelerator based Platforms, in: The 16th International Conference on Parallel and Distributed Systems (ICPADS), Shanghai, China, 2010. doi:10.1109/ICPADS.2010.129.

[27] L. Stanisic, E. Agullo, A. Buttari, A. Guermouche, A. Legrand, F. Lopez, B. Videau, Fast and Accurate Simulation of Multithreaded Sparse Linear Algebra Solvers, in: The 21st IEEE International Conference on Parallel and Distributed Systems, Melbourne, Australia, 2015.
URL https://inria.hal.science/hal-01180272

[28] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, StarPU: a unified platform for task scheduling on heterogeneous multicore architectures, Concurrency and Computation: Practice and Experience 23 (2) (2011) 187–198. doi:10.1002/cpe.1631.
URL https://inria.hal.science/inria-00550877

[29] T. Cojean, A. Guermouche, A. Hugo, R. Namyst, P.-A. Wacrenier, Resource Aggregation for Task-based Cholesky Factorization on top of Modern Architectures, Parallel Computing 83 (2019) 73–92.

[30] D. Alvarez, V. Beltran, Accelerating task-based iterative applications, in: Proceedings of the 51st International Conference on Parallel Processing (ICPP), ACM, 2022, pp. 1–10.
URL https://arxiv.org/abs/2208.06332

[31] R. Pereira, A. Roussel, P. Carribault, T. Gautier, Investigating dependency graph discovery impact on task-based mpi+openmp applications performances, in: Proceedings of the 52nd International Conference on Parallel Processing (ICPP), 2023, pp. 163–172.

[32] C. Yu, S. Royuela, E. Quinones, Taskgraph: A low contention openmp tasking framework, IEEE Transactions on Parallel and Distributed Systems 34 (8) (2023) 2325–2336. doi:10.1109/TPDS.2023.3263077.

[33] C. Augonnet, A. Alexandrescu, A. Sidelnik, M. Garland, Cudastf: Bridging the gap between cuda and task parallelism, in: SC24: International Conference for High Performance Computing, Networking, Storage and Analysis, 2024, pp. 1–17. doi:10.1109/SC41406.2024.00049.

[34] C. Augonnet, S. Thibault, R. Namyst, Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures, in: 3rd Workshop on Highly Parallel Processing on a Chip (HPPC 2009), Delft, Netherlands, 2009.

[35] O. Beaumont, J. Langou, W. Quach, A. Shilova, A Makespan Lower Bound for the Scheduling of the Tiled Cholesky Factorization based on ALAP Schedule, in: EuroPar 2020 - 26th International European Conference on Parallel and Distributed Computing, Proceedings of EuroPar 2020, Springer, Warsaw / Virtual, Poland, 2020.

[36] H. Topcuoglu, S. Hariri, M.-Y. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, IEEE Transactions on Parallel and Distributed Systems 13 (3) (2002) 260–274. doi:10.1109/71.993206.

[37] O. Beaumont, L.-C. Canon, L. Eyraud-Dubois, G. Lucarelli, L. Marchal, C. Mommessin, B. Simon, D. Trystram, Scheduling on two types of resources: A survey, ACM Comput. Surv. 53 (3) (May 2020). doi:10.1145/3387110.
URL https://doi.org/10.1145/3387110

[38] O. Beaumont, A. Legrand, L. Marchal, Y. Robert, Steady-state scheduling on heterogeneous clusters: why and how?, in: 18th International Parallel and Distributed Processing Symposium, 2004. Proceedings., 2004, pp. 171–. doi:10.1109/IPDPS.2004.1303171.

[39] S. Tomov, R. Nath, H. Ltaief, J. Dongarra, Dense linear algebra solvers for multicore with gpu accelerators, in: 2010 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010, pp. 1–8. doi:10.1109/IPDPSW.2010.5470941.

[40] M. Gonthier, L. Marchal, S. Thibault, Taming data locality for task scheduling under memory constraint in runtime systems, Future Generation Computer Systems (2023).

[41] C. Flint, L. Paillat, B. Bramas, Automated prioritizing heuristics for parallel task graph scheduling in heterogeneous computing, PeerJ Computer Science 8 (2022) e969. doi:10.7717/peerj-cs.969.
URL https://doi.org/10.7717/peerj-cs.969

[42] B. Johnston, N. R. Miniskar, A. Young, M. A. H. Monil, S. Lee, J. S. Vetter, Iris: Exploring performance scaling of the intelligent runtime system and its dynamic scheduling policies, in: 2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2024, pp. 58–67. doi:10.1109/IPDPSW63119.2024.00017.

[43] K. Chronaki, A. Rico, M. Casas, M. Moretó, R. M. Badia, E. Ayguadé, J. Labarta, M. Valero, Task scheduling techniques for asymmetric multicore systems, IEEE Transactions on Parallel and Distributed Systems 28 (7) (2017) 2074–2087. doi:10.1109/TPDS.2016.2633347.

[44] R. Bleuse, T. Gautier, J. Lima, G. Mounié, D. Trystram, Scheduling data flow program in xkaapi: A new affinity based algorithm for heterogeneous architectures, in: EuroPar 2014 - 26th International European Conference on Parallel and Distributed Computing, Vol. 8632 of Lecture Notes in Computer Science, Springer-Verlag, 2014. doi:10.1007/978-3-319-09873-9_47.

[45] S. Shirzad, R. Tohid, A. Kheirkhahan, B. Wagle, H. Kaiser, Understanding the effect of task granularity on execution time in asynchronous many-task runtime systems, in: Euro-Par 2021: Parallel Processing Workshops: Euro-Par 2021 International Workshops, Lisbon, Portugal, August 30-31, 2021, Revised Selected Papers, Springer-Verlag, Berlin, Heidelberg, 2021, p. 456–467. doi:10.1007/978-3-031-06156-1_36.
URL https://doi.org/10.1007/978-3-031-06156-1_36

[46] A. Tzannes, G. C. Caragea, R. Barua, U. Vishkin, Lazy binary-splitting: A run-time adaptive work-stealing scheduler, in: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), ACM, 2010, pp. 179–190. doi:10.1145/1693453.1693479.

[47] A. Duran, J. Corbalan, E. Ayguade, An adaptive cut-off for task parallelism, in: SC '08: Proceedings of the 2008 ACM/IEEE Conference

on Supercomputing, IEEE, 2008, pp. 1–11. doi:10.1109/SC.2008.5213927.

[48] E. Ayguadé, J. Beyer, A. Duran, R. Ferrer, G. Haab, K. Li, F. Massaioli, An extension to improve openmp tasking control, in: M. Sato, T. Hanawa, M. S. Müller, B. M. Chapman, B. R. de Supinski (Eds.), Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 56–69.

[49] S. Iwasaki, K. Taura, A static cut-off for task parallel programs, in: Proceedings of the 2016 International Conference on Parallel Architectures and Compilation Techniques (PACT), ACM, 2016, pp. 139–150. doi:10.1145/2967938.2967968.

[50] P. Thoman, H. Jordan, T. Fahringer, Adaptive granularity control in task parallel programs using multiversioning, International Journal of High Performance Computing Applications 29 (4) (2015) 422–438. doi:10.1177/1094342014561209.

[51] U. A. Acar, V. Aksenov, A. Charguéraud, M. Rainey, Provably and practically efficient granularity control, in: Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), ACM, 2019, pp. 214–228. doi:10.1145/3293883.3295725.

[52] B. Wagle, M. A. H. Monil, K. Huck, A. D. Malony, A. Serio, H. Kaiser, Runtime adaptive task inlining on asynchronous multitasking runtime systems, in: Proceedings of the 48th International Conference on Parallel Processing (ICPP), ACM, 2019, pp. 76:1–76:10. doi:10.1145/3337821.3337915.

[53] A. S. Tuft, T. Weinzierl, M. Klemm, Detrimental Task Execution Patterns in Mainstream OpenMP® Runtimes, Springer Nature Switzerland, 2024, p. 210–224. doi:10.1007/978-3-031-72567-8_14.
URL http://dx.doi.org/10.1007/978-3-031-72567-8_14

[54] R. Yadav, S. Sundram, W. Lee, M. Garland, M. Bauer, A. Aiken, F. Kjolstad, Fusion: Merging similar tasks in task-based runtimes, in: Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume

1, ASPLOS '25, Association for Computing Machinery, New York, NY, USA, p. 182–197. doi:10.1145/3669940.3707216.
URL https://doi.org/10.1145/3669940.3707216

[55] E. Agullo, A. Buttari, A. Guermouche, F. Lopez, Multifrontal QR factorization for multicore architectures over runtime systems, in: F. Wolf, B. Mohr, D. an Mey (Eds.), Euro-Par 2013 Parallel Processing - 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings, Vol. 8097 of Lecture Notes in Computer Science, Springer, 2013, pp. 521–532.

[56] E. Agullo, A. Buttari, A. Guermouche, F. Lopez, Task-based multifrontal QR solver for gpu-accelerated multicore architectures, in: 22nd IEEE International Conference on High Performance Computing, HiPC 2015, Bengaluru, India, December 16-19, 2015, IEEE Computer Society, 2015, pp. 54–63.

[57] G. Iooss, C. Alias, S. Rajopadhye, Monoparametric Tiling of Polyhedral Programs, International Journal of Parallel Programming 49 (2021) 376–409. doi:10.1007/s10766-021-00694-2.
URL https://inria.hal.science/hal-02493164