Programming Heterogeneous Architectures Using Hierarchical Tasks

Mathieu Faverge, Nathalie Furmento, Abdou Guermouche, Gwenolé Lucas, Raymond Namyst, Samuel Thibault, Pierre-andré Wacrenier

Received: 21-Oct-2021 | Revised: 17-Jan-2022 | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

Programming Heterogeneous Architectures Using Hierarchical Tasks

Mathieu Faverge[1,3,4] | Nathalie Furmento[2,3,5] | Abdou Guermouche[2,3,4] | Gwenolé Lucas[2,3] | Raymond Namyst[2,3,4] | Samuel Thibault[2,3,4] | Pierre-André Wacrenier[2,3,4]

[1]Bordeaux INP, Bordeaux, France
[2]Inria Bordeaux Sud-Ouest, Bordeaux, France
[3]LaBRI, Bordeaux, France
[4]Université de Bordeaux, Bordeaux, France
[5]CNRS, Bordeaux, France

Correspondence
Gwenolé Lucas. Email: gwenole.lucas@inria.fr

Summary

Task-based systems have become popular due to their ability to utilize the computational power of complex heterogeneous systems. A typical programming model used is the Sequential Task Flow (STF) model, which unfortunately only supports static task graphs. This can result in submission overhead and a static task graph that is not well-suited for execution on heterogeneous systems. A common approach is to find a balance between the granularity needed for accelerator devices and the granularity required by CPU cores to achieve optimal performance. To address these issues, we have extended the STF model in the StarPU [1] runtime system by introducing the concept of hierarchical tasks. This allows for a more dynamic task graph and, when combined with an automatic data manager, it is possible to adjust granularity at runtime to best match the targeted computing resource. That data manager makes it possible to switch between various data layout without programmer input and allows us to enforce the correctness of the DAG as hierarchical tasks alter it during runtime. Additionally, submission overhead is reduced by using large-grain hierarchical tasks, as the submission process can now be done in parallel. We have shown that the hierarchical task model is correct and have conducted an early evaluation on shared memory heterogeneous systems using the Chameleon [2] dense linear algebra library.

KEYWORDS:
Multicore; accelerator; GPU; heterogeneous computing; task graph; programming model; runtime system; dense linear algebra

1 | INTRODUCTION

Recent advancements in High Performance Computing systems have led to the widespread use of heterogeneous multicore architectures. As a result, many researchers have focused on developing runtime systems that provide portable programming techniques and tools to effectively utilize this complex hardware. There are now many runtime systems that have been fully implemented for both regular homogeneous multicore systems and complex heterogeneous systems. Standards such as OpenMP (version 4.0 and above) have begun to support the task-based paradigm, in which applications are represented as a Direct Acyclic Graph (DAG) of tasks.

However, the task-based paradigm presents several challenges when trying to efficiently exploit heterogeneous platforms. The computing resources of heterogeneous platforms have diverse characteristics and requirements. For example, GPU devices typically perform best with large data sets, while conventional CPU cores reach peak performance with fine-grain kernels working on a reduced memory footprint. Additionally, systems typically have a larger number of CPU units than GPUs. This means that having more small tasks may be necessary to increase performance.

Several efforts have been made to address this problem, such as finding the best trade-off between the optimal granularity of each device [2, 3, 4, 5], or by aggregating CPU cores to process a task that was meant to be executed by an accelerator like a GPU [6, 7]. Alternatively, some preliminary work has considered splitting the tasks on CPU cores [8]. While these approaches can be efficient in specific contexts such as dense linear algebra, they suffer from the fact that the task graph is static, making it impossible to select an alternative granularity for a given operation at runtime. For example, when designing linear algebra solvers based on low-rank approximation algorithms, it is almost impossible to predict the right DAG to ensure good numerical accuracy [9, 10, 11, 12, 13].

In general, most modern task-based runtime systems suffer from a lack of dynamism in task-graph generation. Some programming models, such as [14, 8, 15], support just-in-time DAG submission, but the generation either follows static rules or requires a significant amount of programming effort. Another major limitation of using task-based runtime systems is the overhead imposed by the runtime system to manage large applications. This is closely linked to the size of the internal DAG representation as tasks are submitted. Many runtime systems that rely on the STF model [16] (such as OPENMP, STARSS, STARPU) build this representation at submission time and may be negatively impacted by a large number of non-ready tasks. On the other hand, programming models where the task graph is discovered using a high-level description of dependencies never build this representation and are less affected, but tend to be more difficult to use. An example of such a programming model is the parameterized task graph (PTG) [17] used by PARSEC [18]. Finally, regarding the sequential task flow model, task submission is generally done by a single thread, which may represent another bottleneck for the execution of large task graphs.

In this paper, we propose a new type of task, called hierarchical tasks, which can transform themselves into a new task graph dynamically at runtime. Programmers only need to provide hints indicating which tasks can be transformed into hierarchical tasks. The runtime system can then delay the submission of parts of the task graph to support dynamic implementation selection, parallelize the task insertion process, and greatly reduce the number of tasks in the runtime system. Our approach is similar to what is done in OPENMP for nested task-based parallelization schemes. However, we extend this approach to handle heterogeneous platforms while expressing fine-grain dependencies. In order to express more parallelism in sub-DAGs inserted with hierarchical tasks, we might want to partition and unpartition data. This is possible thanks to an advanced data manager, which can dynamically and asynchronously change the data layout. The system performs these operations automatically by inserting data management tasks that enforce data consistency. We also rely on this data manager to ensure that a DAG involving hierarchical tasks is consistent with a DAG created with the sequential task flow model, and therefore correct.

The proposed model associated with these hierarchical tasks address the abovementioned issues: 1) How to make the task graph more dynamic? 2) How to reduce the overhead of the runtime system? 3) How to overcome the intrinsic limitation of the sequential task flow submission process? While this model is generic and targets distributed heterogeneous architectures, this paper focuses on an initial implementation for shared memory heterogeneous architectures.

The contribution of this paper is two-fold: 1) We present an advanced data management engine that supports asynchronous data layout modification, and 2) We show how we extend the sequential task flow model to support hierarchical tasks and present our implementation within the STARPU runtime system.

The remainder of the paper is organized as follows. We present a general context with the sequential task flow model and the STARPU runtime system. Afterward, we describe the proposed advanced data manager, a crucial feature for implementing the hierarchical task model. We then introduce our hierarchical task paradigm and show the correctness of the model. Finally, we present a set of experiments showing the interest in the approach when performing dense linear algebra operations implemented within the CHAMELEON library [2].

2 | RELATED WORK

In recent years, a significant amount of attention has been devoted to designing advanced runtime systems for modern heterogeneous architectures. Most of these efforts fall within the task-based paradigm, which is a natural way for scientific programming

workflow (*e.g.* CILK [19], OPENMP or INTEL TBB [20] for multicore machines, APC [21], CHARM++ [22, 23], HPX [24], KAAPI/XKAAPI [25], LEGION [14], PARSEC [18], STARPU [1] or STARSS [26] for heterogeneous configurations).

The common point among these runtime systems is that they all use high-level descriptions of dependencies to build the task graph at runtime and then schedule the corresponding computations on available resources. Various approaches are used to construct the task graph. For instance, most of the previously mentioned runtime systems rely on the so-called Sequential Task-Flow model (e.g. OPENMP, STARSS, STARPU) to build the task graph. By using data access modes and a sequential submission order, dependencies between tasks can be inferred through data dependency analysis [16]. On the other hand, runtime systems such as PARSEC are based on the parameterized task graph programming model (PTG) [17], where the task graph is unrolled at runtime, using a high-level description of the dataflow corresponding to the computations. Other runtime systems use different paradigms to express computations, such as LEGION, which describes logical data regions representing the data flow and dependencies between tasks. The various programming models have different levels of ease of use and the amount of overhead they impose on the underlying runtime system. For example, the sequential task flow model is easy to use as programmers only need to provide the sequential implementation of their application and then add data access modes. However, this method comes with higher overhead in the runtime system. On the other hand, the parameterized task graph approach requires users to express their computations in a subtle high-level formalism where the dataflow is explicitly described, but has less overhead on the runtime system [27].

Several efforts have been made to address the problem of reducing the overhead of task-based runtime systems, which mainly focus on those based on the sequential task flow model or increasing the amount of parallelism provided by these systems. In [28], the authors analyzed the limiting factors in the scalability of a task-based runtime system and proposed individual solutions for each of the identified challenges, such as a wait-free dependency system and a scalable scheduler design based on delegation instead of work-stealing. Other approaches focus on advanced dependency management. For example, in [29], the authors proposed an eager approach for releasing data dependencies, where tasks are launched for execution as soon as their data requirements are met instead of waiting for predecessor tasks to finish execution. In [7], worksharing tasks were introduced, which internally leverage worksharing techniques to exploit fine-grained structured loop-based parallelism without requiring a barrier. The closest contribution to our proposition, with regard to task dependencies, was introduced in [30] with the concept of weak dependencies, which is an extension of the OPENMP model that supports fine-grained dependencies not only between sibling tasks but also between tasks with any family relationship. Our contribution is a generalization of the weak dependency concept to the heterogeneous case where the underlying hardware does not ensure memory consistency, thus requiring an advanced data manager (see Section 4). Additionally, in ParSEC [8] and XKaapi [31], some preliminary work targeting heterogeneous architectures has considered splitting tasks when assigned to CPU cores.

	Hierarchical tasks support		
Runtime	Fine-grain Dependencies	Automatic Data Management	Heterogeneity
PaRSEC [8]	✗	✗	✓
OmpSs [30]	✓	✗	✗
TaskFlow [32]	✗	✗	✓
IRIS [33]	✗	✓	✓
libtask [12]	✗	✓	✓

TABLE 1 Summary of of hierachical tasks support in task-based runtime systems.

Several efforts have been made to allow task-based runtime systems to have more dynamic capabilities in terms of task management and generation. In TaskFlow [32], advanced tasking schemes are introduced, including dynamic, composable, and conditional tasking. Dynamic tasking, in particular, allows for the dynamic generation of a sub-DAG from a given task. However, synchronization is added at the end of each hierarchical task to ease dependency management, and programmers are responsible for data management and changing data layout as needed. In [33], the IRIS runtime is introduced, which can perform dynamic task partitioning either through user input or automatically via a polyhedral compiler, but no details are provided on

how dependencies are handled in this context. Finally, libtask, an advanced runtime system supporting hierarchical tasks in the context of low-rank linear algebra solvers is presented in [12]. This work introduces hierarchical tasks, and dependencies are expressed at the finest level. The correctness of the produced DAG is considered through extra automatic dependencies, but data management is straightforward as data partitioning is performed statically at the beginning of the execution.

Table 1 provides a summary of how the runtime systems mentionned in this section compare with respect to the functionality we want to achieve with hierarchical tasks. We focus on three main features. The first one is the presence of fine-grain dependencies between distinct sub-graphs. The second one is a data manager that can adjust the data layout around hierarchical tasks without user supervision. The third one is the support of heterogeneous architectures.

3 | BACKGROUND

The subsection below presents the basics of the sequential task flow model and the corresponding sequential consistency. We then briefly present our target runtime system StarPU.

3.1 | The Sequential Task Flow Model

The Sequential Task Flow (STF) programming model entirely relies on Sequential Consistency using only implicit dependencies. The STF model, therefore, simply consists of submitting a sequence of tasks through a non-blocking function call that delegates the execution of the task to the runtime system. Upon submission, the runtime system adds the task to the current DAG, and its dependencies are automatically computed through data dependency analysis [34]. The actual execution of the task is then postponed until its dependencies are satisfied. This paradigm is sometimes referred to as *Superscalar* since it mimics the operation of superscalar processors where instructions are issued sequentially from a single stream but can be executed in a different order and, possibly, in parallel depending on their mutual dependencies.

```
call F(x, y)
call G(x, u)
call H(y, z)
```

```
call submit(F, x:RW, y:RW)
call submit(G, x:R, u:RW)
call submit(H, y:R, z:RW)
call wait_tasks_completion()
```

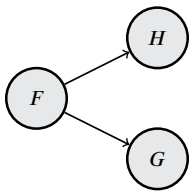


FIGURE 1 Pseudo-code for a dummy sequential algorithm (*left*), corresponding STF version (*center*) and subsequent DAG (*right*).

Figure 1 shows a dummy sequential algorithm and its corresponding STF version. Instead of making three function calls (`F, G, H`), the equivalent STF submits the three corresponding tasks. The data onto which these functions operate as and their access mode (Read, Write or Read/Write) are also specified. Because task `G` accesses data `x` after task `F` has accessed it in Write mode, the runtime infers a dependency between tasks `F` and `G`. Similarly, a dependency is inferred between tasks `F` and `H` due to data `y`. Figure 1 (*right*) shows the DAG corresponding to this STF dummy code. In the STF model, one thread is responsible for submitting the tasks; we refer to this thread as the *master thread*. *Worker threads* instead achieve the execution of tasks. The function called at the end of the STF pseudo-code is simply a barrier that prevents the master thread from continuing until all the submitted tasks are executed.

We present in Sections 4 and 5 how hierarchical tasks can be introduced in task-based runtime systems relying on the STF model without breaking the sequential consistency (i.e. ensuring a task execution order coherent with a sequential execution) to ensure the correctness of the DAG.

3.2 | The StarPU Runtime System

StarPU [1] is a library that provides programmers with a portable interface for scheduling dynamic graphs of tasks onto a heterogeneous set of processing units (*i.e.* CPUs and GPUs). The two basic principles of StarPU are, first, that tasks can have

several implementations for some or each of the various heterogeneous processing units available in the machine. Secondly the runtime system handles necessary data transfers to these processing units transparently. StarPU tasks are defined as multi-version kernels, gathering the different implementations available for CPUs and GPUs, associated to a set of input/output data. To avoid unnecessary data transfers, StarPU allows multiple copies of the same registered data to reside simultaneously in different memory locations as long as it is not modified. Asynchronous data prefetching and write-back is also used to hide memory latencies.

StarPU is a platform for developing, tuning and experimenting with various task scheduling policies in a portable way. It provides an abstract view of the machine by relying on the notion of worker to describe the computing resource (*e.g.* CPU, GPU) in charge of executing the tasks. Implementing a scheduler consists of creating a set of queues, associating them with the different processing units, and defining the code that is triggered each time a new task is ready to be executed, or when a processing unit is about to go idle. Various designs can be used to implement queues (*e.g.* FIFOs or stacks), and they can be organized according to different topologies. Several built-in schedulers are available, ranging from greedy and work-stealing based policies to more elaborate schedulers implementing variants of the Minimum Completion Time (MCT) policy [35].

4 | AUTOMATIC DATA MANAGEMENT

Data handling is at the heart of StarPU to automatically infer dependencies between tasks in the STF model and automatically manage data transfers between the different memory banks of a distributed/heterogeneous system. Applications must register the data that are handled by the tasks. To do so, StarPU provides an opaque data structure called *handle*, which is an abstract view of registered data. Handles along an access mode (read-only, read-write, ...) are used as task parameters. A task must access data through the associated handle.

To ease data manipulation, StarPU brings the notion of *data filter*, a tool to partition data associated with a handle into sub-data parts associated with new sub-handles. Indeed, instead of registering all data sub-sets independently, it is often more convenient to register a large piece of data and to recursively partition it.

Once a handle is partitioned, we can remark that the same data can be designated simultaneously by several different handles. This may be very convenient when programmers want to use data with several views. Moreover, data in read-only access mode can advantageously be accessed simultaneously at different partitioning levels by several tasks on other devices. However, when data is accessed in write access mode, this access must be exclusive for coherency purposes. This property is ensured by StarPU when a single partitioning is used for data, but may be violated when several handles point to the same data. To deal with this problem, StarPU provides functions to invalidate other handles ensuring they can not be used to access their underlying data, and unpartition sub-handles back into the main handle to gather the sub-data.

However, users' feedback showed that maintaining consistency between different partitions is difficult. This is why we propose a mechanism to automate the management of several simultaneous partitions. This is done by letting StarPU automatically insert partition or unpartition tasks as needed.

We will now present the modifications we made in StarPU to allow this automatic data management. First, programmers need to define the partitioning scheme through the plan operation, which aims to declare the partitioning to StarPU, and can be seen as the declaration of a new set of sub-handles. One can even partition recursively and use handles at different levels of the recursion. Once a plan is performed, it is possible to submit tasks using the initial handle or any of the sub-handles, even if the actual partitioning has not been done yet. Furthermore, several partitioning schemes can be planned simultaneously as illustrated in Figure 2 (see the lines 12 to 25). The data manager will then handle the actual partitioning tasks and data coherency. The given example shows a matrix on which two partition plans are defined, one with vertical stripes and one with horizontal stripes. The matrix is first initialized through its root handle, then modified using vertical partitioning, and checks are performed in horizontal and vertical stripes.

This mechanism is very natural to use for the programmer because it allows them to deal with the registration and partitioning of their data once and for all. They are then free to use any handle within the tasks they insert and the runtime system will adapt the data layout by itself.

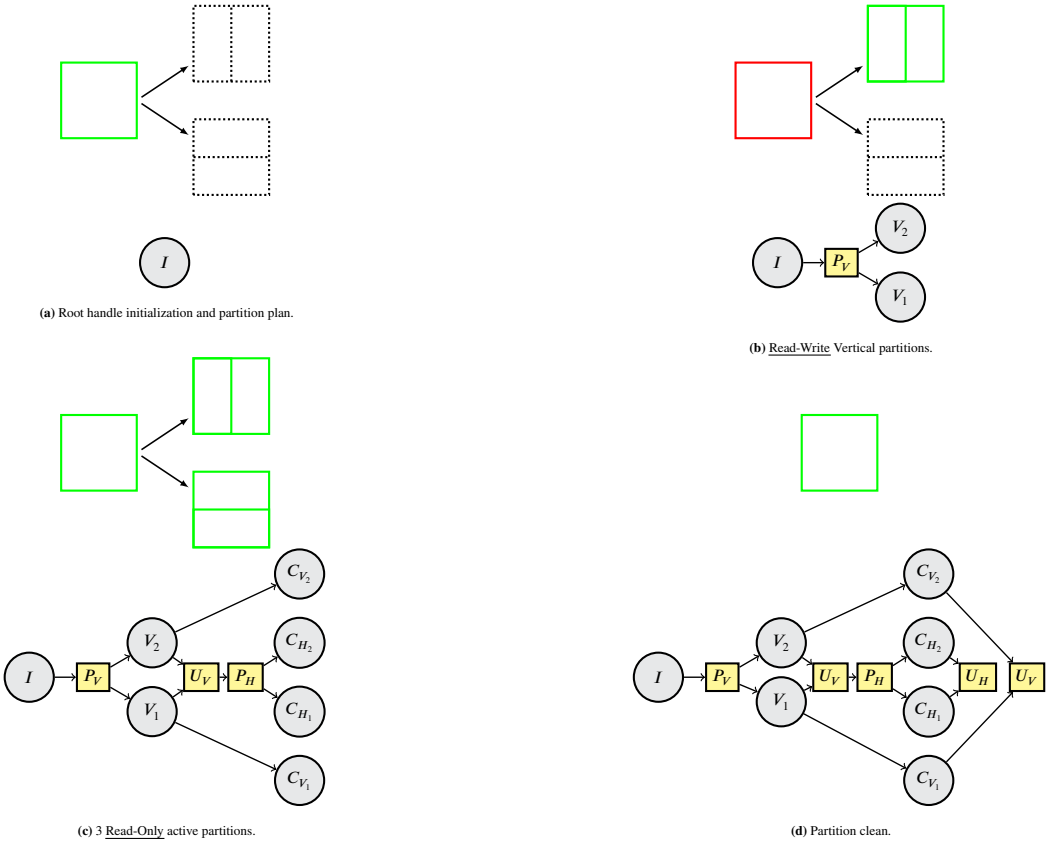
At runtime, StarPU will introduce coherency synchronization. When a task is ready to be executed, StarPU must ensure that the partition associated with each handle it uses is valid. StarPU will allow different partitioning to coexist if data is accessed in read-only mode. As soon as data is accessed in read-write mode, StarPU will automatically (and recursively) unpartition sub-data and activate only the partitioning leading to the handle being written to.

```
1  #define PARTS 2
2  void do(int matrix[NX][NY])
3  {
4    starpu_data_handle_t handle;
5    starpu_data_handle_t v_handle[PARTS];
6    starpu_data_handle_t h_handle[PARTS];
7
8    /* Declare the whole matrix to StarPU */
9    starpu_matrix_data_register(&handle, STARPU_MAIN_RAM,
10     (uintptr_t)matrix, NX, NX, NY, sizeof(matrix[0][0]));
11
12   /* Partition the matrix in PARTS vertical slices */
13   struct starpu_data_filter vertF = {
14     .filter_func = starpu_matrix_filter_block,
15     .nchildren = PARTS
16   };
17   starpu_data_partition_plan(handle, &vertF, v_handle);
18
19   /* Partition the matrix in PARTS horizontal slices */
20   struct starpu_data_filter horF = {
21     .filter_func = starpu_matrix_filter_vertical_block,
22     .nchildren = PARTS
23   };
24   starpu_data_partition_plan(handle, &horF, h_handle);
25
26   /* Fill the matrix */
27   starpu_task_insert(&initialize, STARPU_W, handle, 0);
28
29   /* Modify the values via the vertical slices */
30   for (unsigned i = 0; i < PARTS; i++)
31     starpu_task_insert(&modify, STARPU_RW, v_handle[i], 0);
32
33   /* check the values via both horizontal and vertical slices */
34   for (unsigned i = 0; i < PARTS; i++)
35     starpu_task_insert(&check, STARPU_R, h_handle[i], 0);
36   for (unsigned i = 0; i < PARTS; i++)
37     starpu_task_insert(&check, STARPU_R, v_handle[i], 0);
38
39   /*  Unregister data from StarPU. */
40   starpu_data_partition_clean(handle, PARTS, v_handle);
41   starpu_data_partition_clean(handle, PARTS, h_handle);
42   starpu_data_unregister(handle);
43 }
```

FIGURE 2 Automatic unpartitioning example.

To illustrate the behavior of the code given in Figure 2, we provide in Figure 3 the evolution of the data layout and the task graph by emphasizing the submission of automatic partition/unpartition tasks by the runtime system. We can see that after executing line 27 (see Figure 3a), the plan operations have been performed (see the dotted matrix layouts), and the initialization task has been submitted. When the first task using a vertical stripe, which modifies the matrix, is submitted on line 31, the runtime system will automatically insert the corresponding *partitioning* task (see Figure 3b). The same scheme is then applied at the submission of the tasks working on the horizontal layout (at line 35) and vertical layout (at line 37) in read-mode. One should note that C_{v_1} and C_{v_2} share the same vertical layout as V_1 and V_2, so no partition operation is needed for these tasks. On the contrary, tasks C_{H_1} and C_{H_2} do not share any handles with those using the vertical layout. However, the data manager knows these handles share a joint ancestor (the whole matrix). Thus, it will insert, the unpartition/partition tasks as needed to make the data available to the tasks using the horizontal layout. This is illustrated in Figure 3c where the U_v and P_h tasks are inserted, making the tasks using the horizontal layout depend on them. Finally, when the partition needs to be cleaned, the final unpartition task is inserted (see Figure 3d).

The previous example illustrates the general behavior of the data manager. More precisely, during the submission of tasks, each handle in the partitioning hierarchy can be either *inactive* (one cannot access the piece of data), *read-write-active* (one can read/write to the piece of data or a sub-part of it), or *read-only-active* (one can only read from the piece of data or a sub-part of it). The main handle at the root of the partitioning hierarchy is always *read-write-active*. Each handle in the hierarchy, when active, is additionally either *unpartitioned* (one can read/write the piece of data itself), *read-write-partitioned* (one can only write to the sub-pieces of data), or *read-only-partitioned* (one can read the piece of data or sub-pieces of data) ; when it is partitioned, its children sub-handles in the hierarchy are active.



(a) Root handle initialization and partition plan.

(b) Read-Write Vertical partitions.

(c) 3 Read-Only active partitions.

(d) Partition clean.

FIGURE 3 DAG and data layout corresponding to the code provided in Figure 2. Dotted border stands for inactive, solid border stands for active. Red border stands for read-write partitioned. Green border stands for read-only partitioned or unpartitioned.

When submitting a task that accesses a handle within the hierarchy, StarPU will automatically ensure that the handle is active. This possibly requires recursively making its ancestors active by submitting partitioning tasks for them, possibly starting right from the root handle of the hierarchy. This also possibly requires recursively submitting unpartitioning tasks for some sub-handles which were previously written to. In the case of the transition from Figure 3b to Figure 3c, StarPU indeed had to submit the unpartition of the root handle, and repartition it. It should be noted that all this mechanism is done completely asynchronously: StarPU just inserts data coherency synchronization tasks within the task graph.

5 | THE HIERARCHICAL TASK PARADIGM

To extend the sequential task flow model with hierarchical tasks, we first need to define what precisely a hierarchical task is. Formally, a hierarchical task is simply a regular task that can, at runtime, submit a sub-DAG instead of performing actual computations. Processing a hierarchical task consists of submitting its corresponding task sub-graph, its outgoing dependencies can be released at the end of that submission process. Furthermore, to ensure portability concerning heterogeneous platforms, coherency synchronization tasks are submitted along the sub-graph to ensure a correct execution by connecting the sub-DAG with the rest of the DAG. In the end, hierarchical tasks represent an elegant answer to 1) the problem of adapting the granularity of tasks to the device executing them, 2) the question of the reduction of the number of active tasks in the runtime system, 3) the problem of the dynamic selection of the implementation of a given operation in the application. Introducing hierarchical tasks in a task-based runtime system like StarPU needs to respect the following constraints, which aim at having a general implementation of such a paradigm.

1. The depth of the hierarchy is not limited (a hierarchical task can spawn a DAG that contains hierarchical tasks).
2. Programmers express their task graph at the highest level and only annotate some tasks as possibly hierarchical.

3. Data management needs to be transparent to programmers.

4. Task dependencies always have to be inferred at the finest grain (i.e. the deepest level of the hierarchy).

Properties 1 and 2 ensure a general programming interface with no limitations from the programmer's point of view. Property 3 is related to data layout management. It is necessary for the heterogeneous case where memory is not shared between all the computing resources, meaning that explicit data movements must be performed. This property will be handed to the advanced data manager introduced in Section 4. Finally, Property 4 is related to parallelism management. It only says that we do not want a fork-join approach, as there is no barrier at the end of hierarchical tasks.

To use hierarchical tasks, the programmer inserts a task that works on the coarsest level of data and adds a few hints to make it hierarchical. These hints consist of a graph generating function and the sub-data on which the sub-graph will work. Figure 4 is an example of how the code from Figure 2 could be extended to use hierarchical tasks. Instead of using `for` loops to submit tasks working on the sub-data, we move them into two new functions `modify_gen_dag` and `check_gen_dag` and submit a single task, working on the main data, in their place. These functions, with the appropriate sub-data, are passed as arguments to each corresponding task insertion, making these tasks hierarchical for StarPU.

```
24  [...]
25
26  /* Fill the matrix */
27  starpu_task_insert(&initialize, STARPU_W, handle, 0);
28
29  /* Modify the values via the vertical slices */
30  starpu_task_insert(&modify, STARPU_RW, handle,
31                     STARPU_GEN_DAG, &modify_gen_dag,
32                     STARPU_GEN_DAG_ARG, v_handle, 0);
33
34  /* check the values via both horizontal and
35     vertical slices */
36  starpu_task_insert(&check, STARPU_R, handle,
37                     STARPU_GEN_DAG, &check_gen_dag,
38                     STARPU_GEN_DAG_ARG, v_handle, 0);
39  starpu_task_insert(&check, STARPU_R, handle,
40                     STARPU_GEN_DAG, &check_gen_dag,
41                     STARPU_GEN_DAG_ARG, h_handle, 0);
42
43  [...]
```

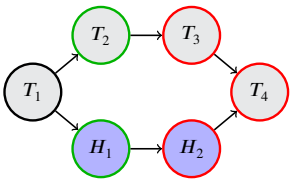
```
void modify_gen_dag(starpu_data_handle_t *handles) {
  for (unsigned i = 0; i < PARTS; i++)
    starpu_task_insert(&modify,
                       STARPU_RW, handles[i], 0);
}
```

```
void check_gen_dag(starpu_data_handle_t *handles) {
  for (unsigned i = 0; i < PARTS; i++)
    starpu_task_insert(&check,
                       STARPU_R, handles[i], 0);
}
```

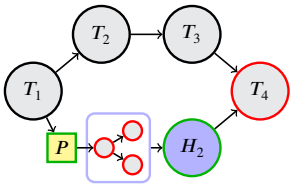
FIGURE 4 Excerpt from Figure 2, rewritten to use hierarchical tasks.

Figure 5a is an execution scenario for a given task graph, where some tasks have been tagged as possibly hierarchical (shown as blue nodes). 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 not-ready task has a **red** border (some dependencies are unsatisfied), 3) an already executed task has a **black** border. Thus, we can see in Figure 5a that T_1 has completed its execution, making T_2 and H_1 ready, while the remaining tasks are not ready for execution. T_2 and T_3 execute as normal tasks, while H_1 is processed, i.e. its corresponding sub-DAG is submitted, resulting in Figure 5b. The dependency between H_1 and H_2 is then released, making H_2 ready for processing. Furthermore, we can see that after the processing of H_2 (see Figure 5c), the dependencies between the resulting submitted tasks are inferred by the runtime system at the deepest level of the hierarchy. This will allow for a good pipelining of the processing of the two task graphs resulting from H_1 and H_2.

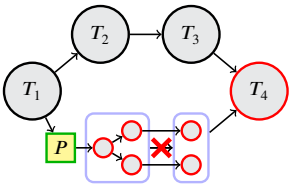
We now have to consider how the data coherency will be achieved between the DAG and the sub-DAGs. Introducing hierarchical tasks in a task-based runtime system requires changing the granularity of data dynamically at runtime each time a hierarchical task has to be executed. Indeed, to be able to express parallelism within the hierarchical task, the task's input data has to be either partitioned or unpartitioned (see Section 4 for more details) depending on the level of the task in the hierarchy. An example of this aspect is provided in Figure 6, where a DAG composed of a single hierarchical task (implementing a matrix product (GEMM)) is provided. We can see that when task H_1 is processed, it submits its corresponding DAG. However, to exhibit parallelism, data has to be partitioned (in this case, the original matrices are partitioned in a 4 by 4 layout). This illustrates the fact that introducing hierarchical tasks requires advanced partitioning/unpartitioning operations for each input and output data.



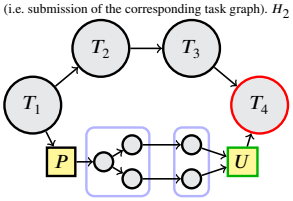
(a) Initial DAG.



(b) Processing of H_1 (i.e. submission of the corresponding task graph). H_2 is not processed yet.

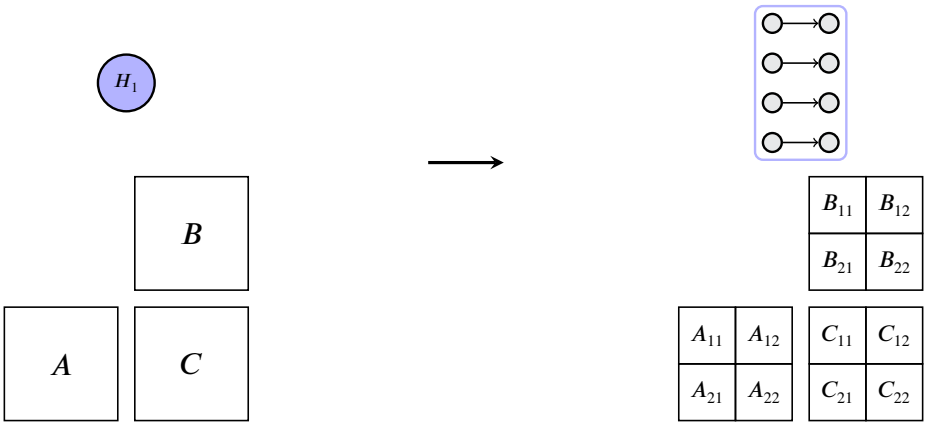


(c) Processing of H_2. Dependencies between H_1 and H_2 expressed at the deepest level.



(d) Automatic insertion of the unpartition task U.

FIGURE 5 Example of a DAG with 2 hierarchical tasks and 4 regular tasks.



(a) Initial hierarchical task and its corresponding data layout.

(b) Data partitioning and the corresponding task graph.

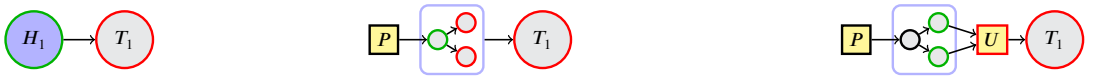
FIGURE 6 Example of a hierarchical task corresponding to a matrix product operation $C = C + A \times B$ and its corresponding DAG.

Thus, the general approach is to have a data management operation (either partitioning or unpartitioning) preceding each individual task (resp. hierarchical task) when needed. We propose to automatically insert a data management task ahead of a task requiring data not in the correct layout by relying on the data manager introduced in Section 4. Coming back to Figure 5, we can see, in Figure 5b, the insertion of the partitioning task P ahead of the sub-graph produced by H_1. This can be done either during the processing of H_1 or delayed until the first task of the task graph corresponding to H_1 is ready for processing. Conversely, the unpartitioning task U is inserted before the final task T_4. The insertion of such a task can be delayed as late as the moment when all the task dependencies of T_4 are fulfilled. It is important to emphasize that the insertion of the unpartition task needs to know whether the task T_4 is hierarchical or regular. These data management tasks are inserted automatically by the runtime ahead of the task requiring access to data whenever the current layout of the data is not the one needed by the task (see Section 4 for more details). We can also notice that there is no data management task between the sub-graphs produced by H_1 and H_2 since they share the same data layout. Finally, it is essential to emphasize that hierarchical tasks are executed when their dependencies are fulfilled. However, the actual computations tasks submitted by these hierarchical tasks are executed whenever they are ready. Thus, we need to ensure a correct order of the actual computations.

We show in the next section how we ensure that the DAG generated when relying on hierarchical tasks is coherent with the sequential task flow model, even though we are now able to make task submission parallel, since the processing of various hierarchical tasks can be performed concurrently.

5.1 | Ensuring the Correctness of the DAG

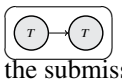
We will now show why the hierarchical task model to extend the sequential task flow model produces a correct DAG whatever the depth of the hierarchy and the interactions between regular and hierarchical tasks. Firstly, as stated above, the sequential task flow model infers the dependencies from data access modes of individual tasks while relying on sequential consistency (the insertion order of tasks provides a natural ordering to infer task dependencies). Thus, when introducing hierarchical tasks, one major issue is ensuring that the DAG produced after processing all the hierarchical tasks is correct with respect to a sequential submission scheme (in the end, we want the DAG to be equivalent to the one using a single submission thread). Indeed, when relying on hierarchical tasks, the submission can now be made in parallel, while in the sequential task flow model, the submission is made by a single entity. In other words, how can the correctness of the DAG be ensured?



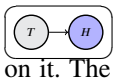
(a) Initial DAG. **(b)** Processing of H_1. **(c)** Insertion of the unpartition task when T_1 is ready.

FIGURE 7 Example of a scenario where a task follows a hierarchical task.

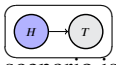
We must show that the dependencies respect the sequential task flow model to answer this question. We discuss four simple scenarios, which are building blocks for any general DAG to show its correctness.



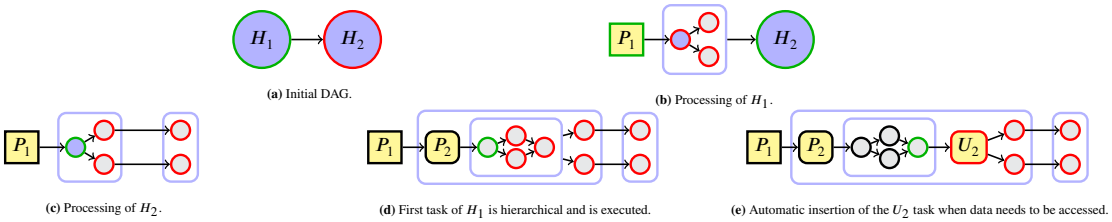
This is a straightforward scenario that corresponds to the sequential task flow model. The sequential consistency and the submission order of the tasks will ensure the correctness of the DAG.



This represents the case, where a regular task in the original DAG is followed by a hierarchical one that depends on it. The sequential task flow model used for submitting the original DAG ensures that the hierarchical task cannot start its processing before the regular task has been completed, ensuring the dependencies' correctness.

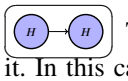


This represents the case where a hierarchical tasks precedes a regular task in the original DAG. An example of this scenario is provided in Figure 7. The main problem in this scenario is that the regular task is by construction submitted before the tasks resulting from the hierarchical task (H_1 in Figure 7). This may violate the order required by sequential consistency. However, in this specific situation, the hierarchical task has changed the data layout before it starts its processing (see Figure 7b). Thus, the task following the hierarchical task (T_1 in Figure 7) will request the data layout to be changed. The data manager will then automatically submit data management tasks to turn back data to their original layout. These data management tasks will be inserted ahead of the task in the DAG and will depend on the data produced by the DAG resulting from the processing of the hierarchical task. This corresponds to the configuration depicted in Figure 7c where the unpartition task U is inserted so that T_1 depends on U, which needs the data produced by the previous tasks. Therefore, the data management tasks will ensure that the regular task T cannot start its execution before the completion of the DAG submitted by the hierarchical task. This illustrates the data manager's central role, which has been discussed in Section 4.



(a) Initial DAG. **(b)** Processing of H_1. **(c)** Processing of H_2. **(d)** First task of H_1 is hierarchical and is executed. **(e)** Automatic insertion of the U_2 task when data needs to be accessed.

FIGURE 8 Example of a chain of two hierarchical tasks.



This represents the case where in the original DAG, a hierarchical task is followed by another one that depends on it. In this case, since the dependency between the two hierarchical tasks is not released until the first one has completed its processing, the tasks resulting from the two hierarchical tasks are correctly ordered, making the dependencies between these tasks coherent with the sequential consistency. This is illustrated in Figure 8 where initially, two hierarchical tasks H_1 and H_2 are submitted (see Figure 8a). Then H_1 is processed (see Figure 8b). Note that in the example, we assume that the data was previously unpartitioned, and thus a data partitioning task P_1 is needed before the DAG corresponding to H_1. Afterward, H_2 is processed (see Figure 8c) and it does not require any data layout modification. Note that, each individual task produced by a hierarchical task can itself be hierarchical, and the same rules can be applied recursively to ensure the correctness of the DAG. This is illustrated in Figure 8d where the first task submitted by H_1, which will be referred to as H_{11}, is decided to be hierarchical (at runtime) and is executed. We can also see the partitioning task P_2, which was automatically inserted by the data manager to partition the data further. The resulting task graph is coherent with the sequential task flow paradigm. Finally, note that the unpartitioning U_2 operation corresponding to P_2 is submitted in a second step when the final tasks of H_1 require data (see Figure 8e). This automatic insertion of U_2 is vital to turn back data into their original layout before the final tasks of H_1 can start their execution.

6 | EXPERIMENTAL EVALUATION

To illustrate the potential of hierarchical tasks, we apply them in a dense linear algebra context using the CHAMELEON library [2]. The objective is to validate the hierarchical task model by using it to create correct DAGs with various level of task granularity. We use various matrix partitioning schemes to decide where to place hierarchical tasks to adress certain limitations in the applications we chose and potentially improve their performances. To do so, we adapted the matrix descriptors to describe hierarchical data structures. Three partitioning schemes have been implemented for these experiments. They are described further down for the experiment they were used in. As explained in Section 4, all these partitions are only planned, and the decision to submit a hierarchical task is taken at runtime. That decision is based on the data structure of the tiles involved in the computations. A sub-graph will be submitted if there is a partitioning plan for all of them. Otherwise, the tasks will operate at the highest joint data level.

Concerning hierarchical variants, we will use the following notation: $x/y/z/...$. This means that each initial tile is of size x and is partitioned according to one of the three schemes mentioned above into tiles of size y, which are in turn split into tiles of size z etc.

The following experiments were conducted on two architectures, presented in Table 2.

Name	Processor	GPU	Memory
INTEL-V100	2 x INTEL XEON GOLD 6142, 16 cores, 2.6GHz	2 x NVIDIA V100 (16GB)	384GB
AMD-A100	2 x AMD ZEN3 EPYC 7513, 32 cores, 2.6GHz	2 x NVIDIA A100 (40GB)	512GB

TABLE 2 Characteristics of the considered platforms.

Figure 9 shows the performance of a single dense DGEMM kernel for different matrix sizes (the test involves three square matrices) on both platforms. We present with a solid line the actual performance for both a single CPU and a single GPU. We can see that on the CPU side, the processor's peak performance is achieved for matrix sizes not exceeding 500. Providing larger matrices does not improve the absolute performance on a single core. On the GPU side, we can observe that the larger the matrix size, the higher the performance. This experiment aims to find the best trade-off for the task granularity between the CPU cores and GPU devices. Having a granularity of tasks not large enough will limit GPU performance. On the other hand, using very coarse grain tasks limits parallelism and may affect CPU performance. Thus, in future experiments, we will mainly use a tile size of 2880 (resp. 5700) on the INTEL-V100 (resp. AMD-A100) platform.

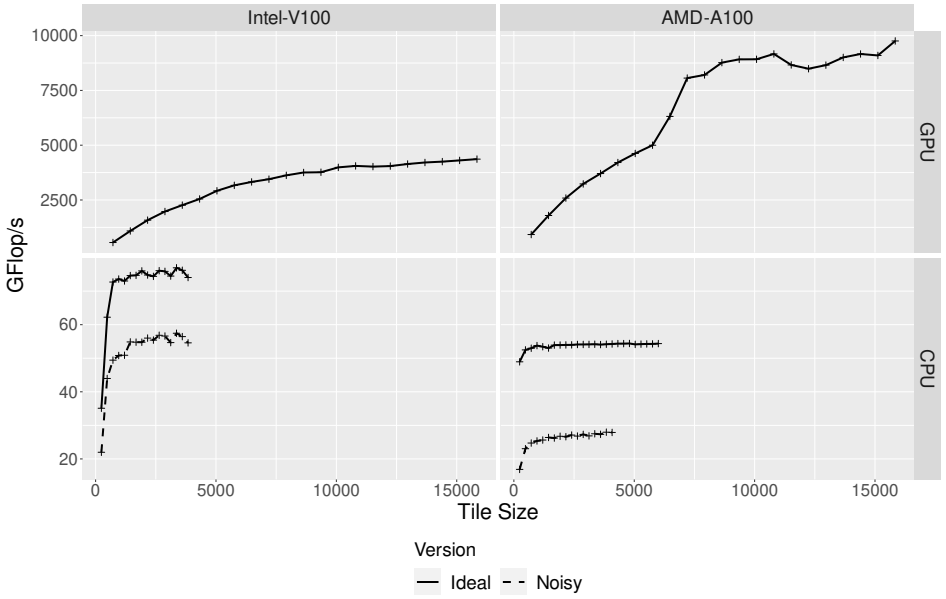


FIGURE 9 Performance of a matrix product (DGEMM) kernel. In the ideal setting a single kernel runs on a single ressource (CPU core or GPU device). In the noisy setting every CPU core runs 10 independent DGEMM kernels.

The dotted line is the normalized performance of a batch of independent DGEMM kernels spread evenly across all CPU cores. This accounts for the performance degradation resulting from memory contention in shared memory hierarchy levels (L3 cache, for example), and for the CPU frequency reduction due to the higher number of concurrent resources exploited.

On both architectures, StarPU and Chameleon are compiled with GCC 10.3.0, Nvidia CUDA 11.2, and the Intel MKL 2020. StarPU has been configured to use a single stream per GPU and to pipeline four events per stream. Unless otherwise specified, we use all CPU cores and GPU devices available on the node.

6.1 | Overhead

To evaluate the overhead induced by hierarchical tasks, we consider the graph of a matrix-matrix multiplication (GEMM) using a tile size of 960. The partitioning scheme used in this experiment recursively splits each matrix tile. An example is shown in Figure 11a with one recursion level: each matrix tile is divided into a finer tile size. Figure 10 compares the graph submission time per computational task in two configurations. The '960' curve represents the non-hierarchical case. The '960/960' curve shows the worst possible scenario: the DAG is composed only of hierarchical tasks and each submits exactly one task when processed. This doubles the number of tasks submitted and heavily increases the workload of the data manager, making the submission time per computational task roughly 3.5 times slower. Finally, the '2880/960' curve is a more realistic scenario, where the graph is first submitted at coarse grain (with a tile size of 2880) and then refined down to the same granularity as the previous configurations (960). In this case, each hierarchical task submits $\lceil 2880/960 \rceil^3 = 27$ regular tasks when processed, thus amortizing the overhead induced by the management of hierarchical tasks.

6.2 | Dense matrix product (GEMM)

For the matrix product kernel $C = \alpha AB + \beta C$, the main objective of the hierarchical scheme is to balance the workload between CPU cores and GPU devices based on their respective computing power. A custom partitioning scheme is used. We first define a computing power for each resource, $\mathcal{P}_{CPU}$, and $\mathcal{P}_{GPU}$ respectively, and then we estimate the computing power ratio of the CPU resources:

$$R = \frac{\#_{CPU} \times \mathcal{P}_{CPU}}{(\#_{CPU} \times \mathcal{P}_{CPU}) + (\#_{GPU} \times \mathcal{P}_{GPU})}.$$

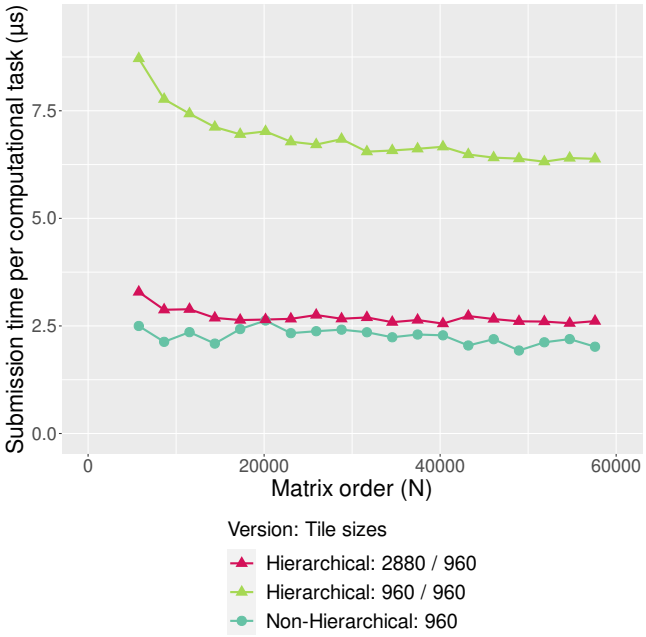
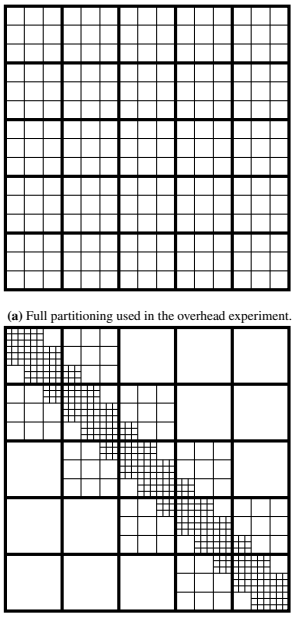


FIGURE 10 Submission cost of computational tasks for DGEMM with all tiles partitioned on INTEL-V100.



(a) Full partitioning used in the overhead experiment.

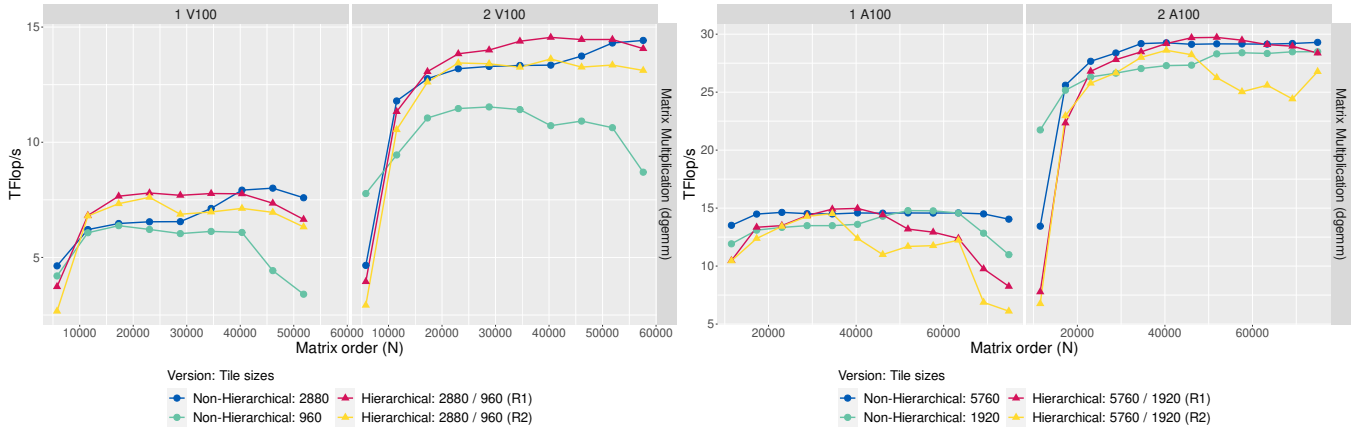
(b) 3 levels diagonal partitioning.

FIGURE 11 Illustration of the matrices layout used in the experiments.

We finally compute the number of tiles for which the partition planning has to be done with respect to R: $\lfloor R \times Nt^2 \rfloor$, where Nt^2 is the total number of tiles of the C matrix. Table 3 summarizes the different values of R we selected for our experiments. $R1$ is computed from the best performances reached for tiled matrix multiplication using all the CPU cores without the GPU devices and vice-versa. $R2$ is computed from the plateau values from Figure 9, using the noisy setting in the CPU case. We evaluate the behavior of the GEMM operation on those matrices, using one and two GPUs (Figures 12a and 12b). On the INTEL-V100 platform, this strategy allows the hierarchical versions to outperform the standard ones on medium matrix sizes by providing the CPU cores of the node with tasks working on smaller tiles. When using large tiles (2880) on bigger matrix sizes, STARPU's scheduler can start affecting more work to the CPUs without penalizing execution time and catches up with the hierarchical versions. On the AMD-A100 platform, the performance ratio between the CPU cores and the GPU devices is so high that the contribution of the CPU cores is marginal. Once the matrices become large enough, the percentage of smaller tiles becomes a disadvantage and negatively impacts the performance. This could be compensated by partitioning fewer tasks as the matrix size increase. However, even in this situation, the hierarchical version using $R1$ can achieve similar performance to standard CHAMELEON. A common trend between both platforms is that the $R1$ ratio provides better results than $R2$, because it was computed using values more representative of the disparity in computing power between CPUs and GPUs for this type of operation. Generally speaking, the dense matrix product is a very regular operation that is rather unfavorable to the use of hierarchical tasks. Despite that, the hierarchical variants have a good behavior and can achieve high performance with a simple partitioning strategy.

Platform	INTEL-V100				AMD-A100			
	$\mathcal{P}_{CPU}$	$\mathcal{P}_{GPU}$	1 GPU	2 GPU	$\mathcal{P}_{CPU}$	$\mathcal{P}_{GPU}$	1 GPU	2 GPU
$R1$	50.9	6.65e3	19%	10%	25.0	14.5e3	10%	5%
$R2$	55.0	4.50e3	26%	15%	28.0	9.5e3	16%	9%

TABLE 3 Performances in GFlop/s used to compute $R1$ and $R2$ for each individual computational unit, and resulting percentages of tiles partitioned for the CPUs in the hierarchical version in Figure 12.

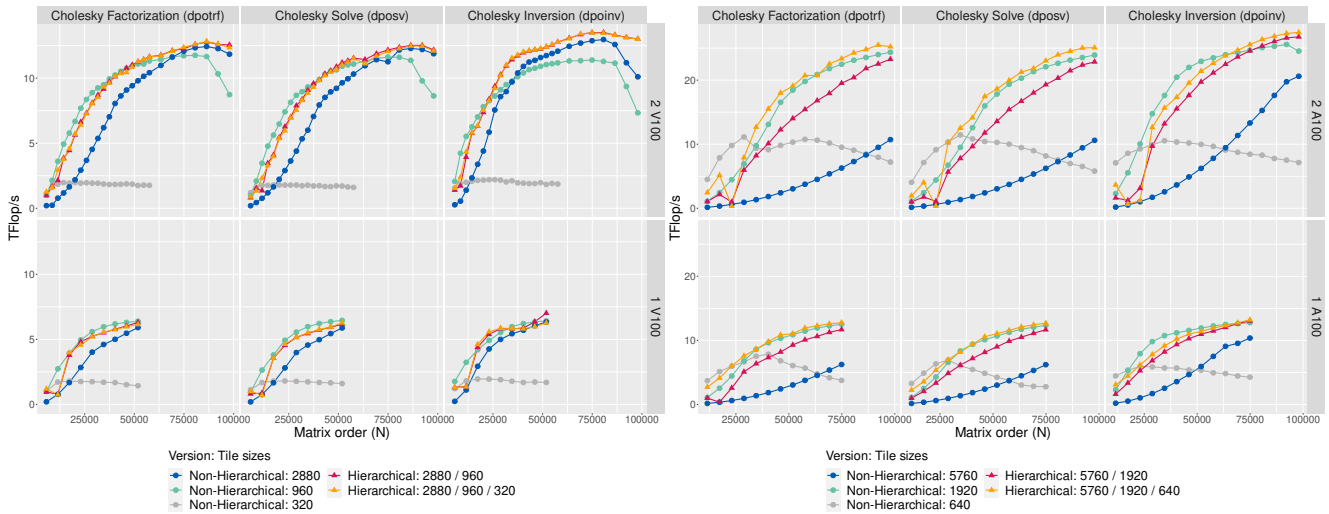


(a) INTEL-V100 platform.

(b) AMD-A100 platform.

FIGURE 12 Comparison of the hierarchical strategy to the original one on the matrix-matrix multiplication (DGEMM) kernel, with a fixed percentage of hierarchical tasks (see Table 3).

We present experimental results in the forthcoming section illustrating the hierarchical tasks' behavior on more advanced dense linear algebra operations to illustrate the flexibility of hierarchical tasks better. First, we consider operations relying on Cholesky decomposition (POTRF): POSV (linear system solving, in this case, of a single vector) and POINV (matrix inversion). These operations have complex task graphs, and in the case of POINV, validate the anti-dependency problem (*WRITE* after *READ*). Furthermore, we provide an experimental evaluation using the LU factorization without partial pivoting (GETRF_nopiv) which has a wider task graph than the Cholesky factorization and for which the criticality of some tasks (mainly those corresponding to the tiles on the diagonal) can strongly affect performance if delayed.



(a) Performance results on the INTEL-V100 platform.

(b) Performance results on the AMD-A100 platform.

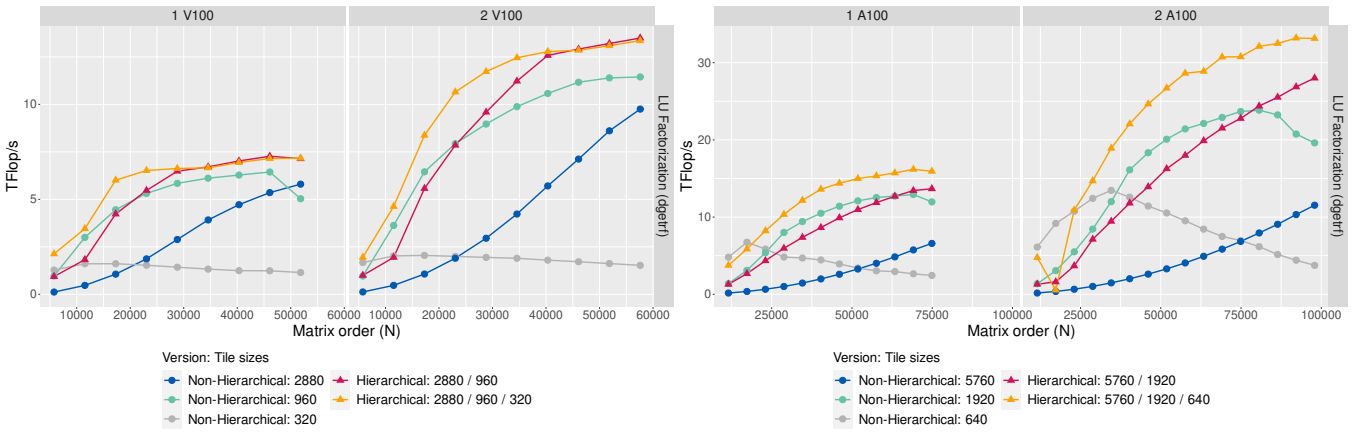
FIGURE 13 Performance evaluation of Cholesky type operations (DPOTRF, DPOSV, DPOINV) with diagonal distribution of the hierarchical tasks.

6.3 | Dense Cholesky factorization-based operations

The partitioning scheme we implemented for the following experiments splits recursively the matrix along the diagonal, sub-diagonal and superdiagonal, to mimic $\mathcal{H}$-Matrices algorithms and use finer granularities on the critical path. An example is shown in Figure 11b. Figure 13 shows the results of operations relying on the Cholesky decomposition: POTRF (actual Cholesky decomposition), POSV (linear system solving, with a single right-hand side) and POINV (matrix inversion). Contrary to the previous DGEMM experiment, this partitioning scheme doesn't simply attempt to supply all resources with the right amount of work. Instead, it tries to improve performance on the critical path of the operation, because CHAMELEON requires all POTRF kernels (which are on the critical path of the factorization and located on the diagonal tiles) on CPU cores. This is illustrated in Figure 13 where we can see that the granularity having the best behavior used for the standard version use smaller tiles than the one relying on the hierarchical tasks on both platforms. Thanks to hierarchical tasks, we can partition the tiles along the diagonal and split those large tasks into sub-graphs with a smaller granularity, allowing for better CPU utilization on the critical path. For all the kernels and platforms, enabling hierarchical tasks enhances performance for most matrix sizes. We can also observe that the regular version can catch up for huge matrix sizes. The sudden drop observed at the end of some non-hierarchical curves is explained by a conflict between the STARPU scheduler data prefetching and eviction in GPU memory, as explained in [36]. The experimental results illustrate the interest in hierarchical tasks for tackling the granularity problem of heterogeneous architectures. This highlights the fact that hierarchical tasks can find a better trade-off to fully exploit the whole platform: each resource is used with a task granularity adapted to its characteristics.

6.4 | Dense LU factorization without pivoting

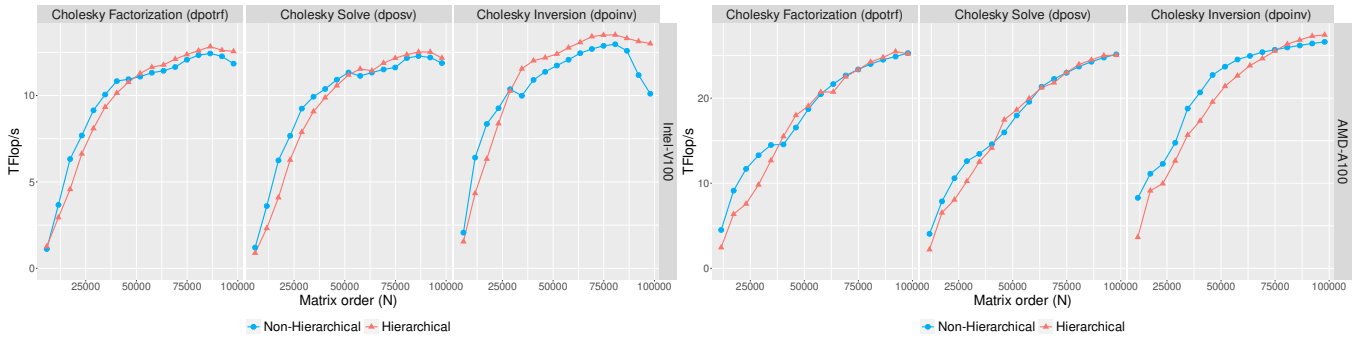
We provide in Figure 14 an experimental evaluation of hierarchical tasks to enhance the dense LU factorization without pivoting (DGETRF_nopiv). This dense linear algebra kernel exhibits a high amount of parallelism. It requires the tiles on the diagonal to be processed as fast a possible to release many dependencies. Once again, we can observe that hierarchical tasks improve performance on both platforms. We can also observe that the non-hierarchical version of CHAMELEON achieves lower performance because of the critical tasks on the diagonal: since the matrix partitioning is uniform, the best tiling uses a large granularity to exploit the GPU devices fully. Finally, we can observe that the gap between the hierarchical tasks and standard Chameleon is more significant on the more heterogeneous AMD-A100 platform. This is mainly because, on this platform, large tiles (5760) are needed to exploit the potential of the A100 devices fully. Thus, not using hierarchical tasks to adapt the granularity for each device penalizes performance. All in all, the experimental results illustrate the flexibility of the hierarchical task model and its ability to find a better tradeoff to tackle the granularity problem for heterogeneous platforms.



(a) INTEL-V100 platform.

(b) AMD-A100 platform.

FIGURE 14 Performance evaluation of the LU decomposition (DGETRF_nopiv) with diagonal distribution of the hierarchical tasks.



(a) INTEL-V100 platform.

(b) AMD-A100 platform.

FIGURE 15 Performance evaluation of Cholesky type operations (DPOTRF, DPOSV, DPOINV) with diagonal distribution of the hierarchical tasks. The non-hierarchical variant relies on multiple streams. Each point of the plot is the best performance obtained for each variant considering tile size, number of streams, and partitioning scheme (for the hierarchical variant).

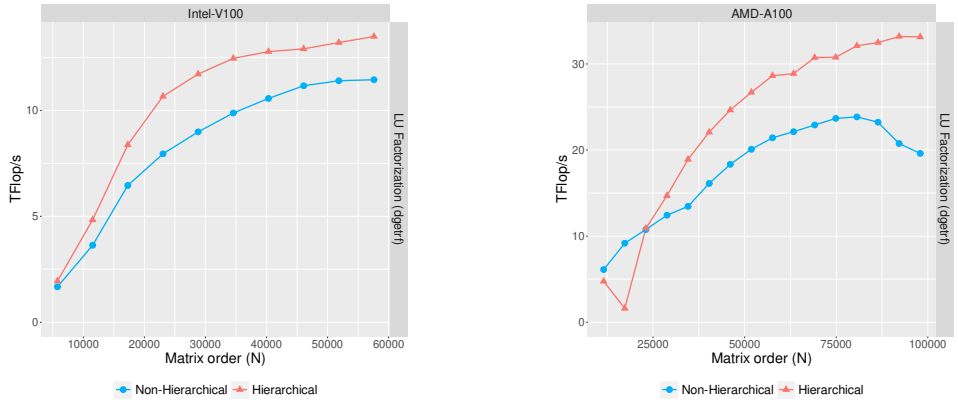
6.5 | Going further

In the previous sections, we fairly compared the standard STARPU task paradigm and the hierarchical one. The results showed the benefits of relying on hierarchical tasks from the performance point of view. Still, we should compare the hierarchical variant with the standard one when using multiple GPU streams to be conclusive. Indeed, GPU streams are, in a sense, a way to achieve task parallelism within the GPU device: kernels submitted on different streams can be executed concurrently on the GPU device. This may enhance GPU occupancy and thus improve performance. In the context of our experimental testbed, using GPU streams will allow the baseline variant to find a better trade-off between the granularities adapted to CPU cores and GPU devices. It will thus be possible to use smaller tile sizes while relying on multiple streams to increase the device efficiency while shortening the critical path (the larger the tile size, the longer the critical path). We report in Figures 15 and 16 the experimental results obtained when relying on streams for the non-hierarchical variant on both platforms using all resources. For both figures, each dot in the plot corresponds to the best performance obtained for the variant while considering different tile sizes (320, 640, 960, 1920, 2880, 5760), number of streams (1, 2, 4, 8) and partitioning scheme for the hierarchical variant (1 and 2 levels of hierarchy). It is important to note that we did not use streams for the hierarchical variant. First, we can observe that the use of GPU streams enhanced the performance of the non-hierarchical variant. We can also see in Figure 15 that both variants have a very close behavior for all three considered kernels. Moreover, the baseline approach relying on streams is slightly better for small to medium sizes, and the hierarchical variant is the best performing one for medium to large matrix sizes.

From the LU factorization without pivoting point of view, we can see in Figure 16 the behavior of both variants. As for the Cholesky-based kernels, the non-hierarchical variant's behavior was improved by using streams. However, for this kernel, the hierarchical variant outperforms the non-hierarchical one. This is mainly because even when using streams, the granularity used by the non-hierarchical variant does not allow exploiting CPU cores fully, affecting the progression on the critical path. These final results illustrate the interest in the hierarchical task paradigm from the performance point of view. The flexibility for choosing task granularities depending on the targeted resources allows one to exploit the underlying hardware's potential fully.

7 | CONCLUSION

The growing complexity of high performance computing platforms has led to the design of highly efficient runtime systems, which aim to abstract the underlying platform. These runtime systems use various programming models to express computations. One of the most widely used models is the so-called sequential task flow model, where the task graph corresponding to the computation is submitted sequentially by a single entity. The runtime system then infers dependencies based on the data access-modes provided by programmers at task submission. While the sequential task flow model has been shown to be effective for highly regular applications, it has limitations when computations become irregular or the task graph cannot be fully known at submission time. Additionally, the entirely sequential submission process can result in significant overhead for complex applications.



(a) INTEL-V100 platform. **(b)** AMD-A100 platform.

FIGURE 16 Performance evaluation of the LU decomposition (DGETRF_nopiv) with diagonal distribution of the hierarchical tasks. The non-hierarchical variant relies on multiple streams. Each point of the plot is the best performance obtained for each variant considering tile size, number of streams, and partitioning scheme (for the hierarchical variant).

In this paper, we propose an extension of the sequential task flow model and an upgrade of the underlying runtime system to overcome these limitations. Our approach introduces a new type of task, called the hierarchical task, which can submit a new sub-graph of tasks at runtime. This allows for the possibility of parallel submission and delays the submission of parts of the task graph, enabling dynamic implementation selection. Additionally, to ensure that the parallel submission process still produces a valid directed acyclic graph (DAG), we introduce a new automatic data manager that handles data layout dynamically by submitting data management tasks at the appropriate time. We also demonstrate the correctness of our model and present a set of results that illustrate the potential of the hierarchical task model.

We plan to expand upon this work in several ways. First, we need to examine the hierarchical tasks from a scheduling perspective. We aim to answer the question of when a hierarchical task should be processed. The solution to this problem must take into account the number of tasks in the system and the work assigned to each resource. We will also consider the problem of choosing which sub-graph to submit when a hierarchical task is processed. To select the most appropriate implementation, we require advanced performance models that must be designed. Finally, the task graph resulting from a hierarchical task must be scheduled efficiently. Overall, addressing all of these issues will improve the behavior of the runtime system when using hierarchical tasks. More broadly, we aim to investigate how this model can be used to implement advanced irregular algorithms, such as linear algebra solvers based on low-rank approximation, sparse solvers and iterative solvers.

In the long-term, we believe that extending the hierarchical task model to the distributed memory context will be an elegant solution to the scalability of task-based runtime systems.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are available from the corresponding author upon request.

CONFLICT OF INTEREST STATEMENT

The authors declare no potential conflict of interests.

ACKNOWLEDGEMENT

This paper is an extended version of [37] presented at the HeteroPar'22 workshop. This work is supported by the french ANR through the Solharis project under the grant (ANR-19-CE46-0009). 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. Augonnet C, Thibault S, Namyst R, Wacrenier PA. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurr. Comput. : Pract. Exper.* 2011; 23: 187–198.

2. Agullo E, Augonnet C, Dongarra J, et al. A Hybridization Methodology for High-Performance Linear Algebra Software for GPUs. *GPU Computing Gems, Jade Edition* 2011; 2: 473-484.

3. Buttari A, Langou J, Kurzak J, Dongarra J. A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures.; 2007.

4. Bosilca G, Bouteiller A, Danalis A, et al. Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA. In: *IEEE IPDPS Workshops and Phd Forum*; 2011: 1432-1441.

5. Valero-Lara P, Catalán S, Martorell X, Usui T, Labarta J. sLASs: A fully automatic auto-tuned linear algebra library based on OpenMP extensions implemented in OmpSs. *J. of Parallel and Distributed Computing* 2020; 138: 153-171.

6. Cojean T, Guermouche A, Hugo A, Namyst R, Wacrenier P. Resource aggregation for task-based Cholesky Factorization on top of modern architectures. *Parallel Comput.* 2019; 83: 73–92.

7. Maroñas M, Sala K, Mateo S, Ayguadé E, Beltran V. Worksharing Tasks: An Efficient Way to Exploit Irregular and Fine-Grained Loop Parallelism. In: *Proc. of HiPC'19*; 2019: 383-394.

8. Wu W, Bouteiller A, Bosilca G, Faverge M, Dongarra J. Hierarchical DAG scheduling for Hybrid Distributed Systems. In: *Proc. of IPDPS'15*; 2015: 156–165.

9. Carratala-Saez R, Christophersen S, Aliaga JI, Beltran V, Borm S, Quintana-Orti ES. Exploiting nested task-parallelism in the H-LU factorization. *Journal of Computational Science* 2019; 33: 20-33.

10. Akbudak K, Ltaief H, Mikhalev A, Keyes D. Tile Low Rank Cholesky Factorization for Climate/Weather Modeling Applications on Manycore Architectures. In: Kunkel JM, Yokota R, Balaji P, Keyes D., eds. *High Performance Computing*Springer International Publishing; 2017; Cham: 22–40.

11. Carratalá-Sáez R, Faverge M, Pichon G, Sylvand G, Quintana-Ortí ES. Tiled Algorithms for Efficient Task-Parallel $\mathcal{H}$-Matrix Solvers. In: *2020 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2020, New Orleans, LA, USA, May 18-22, 2020*IEEE; 2020: 757–766.

12. Augonnet C, Goudin D, Kuhn M, Lacoste X, Namyst R, Ramet P. A Hierarchical Fast Direct Solver for Distributed Memory Machines with Manycore Nodes. tech. rep., ; : 2019.

13. Börm S, Christophersen S, Kriemann R. Semi-Automatic Task Graph Construction for H-Matrix Arithmetic. *CoRR* 2019; abs/1911.07531.

14. Bauer M, Treichler S, Slaughter E, Aiken A. Legion: expressing locality and independence with logical regions. In: *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*; 2012: 66.

15. Faverge M, Herrmann J, Langou J, Lowery B, Robert Y, Dongarra J. Designing LU-QR hybrid solvers for performance and stability. In: *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2014)*; 2014; Phoenix, United States.

16. Allen R, Kennedy K. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann . 2002.

17. Cosnard M, Jeannot E, Yang T. SLC: Symbolic scheduling for executing parameterized task graphs on multiprocessors. In: *Proc. of ICPP'99*; 1999: 413-421.

18. Bosilca G, Bouteiller A, Danalis A, Faverge M, Hérault T, Dongarra JJ. PaRSEC: Exploiting Heterogeneity to Enhance Scalability. *Computing in Science and Engineering* 2013; 15(6): 36–45.

19. Frigo M, Leiserson C, Randall K. The implementation of the Cilk-5 multithreaded language. *SIGPLAN Not.* 1998; 33(5): 212–223.

20. Reinders J. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly . 2007.

21. Hartley TDR, Saule E, Çatalyürek ÜV. Improving performance of adaptive component-based dataflow middleware. *Parallel Computing* 2012; 38(6-7): 289-309.

22. Kunzman DM, Kalé LV. Programming heterogeneous clusters with accelerators using object-based programming. *Scientific Programming* 2011; 19(1): 47-62.

23. Acun B, Gupta A, Jain N, et al. Parallel Programming with Migratable Objects: Charm++ in Practice. In: *Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis*SC '14. IEEE Press; 2014: 647–658.

24. Kaiser H, Diehl P, Lemoine AS, et al. HPX - The C++ Standard Library for Parallelism and Concurrency. *Journal of Open Source Software* 2020; 5(53): 2352. doi: 10.21105/joss.02352

25. Hermann E, Raffin B, Faure F, Gautier T, Allard J. Multi-GPU and Multi-CPU Parallelization for Interactive Physics Simulations. In: *Euro-Par 2010 - Parallel Processing*. 6272. Springer. 2010 (pp. 235-246).

26. Ayguadé E, Badia R, Igual F, Labarta J, Mayo R, Quintana-Ortí E. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In: *Euro-Par 2009*; 2009: 851–862.

27. Hoque R, Herault T, Bosilca G, Dongarra J. Dynamic Task Discovery in PaRSEC: A Data-Flow Task-Based Runtime. In: *Proc. of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*ScalA '17. ; 2017.

28. Álvarez D, Sala K, Maroñas M, Roca A, Beltran V. Advanced Synchronization Techniques for Task-Based Runtime Systems. In: *Proc. of PPoPP '21*; 2021: 334–347.

29. Elshazly H, Lordan F, Ejarque J, Badia RM. Accelerated execution via eager-release of dependencies in task-based workflows. *The International Journal of High Performance Computing Applications* 2021; 35(4): 325-343.

30. Perez JM, Beltran V, Labarta J, Ayguadé E. Improving the Integration of Task Nesting and Dependencies in OpenMP. In: *Proc. of IPDPS'17*; 2017: 809-818.

31. Gautier T, Lima JVF, Maillard N, Raffin B. XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures. In: *Proc. of IPDPS'13*; 2013: 1299–1308.

32. Huang TW, Lin DL, Lin CX, Lin Y. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. *IEEE Transactions on Parallel and Distributed Systems* 2021: 1-1.

33. Kim J, Lee S, Johnston B, Vetter JS. IRIS: A Portable Runtime System Exploiting Multiple Heterogeneous Programming Systems. In: *Proc. of HPEC'21*; 2021: 1-8.

34. Allen R, Kennedy K. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morga!n Kaufmann . 2002.

35. Topcuouglu H, Hariri S, Wu MY. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems* 2002; 13(3): 260-274. doi: 10.1109/71.993206

36. Stanisic L, Thibault S, Legrand A, Videau B, Méhaut JF. Modeling and Simulation of a Dynamic Task-Based Runtime System for Heterogeneous Multi-Core Architectures. In: *Euro-par 2014*; 2014: 50-62

37. Faverge M, Furmento N, Guermouche A, et al. Programming Heterogeneous Architectures Using Hierarchical Tasks. In: *Euro-Par 2022: Parallel Processings, International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms*; 2022.