Programming Heterogeneous Architectures Using Hierarchical Tasks

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

Programming Heterogeneous Architectures Using Hierarchical Tasks

M. Faverge, N. Furmento, A. Guermouche, G. Lucas, R. Namyst, S. Thibault, P.-A. Wacrenier

RESEARCH REPORT
N° 9466
March 2022
Project-Team Storm and Hiepacs

ISSN 0249-6399 ISRN INRIA/RR--9466--FR+ENG

Programming Heterogeneous Architectures Using Hierarchical Tasks

M. Faverge, N. Furmento, A. Guermouche, G. Lucas, R. Namyst, S. Thibault, P.-A. Wacrenier

Project-Team Storm and Hiepacs

Research Report n° 9466 — March 2022 — 20 pages

Abstract: Task-based systems have gained popularity because of their promise of exploiting the computational power of complex heterogeneous systems. A common programming model is the so-called *Sequential Task Flow* (STF) model, which, unfortunately, has the intrinsic limitation of supporting static task graphs only. This leads to potential submission overhead and to a static task graph which is not necessarily adapted for execution on heterogeneous systems. A standard approach is to find a trade-off between the granularity needed by accelerator devices and the one required by CPU cores to achieve performance. To address these problems, we extend the STF model in the StarPU runtime system to enable tasks subgraphs at runtime. We refer to these tasks as *hierarchical tasks*. This approach allows for a more dynamic task graph. This extended model combined with an automatic data manager allows to dynamically adapt the granularity to meet the optimal size of the targeted computing resource. We show that the hierarchical task model is correct and we provide an early evaluation on shared memory heterogeneous systems, using the Chameleon dense linear algebra library.

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

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

Programmation des Architectures Hétérogènes à l'Aide de Tâches Hiérarchiques

Résumé : Les systèmes à base de tâches ont gagné en popularité du fait de leur capacité à exploiter pleinement la puissance de calcul des architectures hétérogènes complexes. Un modèle de programmation courant est le modèle de *soumission séquentielle de tâches (Sequential Task Flow, STF)* qui malheureusement ne peut manipuler que des graphes de tâches statiques. Ceci conduit potentiellement à un surcoût lors de la soumission, et le graphe de tâches statique n'est pas nécessairement adapté pour s'exécuter sur un système hétérogène. Une solution standard consiste à trouver un compromis entre la granularité permettant d'exploiter la puissance des accélérateurs et celle nécessaire à la bonne performance des CPUs. Pour répondre à ces problèmes, nous proposons d'étendre le modèle STF fourni par le support d'exécution StarPU en y ajoutant la possibilité de transformer certaines tâches en sous-graphe durant l'exécution. Nous appelons ces tâches des *tâches hiérarchiques*. Cette approche permet d'exprimer des graphes de tâche plus dynamiques. En combinant ce nouveau modèle à un gestionnaire automatique des données, il est possible d'adapter dynamiquement la granularité pour fournir une taille optimale aux différentes ressources de calcul ciblées. Nous montrons dans ce rapport que le modèle des tâches hiérarchiques est valide ainsi qu'une première évaluation de ses performances en utilisant la bibliothèque d'algèbre linéaire dense Chameleon.

Mots-clés : Multi-cœurs; Accélérateurs; GPU; Calcul hétérogène; Graphe de tâches; Modèle de programmation; Support d'exécution; Algèbre linéaire dense

1 Introduction

Due to the recent evolution of High Performance Computing systems toward heterogeneous multicore architectures, many research efforts have recently been devoted to the design of runtime systems that support portable programming techniques and tools to exploit the complex hardware. Runtime systems with mature implementations are now available both for regular homogeneous multicore systems and for complex heterogeneous systems. Standards like OpenMP (since version 4.0) support the task-based paradigm with applications represented as direct acyclic graph (DAG) of tasks.

However, the task-based paradigm poses several problems when trying to exploit heterogeneous platforms efficiently. First, the computing resources of heterogeneous platforms have diverse characteristics and requirements. For instance, GPU devices typically favor large data sets, whereas conventional CPU cores reach peak performance with fine-grain kernels working on a reduced memory footprint. Additionally systems usually have a much larger number of CPU units than GPUs, having more small tasks may therefore be important to increase performance. Several efforts have tried to tackle this problem either by finding the best trade-off between the optimal granularity of each device [1–4], or by aggregating CPU cores to process a task which was meant to be executed by an accelerator like a GPU [5, 6]. Alternatively, some preliminary work has considered splitting the tasks on CPU cores [7]. Even though these approaches are efficient in specific contexts like dense linear algebra, they suffer from the fact that the task graph is static in the sense that it is not possible to select an alternative granularity for a given operation at runtime. As an example, when designing linear algebra solvers based on low-rank approximation algorithms, it is almost impossible to statically predict the right DAG to ensure good numerical accuracy [8–12].

In a more general context, most of the programming models exhibited by modern task-based runtime systems suffer from the lack of dynamism of task-graph generation. Some programming models such as [7, 13, 14] support just-in-time DAG submission but the generation either still follows static rules or requires a huge programming effort.

Another major limitation of using task-based runtime systems is the overhead imposed by the runtime system itself to manage large applications. This is strongly linked to the size of the internal DAG representation as the tasks are submitted. Most of the runtime systems relying on the STF model [15] (e.g. OpenMP, StarSS, StarPU) build this representation at submission time and may be penalized 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 the dependencies never build this representation and are less penalized but tend to be more tricky to use. An example of such a programming model is the parameterized task-graph (PTG) [16] used by PaRSEC [17]. Finally, concerning the sequential task flow model, the submission of the tasks 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, namely the *hierarchical tasks*, which are tasks that can transform themselves into a new task-graph dynamically at runtime. Programmers only need to provide hints stating which tasks can be transformed into a hierarchical task. The runtime system can then delay the submission of parts of the task graph to support dynamic implementation selection, to parallelize the task insertion process, and to strongly reduce the number of tasks in the runtime system. The approach we propose is similar to what is done in OpenMP for nested task-based parallelization scheme. However, we extend this approach to handle heterogeneous platforms while expressing fine grain dependencies. This is possible thanks to an advanced data manager which can dynamically and asynchronously change the data layout.

The proposed model associated to these *hierarchical tasks* addresses the issues mentioned

above: 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, in this paper, we focus on an initial implementation for shared memory heterogeneous architectures.

The contribution of the paper is two-fold : 1) We present an advanced data management engine which supports asynchronous data layout modification, 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 first present a general context where we present both the sequential task flow model and the StarPU runtime system. Afterwards, we describe the proposed advanced data manager which is a key feature for the implementation of 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 of the approach when performing dense linear algebra operations implemented within the Chameleon library [1].

2 Related Work

A lot of attention has been given in the past years to the design of advanced runtime systems targeting modern heterogeneous architectures. Most of these efforts fall within the task-based paradigm which is a natural way for programming scientific workflow (*e.g.* Cilk [18], OpenMP or Intel TBB [19] for multicore machines, APC [20], Charm++ [21,22], HPX [23], KAAPI/XKAAPI [24], Legion [13], PaRSEC [17], StarPU [25] or StarSS [26] for heterogeneous configurations).

The common point between these runtime systems is the fact 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. Several approaches are used to build the task graph. For instance, most of the previously cited runtime systems rely on the so-called *Sequential Task-Flow* model (e.g. OpenMP, StarSS, StarPU) to build the task graph: by relying on data access-modes and a sequential submission order, dependencies between tasks can be inferred through data dependency analysis [15]. On the other hand, runtime systems such as PaRSEC are based on the parameterized task-graph programming model (PTG) [16] where the task graph is unrolled at runtime using a high-level description of the dataflow corresponding to the computations. Alternatively, other runtime systems use a different paradigm for expressing computations. Legion describes logical regions of data which are used to express the data flow and dependencies between tasks. All these programming models differ with respect to usability and the overhead induced on the underlying runtime system. For instance, the sequential task flow paradigm has a natural way of inferring dependencies since the programmers only have to provide the sequential implementation of their application and then add data access-modes at the cost of a higher runtime system overhead. 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 while the runtime system overhead remains small [27].

Several efforts have targeted the problem of reducing the overhead of task-based runtime systems (mainly for those based on the sequential task flow model) or enhancing the amount of parallelism provided by such systems. In [28], authors analyze the limiting factors in the scalability of a task-based runtime system and propose individual solutions for each of the listed challenges, including a wait-free dependency system and a scalable scheduler design based on delegation instead of work-stealing. Alternative approaches consider advanced dependency man-

agement. For instance, in [29] authors propose an eager approach for releasing data dependencies. Following this approach, the execution of tasks will not be delayed until their predecessor tasks completely finish their execution. Instead, tasks will be launched for execution as soon as their data requirements are available. Alternatively, [6] introduces worksharing tasks. These are tasks that internally leverage worksharing techniques to exploit fine-grained structured loop-based parallelism without requiring a barrier. However the closest contribution to our proposition from the perspective of task dependencies was introduced in [30] as the concept of *weak dependencies*. It is an extension of the OpenMP model which enhances the dataflow model of OpenMP by supporting 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 memory consistency is not ensured by the underlying hardware.

From the point of view of advanced/dynamic task management and generation, several efforts have been made to allow task-based runtime systems to have a more dynamic expressivity. In TaskFlow [31], advanced tasking schemes are introduced including dynamic, composable and conditional tasking. Dynamic tasking, in particular, allows to dynamically generate a sub-DAG from a given task. However, a synchronization is added at the end of each hierarchical task to ease the dependencies management. Furthermore, data management must be handled by the programmers: it is their responsibility to change the layout of data when needed. In [32], authors introduce the IRIS runtime which has the ability to perform dynamic task partitioning (either performed by the user or automatically via a polyhedral compiler). However, no details were provided to illustrate how dependencies are handled in this context. Finally, an advanced runtime system supporting hierarchical tasks in the context of low-rank linear algebra solvers is presented in [11]. In this work, hierarchical tasks are introduced and the dependencies are expressed at the finest level. The correctness of the produced DAG is considered through automatic extra dependencies. However, the data management is straightforward since the partitioning of data is performed statically at the beginning of the execution.

3 Background

We remind in the subsection below the basics of the sequential task flow model and the corresponding sequential consistency. We then provide a brief presentation of our target runtime system StarPU.

3.1 The Sequential Task Flow Model

The *Sequential Task Flow* (STF) programming model consists in fully relying on the *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 along with its dependencies which are automatically computed through data dependency analysis [33]. The actual execution of the task is then postponed until its dependencies are satisfied. This paradigm is also sometimes referred to as *Superscalar* since it mimics the operation of superscalar processors where instructions are issued sequentially from a single stream but can actually be executed in a different order and, possibly, in parallel depending on their mutual dependencies.

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 well as their access mode (Read, Write or Read/Write) are also specified. Because task `G` accesses data `x` after task `F` has accessed it in

```
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()
```

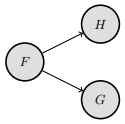


Figure 1: Pseudo-code for a dummy sequential algorithm (*left*), corresponding STF version (*center*) and subsequent DAG (*right*).

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 in charge of submitting the tasks; we refer to this thread as the *master thread*. The execution of tasks is instead achieved by *worker threads*. The function called at the end of the STF pseudo-code is simply a barrier that prevents the master thread from continuing until all of 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 [25] 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 firstly that tasks can have several implementations, for some or each of the various heterogeneous processing units available in the machine, and secondly that necessary data transfers to these processing units are handled transparently by the runtime system. 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 at the same time 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 in creating a set of queues, associating them with the different processing units, and defining the code that is triggered each time a new task becomes ready to be executed, or each time 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 [34].

4 Automatic Data Management

Data handling is at the heart of StarPU both to automatically infer dependencies between tasks in the STF model and to automatically manage data transfers between the different memory banks of a distributed/heterogeneous system. In order to benefit from these automations, 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 a registered data. Handles

along an access mode (read-only, read-write, ...) are used as task parameters. It is mandatory for a task to access a 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 subdata parts associated with new subhandles. Indeed, instead of registering all data subsets 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 observe that a same data can be designated simultaneously by several handles. This may be very convenient when programmers want to use a data with several views. Moreover, data in read-only access mode can advantageously be accessed simultaneously at different partitioning levels by several tasks running on different devices. However, when a data is accessed in write access mode, this access must be exclusive for coherency purpose. This property is ensured by StarPU when a single partitioning is used for a 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 to ensure they cannot be used to access their underlying data, and to unpartition subhandles back into the main handle to gather the subdata.

However, users feedback showed that maintaining consistency between different partitions is a difficult task. 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 that we made in StarPU to allow these automatic data management. First, programmers need to define the partitioning scheme through the *plan* operation which aims at declaring the partitioning to StarPU, and can be seen as the declaration of a new set of subhandles. 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 subhandles 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 ranging from line 12 to line 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 then first initialized through its root handle, then modified using the vertical partitioning, and finally checks are performed in both horizontal and vertical stripes.

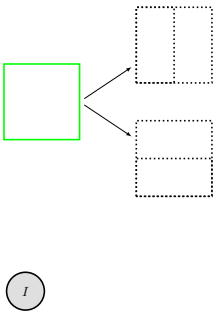
At runtime, StarPU will introduce coherency synchronization: when a task is ready to be executed, StarPU must ensure that the partition associated with each handle it uses is valid. If a data is accessed in read-only mode, StarPU will allow different partitioning to coexist. As soon as a data is accessed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

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 actually 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 that these handles share a common ancestor (the whole matrix) and thus it will insert as needed the unpartition/partition

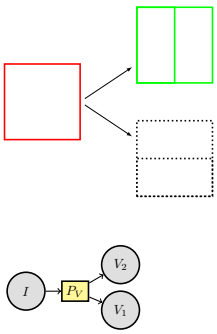
```
#define PARTS 2
void do(int matrix[NX][NY])
{
  starpu_data_handle_t handle;
  starpu_data_handle_t v_handle[PARTS];
  starpu_data_handle_t h_handle[PARTS];

  /* Declare the whole matrix to StarPU */
  starpu_matrix_data_register(&handle, STARPU_MAIN_RAM,
    (uintptr_t)matrix, NX, NX, NY, sizeof(matrix[0][0]));

  /* Partition the matrix in PARTS vertical slices */
  struct starpu_data_filter vertF = {
    .filter_func = starpu_matrix_filter_block,
    .nchildren = PARTS
  };
  starpu_data_partition_plan(handle, &vertF, v_handle);

  /* Partition the matrix in PARTS horizontal slices */
  struct starpu_data_filter horF = {
    .filter_func = starpu_matrix_filter_vertical_block,
    .nchildren = PARTS
  };
  starpu_data_partition_plan(handle, &horF, h_handle);

  /* Fill the matrix */
  starpu_task_insert(&initialize, STARPU_W, handle, 0);

  /* Modify the values via the vertical slices */
  for (unsigned i = 0; i < PARTS; i++)
    starpu_task_insert(&modify, STARPU_RW, v_handle[i], 0);

  /* check the values via both horizontal and vertical slices */
  for (unsigned i = 0; i < PARTS; i++)
    starpu_task_insert(&check, STARPU_R, h_handle[i], 0);
  for (unsigned i = 0; i < PARTS; i++)
    starpu_task_insert(&check, STARPU_R, v_handle[i], 0);

  /*  Unregister data from StarPU. */
  starpu_data_partition_clean(handle, PARTS, v_handle);
  starpu_data_partition_clean(handle, PARTS, h_handle);
  starpu_data_unregister(handle);
}
```

Figure 2: Automatic unpartitioning example.

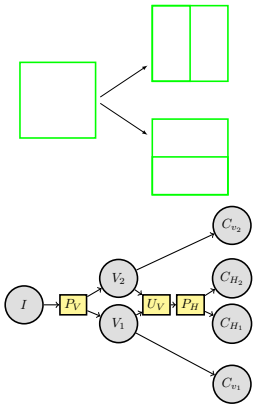
tasks to make the data available to the tasks using the horizontal layout. This is illustrated in Figure 3c where the U_v and P_h tasks are inserted, making the tasks using the horizontal layout depend on them. Finally, when the partition needs to be cleaned, the final unpartition task is inserted (see Figure 3d).



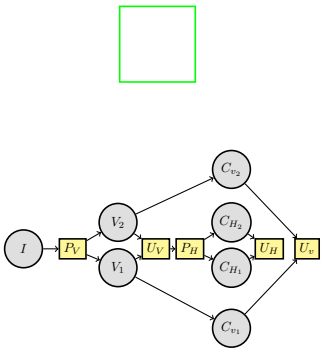
(a) Root handle initialization and partition plan.



(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*.

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 subpart of it), or *read-only-active* (one can only read from the piece of data or a subpart 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 subpieces of data), or *read-only-partitioned* (one can read the piece of data or subpieces of data) ; when it is partitioned, its children subhandles in the hierarchy are active. 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 subhandles which were previously written to. In the case of the transition from Figure 3b to Figure 3c, StarPU indeed had to submit the unpartition task 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

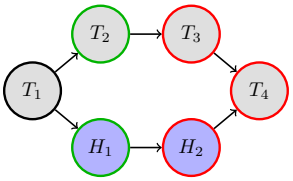
In order to extend the sequential task flow model with hierarchical tasks we first need to define what exactly a hierarchical task is. In a formal way, a hierarchical task is simply a regular task that can, at runtime, submit a sub-DAG instead of performing actual computations. Processing a hierarchical task consists in the submission of its corresponding task subgraph, its outgoing dependencies can be released at the end of that submission process. Furthermore, in order to ensure the portability with respect to 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 amount 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).

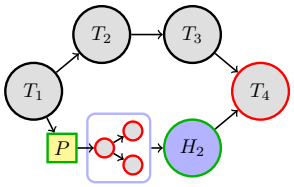
Properties 1 and 2 ensure a general programming interface with no limitation from the programmer's point of view. Property 3 is related to the data layout management. It is necessary for the heterogeneous case where memory is not shared between all the computing resources meaning that explicit data movements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

We present in Figure 4a 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 4a 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 subDAG is submitted, resulting to Figure 4b. 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 4c) the dependencies between the resulting submitted tasks are inferred by the runtime system at

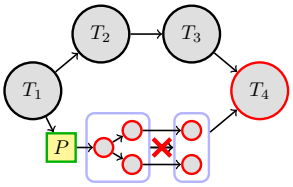
the deepest level of the hierarchy. This will allow for a good pipelining of the execution of the two task graphs resulting from H_1 and H_2.



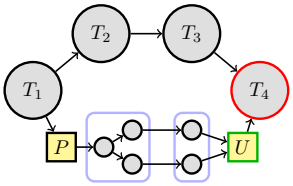
(a) Initial DAG.



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



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

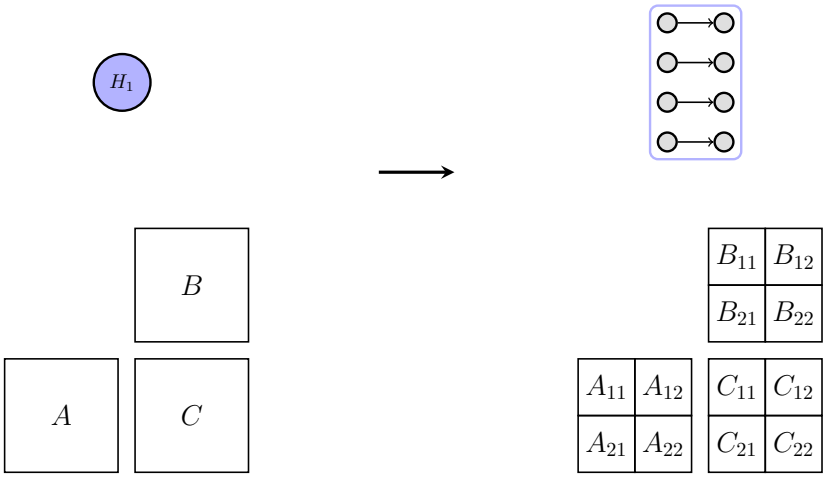


(d) Automatic insertion of the unpartition task U.

Figure 4: Example of a DAG with 2 hierarchical tasks and 4 regular tasks.

We now have to consider how the data coherency will be achieved between the DAG and the subDAGs. Introducing hierarchical tasks in a task-based runtime system requires to change the granularity of data dynamically at runtime each time a hierarchical task has to be executed. Indeed, in order to be able to express parallelism within the hierarchical task, the input data of the task 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 5 where a DAG composed of a single hierarchical task (implementing a matrix product (GEMM)) is provided. We can see that when the task H_1 is processed, it submits its corresponding DAG. However, in order 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.

Thus the general approach is to have a data management operation (either partitioning or unpartitioning) preceding each individual task (resp. hierarchical task) when needed. The approach we propose is to automatically insert a data management task ahead of a task requiring data which are not in the correct layout by relying on the data manager introduced in Section 4. Coming back to Figure 4, we can see, in Figure 4b, the insertion of the partitioning task P ahead of the subgraph 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 execution. 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 however 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 the 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 subgraphs produced by H_1 and H_2 since they share the same data layout. Finally, it is important to emphasize that hierarchical tasks are processed 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



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

(b) Data partitioning and the corresponding task graph.

Figure 5: Example of a hierarchical task corresponding to a matrix product operation $C = C + A * B$ and its corresponding DAG.

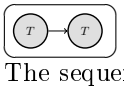
of the actual computations.

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.

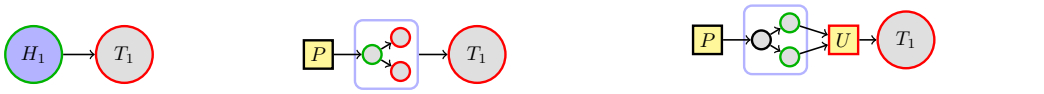
5.1 Ensuring the Correctness of the DAG

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 tasks and hierarchical ones. First of all, 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 to ensure that the DAG produced after the processing of all the hierarchical tasks is correct with respect to a sequential execution (at the end we want the DAG to be equivalent to the one using a single submission thread). This concern is mainly due to the fact that when relying on hierarchical tasks, the submission can now be done in parallel while in the sequential task flow model, the submission is done by a single entity. The problem is to answer the following question : How to ensure the correctness of the DAG?

To provide a solution to this problem we need to show that the dependencies respect the sequential task flow model. We discuss four simple scenarios which are building blocks for any general DAG to show its correctness.

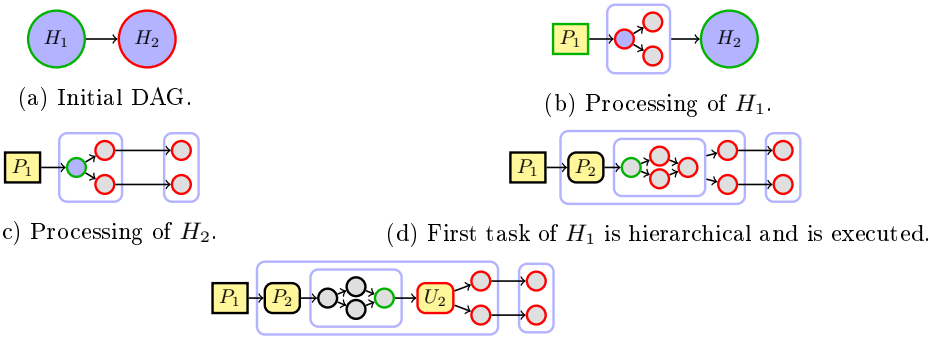


This is a straightforward scenario that corresponds to the sequential task flow model. The sequential consistency together with the submission order of the tasks will ensure the correctness of the DAG.



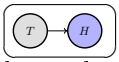
(a) Initial DAG. (b) Processing of H_1. (c) Insertion of the unpartition task when T_1 is ready.

Figure 6: Example of a scenario where a task follows a hierarchical task.

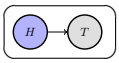


(a) Initial DAG. (b) Processing of H_1. (c) Processing of H_2. (d) First task of H_1 is hierarchical and is executed. (e) Automatic insertion of the U_2 task when data needs to be accessed.

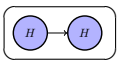
Figure 7: Example of a chain of two hierarchical tasks.



This represents the case where in the original DAG, a regular task is followed by a hierarchical one which 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, which ensures the correctness of the dependencies.



This represents the case where a regular task is preceded by a hierarchical one in the original DAG. An example of this scenario is provided in Figure 6. 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 6). This may violate the order required by the sequential consistency. However, in this specific situation, the hierarchical task has changed the data layout before it starts its execution (see Figure 6b). Thus the task following the hierarchical task (T_1 in Figure 6) 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 execution of the hierarchical task. This corresponds to the configuration depicted in Figure 6c where the unpartition task U is inserted in such a way that T_1 depends on U which itself 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 central role of the data manager which has been discussed in Section 4.



This represents the case where in the original DAG, a hierarchical task is followed by another one which 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 7 where initially two hierarchical tasks H_1 and H_2 are submitted (see Figure 7a). Then H_1 is processed (see Figure 7b). 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. Afterwards, H_2 is processed (see Figure 7c) 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 7d 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 processed. We can also see the partitioning task P_2 which was automatically inserted by the data manager to further partition the data. 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 requires data (see Figure 7e). This automatic insertion U_2 is important 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 [1]. To do so, we extended the matrix descriptors in order to describe a hierarchical partitioning of the matrix tiles. Note that as explained in Section 4, all these partitions are only planned and will be enforced, if needed, at runtime. We remind that Chameleon relies on tiled algorithms and that each basic operation involves a given number of tiles. In the following experiments, the decision to process a hierarchical task is made as early as possible (at submission time) and based on the data structure of the tiles involved in that task. If all of them can be partitioned, then a subgraph will be submitted. Otherwise, the task will operate at the highest common data level. The following experiments were conducted on an architecture composed of 2 Intel Xeon Gold 6142 of 16 cores each running at 2.6GHz, 2 Nvidia V100, and 384GB of memory. The tile sizes used are the ones providing the best asymptotic performance for CPUs only (960) and for hybrid CPU-GPU configuration (2880). Additionally, we provide results for tile size of 320 that provides the best performances on CPU configurations for small matrices. Concerning hierarchical variant we will use the following notation $x/y/z/...$ meaning that each initial tile is of size x and is partitioned into tiles of size y which are in turn split into tiles of size z etc. StarPU has been configured to use a single stream per GPU and to pipeline four events per stream. This limits the performance when using a tile size of 320, but increasing the number of stream has little impact when using a tile size of 960 or 2880. We use the DMDA scheduling policy, a minimum completion time algorithm relying on performance models.

6.1 Overhead of Hierarchical Tasks

To evaluate the overhead induced by hierarchical tasks, we consider the graph of a matrix-matrix multiplication (GEMM) using a tile size of 960. In Figure 8, we compare the submission time per computational task for that graph 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 one of them submits exactly one task when processed. This doubles the number of tasks submitted as well as it heavily increases the workload of the data manager making the submission time per computational task roughly 3.5 times slower. Finally,

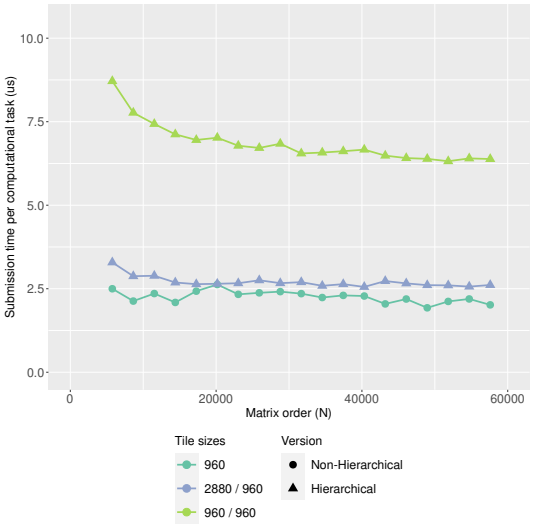


Figure 8: Evolution of the submission cost of computational tasks depending on the use of hierarchical tasks for the matrix-matrix multiplication kernel with all tiles partitioned.

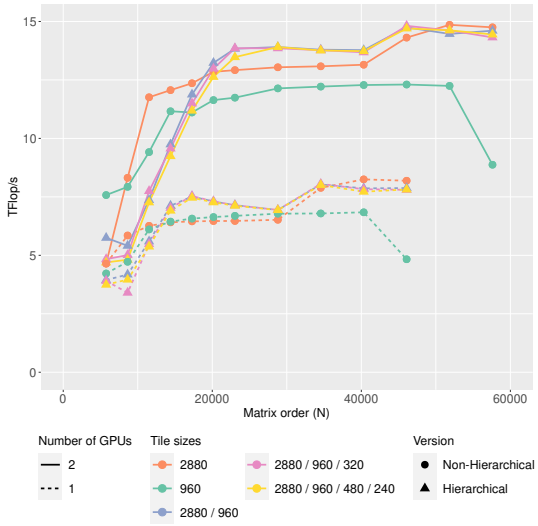


Figure 9: Comparison of the hierarchical strategy to the original one on the matrix-matrix multiplication (DGEMM) kernel with diagonal distribution of the hierarchical tasks.

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 individual hierarchical task submits $\lceil 2880/960 \rceil^3 = 27$ regular tasks when processed which allows to amortize the overhead induced by the management of hierarchical tasks.

6.2 Matrix Multiplication

In the following experiments we use a more realistic partitioning of the matrix where only the diagonal, subdiagonal and superdiagonal tiles are partitioned recursively. We evaluate the behavior of the GEMM operation on those matrices, using one and two GPUs (Figure 9). In both cases, the hierarchical versions lag behind on small matrices, due to the overhead introduced. As the matrix size increases, the amount of kernels using smaller tiles becomes sufficient to feed the CPUs and compensates for that overhead. We can also observe that using more levels of partitioning does not have an impact on performance for this experiment. Eventually, the number of tasks needed for the computation becomes large enough that the '2880' curve can start affecting more work to the CPUs and catches up with the hierarchical curve. All in all, the hierarchical variants have a good behavior and outperforms the regular CHAMELEON implementation while relying on simplistic matrix partitioning. This illustrates the potential of hierarchical tasks at achieving a better trade-off in terms of granularity of computations.

6.3 Cholesky Decomposition

To better illustrate the expressivity of hierarchical tasks, Figure 10 shows results of 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,

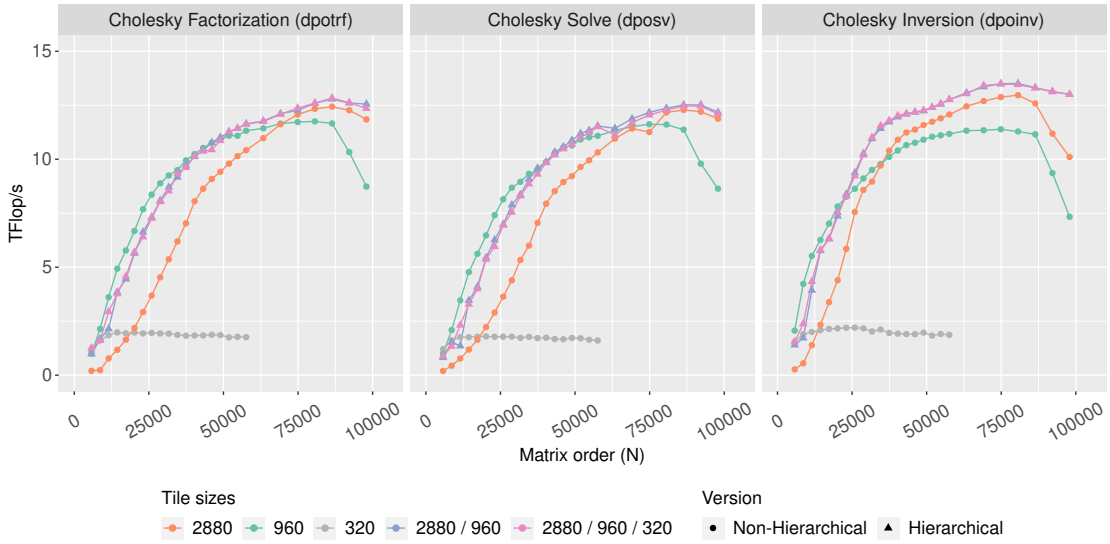


Figure 10: Comparison of the hierarchical strategy to the original one on Cholesky type operations (DPOTRF, DPOSV, DPOINV) kernel with diagonal distribution of the hierarchical tasks.

and in the case of POINV, validate the anti-dependency problem (*WRITE* after *READ*). We observe a similar behavior to the one observed for GEMM. A notable distinction however, is that we now benefit more from our partitioning scheme, because CHAMELEON places all POTRF kernels (which are on the critical path of the factorization) on CPU cores leading to moderate performance before $N \approx 75000$. On the other hand, thanks to hierarchical tasks, we can partition the tiles along the diagonal and split those large tasks into subgraphs with a smaller granularity allowing for better CPU utilization on the critical path. Similarly to the results on GEMM, the hierarchical tasks are sooner able to take advantage of the performance of both GPUs and CPUs resources. 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. This phenomenon is however compensated by the hierarchical variant thanks to the use of different granularities allowing for better resource utilization. The experimental results illustrate the interest of hierarchical tasks for tackling the granularity problem of heterogeneous architectures. We are able to enhance expressivity thanks to hierarchical tasks while improving the overall behavior.

7 Conclusion

In this paper, we proposed an extension of the sequential task flow model together with an upgrade of the underlying runtime system in order to overcome the inherent limitations of the programming model. The approach we propose introduces a new type of tasks, the *hierarchical* tasks, which have the ability to submit at runtime a new sub-graph of tasks. In addition, to ensure that the parallel submission process still produces a valid DAG, we introduce a new automatic data manager whose goal is to handle data layout dynamically by submitting data management tasks at the right moment. We show also why our model is correct and present a set of results illustrating the interest of the hierarchical task model.

In the near future, we plan to extend this work in several ways. First of all, we need to consider the hierarchical tasks from the scheduling point of view. We want to answer the question "when does a hierarchical task need to be processed?". The solution to this problem needs to consider the amount of tasks in the system and the work assigned to each resource. Additionally, we will consider the problem of choosing which subgraph has to be submitted when a hierarchical task is processed. Indeed, to be able to select the most adapted implementation, we need advanced performance models which have yet to be designed. Finally, the task graph resulting from the processing of a hierarchical task has to be efficiently scheduled. All in all, considering all the problems mentioned above will allow to improve the behavior of the runtime system when relying on hierarchical tasks. More generally, we want to investigate how this model can be used to implement advanced irregular algorithms like linear algebra solvers based on low-rank approximation or sparse solvers. In the long run, we believe that extending the hierarchical task model to the distributed memory context will be an elegant answer to the scalability problem of task-based runtime systems.

Acknowledgement

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] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov, "A hybridization methodology for high-performance linear algebra software for gpus," *GPU Computing Gems, Jade Edition*, vol. 2, pp. 473–484, 2011.

[2] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," 2007. [Online]. Available: http://www.citebase.org/abstract?id=oai:arXiv.org:0709.1272

[3] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Luszczek, and J. Dongarra, "Dense linear algebra on distributed heterogeneous hardware with a symbolic dag approach," *Scalable Computing and Communications: Theory and Practice*, 2013.

[4] P. Valero-Lara, S. Catalán, X. Martorell, T. Usui, and J. Labarta, "slass: A fully automatic auto-tuned linear algebra library based on openmp extensions implemented in ompss (lass library)," *Journal of Parallel and Distributed Computing*, vol. 138, pp. 153–171, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731519303417

[5] T. Cojean, A. Guermouche, A. Hugo, R. Namyst, and P. Wacrenier, "Resource aggregation for task-based cholesky factorization on top of modern architectures," *Parallel Comput.*, vol. 83, pp. 73–92, 2019.

[6] M. Maroñas, K. Sala, S. Mateo, E. Ayguadé, and V. Beltran, "Worksharing tasks: An efficient way to exploit irregular and fine-grained loop parallelism," in *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2019, pp. 383–394.

[7] W. Wu, A. Bouteiller, G. Bosilca, M. Faverge, and J. Dongarra, "Hierarchical dag scheduling for hybrid distributed systems," in *29th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. Hyderabad, India: IEEE Computer Society, May 2015, pp. 156–165.

[8] R. Carratala-Saez, S. Christophersen, J. I. Aliaga, V. Beltran, S. Borm, and E. S. Quintana-Orti, "Exploiting nested task-parallelism in the h-lu factorization," *Journal of Computational Science*, vol. 33, pp. 20–33, 2019.

[9] K. Akbudak, H. Ltaief, A. Mikhalev, and D. Keyes, "Tile low rank cholesky factorization for climate/weather modeling applications on manycore architectures," in *High Performance Computing*, J. M. Kunkel, R. Yokota, P. Balaji, and D. Keyes, Eds. Cham: Springer International Publishing, 2017, pp. 22–40.

[10] R. Carratalá-Sáez, M. Faverge, G. Pichon, G. Sylvand, and E. S. Quintana-Ortí, "Tiled algorithms for efficient task-parallel ħ-matrix solvers," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2020, New Orleans, LA, USA, May 18-22, 2020*. IEEE, 2020, pp. 757–766.

[11] C. Augonnet, D. Goudin, M. Kuhn, X. Lacoste, R. Namyst, and P. Ramet, "A hierarchical fast direct solver for distributed memory machines with manycore nodes," Research Report, Oct. 2019. [Online]. Available: https://hal-cea.archives-ouvertes.fr/cea-02304706

[12] S. Börm, S. Christophersen, and R. Kriemann, "Semi-automatic task graph construction for h-matrix arithmetic," *CoRR*, vol. abs/1911.07531, 2019. [Online]. Available: http://arxiv.org/abs/1911.07531

[13] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "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, p. 66.

[14] M. Faverge, J. Herrmann, J. Langou, B. Lowery, Y. Robert, and J. Dongarra, "Designing LU-QR hybrid solvers for performance and stability," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2014)*, Phoenix, United States, May 2014. [Online]. Available: https://hal.inria.fr/hal-00930238

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

[16] M. Cosnard, E. Jeannot, and T. Yang, "Slc: Symbolic scheduling for executing parameterized task graphs on multiprocessors," in *Proceedings of the 1999 International Conference on Parallel Processing*, 1999, pp. 413–421.

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

[18] M. Frigo, C. Leiserson, and K. Randall, "The implementation of the cilk-5 multithreaded language," *SIGPLAN Not.*, vol. 33, no. 5, pp. 212–223, 1998.

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

[20] T. D. R. Hartley, E. Saule, and Ü. V. Çatalyürek, "Improving performance of adaptive component-based dataflow middleware," *Parallel Computing*, vol. 38, no. 6-7, pp. 289–309, 2012.

[21] D. M. Kunzman and L. V. Kalé, "Programming heterogeneous clusters with accelerators using object-based programming," *Scientific Programming*, vol. 19, no. 1, pp. 47–62, 2011.

[22] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale, "Parallel programming with migratable objects: Charm++ in practice," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. IEEE Press, 2014, p. 647–658.

[23] 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, and T. Zhang, "Hpx - the c++ standard library for parallelism and concurrency," *Journal of Open Source Software*, vol. 5, no. 53, p. 2352, 2020. [Online]. Available: https://doi.org/10.21105/joss.02352

[24] E. Hermann, B. Raffin, F. Faure, T. Gautier, and J. Allard, "Multi-gpu and multi-cpu parallelization for interactive physics simulations," in *Euro-Par 2010 - Parallel Processing*, 2010, vol. 6272, pp. 235–246.

[25] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurr. Comput. : Pract. Exper.*, vol. 23, pp. 187–198, Feb. 2011.

[26] E. Ayguadé, R. Badia, F. Igual, J. Labarta, R. Mayo, and E. Quintana-Ortí, "An Extension of the StarSs Programming Model for Platforms with Multiple GPUs," in *Euro-Par 2009*, 2009, pp. 851–862.

[27] R. Hoque, T. Herault, G. Bosilca, and J. Dongarra, "Dynamic task discovery in parsec: A data-flow task-based runtime," in *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ser. ScalA '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3148226.3148233

[28] D. Álvarez, K. Sala, M. Maroñas, A. Roca, and V. Beltran, "Advanced synchronization techniques for task-based runtime systems," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 334–347.

[29] H. Elshazly, F. Lordan, J. Ejarque, and R. M. Badia, "Accelerated execution via eager-release of dependencies in task-based workflows," *The International Journal of High Performance Computing Applications*, vol. 35, no. 4, pp. 325–343, 2021.

[30] J. M. Perez, V. Beltran, J. Labarta, and E. Ayguadé, "Improving the integration of task nesting and dependencies in openmp," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 809–818.

[31] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, "Taskflow: A lightweight parallel and heterogeneous task graph computing system," *IEEE Transactions on Parallel and Distributed Systems*, pp. 1–1, 2021.

[32] J. Kim, S. Lee, B. Johnston, and J. S. Vetter, "Iris: A portable runtime system exploiting multiple heterogeneous programming systems," in *Proceedings of the 25th IEEE High Performance Extreme Computing Conference*, ser. HPEC '21, 2021, pp. 1–8.

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

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

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399