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

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

LaBRI/Inria/University of Bordeaux/CNRS/Bordeaux INP
firstname.lastname@inria.fr

Abstract. Task-based systems have gained popularity as they promise to exploit 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 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 of StarPU [5] to enable tasks subgraphs at runtime. We refer to these tasks as *hierarchical tasks*. This approach allows for a more dynamic task graph. Combined with an automatic data manager, it allows to dynamically adapt the granularity to meet the optimal size of the targeted computing resource. We show that the model is correct and we provide an early evaluation on shared memory heterogeneous systems, using the Chameleon [1] dense linear algebra library.

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

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. GPU devices typically favor large data sets, whereas conventional CPU cores reach

peak performance with fine-grain kernels working on a reduced memory footprint. Systems usually have a much larger number of CPU units than GPUs, having more small tasks may lead to better performance. Several efforts have tried to tackle this problem either by finding the best trade-off between the optimal granularity of each device [1,7,17], or by aggregating CPU cores to process a task which was meant to be executed by an accelerator like a GPU [9,15]. Alternatively, some preliminary work has considered splitting the tasks on CPU cores [18]. Even though these approaches are efficient in specific contexts like dense linear algebra, they suffer from the fact that the task graph is static and does not allow 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 [2,6,8].

These runtime systems 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. 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 [3] ensuring the so-called *Sequential Consistency* at runtime. Other runtime systems such as PaRSEC use the parameterized task-graph programming model (PTG) [10] 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 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.

In this paper, we propose a new type of task, namely the *hierarchical tasks*, which 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. This approach is similar to what is done in OpenMP for the nested task-based parallelization scheme. However, we extend it 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. Our contribution 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.

2 Related Work

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. [4] analyzes the limiting factors in the scalability of a task-based runtime system and proposes 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 management. For instance, [11] proposes 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, [15] introduces worksharing tasks. These are tasks that internally leverage worksharing techniques to exploit fine-grained structured loop-based parallelism without requiring a barrier.

The closest contribution to our proposition from the perspective of task dependencies was introduced in [16] as the concept of *weak dependencies*. It is an extension of the OpenMP task-nesting model which enhances the dataflow model of OpenMP by supporting fine-grained dependencies between any set of tasks. Our contribution is a generalization of the weak dependency concept to the heterogeneous case where memory consistency is not ensured by the underlying hardware, thus needing an advanced data manager (see Section 3). Alternatively, some preliminary work targeting heterogeneous architectures has considered splitting the tasks when assigned to CPU cores in the context of ParSEC [18] and XKaapi [12].

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 expressiveness. In TaskFlow [13], 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. [14] introduces 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 [6]. In this work, hierarchical tasks are introduced and the

dependencies are expressed at the finest level. However, the data management is straightforward since the partitioning of data is performed statically at the beginning of the execution.

3 Automatic Data Management

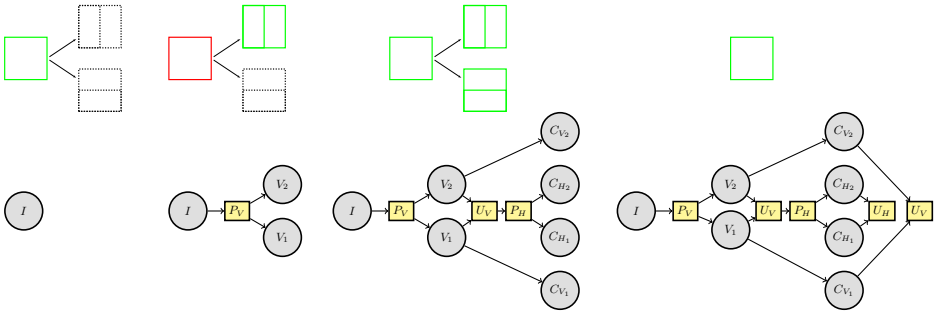
Data handling is at the heart of StarPU both to automatically infer dependencies between tasks in the STF model and to automatically manage data transfers between the different memory banks of a distributed/heterogeneous system. To benefit from these automation, applications must register the data that 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 are coupled with an access mode (read-only, read-write, ...) and are used as task parameters. It is mandatory for a task to access a piece of 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 the same piece of data can be designated simultaneously by several handles. Data in read-only access mode can advantageously be accessed simultaneously at different partitioning levels by several tasks. 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.

We propose a mechanism to automate the management of several simultaneous partitions. This mechanism enhances StarPU such that it automatically inserts partition or unpartition tasks as needed. First, programmers need to define the partitioning scheme through the *plan* operation which declares the partitioning to StarPU, and can be seen as the declaration of a new set of subhandles. 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.

The data manager will then handle the actual partitioning tasks and data coherency. 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. Figure 1 shows a matrix on which two partition plans are defined. The matrix is first

initialized through its root handle, then modified using the vertical partitioning, and finally checks are performed in both horizontal and vertical stripes.

Figure 1a shows the state of the DAG and the data-layout after the execution of the plan operations and the insertion of the initialization task. With the first task using a vertical stripe, StarPU will automatically insert the corresponding *partitioning* task (see Figure 1b). The same scheme is then applied when submitting tasks working on the horizontal layout and vertical layout 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 tasks to make the data available to the tasks using the horizontal layout. This is illustrated in Figure 1c 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 1d).



(a) Step 1. (b) Step 2. (c) Step 3. (d) Step 4.

Fig. 1: Example of the behavior of the automatic data manager. 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*. *Step 1.* Root handle initialization and partition plan, *Step 2. Read-Write* Vertical partitions, *Step 3.* 3 *Read-Only* active partitions, *Step 4.* Partition clean.

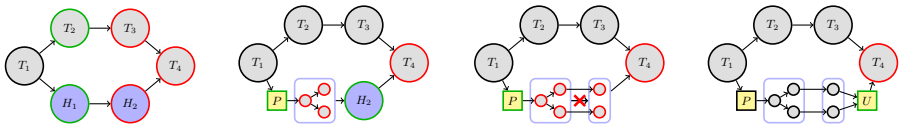
The previous example illustrates the general behavior of the data manager. More precisely, during the submission of tasks, each handle in the partitioning hierarchy can be either *inactive* (one cannot access the piece of data), *read-write-active* (one can read/write to the piece of data or a subpart 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 1b to Figure 1c, StarPU indeed had to submit the unpartition task of the root handle, and repartition it.

4 The Hierarchical Task Paradigm

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. To ensure the portability with 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. 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 needs to respect the following constraints which aim at having a general implementation of such a paradigm. First of all, the depth of the hierarchy is not limited. Secondly, Programmers express their task-graph at the highest level and only annotate some tasks as possibly hierarchical. Thirdly, data management needs to be transparent to programmers. Finally, task dependencies always have to be inferred at the deepest level.



(a) Initial DAG. (b) H_1 processed. (c) H_2 processed. (d) Auto. U inserted.

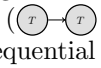
Fig. 2: Example of a DAG with 2 hierarchical tasks and 4 regular tasks.

Figure 2a shows an execution scenario for a given task graph where **blue** tasks could be transformed into hierarchical tasks. The state of each task (i.e. node in the graph) is described by its border: 1) a ready task is **green** (all dependencies are met), 2) a not-ready task is **red** (some dependencies are unsatisfied), 3) an already executed task is **black**. Thus, we can see in Figure 2a

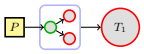
that T_1 has completed its execution making T_2 and H_1 ready for execution. T_2 and T_3 execute as normal tasks, while H_1 is processed, i.e. its corresponding subDAG is submitted, resulting to Figure 2b. The dependency between H_1 and H_2 is then released, making H_2 ready for processing. Furthermore, we can see that after the processing of H_2 (see Figure 2c) the dependencies between the resulting submitted tasks are inferred by the runtime system at the deepest level of the hierarchy.

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 processed. We propose 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 3. Figure 2b shows the insertion of the partitioning task P (resp. U) ahead of the subgraph produced by H_1 (resp. T_4). 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 of the actual computations.

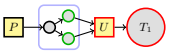
4.1 Ensuring the Correctness of the DAG

We now show why the hierarchical task model to extend the STF model produces a correct DAG regardless of the depth of the hierarchy. First of all, as stated above, the STF model infers the dependencies from data access modes of individual tasks while relying on the sequential consistency. Introducing hierarchical tasks makes the submission process parallel while in the STF model, the submission is done by a single entity. We show that the dependencies respect the STF model by discussing four simple scenarios which are building blocks for any general DAG to show its correctness. The two first scenarios ( and ) will not be discussed since they inherently respect the sequential consistency.

(a) Initial DAG.

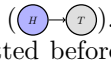


(b) Processing of H_1.

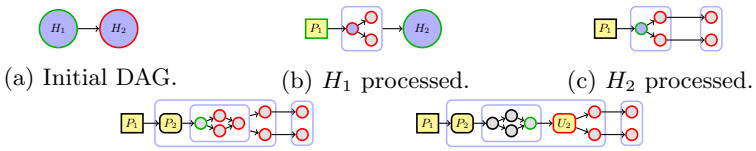


(c) Insertion of U when T_1 is ready.

Fig. 3: Example of a scenario where a task follows a hierarchical task.

Task following hierarchical task. Figure 3 illustrates this scenario (). The main problem is that the regular task is by construction submitted before the tasks resulting from the hierarchical task (H_1 in Figure 3). This may violate the order required by the sequential consistency. However, the hierarchical task has changed the data layout before it starts its execution (see Figure 3b). Thus the

task following the hierarchical task (T_1 in Figure 3) 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 (see Figure 3c). 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.



(a) Initial DAG. (b) H_1 processed. (c) H_2 processed.

(d) H_{11} processed. (e) Auto. U_2 inserted.

Fig. 4: Example of a chain of two hierarchical tasks.

Hierarchical task following hierarchical task. Figure 4 illustrates this scenario (). 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 4 where initially two hierarchical tasks H_1 and H_2 are submitted (see Figure 4a). Then H_1 is processed (see Figure 4b). 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 4c) 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 4d where the first task submitted by H_1, which will be referred to as H_{11}, is decided to be hierarchical and is processed. We can also see the partitioning task P_2 which was automatically inserted by the data manager. The resulting task-graph is coherent with the STF paradigm.

5 Experimental Evaluation

To illustrate the potential of hierarchical tasks for handling the coexistence of multiple levels of granularity, we apply them in a dense linear algebra context[1] 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 3, all these partitions are only planned and will be enforced, if needed, at runtime. The following experiments were conducted on an

[1] https://gitlab.inria.fr/starpu/starpu-papers/heteropar2022 for replication.

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 variants 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, to pipeline four events per stream and to use the DMDA scheduler.

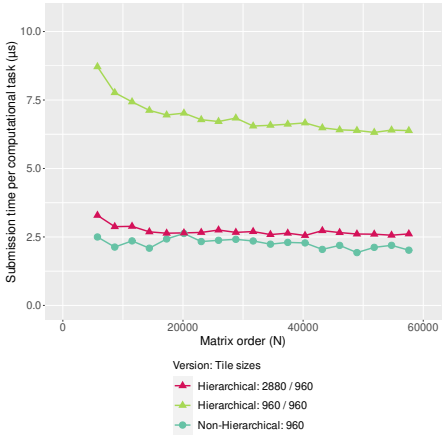


Fig. 5: Submission cost of computational tasks for DGEMM with all tiles partitioned.

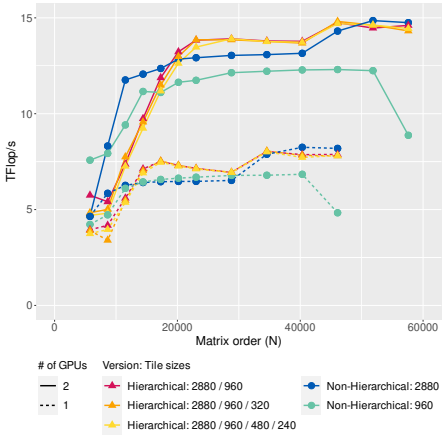


Fig. 6: Performance evaluation of DGEMM with diagonal distribution of the hierarchical tasks

To evaluate the overhead induced by hierarchical tasks, we consider the graph of a matrix-matrix multiplication (GEMM) using a tile size of 960. Figure 5 compares 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 heavily increasing 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 individual hierarchical task submits $\lceil 2880/960 \rceil^3 = 27$ regular tasks when processed, thus amortizing the overhead induced by the management of hierarchical tasks.

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 6). 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 outperform the regular CHAMELEON implementation while relying on simplistic matrix partitioning.

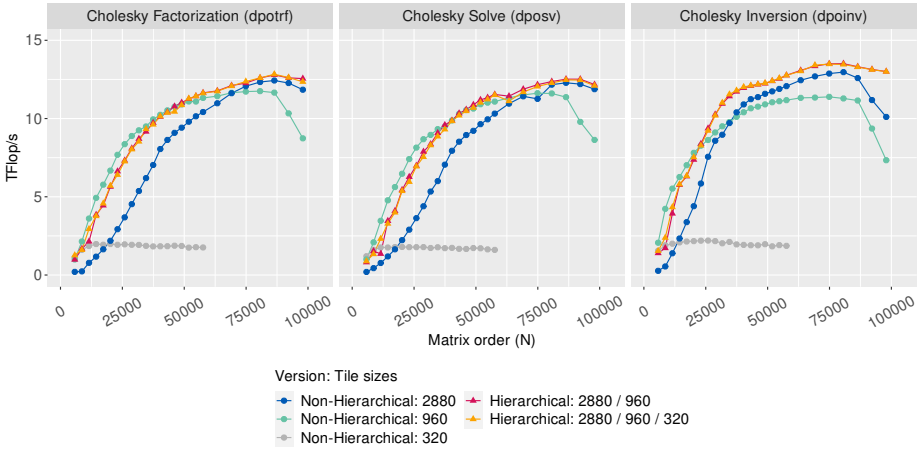


Fig. 7: Performance evaluation of Cholesky type operations (DPOTRF, DPOSV, DPOINV) with diagonal distribution of the hierarchical tasks.

To better illustrate the expressiveness of hierarchical tasks, Figure 7 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, 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. The experimental results illustrate the interest of hierarchical tasks for tackling the granularity problem of heterogeneous architectures.

6 Conclusion

In this paper, we propose an extension of the STF model together with an upgrade of the underlying runtime system in order to overcome the inherent limitations of the programming model. Our approach 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.

In the near future, we plan to extend this work in several ways. We first need to consider the hierarchical tasks from the scheduling point of view, and answer the question "when does a hierarchical task need to be processed?". This requires 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. 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. 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.

Acknowledgment

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[2].

References

1. Agullo, E., Augonnet, C., Dongarra, J., Ltaief, H., Namyst, R., Thibault, S., Tomov, S.: A Hybridization Methodology for High-Performance Linear Algebra Software for GPUs. GPU Computing Gems, Jade Edition **2**, 473–484 (2011)
2. Akbudak, K., Ltaief, H., Mikhalev, A., Keyes, D.: Tile Low Rank Cholesky Factorization for Climate/Weather Modeling Applications on Manycore Architectures (2017)

[2] https://www.plafrim.fr

3. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures: A Dependence-Based Approach. Morgan Kaufmann (2002)
4. Álvarez, D., Sala, K., Maroñas, M., Roca, A., Beltran, V.: Advanced Synchronization Techniques for Task-Based Runtime Systems. In: Proc. of PPoPP '21. p. 334–347 (2021)
5. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. Concurr. Comput. : Pract. Exper. **23**, 187–198 (Feb 2011)
6. 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. (Oct 2019), https://hal-cea.archives-ouvertes.fr/cea-02304706
7. Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Haidar, A., Herault, T., Kurzak, J., Langou, J., Lemarinier, P., Ltaief, H., Luszczek, P., YarKhan, A., Dongarra, J.: Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA. In: IEEE IPDPS Workshops and Phd Forum. pp. 1432–1441 (2011)
8. Carratala-Saez, R., Christophersen, S., Aliaga, J.I., Beltran, V., Borm, S., Quintana-Orti, E.S.: Exploiting nested task-parallelism in the H-LU factorization. Journal of Computational Science **33**, 20–33 (2019)
9. Cojean, T., Guermouche, A., Hugo, A., Namyst, R., Wacrenier, P.: Resource aggregation for task-based Cholesky Factorization on top of modern architectures. Parallel Comput. **83**, 73–92 (2019)
10. Cosnard, M., Jeannot, E., Yang, T.: Slc: Symbolic scheduling for executing parameterized task graphs on multiprocessors. In: Proc. of ICPP'99. pp. 413–421 (1999)
11. Elshazly, H., Lordan, F., Ejarque, J., Badia, R.M.: Accelerated execution via eager-release of dependencies in task-based workflows. The International Journal of High Performance Computing Applications **35**(4), 325–343 (2021)
12. Gautier, T., Lima, J.V.F., Maillard, N., Raffin, B.: Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In: Proc. of IPDPS'13. pp. 1299–1308 (2013)
13. Huang, T.W., Lin, D.L., Lin, C.X., Lin, Y.: Taskflow: A lightweight parallel and heterogeneous task graph computing system. IEEE Transactions on Parallel and Distributed Systems pp. 1–1 (2021)
14. Kim, J., Lee, S., Johnston, B., Vetter, J.S.: Iris: A portable runtime system exploiting multiple heterogeneous programming systems. In: Proc. of HPEC'21. pp. 1–8 (2021)
15. 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. pp. 383–394 (2019)
16. Perez, J.M., Beltran, V., Labarta, J., Ayguadé, E.: Improving the Integration of Task Nesting and Dependencies in OpenMP. In: Proc. of IPDPS'17. pp. 809–818 (2017)
17. 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 **138**, 153–171 (2020)
18. Wu, W., Bouteiller, A., Bosilca, G., Faverge, M., Dongarra, J.: Hierarchical DAG scheduling for Hybrid Distributed Systems. In: Proc. of IPDPS'15. pp. 156–165 (2015)