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

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

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

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

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

Abstract. Task-based programming models significantly improve the efficiency of parallel systems. The Sequential Task Flow (STF) model focuses on static task sizes within task graphs, but determining optimal granularity during graph submission is tedious. To overcome this, we extend StarPU's STF recursive tasks model, enabling dynamic transformation of tasks into subgraphs. Early evaluations on homogeneous shared memory reveal that this just-in-time adaptation enhances performance.

Keywords: Task-based programming · Granularity · Runtime System

1 Introduction

Heterogeneous architectures play a crucial role in the development of high-performance computers. The effective utilisation and capability to achieve portable performance relies on Runtime Systems (RS). Most of them use task-based parallelism. With this model, the application is represented as a graph of small units of work called tasks. A widely used task-based paradigm is the Sequential Task Flow (STF) model as seen in frameworks like OpenMP, StarPU or StarSs. However, the STF model has some limitations, notably its reliance on sequential task submission. This constraint can pose challenges in dynamically adjusting task granularity during execution. To overcome these issues, we extend the STF model by using recursive task parallelism. A recursive task can either undergo normal execution or be split, transforming into a subgraph of tasks during execution. This extension prompts two critical questions: firstly, the technical implementation of recursive tasks in a Runtime System; and secondly, the determination of which tasks should be split.

The first question has been previously addressed in [3]. In this paper, we focus on addressing the second question by using the recursive tasks of StarPU [1]. Therefore, our contributions in this paper are as follows:

- Introduction of a novel decision-tool, referred to as the splitter, designed to decide which tasks should be split.
- Presentation of a preliminary experimental evaluation that illustrates the potential efficiency of our approach.

By exploring these aspects, we aim to enhance the understanding of recursive task parallelism within the context of STF models, paving the way for improved adaptability and performance in heterogeneous computing environments.

2 Granularity Challenges within the STF Model

The STF model relies on sequential consistency to automatically infer dependencies between tasks through the analysis of data accesses. Its capability to streamline the utilisation of heterogeneous machines, combined with its seemingly straightforward design, positions it as a model attracting increased attention. However, it is important to note that this model comes with inherent limitations. Firstly, the submission of large DAGs presents challenges: submitting tasks well before their execution can result in unnecessary system congestion. Attempting to regulate submissions by periodically suspending them is not without risks, as suboptimal submission order may lead to idle periods. Secondly, the utilisation of different types of processing units (PUs) poses challenges, as a one-size-fits-all approach becomes not applicable. GPUs excel with coarse-grained parallelism, while CPUs require finer-grained parallelism to effectively utilise all cores. Thirdly, even with homogeneous computing units, accommodating varying granularities is essential. Large tasks are generally the most efficient, yet transforming such tasks into subgraphs may increase parallelism. Thus, optimising the overall application completion time may entail considering multiple granularities.

To tackle granularity issues, Runtime Systems (RS) can allow a task to be executed on multiple CPU cores, as shown in StarPU [2]. This makes CPUs competitive with GPUs on coarse-grained tasks, but this technique does not create different granularities. Programmers can also submit tasks with varying granularities. However, this manual approach is labour-intensive and does not adapt well to the dynamic nature of the graph, since the optimal granularity may not be known at submission time. Fortunately, RS provide features that enable tasks to become subgraphs at execution time. These contributions can be classified based on their approach to handle heterogeneity, their expression of dependencies, and their methods to manage data. OmpSs [5] introduces weak dependencies to establish fine-grained relationships between tasks and a subgraph. TaskFlow [4] introduces advanced tasking schemes to enable the dynamic generation of task subgraphs. Additionally, ParSEC allows hierarchical Directed Acyclic Graphs (DAGs) to achieve high performance on hybrid distributed systems [6].

Recursive tasks have recently been introduced in StarPU [3], where programmers describe a data hierarchy and submit tasks that can be transformed into DAGs that will work on sub-data. StarPU's key feature is an automatic data manager without spurious synchronisation in a heterogeneous context. In the case of task splitting, the data manager automatically introduces a task called *Partition Task* (PT) to partition the input data for sub-DAG tasks. Conversely, when a task is not split, and the data was previously partitioned, the data manager automatically introduces an *Unpartition Task* (UT) to wait for output data from sub-DAG tasks and gather them, as illustrated in Figure 1. On the one hand, we have the *normal* dependencies, related to the dependencies introduced by the sequential consistency. For a recursive task, these are released when the subgraph has been submitted. On the other hand, we have the *recursive* dependencies, which are introduced by PT and UT.

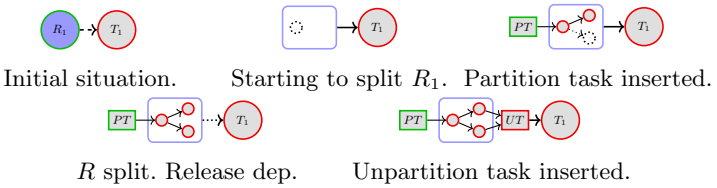

Fig. 1: Recursive task graph processing by StarPU's data manager.

3 Just-in-time Task Splitting in StarPU

This section presents the integration of a new component, the splitter, within StarPU's scheduling framework. The role of the splitter is to receive a potential recursive task as input and ascertain whether it evolves into a recursive form or remains a regular task. The submission of the subgraph is then done by a worker. Following this, we address three pivotal questions.

The first question concerns **which** task to split, by identifying the relevant criteria, aiming to optimise the system's overall efficiency. Striking a balance between task efficiency and parallelism is crucial, as coarse-grain parallelism tends to be more efficient but potentially less parallel. To make the decision to split a task, two criteria are considered: the efficiency of the splitting in terms of potential created parallelism and the need for parallelism, particularly when there are a limited number of tasks to complete.

Having determined the parameters that need to be taken into account, the second question focuses on **when** to split tasks. The timing significantly influences the quality of the information used for decision-making. Ensuring the alignment of the recursive task-splitting process with the progression of computations is essential. Splitting tasks too early may overload the system, while delayed decisions may lead to idle periods.

Task-splitting decisions can be made at various stages of the RS workflow, including the submission stage. While splitting at submission minimises overhead, it has two drawbacks: (1) it does not significantly reduce the number of tasks stored in the system, and (2) the decision to split a task could be made far in advance of its execution, lacking comprehensive knowledge of the machine's future state. Hence, it is advantageous to delay the decision-making process until a recursive task has fulfilled all its normal dependencies. However, exercising this precaution alone may not be entirely effective. Let us consider a sequence of recursive tasks denoted as $R_1 \rightarrow R_2 \cdots \rightarrow R_k$. Let us assume that all tasks preceding R_k are split. The decision to split R_k can be made before the execution of the subtasks generated by R_1. Without additional safeguards, the splitting decision might occur well before the computation progresses significantly.

Therefore, in order to harmonise the task splitting decision with the computation progress, we have developed a technique to postpone the release of outgoing normal dependencies of a recursive task. Instead of releasing these dependencies immediately after the submission of its subgraph, the release is deferred

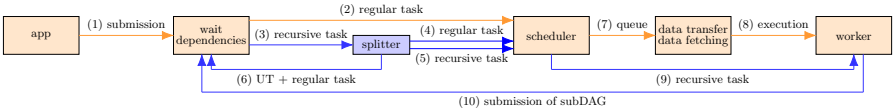

Fig. 2: Scheme of the insertion on existing architecture of our solution. Existing components and paths in orange, added in blue.

until the execution of one of its subtasks. Thus, the subgraph submission of a sequence of recursive tasks will occur gradually: a recursive task R_2 following a split recursive task R_1 will remain blocked until the execution of a R_1 subtask.

The third question is related to the design of the RS, specifically determining **where** to place the splitter within the workflow. We have already seen that the splitter component should be placed later than the submit phase. In addition, unlike regular tasks, the execution of a recursive task does not require any data transfer. Therefore, to avoid potentially redundant transfers, it is advisable to make the task splitting decision before the data prefetching stage.

As shown in Figure 2, we position the splitter at the initial stage of the scheduler. The handling of a recursive task is as follows: upon submission by the application (1), the recursive task is given to the splitter (3) once its dependencies are satisfied. The splitter then takes its decision. If the task requires splitting, it is assigned directly to the scheduler (5), which then places it in the execution queue of a worker (9). The processing of this recursive task results in the submission of a sub-DAG (10). If the task is not split, the recursive task is transformed into a regular task. In the scenario where its data is not split, the regular task can be executed as usual (4). Alternatively, if its data is processed by subtasks at a lower recursive level, an unpartitioning task (6) is automatically submitted to maintain sequential consistency.

4 Study Case: Cholesky Factorisation

In this section, we illustrate our solution on homogeneous systems by making a performance study case with the Cholesky factorisation. The platform has two Intel(R) Xeon(R) Gold 6240 CPU @ 2.60GHz, having 18 cores each. We compare three tile sizes: the coarse one (1120), the medium one (560) and the fine one (280). A task is split if the number of ready tasks is less than four times the number of cores, and the split efficiency is better than 50%. The locality-aware work-stealing scheduler (*lws*) from StarPU was used for all the experiments.

Figure 3 compares the non-recursive version using coarse, medium and fine tile sizes, with recursive versions which choose the granularity dynamically. Three recursive variants are considered : 1) the fully recursive variant where each task is split to the fine granularity (280), 2) the critical path variant where a task is split if it is on the Critical Path, 3) the splitter recursive version relies on the criteria presented above. We observe that our criteria allow a speed-up of approximately

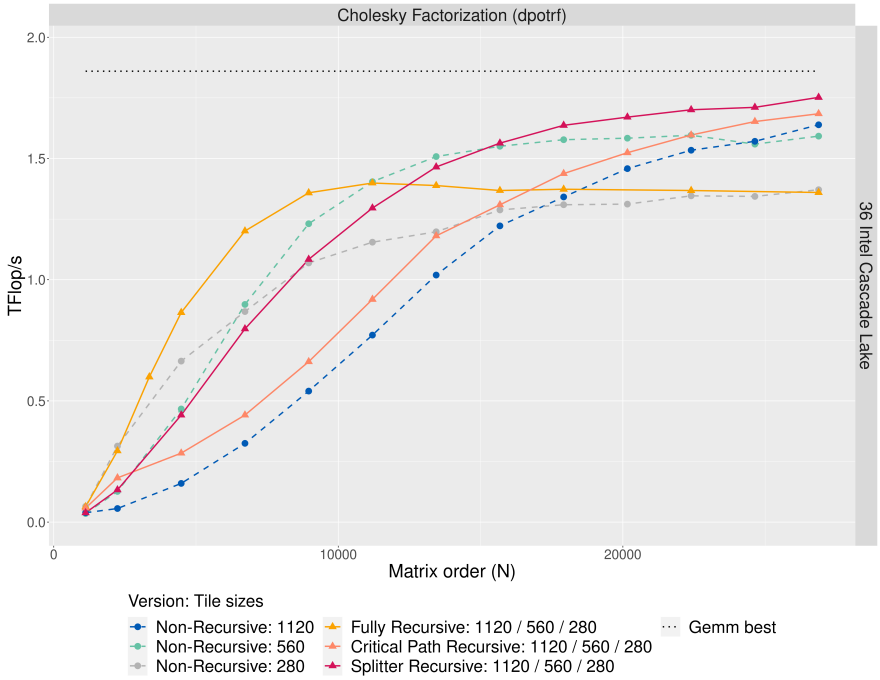

Fig. 3: Cholesky Factorisation performance according to matrix order, tile size and recursive version.

10 % compared to the best non-recursive version (tile size 1120), and a speed-up of 5 % over the Critical Path recursive version. The fully recursive version is the fastest for small matrices because all tasks are split; and with recursive tasks, task submission is parallelised, thus reducing the submission bottleneck. Finally, when comparing small matrix sizes, the splitter-recursive version makes a good trade-off between efficiency and parallelism, and succeeds in obtaining performance close to the best.

Figure 4 shows the available number of floating operations all along the execution of a Cholesky factorisation for a matrix of order 26 880. The colours represent the level of the floating operations, *i.e.* in blue (resp. orange, green), we represent the available floating operations for coarse (resp. medium, fine) grain tasks. We observe that the splitter policy is able to react whenever the number of available operations drops, by creating smaller tasks.

5 Conclusion

The increasing complexity of computing platforms has led to the development of advanced runtime systems that aim to separate problem expression from execu-

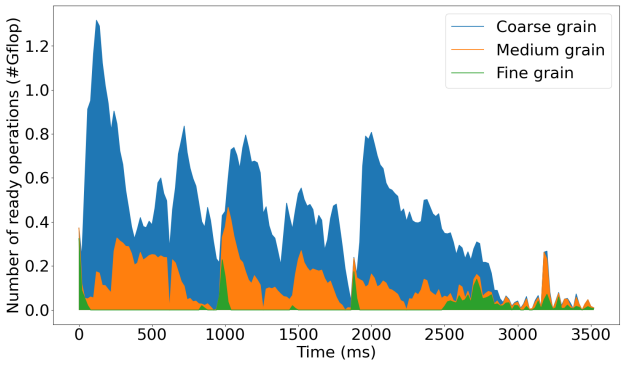

Fig. 4: Evolution of the number of flops during a Cholesky execution for a matrix of order 26 880 according to the task level.

tion. Many of these systems use tasks, with some adopting the Sequential Task Flow paradigm. While powerful, this paradigm has limitations that we address by extending StarPU's recursive tasks. This extension allows StarPU to smartly select which tasks to split, and shows promising preliminary results.

Our ongoing efforts aim to extend these results to heterogeneous scenarios where tasks need to be differentiated for CPUs and GPUs. We are investigating three different plans: one optimised for efficiency when tasks are abundant, another that prioritises parallelism in the presence of a slowing-down task on the critical path, and a balanced approach to distribute work based on computational power. Additionally, we are exploring the extension of recursive tasks for a distributed context. We seek to minimise the impact of communication tasks on the runtime system and to dynamically adapt execution strategies.

References

1. Augonnet, C., , Thibault, S., Namyst, R., Wacrenier, P.A.: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In: Euro-Par Parallel Processing (2009)
2. Cojean, T., Guermouche, A., Hugo, A., Namyst, R., Wacrenier, P.A.: Resource Aggregation for Task-based Cholesky Factorization on top of Modern Architectures. Parallel Computing **83** (2019)
3. Faverge, M., Furmento, N., Guermouche, A., Lucas, G., Namyst, R., Thibault, S., Wacrenier, P.A.: Programming Heterogeneous Architectures Using Hierarchical Tasks. Concurrency and Computation: Practice and Experience **35**(25) (2023)
4. Huang, T.W., Lin, D.L., Lin, C.X., Lin, Y.: Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. IEEE TPDS **33**(6) (2022)
5. Perez, J.M., Beltran, V., Labarta, J., Ayguadé, E.: Improving the Integration of Task Nesting and Dependencies in OpenMP. In: IPDPS (2017)
6. Wu, W., Bouteiller, A., Bosilca, G., Faverge, M., Dongarra, J.: Hierarchical DAG scheduling for Hybrid Distributed Systems. In: IPDPS (2015)