Assembly Operations for Multicore Architectures using Task-Based Runtime Systems

Damien Genet[1], Abdou Guermouche[2], and George Bosilca[3]

[1] INRIA, Bordeaux, France
[2] INRIA, LaBRI, Univ. Bordeaux, Bordeaux, France
[3] University of Tennessee, Knoxville, USA

Abstract. Traditionally, numerical simulations based on finite element methods consider the algorithm as being divided in three major steps: the generation of a set of blocks and vectors, the assembly of these blocks in a matrix and a big vector, and the inversion of the matrix. In this paper we tackle the second step, the block assembly, where no parallel algorithm is widely available. Several strategies are proposed to decompose the assembly problem while relying on a scheduling middle-ware to maximize the overlap between stages and increase the parallelism and thus the performance. These strategies are quantified using examples covering two extremes in the field, large number of non-overlapping small blocks for CFD-like problems, and a smaller number of larger blocks with significant overlap which can be met in sparse linear algebra solvers.

1 Introduction

The increasing parallelism and complexity of hardware architectures requires the High Performance Computing (HPC) community to develop more and more complex software. To achieve high levels of optimization and fully benefit of their potential, not only the related codes are heavily tuned for the considered architecture, but the software is often designed as a single entity that aims to simultaneously cope with both the algorithmic and architectural needs. If this approach may indeed lead to extremely high performance, it is at the price of a tremendous development effort, a lesser portability and a poor maintainability.

Alternatively, a more modular approach can be employed. The numerical algorithm is described at a high level, independently of the hardware architecture, as a Directed Acyclic Graph (DAG) of tasks where a vertex represents a task and an edge represents a dependency between tasks. A second layer is in charge of taking the scheduling decisions. Based on these decisions, a runtime system will perform the actual execution of the tasks, maintaining data consistency and ensuring that dependencies are satisfied. The fourth layer consists of the optimized code for the related tasks on the underlying architectures. This approach is starting to give successful results in various domains going from very regular applications [16, 3, 7] to very irregular ones [14, 2, 1]. However, building such complex applications on top of task-based runtime systems requires algorithmic modifications of some core kernels of the application so that the flexibility offered by the runtime system can be fully exploited. More precisely, these operations need to be expressed as a task graph having enough parallelism to allow the runtime system to overcome all the synchronizations/race conditions which can be met with regular implementations of these kernels.

In this paper, we will focus on a specific operation, namely assembly operation, which can be met in various application fields: *finite elements* (FEM) methods, multifrontal sparse direct solvers, etc. This operation, even if not costly in terms of operations count, is memory-bound and often a performance bottleneck when the number of computational resources increases. Assembly operations can be viewed as scatter/add operations used to process dense contribution blocks to update a global, dense or sparse, matrix. This work is a first step toward a larger context where numerical simulations will be expressed in a task-based paradigm in order to diverge from the traditional fork-join model and relax synchronizations. Our contributions are : 1) A tiled version (which enhances parallelism) of the assembly operation is introduced and implemented on top of two task-based runtime systems. 2) Several *priority based* dynamic scheduling techniques which aim at reducing the makespan of the assembly operation are presented. 3) An experimental study concerning two application fields, namely FEM applications and multifrontal sparse direct solver, is presented.

The remainder of the paper is organized as follows. After a presentation of existing techniques for parallelizing assembly operations, we will introduce our tiled version of the assembly operations and show how it can be expressed in two different task-based paradigms. Finally, we will evaluate our proposed approaches and compare them with state-of-the-art techniques.

2 Related Work

Considering the increasing complexity of modern high performance computing platforms, the need for a portable layer that will insulate the algorithms and their developers from the rapid hardware changes becomes critical. Recently, this portability layer appeared under the denomination of task-based runtime. A lot of initiatives have emerged in the past years to develop efficient runtime systems for modern architectures. As stated above, most of these runtime systems use a task-based paradigm to express concurrency and dependencies by employing a task dependency graph to represent the application to be executed: PaRSEC [8], SMPSs [6], StarPU [5], etc. The main differences between all the approaches are related to whether or not they manage data movements between computational resources, to which extent they focus on task scheduling, and how task dependencies are expressed. These task-based runtime systems aim at performing the actual execution of the tasks, both ensuring that the DAG dependencies are satisfied at execution time and maintaining data consistency. Most of them are designed to allow writing a program independently of the architecture and thus require a strict separation of the different software layers: high-level algorithm, scheduling, runtime system, actual task implementation. Among these frameworks, we will focus in this paper on the StarPU and the PaRSEC runtime systems. The dense linear algebra community has strongly adopted such a modular approach lately [16, 3, 7] and delivered subsequent production-level solvers. As a result, performance portability is achieved thanks to the hardware abstraction layer introduced by runtime systems. More recently, this approach was considered in more complex/irregular applications : sparse direct solvers [14, 2], fast multipole methods [1], etc. The obtained results are promising and illustrate the interest of such a layered approach.

From the numerical simulation point of view, more precisely finite element methods, significant efforts have been made to exploit modern heterogeneous architectures (i.e. multicore systems equipped with accelerators) [13, 11]. The main idea is to be able to have efficient implementations of the core kernels needed by the numerical simulation namely assembly operations, linear systems solution, etc, for these architectures. We believe that these efforts are necessary to understand the bottlenecks to obtain a good performance on such heterogeneous architectures. However, we think that the modular approach proposed in this paper, coupled with a fine grain task-based expression of the application will ensure performance portability on any heterogeneous execution platform.

3 Background

3.1 Assembly operations on multicore systems

```
1: Initialize the matrix $A$
2: for each contribution block $c$ do
3:     for each entry $c[i][j]$ of $c$ do
4:         $A[rmap(c,i), cmap(c,j)] += c[i][j]$
5:     end for
6: end for
```

Algorithm 1: Assembly operation.

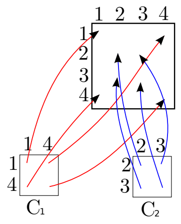


Fig. 1: Assembly operation with 2 contribution blocks.

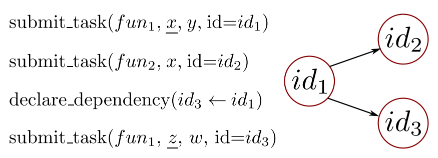
From a general point of view, assembly operations can be viewed as scatter-add operations of each contribution on the matrix following the scheme depicted in Algorithm 1. This operation is commutative and contributions can be treated in any order. For each contribution block, each entry is summed with the corresponding entry of the matrix A. The association between elements of the contribution blocks and entries of A are determined using indirection arrays *rmap* and *cmap* which store the correspondence between local indices within the contribution block and global indices within the matrix A. For example, if we consider the assembly operation the contribution block c_1 (which is a 2 by 2 matrix) presented in Figure 1, $rmap(c_1, 1)$ (resp. $cmap(c_1, 1)$) will be equal to 1 while $rmap(c_1, 2)$ (resp. $cmap(c_1, 2)$) will be equal to 4.

Recently, a lot of work has targeted the implementation of efficient assembly operations for finite element methods running on multicore architectures which may be enhanced with accelerators. The main issue with the parallelization of assembly operations comes from the race conditions which occur when two different contribution blocks need to update the same entry of the global matrix. A naive parallelization scheme of the assembly operation is to process the contribution blocks in a sequential way using a parallel implementation of the assembly of a block. This strategy requires the contribution blocks to be large enough to ensure performance.

Moreover, the approach suffers from the lack of scalability: only intra-block parallelism is exploited. More Recently, in [9] Cecka *et al.* introduced a parallelization approach based on a coloring of the contribution blocks where contribution blocks having the same color can be treated in parallel. This property is guaranteed by the fact that blocks having the same color do not contribute to the same entries of the global matrix. This idea has been pushed further by Markall *et al.* in [15] by improving the coloring scheme in a way such that the number of colors used is reduced. Lately, Hanzlikova *et al.* proposed in [12] an approach which extends the work from Cecka by using extra storage to avoid synchronizations needed to prevent race conditions.

3.2 The StarPU runtime system

As most modern task-based runtime systems, StarPU aims at performing the actual execution of the tasks, both ensuring that the DAG dependencies are satisfied at execution time and maintaining data consistency. The particularity of StarPU is that it was initially designed to write a program independently of the architecture and thus requires a strict separation of the different software layers: high-level algorithm, scheduling, runtime system, actual code of the tasks. We refer to Augonnet *et al.* [5] for the details and present here a simple example containing only the features relevant to this work. Assume we aim at executing the sequence $fun_1(\underline{x}, y)$; $fun_2(x)$; $fun_1(\underline{z}, w)$, where $fun_{i,i\in\{1,2\}}$ are functions applied on w, x, y, z data; the arguments corresponding to data which are modified by a function are underlined. A task is defined as an instance of a function on a specific set of data. The set of tasks and related data they operate on are declared with the **submit_task** instruction. This is a non blocking call that allows one to add a task to the current DAG and postpone its actual execution to the moment when its dependencies are satisfied. Although the API of a runtime system can be virtually reduced to this single instruction, it may be convenient in certain cases to explicitly define extra dependencies. For that, identification tags can be attached to the tasks at submission time and dependencies are declared between the related tags with the **declare_dependency** instruction. For instance, an extra dependency is defined between the first and the third task in Figure 2 (left). Figure 2 (right) shows the resulting DAG built (and executed) by the runtime. The $id_1 \rightarrow id_2$ dependency is implicitly inferred with respect to the data hazard on x while the $id_1 \rightarrow id_3$ dependency is declared explicitly. Optionally, a priority value can be assigned to each task to guide the runtime system in case multiple tasks are ready for execution at a given moment. In StarPU, the scheduling system is clearly split from the core of the runtime system (data consistency engine and actual task execution). Therefore, not only all built-in scheduling policies can be applied to any high-level algorithm, but new scheduling strategies can be implemented without having to interfere with low-level technical details of the runtime system.



```
submit_task($fun_1$, $\underline{x}$, $y$, id=$id_1$)
submit_task($fun_2$, $x$, id=$id_2$)
declare_dependency($id_3 \leftarrow id_1$)
submit_task($fun_1$, $\underline{z}$, $w$, id=$id_3$)
```

Fig. 2: Basic StarPU-like example (left) and associated DAG (right).

```
1: PING(k) : k = 0 .. N
2: RW A ← (k == 0) ? A(k) : A PONG(k-1)
3:          → A PONG(k)

4: PONG(k) : k = 0 .. N
5: RW A ← A PING(k)
6:          → (k == N) ? A(k) : A PING(k+1)
```

Algorithm 2: Ping-Pong algorithm expressed in the PaRSEC dataflow description

3.3 The PaRSEC runtime system

As described in [8], PaRSEC is a dataflow programming environment supported by a dynamic runtime, capable of alleviating some of the challenges imposed by the ongoing changes at the hardware level. The underlying runtime is a generic framework for architecture-aware scheduling and management of micro-tasks on distributed many-core heterogeneous architectures. The dynamic runtime is only one side of the necessary abstraction, as it must be able to discover concurrency in the application to feed all computing units. To reach the desired level of flexibility, we support the runtime with a symbolic representation for the algorithm, able to expose more of the available parallelism than traditional programming paradigms. The runtime is capable of freely exploiting this parallelism to increase the opportunities for useful computation, predict future algorithm behaviors and increase the occupancy of the computing units.

Algorithm 2 represents a concise dataflow description of a ping-pong application, where a data A(k) is altered by two tasks, PING and PONG, before being written back into the original location A(k). Line 1 defines the task PING and it's valid execution space, $\forall k \in [0..N]$. Line 2 depicts the input value A for the task PING(k), where if k is 0 the data is read from an array A(), otherwise it is the output A of a previous task PONG(k-1). Line 3 describes the output flow of the tasks PING, where the locally modified data A is transmitted to a task PONG(k). This task PONG(k) can be executed in the context of the same process as PING(k) or remotely, the runtime will automatically infer the communications depending on the location of the source and target tasks. Lines 4 to 6 similarly depict the complementary task PONG.

Each task consists in the addition to the dataflow definition depicted in the above algorithm, several possible implementations of the code to be executed on the data, the so called codelets. Each codelet is targeted toward a specific hardware device (CPU, Xeon Phi, GPU) or a specific language or framework (Open CL). The decision of which of the possible codelets to be executed is controlled by a dynamic scheduling, aware of the state of all local computing resources. Once the scheduling decision is taken, the runtime provides the input data located on the specific resource where the task is to be executed, and upon completion will make the resulting data available for any potential successors. As the task flow definition includes a description of the type of use made by a task for each data (read, write or read/write) the runtime can minimize the data movements while respecting the correct data versioning. Not depicted in this short description are other types of collective communication patterns that can be either described, or automatically inferred from the dataflow description.

4 Taskified Assembly Operation

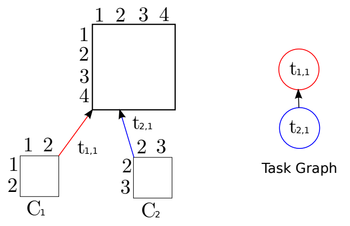


Fig. 3: Naive scheme.

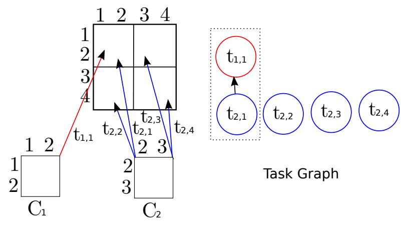


Fig. 4: Tiled scheme.

We introduce in this section a taskified assembly operation where the objective is to enhance parallelism while leaving the management of data constraints and possible race conditions to the underlying runtime system. The main phenomenon which limits the amount of parallelism is the serialization of the assembly of two contribution blocks updating the same block, serialization that prevents possible race conditions. To increase the amount of parallelism, computations must be organized such that conflicting write operations are minimized. A naive approach to express the global assembly operation would be to associate a task to the assembly operation of each contribution block (see Figure 3). In this context, all tasks will be serialized because of the write conflicts on the global matrix. For example, if we consider the assembly operation presented in Figure 3 where this naive scheme is used, the dependency task graph contains 2 tasks (namely $t_{1,1}$ and $t_{2,2}$) which have a write conflict on the global matrix. Note that since the summation operator used during the assembly operation is commutative and associative, the task graph where $t_{1,1}$ is the predecessor of $t_{2,1}$ is also valid. However, for the remaining of this study, we ignore the commutativity of the assembly operation, and will impose a writing order by ordering the tasks generation and declaration. With such an approach, the runtime is now responsible to order the assembly operations with respect to the depicted data dependencies, preventing all conflicts between accesses to the same data.

In order to exhibit more parallelism, one could partition the global matrix into blocks and associate a task to the assembly operation of each contribution block into each tile of the global matrix (see Figure 4). Of course, if a contribution block does not update a tile of the global matrix, the corresponding empty task is not considered. By doing so, the amount of non-conflicting tasks is increased leading to higher degree of parallelism. For example, if we consider now the assembly operation described in Figure 4 where this tile-based scheme is used, we can see that the task graph contains now 5 tasks for which there is only one conflict between $t_{1,1}$ and $t_{2,1}$. When using this scheme, the number of tasks and subsequently the degree of parallelism is strongly linked to blocking factor used for the global matrix. A trade-off needs thus to be found between the needed parallelism and the management overhead induced in the runtime system. The approach to taskify the assembly operation that we propose is a tiled approach where the serialized tasks are sorted according to their computational cost in each chain: the most costly tasks are treated first. The task graph is thus composed by a set of independent chains of tasks. This scheme will be referred to as the *flat assembly operation scheme.*

To overcome the overhead due the management of the large number of tasks, one could decrease the number of tasks for a fixed tile size by merging the chains of the flat assembly scheme into a single tasks. This will produce a fixed number of tasks corresponding to the number of tiles of the global matrix. This approach is similar to [9], in the sense that it builds a set of completely independent tasks preventing all race conditions from occurring. This is illustrated in Figure 4, where the chain is replaced by the dashed box surrounding it. In the rest of the paper, this scheme will be referred to as *no-chain assembly operation scheme.*

4.1 Scheduling strategies for taskified assembly operations

Taskified assembly operations can are expressed using task dependency graphs composed of independent chains of tasks (an example is given in Figure 4). In this paper, we consider dynamic on-line scheduling strategies which are commonly used in various runtime systems. In order to efficiently assign tasks to the computational resources it is important to take into account the weight of each task in terms of workload and give priority to the largest ones (the larger the contribution the higher its priority is). This strategy is used on the set of ready tasks (i.e. tasks for which the corresponding dependencies are satisfied) and each idle processing unit picks the task with highest priority from the set of ready tasks. By doing so, the processing units are constantly working on the critical path of the execution. Varying the tasks priorities allow for further improvement of the scheduling strategy.

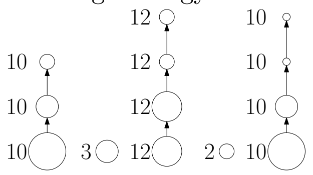


Fig. 5: Fixed priorities.

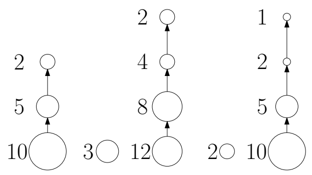


Fig. 6: Adaptive priorities.

A first approach to express the critical aspect of a task regarding the length of the chain it belongs to, is to associate a priority related to the cost of the entire chain. This illustrates in Figure 5, where the priorities of the entire chain are constant, and are computed based on the cost of the entire chain. We will refer to this priority scheme as *fixed priority scheme*.

This priority management can be pushed further so that the priorities, not only take into account the absolute length of the critical path but its current length at the moment where the scheduling decision is taken. Thus, the priority of a task is computed based on the remaining workload on the chain it belongs to. This allows the working units to select the tasks that are currently the most critical. Figure 6 depicts the same example as before using this new priority assignment scheme. This time the tasks belonging to a chain have a priority linked to the length of the remaining part of the chain. We will refer to this priority scheme as *adaptive priority scheme*.

One of the major differences between StarPU and PaRSEC is the way the list of ready tasks is managed. In StarPU, the user divides data, precomputes a list of tasks working on those data, and submits, in advance, all the tasks. This sequential submission of tasks creates implicit dependencies between the tasks. In PaRSEC, the dependencies are explicitly specified by the user, and the tasks are dynamically discovered by the runtime based on completed dependencies and the symbolic description of the algorithm. From the scheduling point of view, StarPU gives the opportunity to the user to write his own scheduler while in PaRSEC, a highly optimized scheduler is provided, where priorities are secondary to enforcing a coherent data locality policy.

5 Experimental results

We evaluate the behavior and performance of our task-based approach on the `riri` platform, composed by 4 Intel E7-4870 processors having 10 cores clocked at 2,40 GHz and having 30 MB of L3 cache. The platform has uniform memory access (UMA) to it's 1 TB of RAM. In all cases the results presented are averages over multiple runs (at least 10), where the outliers have been cleaned. In addition to the results presented here, we also analyzed the standard deviation, but we decided not to report it as is was under the system noise (2%).

We have chosen to illustrate the behavior of our approaches on two different classes of problems. The first class correspond to assembly operations met in finite element methods. We consider in the following study both 2D and 3D finite element continued method applied on structured meshes. The difference between the two cases resides in the connectivity between elements. While on a 2D grid, each element has at most 8 neighbors, in 3D, each hexahedron has 26 neighbors leading to higher overlapping between contribution blocks for the 3D case. The second class correspond to a less structured assembly operations met in a sparse direct method (namely the multifrontal method [10]). The considered configuration has been generated using the `MUMPS` sparse direct solver [4] using input problems coming from the University of Florida Sparse Matrix Collection [4]. To be more precise, we extracted configurations met during the assembly phases

[4] `http://www.cise.ufl.edu/research/sparse/matrices`

needed by the sparse LU factorization. The contribution blocks for these configurations are very irregular with sizes varying from 0.01% to 99% of father's size. Thus, we are not analyzing the task-based implementation asymptotically on large benchmarks, but on real-life cases extracted existing applications. Finally, two parameters will vary in our experiments, the size of the tile and the number of computational resources. The bigger the tile size, the lesser parallelism one will be able to exhibit. Thus, one shall find an acceptable value in sync with the second parameter, the number of computing resources units available.

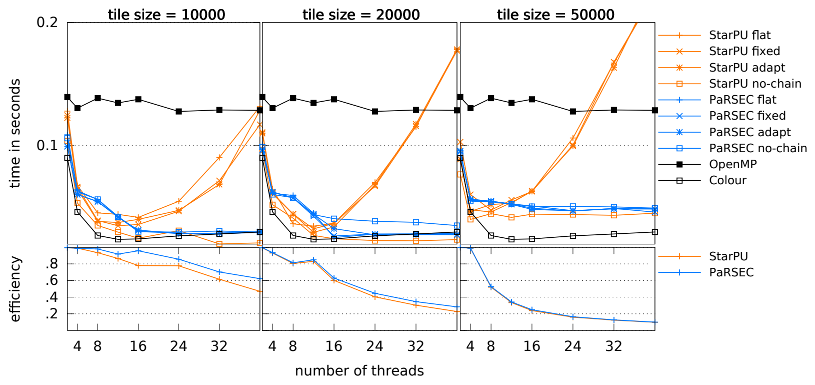


Fig. 7: Comparison of the performance of an assembly operation using a 2D mesh with 2025 blocks of size 121^2 based on the granularity of the operation (as depicted by the tile size). The matrix has 203k entries and respectively 61, 31, and 13 active tiles (from left to right).

Figure 7 depicts the performance of the assembly operation when used in the context of a finite element method application in a 2D mesh case. This corresponds to a case where the overlapping between contribution blocks is small. First of all, we can observe that, by increasing the concurrency (leftmost plot), the taskified assembly operation obtains a very good behavior with all strategies in PaRSEC. Moreover, we observe that the StarPU implementation has a good behavior on a small number of processing units but seems less efficient when the number of resources increases. As shown in the bottom part of the graph this is mainly due to the overhead induced by the management of the tasks, the tasks are not compute intensive enough to amortize the overhead of the StarPU runtime system (which is mainly due to the inference of task dependencies). Similarly, we can notice that independently from this observation, the *no-chain assembly operation scheme* behaves well in both runtime systems mainly because there are no race conditions in this strategy. We can see also, that this strategy gives performance equivalent to the one obtained with the coloring strategy described in [9] and outperforms it in certain configurations (typically when there the global matrix is tiled using fine grain blocks). This illustrates the interest of our taskified assembly scheme on this simple scenario.

In Figure 8 we investigate the behavior of the taskified assembly operation on the two runtime systems in the context of a finite element method application in a 3D mesh case. This time both the size of the contribution blocks and their overlapping increased in comparison with the 2D case.

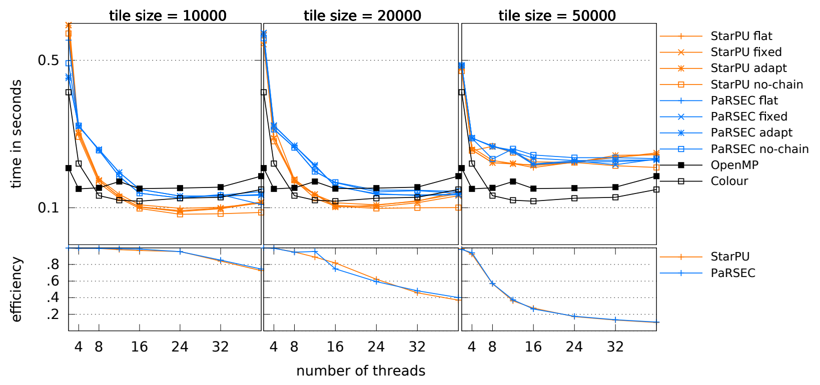


Fig. 8: Comparison of the performance of an assembly operation using a 3D mesh with 512 blocks of size 512^2 based on the granularity of the operation (as depicted by the tile size). The matrix has 185k entries and respectively 121, 44, and 10 active tiles (from left to right).

We can observe that the functioning of our taskified schemes have a good behavior for all tile sizes. Moreover, we can observe that the overhead of the runtime system is negligible compared with the computational cost of the tasks and allow all the strategies to expose a scalable behavior. Concerning the coloring scheme, it is outperformed by all the strategies when the number of computational resources increased. Finally, once again, the *no-chain assembly operation scheme* is the most efficient variant for both runtime systems.

Finally, Figure 9 reports the results gathered in the context of the most irregular and complex case: assembly operations arising in the sparse LU factorization using the multifrontal method. First of all, note that in this case, it is not possible to use the coloring heuristic since the overlapping between contributions blocks may be arbitrarily large (the cost of the coloring heuristic is prohibitive in this case). We can observe that PaRSEC has a good performance with all tiling strategies and all scheduling policies. We can also see that the *adaptive priority* scheduling policy is the one with the most scalable behavior. Finally, we can observe that the overhead induced by the runtime is minimal with PaRSEC. Concerning StarPU, when the granularity of the tiles is small, we measure that the overhead of the runtime system tends to increase with the number of resources leading to a significant performance loss. However, increasing the granularity allows to overcome the runtime overhead and the behavior of StarPU becomes equivalent to the one obtained with PaRSEC. Once again, the *no-chain assembly operation scheme* is the most efficient variant for both runtime systems. Finally, we report also, the behavior of the naive implementation using based on OpenMP where all the global matrix is not tiled and the contribution blocks are treated sequentially using as many threads as provided by the user for each contribution block. We can see, that our taskified assembly scheme is much more stable in terms of behavior and outperforms the OpenMP implementation for most non-trivial cases. Even though our strategies and the OpenMP implementation are extremely close on some experiments, our approach permits to relax synchronizations once integrated into an application, enabling additional overlap between the assembly operations and the rest of the computations and the entire application will benefit. From this perspective, these experimental results illustrate the interest of our taskified scheme.

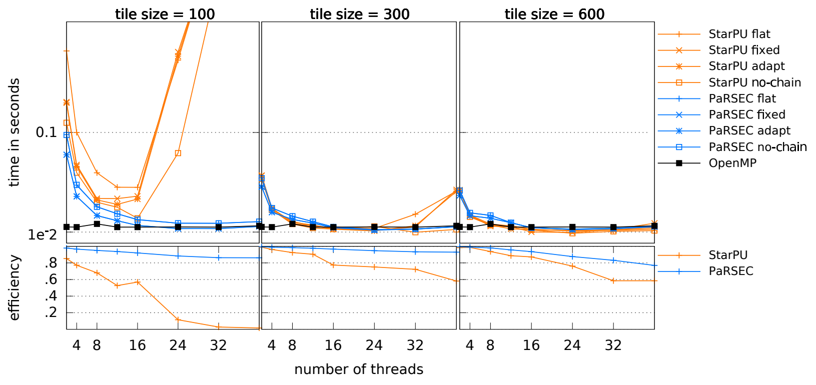


Fig. 9: Comparison of the performance of an assembly operation coming from the `MUMPS` solver based on the granularity of the operation.

6 Conclusion

In this work we evaluated the usability and effectiveness of general-purpose task-based runtime systems for parallelizing the assembly operation, which is a main operation in several application fields. We expressed the assembly operation as tasks with data dependencies between them and provided the resulting task graph to a runtime systems. Several algorithms aiming at enhancing the concurrency while trying to reduce the number of race conditions have been proposed, and they were analyzed under different dynamic constraints: tasks priority and granularity. Overall, the results clearly indicates that for both runtime systems, namely PaRSEC and StarPU, our approach exhibits encouraging performance, especially when the right balance is reached between the task granularity and the overhead of the runtime system.

In the near future, we plan to further extend this work by using accelerators (GPU, Intel Xeon-Phi, etc) to minimize the time-to-solution. This will be done by relying on existing assembly kernels for the different accelerators and leave the data management and scheduling decisions to the runtime systems (the scheduling policies need to be adapted to the heterogeneous context). Moreover, it could be of interest to consider intra-task parallelism which may offer more flexibility to enhance concurrency. In a longer term, this work represents a necessary kernel which will be used to design complex numerical simulation applications on top of modern runtime systems. This will allow the application to run in a more asynchronous way without relying on the classical fork-join paradigm.

Acknowledgments This work was partially supported by the French ANR through the MN (Solhar ANR-13-MONU-007 project) program, and the US Department of Energy through the DE-FG02-13ER26151.

References

1. E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi. Task-based fmm for multicore architectures. *SIAM SISC*, 36(1), 2014.
2. E. Agullo, A. Buttari, A. Guermouche, and F. Lopez. Multifrontal QR factorization for multicore architectures over runtime systems. In *Euro-Par 2013 Parallel Processing - 19th International Conference*, pages 521–532, 2013.
3. E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics*, 180(1), 2009.
4. P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel Computing*, 32(2):136–156, 2006.
5. C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience*, 23:187–198, February 2011.
6. R. M. Badia, J. R. Herrero, J. Labarta, J. M. Pérez, E. S. Quintana-Ortí, and G. Quintana-Ortí. Parallelizing dense and banded linear algebra libraries using SMPSs. *Concurrency and Computation: Practice and Experience*, 21(18), 2009.
7. 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.
8. George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J. Dongarra. PaRSEC: Exploiting heterogeneity to enhance scalability. *Computing in Science and Engineering*, 15(6):36–45, 2013.
9. C. Cecka, A. J. Lew, and E. Darve. Assembly of finite element methods on graphics processors. *Int. J. for Numerical Methods in Engineering*, 85(5):640–669, 2011.
10. I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Transactions on Mathematical Software*, 9:302–325, 1983.
11. Z. Fu, T. J. Lewis, R. M. Kirby, and R. T. Whitaker. Architecting the finite element method pipeline for the GPU. *Journal of Computational and Applied Mathematics*, 257(0):195 – 211, 2014.
12. N. Hanzlikova and E. R. Rodrigues. A novel finite element method assembler for co-processors and accelerators. In *Proceedings of the 3rd Workshop on Irregular Applications: Architectures and Algorithms*, NY, USA, 2013. ACM.
13. P. Huthwaite. Accelerated finite element elastodynamic simulations using the GPU. *Journal of Computational Physics*, 257, Part A(0):687 – 707, 2014.
14. X. Lacoste, M. Faverge, P. Ramet, S. Thibault, and G. Bosilca. Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes. Rapport de recherche RR-8446, INRIA, January 2014.
15. G. R. Markall, A. Slemmer, D. A. Ham, P. H. J. Kelly, C. D. Cantwell, and S. J. Sherwin. Finite element assembly strategies on multi-core and many-core architectures. *International Journal for Numerical Methods in Fluids*, 71(1):80–97, 2013.
16. G. Quintana-Ortí, E. S. Quintana-Ortí, R. A. van de Geijn, F. G. Van Zee, and E. Chan. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Trans. Math. Softw.*, 36(3), 2009.