Towards an efficient Task-based Parallelization over a Runtime System of an Explicit Finite-Volume CFD Code with Adaptive Time Stepping

Jean Marie Couteyen Carpaye, Jean Roman, Pierre Brenner

Towards an efficient Task-based Parallelization over a Runtime System of an Explicit Finite-Volume CFD Code with Adaptive Time Stepping

Jean Marie Couteyen Carpaye[1,2], Jean Roman[1], Pierre Brenner[2]
[1]Inria Bordeaux Sud-Ouest, 200 Avenue de la Vieille Tour, Bordeaux, France
[2]Airbus Defence & Space, 66 Route de Verneuil, 78130, Les Mureaux, France

***Abstract*—FLUSEPA[1] is an advanced simulation tool which performs a large panel of aerodynamic studies. It is the unstructured finite-volume solver developed by Airbus Defence & Space company to calculate compressible, multidimensional, unsteady, viscous and reactive flows around bodies in relative motion. The time integration in FLUSEPA is done using an explicit temporal adaptive method. The current production version of the code is based on MPI and OpenMP. This implementation leads to important synchronizations that must be reduced. To tackle this problem, we present a first study of a task-based parallelization of the solver part of FLUSEPA using the runtime system StarPU and combining up to three levels of parallelism. We validate our solution on the simulation of a take-off blast wave propagation for Ariane 5 launcher.**

I. Introduction

For industrial applications of numerical simulation, the most common architecture is nowadays clusters composed of SMP nodes of multicore processors. To develop parallel codes on those machines, a common way is to rely on MPI [1]. While it is possible to use only one core per process and to rely only on Flat-MPI, this approach does not generally scale and it is even worse for codes with a large potential imbalance during execution. A common way to reduce the number of processes while using the same number of cores consists in using OpenMP [2] inside the SMP nodes, leading to a two level parallelism.

Another problem is the increasing of the heterogeneity of architectures. Accelerators (e.g. GPGPU, Xeon Phi) are now available and the design of efficient industrial applications exploiting distributed heterogeneous systems with non uniform memory accesses is a complex challenge at large scale.

Therefore, applications tend to evolve slowly compared to architectures and achieving performance with a new architecture is a time-consuming task for the developers.

Task-based programming is a good candidate to deal with those issues: describing the problem in a generic manner as a DAG of tasks allows more potential flexibility to exploit the architectures. A runtime system is then in charge of mapping tasks among computational resources (CPU cores and/or accelerators) and of managing memory transfers. By using a powerful abstraction of parallel codes and an efficient runtime system, one can expect to achieve performance quickly on different kinds of architectures (performance portability issue [3],[4]).

[1]Registered trademark in France No. 134009261

The aerospace industry faces a lot of complex problems for which actual experiences are not doable: e.g. take-off blast wave propagation, stability at reentry into Earth atmosphere. Those problems are time-dependent and involve strong shocks.

In this paper, we describe the "taskification" using the runtime system StarPU [3] of the aerodynamic solver of the FLUSEPA code [5] which is an MPMD MPI/OpenMP code. This code is able to compute stage separation. Each body is meshed separately and can move using an ALE formulation [6], leading to mesh intersection computations. The code is also well adapted for unsteady computations. Figure 1 shows a separation computation: each booster and the main stage are meshed separately.

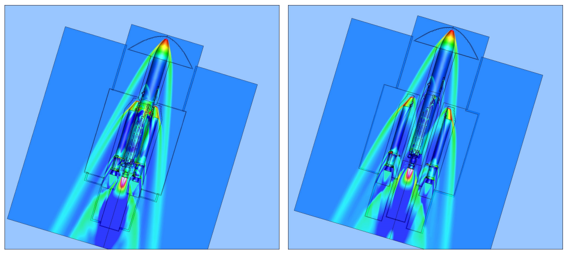


Fig. 1: Booster separation.

The aerodynamic solver of FLUSEPA uses a finite-volume (FV) discretization and an explicit temporal adaptive time stepping scheme. This kind of scheme is well suited for our class of problem because it is conservative and consistent in time ([7], [8], [9]). The method is designed to minimize the computational cost, but it introduces several difficulties for an efficient parallelization (synchronizations, load balancing problems).

The paper is organized as follows. Section II briefly presents the main computational methods used in FLUSEPA and the existing MPI/OpenMP code. Section III presents the runtime system StarPU and Section IV focuses on the task parallelization of the aerodynamic solver. Section V presents an experimental study from an industrial test case, the take-off blast wave propagation. Finally, Section VI gives some conclusions and perspectives for this work.

II. Main numerical methods used in FLUSEPA

The two main computation steps performed in FLUSEPA, the aerodynamic solver and the mesh intersection computations, are linked by a kinematic computation. During aerodynamic computations, external forces apply to the bodies. From the computation of these loads, a kinematic is obtained. If the bodies moved sufficiently since the last mesh intersection computation, the meshes are moved according to the computed kinematic. This general computational framework is summed up in Algorithm 1.

Algorithm 1 General iteration

```
1: Aerodynamic solver computation
2: Computation of a new_kinematic
3: if important motion since last mesh intersection then
4:     Body displacement(new_kinematic)
5:     Intersection computation
6: end if
```

A. Finite-volume aerodynamic solver

Finite-volume method is a discretization technique where the integral formulations of the conservation laws are discretized directly in the physical space. The finite-volume method is naturally conservative. We look for the numerical solution of the compressible Navier-Stokes equations in Reynolds-Averaged form that can be written as :

$$\frac{d}{\partial t}\iiint\limits_{\Omega_{CV}} \mathbf{w} d\Omega = - \oiint\limits_{A_{CV}} \mathbf{F}\mathbf{n} dS + \iiint\limits_{\Omega_{CV}} \mathbf{S} d\Omega \tag{1}$$

where Ω_{CV} is a fixed control volume (3D cell) with boundary A_{CV} (2D faces), $\mathbf{n}$ is the outer-oriented unit normal, $\mathbf{w}$ is the conservative variable vector, $\mathbf{F}$ is the flux density and $\mathbf{S}$ is the source term vector. So, the solver mainly manipulates cells and faces. Field values (e.g. pressure, temperature) are computed for cells and flows are evaluated between faces of cells. See [10] for more details.

B. Temporal adaptive explicit solver

The aerodynamic solver used is explicit and based on a temporal adaptive time stepping scheme. When using an explicit temporal formulation, the maximum allowable time step of a cell is given by its CFL number which depends mostly of the volume of the cell. For an explicit solver, the CFL must be inferior to 1. In classical explicit solvers, the time step is determined by the slowest cell (the cell which has the slowest time step while respecting the CFL condition).

Because we consider complex real size problems, the mesh resolution is not uniform, so using the lowest physical time step would be very penalizing for larger cells. The temporal adaptive algorithm allows to compute each cell near its maximum allowable explicit time step, ranking them in levels; these temporal levels τ are numbered from 0 to a given value θ.

The temporal adaptive method is described in Algorithm 2. At line 1, the maximum allowable time step is computed for each cell. The slowest cell is also found and defines Δt, the minimum time step in the computation. At line 2, according to Δt and its maximum allowable time step, each cell is classified inside a temporal level. Inside a temporal level τ, cells are computed at the same time step which is $2^{\tau} * \Delta t$.

Algorithm 2 Temporal adaptive time stepping scheme in FLUSEPA: one iteration of the aerodynamic solver

```
1: Time step computation
2: Classification of every cell inside a temporal level
3: *Temporal adaptive loop:*
4: for subiteration=1 to $2^\theta + 1$ do
5:     $\tau = 0$
6:     for $tmp = 1$ to $\theta + 1$ do
7:         if $(mod(subiteration - 1, 2^{tmp}) == 0)$ then
8:             $\tau = tmp$
9:         end if
10:    end for
11:    if subiteration>1 then
12:        Intensive Correction (0 to $\tau$)
13:        Intensive interpolation ($\tau + 1$)
14:    end if
15:    *Predictor:*
16:    Gradient computation (0 to $\tau$)
17:    Limitation and flow reconstruction (0 to $\tau$)
18:    Flow repositionning ($\tau + 1$ class)
19:    Riemann Solver (0 to $\tau$)
20:    Flow sum on cells (0 to $\tau$)
21:    if $\tau \neq \theta$ then
22:        Intensive repositionning ($\tau + 1$)
23:    end if
24:    for $\tau' = \tau$ to 0 do
25:        *Corrector:*
26:        Extensive prediction ($\tau'$)
27:        Intensive prediction ($\tau'$)
28:        Gradient computation ($\tau'$)
29:        Limitation and flow reconstruction ($\tau'$)
30:        Flow interpolation ($\tau'$)
31:        Riemann Solver ($\tau'$)
32:        Flow sum on cells ($\tau'$)
33:        Extensive Correction ($\tau'$)
34:        Intensive interpolation ($\tau'$)
35:    end for
36: end for
```

An iteration of the algorithm is composed of multiple subiterations. There are 2^{θ} subiterations, θ being the level of the fastest cells which are the ones that need only one subiteration to reach the time of the end of the iteration.

The levels τ that are computed are determined by lines 5 to 9. For example, with $\theta = 2$, τ will take successively the following values : 2,0,1,0. Figure 2 shows how level τ evolves after each subiteration in this case; the evolving levels τ are in red. The physical time reached after one iteration with $\theta = 2$ is equivalent to 4 iterations with a global time step.

To stay consistent in time, the computations have to be

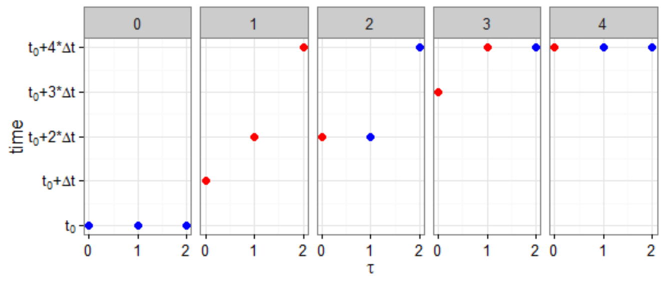


Fig. 2: Time reached after each subiteration for each level τ ($\theta = 2$; 4 subiterations).

performed in a specific order. When computing a flow between two cells, they must be at the same time. The cells can only have neighbor cells of the same level τ or with the level values $\tau - 1$ or $\tau + 1$. If cells are near other cells with a different temporal level, they are positioned at a time that will ensure a consistent computation (Lines 18, 22, 30, 34) [7]. In this way, the computation order is strict and each temporal level is integrated at a specific moment.

The interest of the method depends strongly of the distribution in temporal levels. Let us denote by $\Omega(\tau)$ the set of cells at temporal level τ and by Ω all the cells in the global domain.

The computational cost of a temporal level τ can be estimated by $C(\tau) = 2^{\theta-\tau} * |\Omega(\tau)|$ and the cost of an iteration described by Algorithm 2 is $\sum_{\tau=0}^{\theta} C(\tau)$.

If we compare the computational cost needed to reach the same time with a global time step $2^{\theta} * |\Omega|$, the cost ratio is

$$\frac{2^{\theta} * |\Omega|}{\sum_{\tau=0}^{\theta} 2^{\theta-\tau} * |\Omega(\tau)|}$$

which is superior to 1.

However, this estimation is an upper bound because of the overhead induced by the temporal adaptive method. Several interpolations are not taken into account in this cost ratio, while their importance increases with the number of temporal levels. More temporal levels imply more cells concerned by interpolation and the cells with a small time step are interpolated more often.

C. *MPI-OpenMP version of FLUSEPA*

FLUSEPA uses several kinds of processes to handle the numerical coupling between aerodynamic computations and body movements. Three kinds of processes are used: one process is in charge of coordinating the simulation, some other processes are used to compute the aerodynamic solution and the last kind of processes is dedicated to compute mesh intersections [5].

We focus here on the aerodynamic solver computations.

The code uses unstructured meshes in order to take into account complex geometries for interstages. The elements manipulated by the solver are the cells and the faces between them.

The current parallel version is based on a two-level parallelism: MPI [1] processes associated with a spatial decomposition and OpenMP [2] parallelism inside them.

Making a MPI version of a FV code is usually done by a Domain Decomposition approach. Each process is in charge of a portion of the initial domain and ghost cells are used in order to ensure efficiently communications between subdomains. At the border of a subdomain, faces are duplicated, but each cell belongs to only one subdomain. Figure 3 illustrates this spatial decomposition with 2 subdomains: for the red subdomain, the dark red part corresponds to the border cells, the light red part to inner cells, the dark green cells are the ghost cells of this red subdomain, and finally bold black faces are the duplicated border faces.

MPI communications are only done when necessary according to the temporal level of cells that is currently computed. Border cells are tagged and are computed as soon as possible according to their temporal level. MPI asynchronous communications are used to ensure a good computation-communication overlapping.

The second level of parallelism is achieved in shared memory by using OpenMP directives (OMP DO) applied to loops concerning the cells and faces inside a subdomain.

The main problem of this spatial domain decomposition is the fact that the cells have not the same computational cost which is determined by their temporal level. To ensure a better load balancing, we give a weight to each cell of the subdomain according to its temporal level. However, the temporal level of a cell can change between iterations and a recomputation of a new domain decomposition is needed periodically.

Despite that, the way the time is integrated leads to an important time wasted in synchronizations. The time integration implies a strict order for the cells to be processed depending on their temporal level: neighbor cells must be at the same time during computation.

This temporal locality information is partially lost with the current parallelization. This issue is one of the key elements to justify the development of a task-based version of the aerodynamic solver. By working on subdomains inside each process, we want to be able to capture all the dependencies during computations and to exploit more asynchronism with the help of a runtime system.

III. The StarPU task-based runtime system

There exist different libraries and frameworks to exploit task-based parallelism: e.g. SMPSs [11], StarPU [3] , PaRSEC [4], CnC [12], Legion [13], SuperMatrix [14].

As said in the introduction, the main goal of task-based programming is to ensure performance portability on heterogeneous manycore distributed platforms. In this framework, the algorithm is described as a sequential task flow with data dependencies expressed through read/write attributes for each task parameter. This task flow is translated into a Direct Acyclic Graph (DAG) of tasks: the nodes represent the tasks and the edges between the nodes are the dependencies. Then, a runtime system is in charge of scheduling the tasks over the computational resources (CPU, GPU) and of managing the data transfers.

Good results have been achieved by this approach for dense linear algebra [15] and sparse linear algebra [16]. There are some other works about parallelization of applications over runtimes: S3D over Legion [13], ScalFMM over StarPU [17].

To construct the DAG, there are two common approaches. First, a Parametrized Task Graph (PTG) can be used [18]: tasks are not enumerated but parametrized and dependencies between tasks are explicit. Another way is to use the Sequential Task Flow (STF) model. With STF, tasks are inserted from the main program and the dependencies are computed at task insertion according to data accesses [19]. One advantage of the later model is the fact that tasks can be inserted according to the results of previous computations.

The StarPU runtime system [3] relies on the STF model and computation resources (e.g. CPU, GPU) are seen as workers.

Tasks are inserted from the main program through calls to `starpu_insert_task`. The dependencies between the tasks are then computed by the runtime system according to the data accesses and how they are accessed (read, write, read-write). This implies that the DAG is unrolled during the execution. The task insertion is asynchronous: the computation can start even if not all the tasks have been inserted. When a task is ready (i.e. all its dependencies have been fulfilled), it becomes available to the scheduler. Then, according to the scheduling strategy, workers pop tasks and execute them. Some hints are available to the scheduler: at task insertion, it is possible to give a priority to a task and some schedulers can take advantage of this information. It is also possible to have performance models which will compute a weight for a task according to various parameters (e.g. size of the data, targeted architectures) in order to help the scheduler.

A. *Parallel tasks and worker-contexts*

It is possible to exploit existing OpenMP code within StarPU through the use of context [20]. A context can be seen as a set of computational resources with a scheduling strategy in which tasks can be submitted. Contexts can be nested.

When no scheduling strategy is specified, the context is seen as a worker (called worker-context): when a task is pushed on this worker-context, it is supposed to be executed on all its computational resources.

To use an OpenMP code inside a worker-context, a specific initialization task that will bind OpenMP threads to the CPUs of the worker-context must be used. By default, tasks are inserted in a global context. So, if one wants to create 4 worker-contexts of 4 CPUs, he has to create a main-context with a scheduling strategy that will contain these worker-contexts and insert tasks in this main-context.

B. *Distributed parallelism with StarPU*

There are two ways to use MPI parallelism with StarPU. First, StarPU provides `starpu_mpi_insert_task` that lets StarPU handle all the communications and data transfers. With this approach, all the tasks must be inserted in all the nodes of the cluster.

It is also possible to describe communications explicitly using `starpu_mpi_isend_detached` / `starpu_mpi_irecv_detached`. When using those primitives, communications are consistent with the computation dependencies inferred by task insertion inside the nodes.

IV. MPI + Task design of the aerodynamic solver

A. *Computation Elements*

As the first level of parallelism we can exploit in FLUSEPA is spatial, the natural way to described tasks is to use multiple subdomains. In the DAG, we want to express dependencies in such a way that allows the maximum concurrency. In order to achieve this, we use an algorithmic abstraction called "Computation Element" (CE).

The code mainly manipulates faces and cells and this must be exploited to have a good task parallelization. Four main basic computation patterns can be found in the original code: computation on cells, computation on faces, computation on faces using cell values, computation on cells using face values. However, the way the code has been written does not lead to a clear distinction between those patterns when they were used, and thus a rewritting step has been performed. Most of the computation kernels of the aerodynamic solver will now correspond to one of these computation patterns in order to achieve a well-structured task version (cf. Algorithm 3).

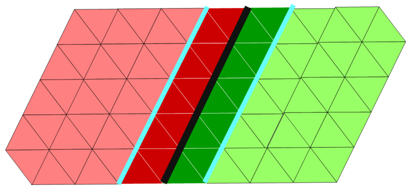


Fig. 3: Illustration of two CEs (a red one and a green one).

The domain partitioning is done with the graph partitioner SCOTCH[2] [21]. For each subdomain created, we associate a CE containing informations to retrieve each field or flow value (See Section II-A). The different components of a CE are shown at Figure 3: there are two CEs, a red one and a green one. Border cells (dark red and dark green) and inner cells (light red and light green) of the CEs are distinguished. In a CE, faces are classified into different categories: faces between inner cells (black), faces between border cells (white), faces between border/inner cells (bold light blue). Between CEs, we also have inter-CE faces (bold black). Because we have a unstructured mesh, all these topological informations are precomputed for each CE after the partitioning step.

B. *Task generation*

In order to generate tasks, we mainly use the different data types we described previously and exploit the associated

[2]https://gforge.inria.fr/projects/scotch/

Algorithm 3 Task insertion for the Aerodynamic Solver

```
 1: Insert tasks for time step computation
 2: Insert tasks for classification of cells in temporal levels
 3: Wait all tasks
 4: Compute hints (scheduling and packing algorithm)
 5: Temporal adaptive loop:
 6: for subiteration=1 to $2^\theta + 1$ do
 7:     Compute $\tau$
 8:     Insert task for predictor (0 to $\tau$)
 9:     for $\tau' = \tau$ to 0 do
10:         Insert task for corrector ($\tau$)
11:     end for
12: end for
13: Wait all tasks
14: Send informations to master process
15: Foreach cells update intensive values
16: Wait all tasks
```

computation patterns. We rely on the STF model of StarPU. For each pattern, we used a "foreach" function that is in charge of generating the right tasks. We mainly exploit Algorithm 2, each line being converted in one or several "foreach".

The task insertion follows Algorithm 3. The time step computation (line 1) and the classification of cells in temporal levels for each CE (line 2) are also done using tasks. Currently, we have one synchronization after the classification of cells in temporal levels. In order to know which task must be inserted, the temporal levels inside each CE must be known. Some CEs do not contain cells of some temporal levels and this fact is exploited to avoid insertion of useless empty tasks.

The foreach functions can generate a high number of tasks and in particular chains (succession of tasks that depend only of one previous task). In order to reduce the number of tasks, we implemented a strategy to pack tasks at runtime.

Computation kernels are written in such a way that they only write one component of a CE and the pack mechanism relies on that. We have 3 kinds of task packs, each one for modifying data either on border cells, inner cells or faces. For each CE, these task packs exist.

The foreach functions know which type of component (face or cell) will be modified. If it is the same at the previous one, tasks will be added to the corresponding pack associated to the CE. Otherwise, the previous packs are inserted as tasks and new packs are created.

Another case that generates chains of tasks happens when cells of a given temporal level are present only in the inner cell component of a CE. For each CE, we check if it is the case at line 4 of Algorithm 3. To handle this particular context, we introduce a 4th kind of pack called "large task pack". As there is no interaction with other CEs, they contains all the tasks for a given temporal level and a given CE, disregarding the type of component currently modified.

Optimizing this point is critical because tasks that work on cells with low temporal levels are numerous and they work on a low number of cells. This fact is shown in Table I which gives the proportion of cells in each temporal level and the proportion of associated computation for the test case we use in Section V. The large majority of cells are of the higher temporal level (τ_4). Few cells belong to low temporal levels, so we can expect that most of them won't be part of the border component of a CE.

	τ_0	τ_1	τ_2	τ_3	τ_4
Cells	0.05%	2.42%	2.95%	5.77%	88.80%
Computation	0.64%	14.61%	8.93%	8.73%	67.10%

TABLE I: Cell distribution and associated computation.

Figure 4 shows a DAG for an iteration with a maximum temporal level $\theta = 2$, three temporal levels of cells, 4 sub-iterations and 6 CEs. Colors represent the different CEs. The red CE contains cells of the three levels, two other CEs (blue and purple) contain τ_1 and τ_2 temporal level cells. The remaining three CEs contain only τ_2 temporal level cells.

The colored diamonds correspond to computations that modify data for faces of a CE, black diamonds for inter-CE faces, small circles for border cells and large circles for inner cells. Triangles represent large task packs: in this case, each large task pack contains more than 10 subtasks. In this DAG, we can notice that τ_0 temporal level cells are only present in the inner component of the red CE as there are large task packs.

Increasing the number of CEs would enlarge the width of the DAG and using more temporal levels would enlarge its height for some CEs. The DAG can really be unbalanced according to the cell distribution in temporal levels inside each CE, so the way the graph is traversed can have a strong impact on the efficiency of the computation.

C. MPI + Task parallelization

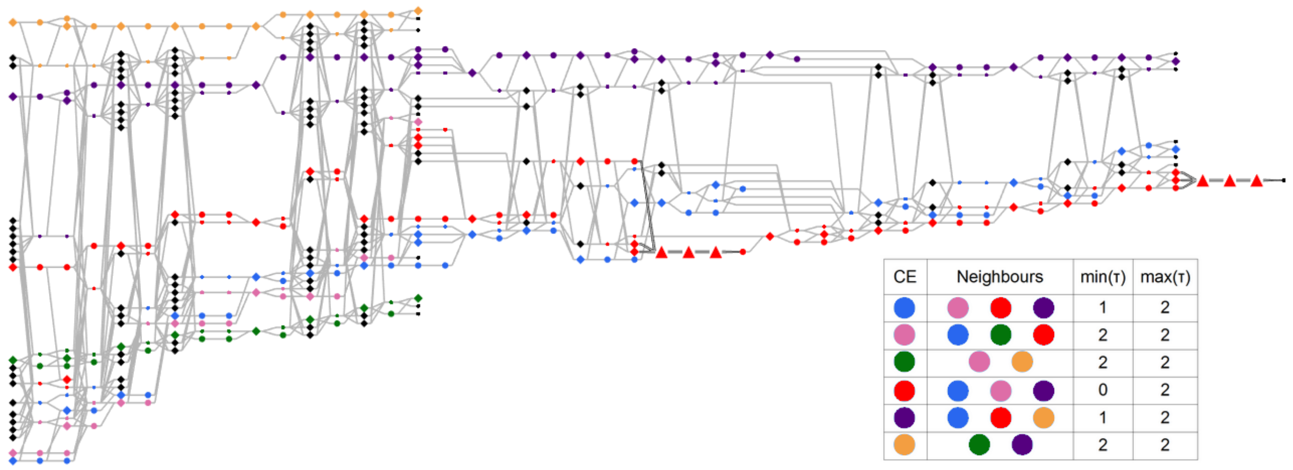
As we insert tasks according to previous computations, we decided to rely on explicit communications.

Each process gets a domain after the domain partitioning. At this moment, for each process, border cells are identified and border faces are duplicated. In order to distinguish border cells of the domain associated to a process and border cells of the component of a CE, we note the first ones "MPI-border cells".

These MPI-border cells are then included to the border component of the CE they belong. After that, each process communicates with its neighbors to complete informations for the common border cells. For each (local CE, foreign CE) couple, we create a border ghost cell component.

For the management of communications, we use for sending data a task that copies border cell content to a temporary buffer and this buffer is sent using `starpu_isend_detached`. For receiving data, the `starpu_irecv_detached` operation is used in a temporary buffer and then, we copy the data in the ghost cell component.

We use a temporary buffer because we do not want to send the whole border component of a CE: some border cells are



CE	Neighbours	min(τ)	max(τ)
🔵	🩷 🔴 🟣	1	2
🩷	🔵 🟢 🔴	2	2
🟢	🩷 🟠	2	2
🔴	🔵 🩷 🟣	0	2
🟣	🔵 🔴 🟠	1	2
🟠	🟢 🟣	2	2

Fig. 4: DAG for a computation with 6 CEs and $\theta = 2$.

not MPI-border cells and when processing low temporal levels, a fraction of the MPI-border cells is concerned.

Before foreach functions that uses cells in read mode, we insert communications as tasks.

Our task-based implementation is able to exploit 3 levels of parallelism : between computation nodes, MPI is used; inside a SMP node, StarPU tasks are used; and tasks can be parallel by using OpenMP.

V. Experimental study

For the experimental validation of this work, we used a cluster with nodes composed of two 8-cores Sandy Bridge and 64 GB of RAM.

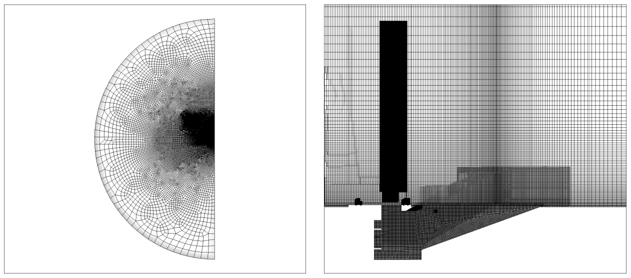
The test case from Airbus Defence & Space is the computation of a take-off blast wave. For an Ariane 5 rocket, boosters provide 90% of the thrust at lift-off and the objective is to compute the resulting blast-wave implied by the ignition of the boosters. The mesh is composed of 10M cells and is finer around area of interest (Figure 5). During take-off blast wave propagation, there are two overpressures: the first one is due to ignition of the boosters, the second is due to ducts. Those events are visible at Figure 6: the comeback of the wave can be seen from the 4th picture.

We consider a case with $\theta = 4$. The cell distribution in temporal levels and the theoretical computation load associated to each level is shown in Table I.

We focus on one iteration of the aerodynamic solver and we study the impact of several parameters: the priority strategy for the scheduling of tasks, the number of CEs, the number of parallel workers. We start by a study in shared memory and then we will evaluate the distributed version of our new solution.

A. *Shared memory study*

1) Impact of priorities and of number of CEs: For this study, we don't use parallel workers, so each CPU is associated to only one worker.



(a) Top view of the mesh (b) Side view of the mesh

Fig. 5: Mesh for the take-off blast wave computation.

We work with the StarPU built-in "prio" scheduler. In this scheduler, tasks are pushed in different priority queues and they are popped by priority order.

We consider two strategies. In the first one, we don't give any priority to any CE. For the second one, we give to tasks a priority according to the CE they belong. We want to give an advantage to the CEs with low temporal levels. However, when giving a priority to a task in StarPU, it doesn't automatically propagate the priority to the predecessors of the task in the DAG. So, it is also necessary to give a good priority to the predecessors of the tasks.

To achieve this goal, we give the highest priority to CEs which contain cells of temporal levels 0 or 1. Then for each CE, we compute a priority : we evaluate the lower distance from this CE to the other CEs previously prioritized. Then we map priorities according to the distance computed: the lower the distance, the higher the priority.

The result for our test case are shown at Figure 7. The reference time (20.02s) is the one for a computation with only 1 CE and one unique parallel worker which uses the 16 cores of the node; this is the configuration that mimics the best the previous OpenMP version in shared memory. The first observation is that this reference configuration with 1 CE is the worst in terms of performance. For a number of CEs from 16

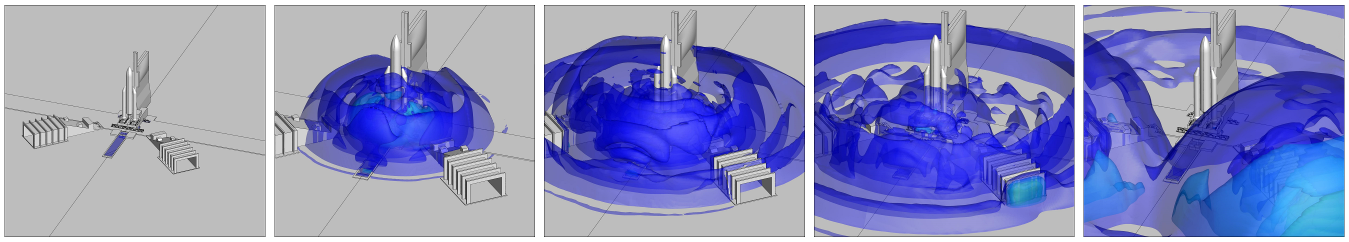


Fig. 6: Take-off blast wave computation.

to 256, we compare the elapsed time with our priority strategy (on the right) and without priority (on the left).

We show the state of the different workers during one iteration. The global size of the bar indicates the time (in seconds) needed to complete one iteration and the fill colors correspond to the proportions spent in each state (executing, sleeping, overhead). The overhead state contains, among other things, time spent to compute dependencies and insert tasks. The sleeping state means that the worker is ready but there is no ready task available.

The TS/GER part corresponds to lines 1 to 3 of Algorithm 3, including the "Wait all tasks" step. The Solver part corresponds to lines 4 to 15. We observe that the time spent in computation (blue and green bar) does not evolve when we increase the number of CEs, or if we use or not the priority strategy. Concerning computations from 16 to 128 CEs, sleeping state (red and dark pink bar) is reduced by decomposing in more CEs and by using the priority strategy. This strategy is beneficial as soon as we use 32 CEs. Concerning the overhead (yellow and gold bar), it evolves linearly with the number of CEs and the priority strategy has no influence. We detail below what happens for 128 and 256 CEs.

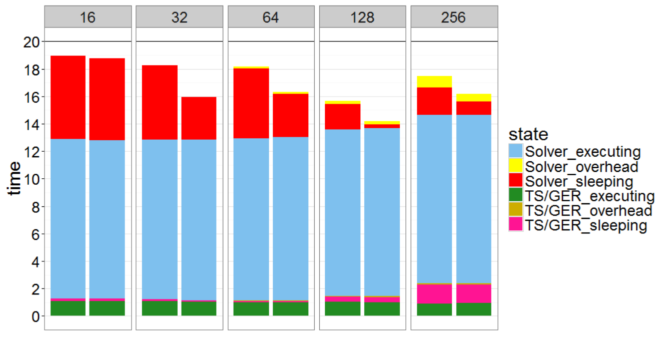


Fig. 7: Impact of Priority Strategy with varying number of CEs.

First, let us consider the computation with 128 CEs to highlight what happens when we use or not the priority strategy. Figure 8 is a Gantt Diagram where each horizontal bar represents a worker. Idle time is always in red, the overhead is in yellow.

The first step of the computation is in green (TS/GER, lines 1 to 3 of Algorithm 3). For the solver part of the computation (lines 4 to 13), tasks are colored according to the priority they have in the computation with priorities. There are four priority levels: from dark blue (high priority task) to light blue (low priority task). The last operation (line 15 of Algorithm 3) is in light green.

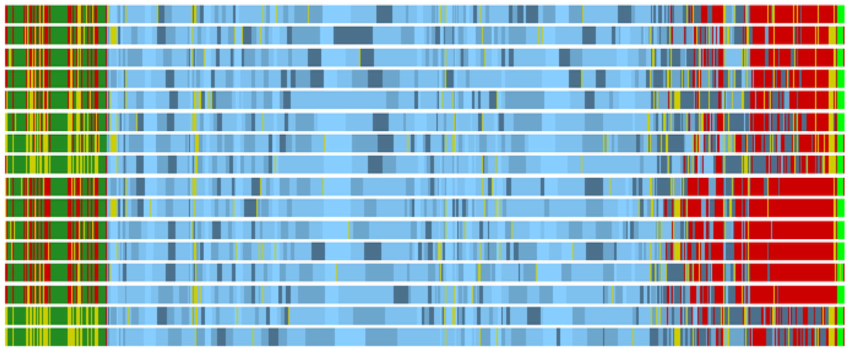
Of course, the same DAG is built and only the order of computations changes when using the priority strategy. In the trace without priorities, we observe that at the end of the computation, there is a large starvation zone: workers spend time in sleeping state (in red) because there is no task available. When using priorities, this starvation is much less important: we perform the DAG traversal in a way that allows more tasks to be available at the end of the computation. It is also noticeable that we managed to finish the scheduling by low priority tasks.

The evolution of ready tasks in the solver is shown at Figure 9. For the two strategies, we represent the number of ready tasks over time. Ready tasks are the ones already available for the workers: their dependencies have been fulfilled and they can be executed. The spike in the end comes from line 15 of Algorithm 3: for each cell component of a CE, a last task is inserted, just after a "Wait for all" (line 13). Some operations that are not done using task for legacy reasons are performed line 14. Traversing the graph without priority unlocks more tasks at start, but in the end, not enough tasks are available to feed the workers. Our strategy manages to keep more ready tasks available until the end.

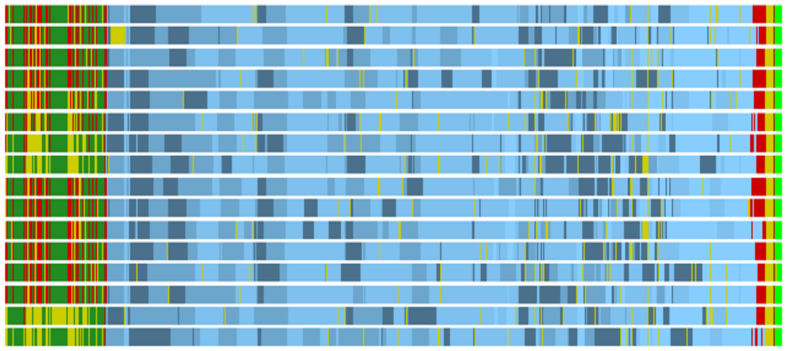
The idle step we observe in the TS/GER part of the computation is due to the fact that with our task granularity, we execute tasks quicker than we insert them. Finally, we can notice that the gain achieved by using the priority strategy is almost 10% (14,18s versus 15,56s).

When using now 256 CEs, we notice that the sleeping time of both the solver and the TS/GER parts increases. This can be explained by the time needed to insert tasks. For example, it takes 13.6 s to insert all the tasks of the solver step (from lines 5 to line 12) and the total computing CPU time is 195.8 s (about 12.2 seconds per worker). So, increasing the number of CEs allows more concurrency and the idle time is reduced. But with 256 CEs, the task insertion becomes so costly that it affects the global computation performance.

In conclusion of this shared memory study, we can say that the way we prioritize the tasks is efficient: we manage to feed the workers as long as possible whereas starvation was important without priority. The task-based description truly allows to take advantage of the irregularity of the computation.



(a) Trace without priorities (t=15.565s)



(b) Trace with priorities (t=14.184s)

Fig. 8: 128 CE computation without or with priorities.

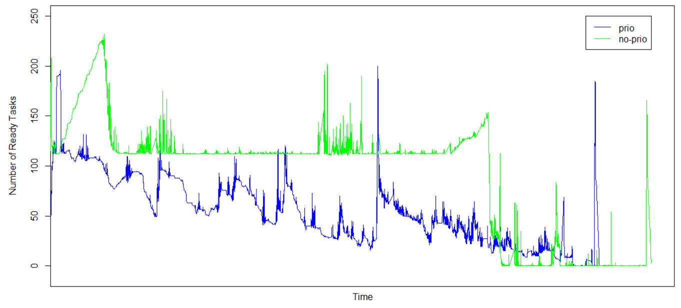


Fig. 9: Evolution of ready tasks (Solver step, 128 CEs).

2) Study with parallel workers: In order to exploit all the CPUs without creating too much tasks, another way is the use of parallel workers. Instead of being executed on only one CPU, a task is then executed on several ones. We rely on OpenMP DO loops for the parallelization of our computational kernels, which comes from the previous OpenMP version of the code.

We consider different numbers and sizes of workers (the size is the number of cores used), from 16 workers of size 1, to one worker that uses all the cores: so we have the relation "(number_of_worker)×(size_of_workers)=16". CPUs that belong to the same worker are on the same socket, except when the 2 sockets are used by one worker. We test configurations from 16 CEs to 128. The previous strategy for scheduling tasks is always used in this study.

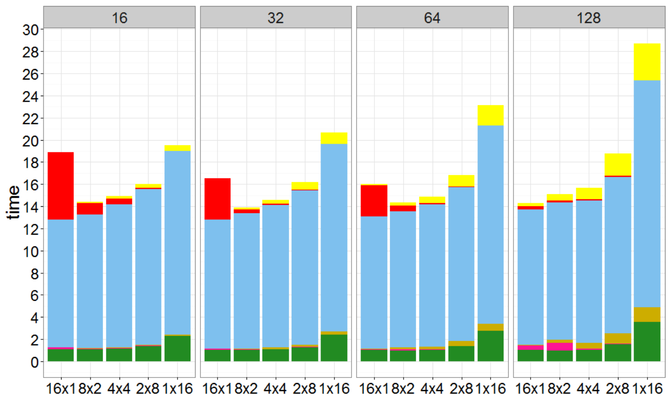
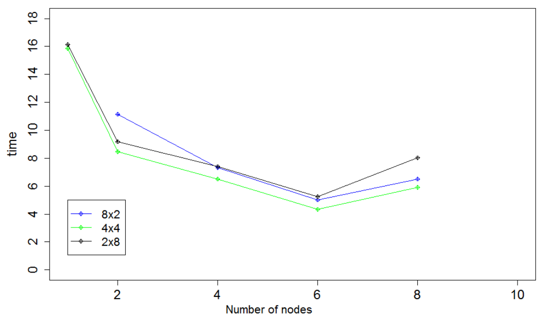
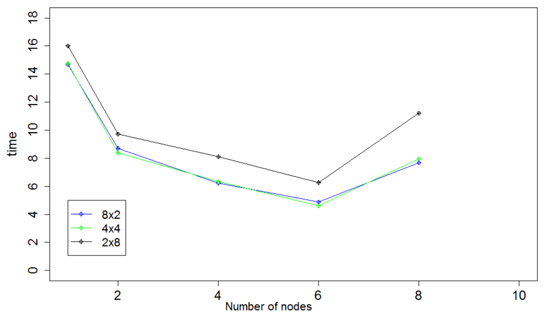


Fig. 10: Time performance with different configurations with parallel workers.

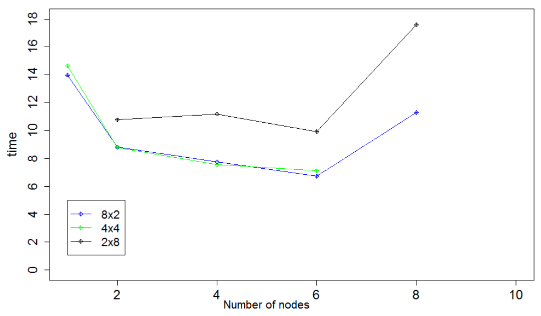
At Figure 10, we first notice that the total time spent in computation tends to increase slightly while we use fewer and larger workers. Indeed, the scalability of our computational kernels is not good enough. However, this is not critical until a worker size equal to 8. When we use the whole node and its two sockets to form only one worker, the situation is more critical.



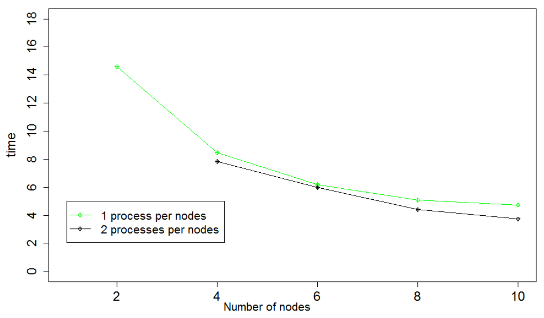
(a) 8 CEs per process



(b) 16 CEs per process



(c) 32 CEs per process



(d) MPI/OpenMP version

Fig. 11: Time spent in the solver with different configurations (X-axis gives the number of nodes).

	TS/GER idle	TS/GER executing	Solver idle	Solver executing	Submission time	Elapsed time
Process #1	0.338	0.197	1.440	2.356	0.608	4.331
Process #2	0.357	0.174	1.559	2.241	1.301	4.331
Process #3	0.351	0.184	1.401	2.395	0.937	4.331
Process #4	0.293	0.243	1.790	2.005	0.218	4.331
Process #5	0.310	0.216	1.599	2.205	0.599	4.331
Process #6	0.344	0.191	1.627	2.169	1.259	4.331

TABLE II: Average time (in seconds) for each process. We use 6 processes, 8 CEs per process, 4 workers and 4 cores per worker.

The overhead evolves linearly with the size of workers, while the idle time is reduced when using parallel workers.

The best overall configuration is obtained for 32 CEs and 8 workers of size 2.

B. A distributed memory version

We still use the same test case as in the shared memory study. We consider here experiments from 2 to 8 MPI processes, with 8 to 32 CEs per process, and we configure our workers in 3 configurations : 8×2, 4×4 and 2×8. Each process is in charge of a portion of the mesh and the domain decomposition takes into account the cost of cells. Figure 11 shows the different elapsed times for the different configurations tested.

We can see that the best performance (4,331s) is obtained with 8 CEs per process and 6 processes configured as 4×4 (4 workers and 4 cores per worker). This case is detailed in Table II. The load balancing is good with a difference of 13% of executing time between Process 1 and Process 4. However, the idle time is important: with only 2 CEs per worker, it is not easy to exploit enough asynchronism and communication-computation overlapping.

Figure 11d shows the elapsed time for the previous MPI/OpenMP version of FLUSEPA. We consider 2 configurations, one with one process per node, and one with one process per socket. Unfortunately, the problem does not fit in memory for a larger number of processes per node, mainly because the memory consumed by process depends of the load balancing. We see that the absolute best performance is obtained for this previous version with 2 processes per node and 10 nodes. But when we consider a number of nodes for which the task-based version is competitive (at most 6), the task-based version gives a better result than the previous one (4.33s versus 5.98s) for 2 processes per node and only 6 nodes.

With 8 MPI processes, we don't reduce the elapsed time in the solver. This is due to the number of CEs which varies from 64 for 8 CEs per process to 256 for 32 CEs per process. When we work on more nodes, the computation time by node decreases, but the overhead stays almost the same. This is the main reason why we cannot use an important number of CEs. Load balancing is also a concern. The size of the problem is a little bit tiny to overcome the overhead induced by the parallelization method.

VI. Conclusion and perspectives

In this paper, we have described a preliminary study towards a task-based distributed version of an aerodynamic solver with time adaptive time step. We validated our implementation with a real-life industrial case.

When under the right conditions (i.e. when the overhead induced is not too high), the implementation shows very promising results. The parallelism offered by the task-based paradigm allows a better exploitation of the computational resources. However, for our current fixed-sized problem, the overhead becomes too high to obtain a gain when we use too much nodes. As we expect to grow the size of our industrial test-case, this is not dramatic. So this result is very promising and the gain of our distributed task-based solution should be more important with a larger size test case.

It is clear that exploiting parallel tasks is useful reducing the number of workers, but this point can be limited by the scalability of the computational kernels.

Right now, the decomposition into CEs is fixed at the start of the computation, but the temporal level of cells evolves during the computation between iterations. Having the possibility to reshape the CEs could lead to an improvement.

Concerning the way we exploit parallel tasks, we tested only the simple case of multiple workers of the same size. It is possible to explore more heterogeneous configurations (e.g. 1×8+2*2×4 : one worker of size 8, and two of size 4). It could even be possible to modify the configuration on the fly regarding current status of computations. To overcome the starvation seen at the end of the iteration, one possible strategy is to gather all CPUs into one worker when there is few ready tasks. Context resizing is possible in StarPU and have been described in [22].

Concerning the distributed version of the code, we have to test the implementation with a bigger test case. If we manage to remove the hard synchronization we have at the end of iterations, we expect to pipeline iterations while applying an asynchronous load balancing scheme.

We intend also to develop a task-based parallelization of the intersection mechanism of FLUSEPA in order to exploit more asynchronism in the whole application when considering booster separation simulations.

References

[1] M. P. Forum, "MPI: A Message-Passing Interface Standard," tech. rep., Knoxville, TN, USA, 1994.

[2] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[3] 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, Special Issue: Euro-Par 2009*, vol. 23, pp. 187–198, Feb. 2011.

[4] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hrault, and J. Dongarra, "PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability," *Computing in Science and Engineering*, vol. 99, p. 1, 2013.

[5] P. Brenner, J.-M. Carrat, and M. Pollet, "Simulation d'interactions aérodynamiques instationnaires autour de plusieurs corps en mouvement relatif," in *AAAF : Les interactions en aérodynamique*, 1998.

[6] C. W. Hirt, A. A. Amsden, and J. L. Cook, "An arbitrary lagrangian-eulerian computing method for all flow speeds," *Journal of Computational Physics*, vol. 14, no. 3, pp. 227–253, 1974.

[7] W. L. Kleb, J. T. Batina, and M. H. Williams, "Temporal adaptive Euler/Navier-Stokes algorithm involving unstructured dynamic meshes," *AIAA journal*, vol. 30, no. 8, pp. 1980–1985, 1992.

[8] R. Lhner, K. Morgan, J. Peraire, and O. A. Zienkiewicz, *Finite element methods for high speed flows*. University College of Swansea Institute for Numerical Methods in Engineering, 1985.

[9] M. M. Pervaiz and J. R. Baron, "Temporal and spatial adaptive algorithm for reacting flows," *Communications in Applied Numerical Methods*, vol. 4, no. 1, pp. 97–111, 1988.

[10] H. K. Versteeg and W. Malalasekera, *An Introduction to Computational Fluid Dynamics: The Finite Volume Method*. Pearson Education, 2007.

[11] A. Duran, J. M. Perez, R. M. Ayguadé, E. amd Badia, and J. Labarta, "Extending the OpenMP tasking model to allow dependent tasks," in *OpenMP in a New Era of Parallelism, 4th International Workshop, IWOMP 2008*, (West Lafayette, IN), Lecture Notes in Computer Science 5004:111-122, May 12-14 2008.

[12] Z. Budimli, M. Burke, V. Cav, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Tairlar, "Concurrent collections," *Sci. Program.*, vol. 18, no. 3, pp. 203–217, 2010.

[13] M. E. Bauer, *Legion: Programming Distributed Heterogeneous Architectures with Logical Regions*. PhD thesis, Stanford University, 2014.

[14] E. Chan, E. S. Quintana-Orti, G. Gregorio Quintana-Orti, and R. van de Geijn, "Supermatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures," in *Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures SPAA'07*, pp. 116–125, 2007.

[15] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra, "Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA," in *12th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC'11)*, 2011.

[16] X. Lacoste, M. Faverge, P. Ramet, S. Thibault, and G. Bosilca, "Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes," in *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing Workshops and Phd Forum (IPDPSW'14), HCW 2014*, 2014.

[17] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi, "Task-based FMM for multicore architectures," *SIAM Journal on Scientific Computing*, vol. 36, no. 1, pp. C66–C93, 2014.

[18] M. Cosnard and M. Loi, "Automatic task graph generation techniques," in *System Sciences, 1995. Proceedings of the Twenty-Eighth Hawaii International Conference on*, vol. 2, pp. 113–122, IEEE, 1995.

[19] R. Allen and K. Kennedy, *Optimizing compilers for modern architectures: a dependence-based approach*, vol. 289. Morgan Kaufmann San Francisco, 2002.

[20] T. Cojean, A. Guermouche, A. Hugo, R. Namyst, and P.-A. Wacrenier, "Exploiting two-level parallelism by aggregating computing resources in task-based applications over accelerator-based machines." https://hal.inria.fr/hal-01181135, 2015.

[21] F. Pellegrini and J. Roman, "Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs," in *High-Performance Computing and Networking*, pp. 493–498, Springer, 1996.

[22] A. Hugo, A. Guermouche, R. Namyst, and P.-A. Wacrenier, "Composing multiple StarPU applications over heterogeneous machines: a supervised approach," in *Third International Workshop on Accelerators and Hybrid Exascale Systems*, (Boston, USA), May 2013.