# Multi-Criteria Mesh Partitioning for an Explicit Temporal Adaptive Task-Distributed Finite-Volume Solver

Alice Lasserre, Jean Marie Couteyen Carpaye, Abdou Guermouche, Raymond Namyst

# Multi-Criteria Mesh Partitioning for an Explicit Temporal Adaptive Task-Distributed Finite-Volume Solver

Alice Lasserre
*Inria*
Bordeaux, France
alice.lasserre@inria.fr

Jean-Marie Couteyen-Carpaye
*Airbus CRT*
Toulouse, France
jean-marie.couteyen-carpaye@airbus.com

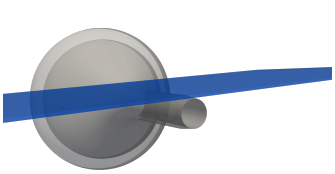
Abdou Guermouche, Raymond Namyst
*univ. Bordeaux*
Bordeaux, France
name.lastname@u-bordeaux.fr

***Abstract*—The aerospace industry is one of the largest users of numerical simulation, which is an essential tool in the field of aerodynamic engineering, where many fluid dynamics simulations are involved. In order to obtain the most accurate solutions, some of these simulations use unstructured finite volume solvers that cope with irregular meshes by using explicit time-adaptive integration methods. Modern parallel implementations of these solvers rely on task-based runtime systems to perform fine-grained load balancing and to avoid unnecessary synchronizations. Although such implementations greatly improve performance compared to a classical fork-join MPI+OpenMP variants, it remains a challenge to keep all cores busy throughout the simulation loop. In this article, we first investigate the origins of this lack of parallelism. We emphasize that the irregular structure of the task graph plays a major role in the inefficiency of the computation distribution. Our main contribution is to improve the shape of the task graph by using a new mesh partitioning strategy. The originality of our approach is to take the temporal level of mesh cells into account during the mesh partitioning phase. We evaluate our approach by integrating our solution in an ArianeGroup production code used by Airbus. We show that our partitioning method leads to a more balanced task graph. The resulting task scheduling is up to two times faster for meshes ranging from 200,000 to 12,000,000 components.**

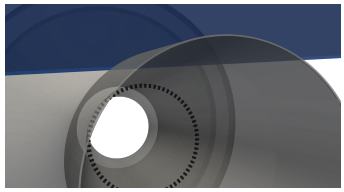
***Index Terms*—scheduling, DAG, task-based applications, heterogeneous systems**

## I. Introduction

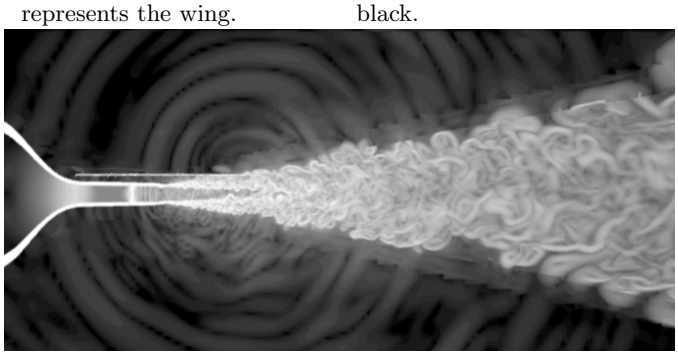
Complex fluid flow phenomena are difficult to analyze using analytical methods or physical experiments alone. Fortunately, numerical simulations provide valuable insights into them. Existing industrial aerodynamics applications include the modelling of launcher stage separation, blast wave propagation during rocket take-off, as well as aircraft propeller noise. To achieve reasonable execution times, these large-scale phenomena require applications to run on distributed clusters of multicore nodes. These geometries are digitally represented by means of meshes in which physical values, such as pressure or temperature, are calculated for each *cell* and the flows between them are evaluated for each *face*. For the purpose of simulation progress, time is advanced in discrete units (i.e. time steps) that determine the updating of these faces and cells. In the context of an explicit solver, the maximum time step allowed for a cell depends mainly on its volume, which can vary considerably from cell to cell in non-homogeneous meshes. When dealing with this kind of mesh, it is advantageous to use a temporal adaptive integration. Parallel implementations of such simulation algorithms have to cope with a significant load imbalance between the different mesh areas. While a task-based approach helps provide more concurrency and a more flexible distribution of the workload between cores, occupying all the processing units used during execution remains difficult.



(a) Installed PPRIME nozzle. The flat plate in blue represents the wing.



(b) Nozzle section with internal cylinder rugosities in black.



(c) Norm of the density gradient into the jet.

Fig. 1: Installed jet noise configuration (*PPRIME* nozzle experiment [15])

This work takes place in the context of the FLUSEPA solver, developed by ArianeGroup, which is one such aerodynamic application, integrating the techniques described above. At Airbus, this code serves the purpose of evaluating jet noise in installed nozzle configurations [15]. Figure 1 illustrates a FLUSEPA simulation which reproduces the

PPRIME nozzle wind tunnel experiment. A flat plane represents the wing, which is clearly visible in Figure 1a. Rugosities are added in order to obtain a turbulence-resolved boundary layer attached to the inside of the nozzle. This article investigates performance optimization through this industrial numerical simulation case, which is a persistent concern in the aerospace field, especially in the context of commercial aircraft.

The main concern of this article is the existence of specific CPU idleness patterns during the solver runtime. First, we provide a thorough analysis of the performance behavior of the application in order to identify the dominant factors leading to CPU inactivity periods observed during executions. We conclude that the intrinsic structure of the task graph limits performance. Therefore, the main innovation of this article lies in the objective of finding the optimal execution through the task graph generation process, contrasting with the conventional practices of scheduling an already-defined graph. Consequently, the second contribution is a 20% improvement in the total execution time directly within the FLUSEPA code integrated into the Airbus production environment. This improvement is achieved by developing a new partitioning strategy, which aims to generate a well-balanced and highly efficient parallel task graph, thereby ensuring balance throughout the entire execution.

Section II provides an explanation of how FLUSEPA operate, with a specific focus on the integrated aerodynamic solver. We then introduce the first contribution, which is to identify the root causes of runtime inactivity. The following section proposes a relevant partitioning heuristic, followed by a description of its implementation. Section VI presents initial findings that indicate a 50% reduction in total execution time on a virtual platform. Finally, the last section provides a direct validation of the strategy within the FLUSEPA production code, leading to a 20% enhancement in the total execution time for the Airbus *PPRIME* nozzle simulation.

## II. The FLUSEPA Fluid Dynamics Solver

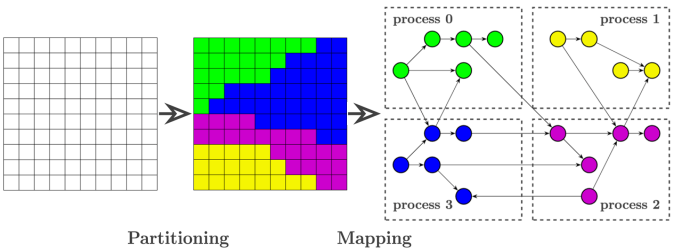



Fig. 2: Simulation process of the FLUSEPA solver. Partitioning of the input mesh leads to the creation of several domains, which are then mapped to the MPI processes. The task graph is constructed using the partitioning performed, with each task running on the process assigned to its extraction domain.

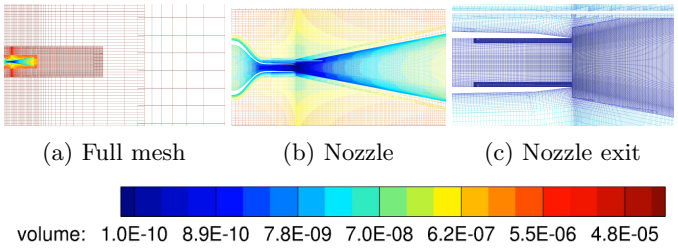
The purpose of this section is to provide an overview of how the FLUSEPA aerodynamic solver works. Industrial aerodynamics simulation codes that model complex fluid phenomena operate on very large meshes and must obviously be parallelized to deliver results in an acceptable time. This article focuses on the FLUSEPA parallel Navier-Stokes solver [8], [9]. As most fluid dynamics applications, FLUSEPA relies on the *finite volume method* as a discretization technique to create a mesh, which is subsequently passed as input to the explicit solver. Afterward, a domain decomposition is performed to explicitely distribute the mesh and assign the associated calculations to the hardware resources. The main steps of FLUSEPA solver operations are depicted in Figure 2. The current implementation of FLUSEPA is based on a task-based execution model [7]. The task scheduling is delegated to the StarPU runtime support [2] and the internode communication to the MPI Library [13]. The granularity of parallelism is affected by the size (or number) of the obtained domains, which also affects the shape of the task graph. Section II-B discusses this. The FLUSEPA solver implements an adaptive time-stepping scheme to advance the simulation over time. Its purpose is to determine the insertion of calculations at runtime to update the values and flows stored in the mesh. The next section provides a more detailed description.

### A. The adaptive time-stepping scheme

The presence of multiple physical scales (boundary-layer turbulence, jet turbulence, acoustics) requires a non-uniform grid resolution. Figure 3 provides a visual representation of the `PPRIME_NOZZLE` case at different zoom levels, with colors representing volume variations. The cells become finer and more densely packed near the nozzle exit and where the turbulence phenomenon occurs. The maximum time step permitted for fine cells is considerably smaller than for far-field cells. This secures a minimum level of precision when resolving these calculations, which are primarily located in the phenomenon's key areas of interest. To avoid penalizing the simulation by calculating all cells with the maximum frequency of the finer cells, an adaptive time-stepping scheme is implemented in the solver.

Cells are assigned a *temporal level* $\tau$, reflecting their maximum allowed time step. Note that the scale of the time step is exponential: the cell calculation time is doubled for each level increase. Each iteration is divided into a set of subiterations that involve only certain temporal levels. At the end of a complete iteration, all cells reach the same final physical time, regardless of their temporal level.

Figure 4 illustrates how this temporal integration scheme works on a mesh with three different temporal levels: 0, 1 and 2. This scheme determines *when and which* cells and their associated faces should be updated during the simulation, and serves as the mechanism for




(a) Full mesh (b) Nozzle (c) Nozzle exit

Fig. 3: Different zoom levels of a slice of the `PPRIME_NOZZLE` mesh. Cells are color-coded according to their volume.

injecting computation into the execution. Given that 2 is the maximum level inside the mesh, the second-order Heun method deduces a division of each iteration into $2^2$ subiterations. Cells with $\tau = 0$ generate calculations at each subiteration (time step $\Delta_t$), as they hold a critical position within the phenomenon and demand the highest precision. Calculations of $\tau = 1$ cells are spaced every odd subiteration. Finally, the $\tau = 2$ cells are calculated only once, in the first subiteration of the iteration. Obviously, cells generate a certain amount of computation depending on their $\tau$ value, leading to cost heterogeneity among them. Each cell can be described in terms of an operating cost $2^{max-\tau}$, where *max* is the highest temporal level within the mesh. Figure 4 clearly confirms that each subiteration injects a different workload, determined by both the maximum allowable $\tau$ of the subiteration and the number of cells involved. Thus, the temporal integration scheme intrinsically introduces load imbalances between subiterations.

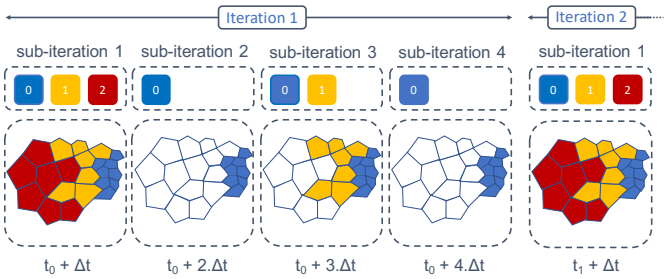



Fig. 4: Decomposition of an iteration into subiterations using an explicit temporal integration method. Since the highest temporal level within the mesh is 2, the iteration is divided into $2^2$ subiterations, each dedicated to handling specific temporal levels. *Active* cells and faces are color-coded to denote their temporal level.

To balance the computational load between the domains assigned to the processes during an iteration, FLUSEPA carries out a partitioning of the mesh on the basis of a weighting of the cells according to their operating costs ($2^{max-\tau}$). The resulting partitions ensure that the overall workload is balanced between them. An interesting observation is that using such cost-based strategies leads to partitions that vary widely in size. Some partitions contain few cells with low temporal level (extremely expensive to compute), compensated by partitions with many cells with high temporal level (inexpensive to compute). When iterated, all these partitions generate an equivalent computational load.

### *B. Overview of the task-based implementation*

Figure 2 shows the general scheme of the application. Starting from the initial mesh, the partitioner is invoked to create domains, which are then distributed to the different processes involved. Subsequently, a task graph is induced from this partitioning, expressing the computations required for the mesh and their associated dependencies.

The task generation process is formalized in Algorithm 1. We focus on a single subiteration, although the same process applies to any subiteration. A subiteration contains as many phases as its $\tau_{max}$ permits. A $\tau_{max} = 1$ involves two phases, one for $\tau = 0$ and one for $\tau = 1$. A phase is therefore dedicated to a specific $\tau$ and generates the tasks for the processing of the cells and faces of its level.

As can be seen in Algorithm 1, temporal levels are traversed in descending order within each subiteration. The first subiteration, therefore, presents a phase sequence in which the first one processes the maximum temporal level 2 (red, as depicted in Figure 4), then the next one the level 1 (orange) and finally the last one the level 0 (blue). Each of these phases iterates, once for face processing and once for cell processing, over the set of domains defined by the previous partitioning. At the domain level, cells are distinguished according to whether or not they are bordered by cells belonging to another domain. If this happens, the cells are considered to be external. The same principle applies to faces. This process in depicted in Algorithm 1 where line 3 iterates on the mesh cells and then the faces. Lines 7, 8 generate the corresponding external and internal tasks for each of the domains. Finally, remember that dependencies are created between a pair of tasks if the calculations involved require either the values of neighboring objects (cells or faces) or the previous values of the elements they process (i.e. those calculated in the previous subiterations or iterations).

The paradigm shift within FLUSEPA towards a task-based approach has proven to be extremely beneficial, delivering substantial performance enhancements when compared to the previous OpenMP loop-based parallelization. Still, a detailed analysis of execution traces, such as the Gantt chart depicted at the upper part of Figure 5, exposed multiple notable instances of core idleness within a simulation iteration. Consequently, the upcoming section provides the initial contribution of this article by investigating the plausible explanations behind the recurring patterns of inactivity.

**Algorithm 1:** Task generation algorithm.

```
   // subiteration loop
1  for s in 0 ... max_subiter do
       // temporal levels loop in descending order
2      for τ = 0..max do
           // object type loop
3          for t in faces,cells do
               // domain loop
4              foreach domain d do
5                  s = {x|type(x) = t and x ∈ d and t_lvl(x) ≤ τ}
                   // Generate a task if there are active objects
6                  if s ≠ ∅ then
7                      Generate a task for external objects of type t
8                      Generate a task for internal objects of type t
```

## III. Investigating core idleness issues

This section identifies the root causes of the unique inactivity patterns found during execution.

### A. FLUSIM: A dedicated testing submodule of FLUSEPA

When investigating phenomena such as jet noise in an installed configuration, the temporal levels of the cells experience minimal evolution across iterations. Hence, optimizing the entire computation is equivalent to optimizing an individual iteration. Due to the complexity and scale of the modeled phenomena, executing the FLUSEPA code to evaluate various configurations lacks efficiency. Booking compute nodes and performing the initialization process is costly, especially for a single iteration. In order to easily evaluate different configurations, we developed a small simulator, referred as FLUSIM, which is able to emulate an iteration of FLUSEPA. With FLUSIM, emulating an iteration for even the largest meshes can be accomplished in a matter of seconds on virtually any cluster, using only a single CPU core. As inputs, FLUSIM takes a cluster configuration, the mesh with the temporal level of each cell, a domain decomposition, and a scheduling strategy. When defining the cluster configuration, we specify the number of nodes and the number of workers per node that we intend to emulate. FLUSIM, just like FLUSEPA, relies on domain decomposition, mesh, and temporal levels to generate the task graph, which is subsequently executed according to the scheduling strategy. FLUSIM does not aim to simulate the full complexity of executing on a given platform, as SimGrid does [6]. No communication or runtime overheads are considered. Our objective is to evaluate the scheduling of diverse DAGs within an idealized environment. As a result, we are able to identify and address the root causes of idleness that are not due to overheads.

Figure 5 displays a side-by-side comparison of two identically parameterized executions, both using the same input mesh `PPRIME_NOZZLE`. For this instance and all subsequent experiments presented in this article, we focus exclusively on a single iteration. It is considered to be a representative illustration of the problem at the simulation's full scale (i.e., this pattern is reproduced at each iteration). The upper temporal diagram corresponds to an actual FLUSEPA execution solver, while the lower one is acquired from the FLUSIM submodule. Tasks are color-coded to indicate the subiteration to which they correspond. We can observe that FLUSIM accurately reproduces the actual execution of FLUSEPA with a 20% variance in the iteration's execution time. Above all, the overall scheduling of the tasks exhibits the same execution patterns, illustrating FLUSIM's accuracy.

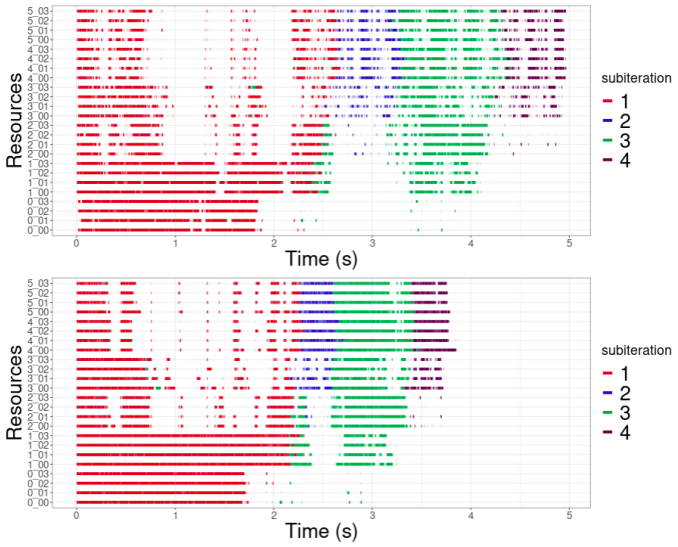



Fig. 5: The relevance of the FLUSIM simulator is illustrated by comparing the outputs of FLUSEPA (top trace) and FLUSIM (bottom trace) executed on 6 MPI processes of 4 cores each. The runs are parameterized identically. `PPRIME_NOZZLE` is partitioned into 12 domains using the operating cost strategy. Tasks are color-coded according to their subiteration. A difference of 20% in execution time is observed between the two.

We present in the next sections both the set of experimental problems we use in the article and an analysis which illustrates that the performance issues met with FLUSEPA are intrinsically related to the shape/topology of the underlying task-graph. Neither the scheduling strategies used, nor the overheads, are responsible for the idle periods which can be seen during FLUSEPA's executions.

### B. Representative input problems

All the meshes featured in this article have been supplied directly by Airbus and can be visualized in Figure I. Despite their seemingly simple geometry, these meshes are typical of the fluid dynamics phenomena modelled by the aerodynamics industry (see Figure 1). `CYLINDER` is the cylindrical one appearing in the left-hand column I. Containing just over 6 million meshes, it models the flow of fluids around a piece of machinery that sits at the

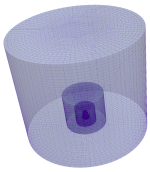
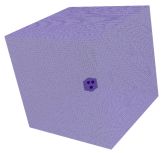
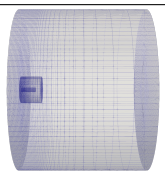
| | CYLINDER | | | | CUBE | | | | PPRIME_NOZZLE | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| |  Total cell count=6400505 | | | |  Total cell count=151817 | | | |  Total cell count=12594374 | | |
| | $\tau=0$ | $\tau=1$ | $\tau=2$ | $\tau=3$ | $\tau=0$ | $\tau=1$ | $\tau=2$ | $\tau=3$ | $\tau=0$ | $\tau=1$ | $\tau=2$ |
| #Cells | 52697 | 273525 | 2088538 | 3985745 | 2953 | 23489 | 514 | 124861 | 1500741 | 4052551 | 7041082 |
| %Cells | 0.8% | 4.3% | 32.6% | 62.3% | 2.0% | 15.5% | 0.3% | 82.2% | 11.9% | 32.2% | 55.9% |
| %Computation | 4.4% | 11.3% | 43.2% | 41.2% | 9.7% | 38.6% | 0.4% | 51.3% | 28.4% | 38.3% | 33.3% |

TABLE I: Test meshes.

center of a series of other cylindrical shapes (see Figure 3). The nerve center of the phenomenon revolves around this specific piece, as it contains all the cells of temporal level 0. As one gets closer to the boundaries of the mesh, the temporal level gradually decreases. Figures 1a,1b and 3 all refer to the PPRIME_NOZZLE test case. It features the same structure as CYLINDER except that it is spread over 12 million meshes and incorporates three temporal levels, instead of the four present in the other two meshes. In the middle column I, CUBE is viewed as a worst case scenario. With its three non-contiguous $\tau = 0$-cell hotspots contained in just over 150,000 cells, the composition of this mesh can be described as complex. From the perspective of parallel code running on a distributed architecture, this intricate geometry proves to be difficult to divide during the first phase of FLUSEPA's operation.

### C. Investigating task scheduling

Given the scale and complexity of the phenomena, FLUSEPA generates extremely large task graphs and relies on the external StarPU library to perform scheduling. Bottlenecks tend to occur in the computation of complex parts of the mesh, creating tasks with many input and output edges. A first step for identifying the origin of ressource-idleness during the execution is to see whether the scheduling policy used is causing the issue. To estimate the performance gain we could achieve with a more clever/advanced task scheduling policy, we use the FLUSIM submodule to examine the scenario where the number of cores per node is unlimited. In practice, this means defining a configuration where the number of cores per node is greater than the maximum number of ready tasks available at any given time. An eager scheduling strategy is optimal for such a configuration, as ready tasks are always scheduled immediately. This makes the total completion time as short as possible. Figure 6 provides executions performed on a 64-node cluster. Each MPI process is assigned to a single domain, the tasks of which are carried out on a virtually unbounded number of cores. To enhance clarity, the displayed trace aggregates all resources assigned to a specific MPI process into a single composite resource. As a result, a unit is deemed inactive solely when all the cores associated with the process are inactive as well. It is clear from this experiment that MPI processes, even in our ideal configuration, still exhibit periods of inactivity. Most importantly, the identifiable pattern from prior executions is clearly apparent. Accordingly, the task scheduling policy is not the primary cause of the observed load imbalance. This section leads us to the conclusion that while it is possible to save execution time on specific configurations by using a more suitable scheduling strategy, the underlying cause of idle time remains.

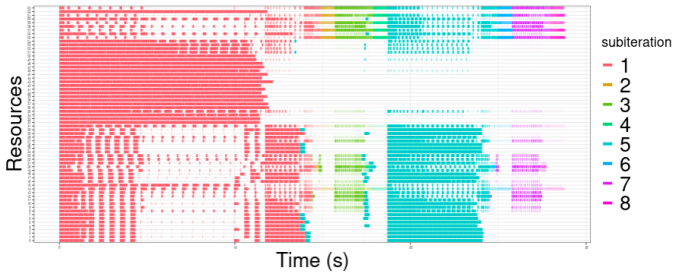


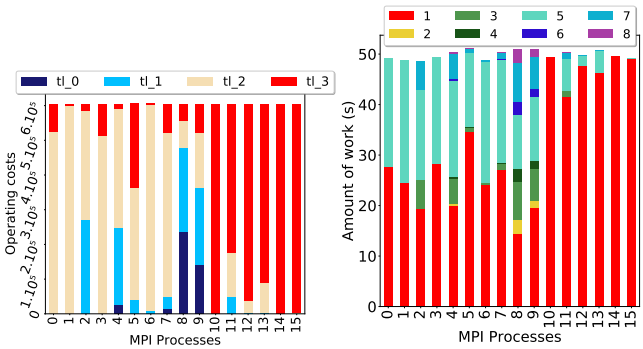

Fig. 6: Execution trace with FLUSIM, which simulates 64 MPI processes (1 domain per process). Each MPI process has a theoretically unrestricted number of cores. Tasks are color-coded according to their subiteration.

Optimizing the behavior of parallel applications based on task-based programming models often consists on the design of custom scheduling strategies and advanced data distribution strategies while considering the task graph as an input of the problem. In the case of FLUSEPA, neither the scheduling policy (see previous section) nor the data distribution (the work is perfectly balanced over the domains) are responsible for the bad behavior of the application. Thus, we consider the problem of finding the best task graph with respect to our resource idleness problem. Consequently, the inherent structure of the graph itself and its effect on execution are the focus of the following sections.

## IV. Temporal levels-aware mesh partitioning

In FLUSEPA (resp. FLUSIM), the task graph generation process is tightly linked to the domain decomposition step. We remind that the default partitioning strategy

aims at balancing the workload between domains (and thus between MPI processes) for a given iteration. However, these processes are overloaded with computations at certain time intervals and completely inactive at others (see Figure 6). According to execution traces, these intervals coincide with the boundaries of subiterations, each characterized by a different set of temporal levels to be calculated. One explanation may be that a domain assigned to an MPI process tends to mostly contain cells having the same temporal level. In fact, the decomposition tends to generate partitions containing either numerous lightweight cells or a few computationally intensive cells. Figure 7a illustrates this behavior, depicting the operating costs allocated to each process for a domain decomposition targeting 16 domains on the `CYLINDER` test case. Costs are color-coded in accordance with their respective temporal levels. Processes are effectively assigned an equivalent workload. However, we can observe that the partitioning strategy builds domains containing very different proportions of cells in each temporal level while keeping the global workload very-balanced. For instance, Processes 10 to 15 are characterized primarily by an abundance of $\tau = 3$ cells (in red). Others feature a preponderance of $\tau = 2$ cells (in light yellow), and so on. It should be noted that the most expensive level ($\tau = 0$) is mainly allocated to two processes in this configuration. As a result, even with perfect load balancing, processes can have *all* the computations of their assigned domains mostly concentrated into a single subiteration. Figure 7b depicts the cumulative computation time completed by each process in every subiteration. Processes 10 through 15 almost entirely perform their *entire cumulative workload during the first subiteration*. These processes are then inactive for the rest of the execution.




(a) Distribution of the total operating costs by temporal level among MPI processes

(b) Cumulative computation time by subiteration among MPI processes

Fig. 7: Domain characteristics for the execution of the `CYLINDER` case using 16 MPI processes (32 cores per process), using the operating-costs partitioning strategy.

The main reason behind these starvation periods stems from the fact that cells from a given temporal level are not uniformly distributed. Since consecutive subtiterations are strongly ordered by task dependencies, a process which has no task to execute during a given subtiteration is forced to wait the completion of the subiteration by its neighbours before entering the next subiteration. This issue is depicted at the scale of task graph generation in Figure 8a. Both domains exhibit a large variation in temporal-level sampling, even though they produce the same workload. In the depicted first subiteration and throughout the entire iteration, this variability is reflected in the irregularity and inconsistency of their active periods.

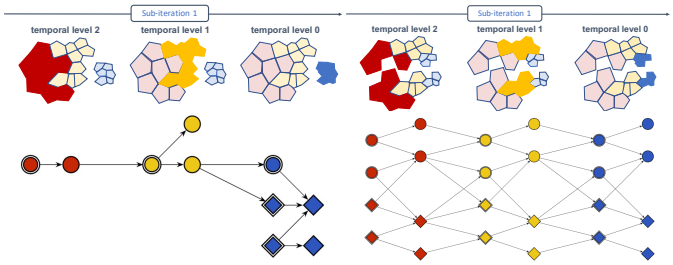
As mentioned above, FLUSEPA addresses cell heterogeneity through an explicit temporal integration method. Each iteration is segmented into multiple subiterations, each of which generates a predefined workload based solely on the number of its active cells. Since the domain decomposition is based only a global load balancing criterion for the whole iteration, there is no guarantee that the workload is uniformly distributed among the subiterations of the iteration. One approach to alleviate this phenomenon would be to drastically decrease the granularity of tasks (i.e., domains). However, this approach would require very low granularity tasks to achieve good load balancing, which would dramatically increase the overhead of managing such numerous tasks.

### *A. Considering of temporal levels in the partitioning scheme*

Improving our application's behavior requires that *each subiteration's workload* is evenly distributed across available resources. The objective of the partitioning should be to balance the workload for each subiteration. A naive approach would be to perform a separate partitioning for each subiteration. This would lead to a dramatic increase in communication volume due to the necessary data redistribution between two successive subiterations. A more reasonable solution to the problem is to consider a partitioning strategy that balances the workload for all subiterations at once.

From the insights provided in earlier sections, it becomes clear that achieving workload balance across all subiterations can be achieved by balancing the number of cells of each temporal level between processes. The idea, then, is to have a partitioning strategy that distributes the cells of each temporal-level class evenly across each domain. The active cells per subiteration are balanced between the domains and therefore between the processes, theoretically resulting in an efficient and well-balanced execution.

Figure 8 depicts how task graph generation is affected by `SC_OC` and `MC_TL` partitioning, focusing on the first subiteration. Let us consider the first phase (in red), which deals with the highest temporal level of the first subiteration. In the case of `SC_OC` in Figure 8a, only one domain has an effect on the insertion of work into the execution, since the other has no cells and faces corresponding to the phase's temporal level. A first red task is therefore




(a) Operating cost partitioning strategy

(b) Temporal level partitioning strategy

Fig. 8: Illustration of task graph generation for the first subiteration. As the initial subiteration includes all temporal levels in the mesh, the solver iterates in reverse order, starting with $\tau = 2$ (in red), $\tau = 1$ (in yellow), and ultimately $\tau = 0$ (in blue) to generate the respective tasks. Double contour tasks correspond to those that compute faces, while the others compute cells. Their shape varies according to their extraction domain.

inserted in the graph for the calculation of the internal faces, and then a second one for the internal cells of the active domain. No external components are processed because none are characterized by the $\tau$ of the phase within this domain. In the case of `MC_TL` in Figure 8b, both domains have an equivalent number of $\tau = 0$, $\tau = 1$ and $\tau = 2$ cells. Note that the phase generates a *fixed* amount of work regardless of the partitioning used. `MC_TL` therefore expresses the mesh calculations at a much more efficient level of granularity. A total of 8 tasks, 4 from each domain, are used to express these computations instead of the 2 created by `SC_OC`. This behavior is repeated for subsequent phases, ensuring that the computations are evenly distributed throughout the subiteration. The effect can be extrapolated to all subiterations and thus to the entire iteration, where each domain inserts tasks into the graph at each subiteration and in a balanced manner among them. As a result, the issue of balancing the workload of all subiterations at once is solved by this temporal-level balancing.

Based on promising initial observations, the following sections describe the implementation of this new approach and conduct an in-depth evaluation of its influence on both FLUSIM and FLUSEPA executions.

## V. Establishing multi-criteria partitioning through Metis

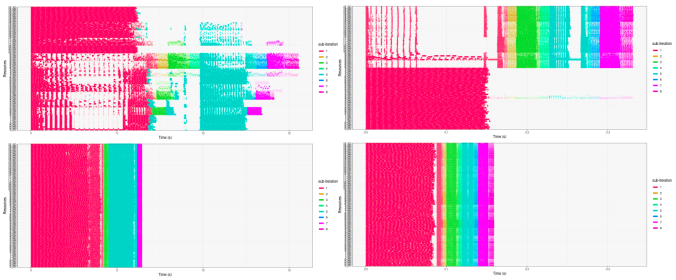
This section describes the implementations made to apply the new strategy to executions. The integration of *automatic multi-criteria partitioning* within the FLUSIM submodule and FLUSEPA itself represents another significant contribution of this article. The Metis library [12] was selected for its support for multi-criteria partitioning. Note that this method differs from the standard practice in multi-criteria approaches, where the initial partitioning step typically relies on a primary criterion, followed by the application of secondary criteria to refine the initial partition.

The new approach is implemented with Metis as follows. Consider the case of a mesh with three temporal levels: 0,1,2. The first step in FLUSEPA (and by transitivity FLUSIM) is to generate a graph from the mesh, where vertices represent cells and edges their associated faces. This graph is then passed to the Metis partitioning tool. Effectively, the goal of this method is to evenly distribute cells according to their temporal level. For this purpose, each vertex (i.e., cell in the mesh) is associated with a binary vector that has as many entries as the total number of temporal levels in the mesh. The only component that is assigned a value of 1 for a given vertex is the one corresponding to the temporal level of its respective cell. Coming back to our example, the vectors are of length 3 because the mesh contains 3 distinct temporal levels. Vectors of the form [1, 0, 0] are assigned to the $\tau = 0$ cells, [0, 1, 0] to the $\tau = 1$ cells, and [0, 0, 1] to the $\tau = 2$ cells. The objective of the partitioning is then to ensure that each constraint (i.e., component of the vectors) is evenly balanced across the target domains. Finally, instead of the k-way approach, we use the so-called *recursive bisection* method for partitioning because it produces higher quality solutions on our meshes.

## VI. Experimental evaluation using FLUSIM

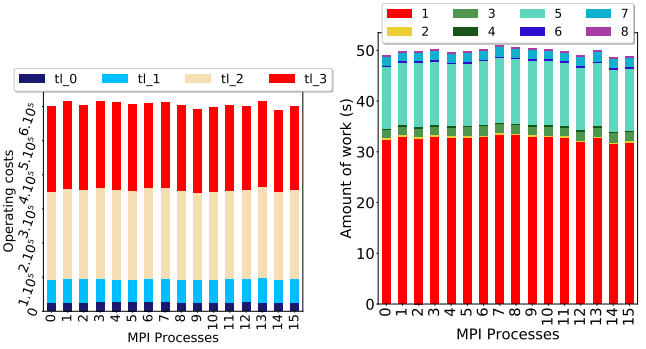
This section presents the preliminary results of applying the temporal-level partitioning strategy in an optimistic environment. It should be noted that all the experiments presented in this section are obtained from the FLUSIM submodule. To begin, the solver input mesh is divided into 128 domains, and the task graph is executed on 512 cores distributed across 16 MPI processes. Using `CYLINDER` and `CUBE` as input, the resulting executions are provided in Figure 9. The top traces depict the execution using the Single-Constraint Operating Cost strategy, now referred to as `SC_OC`. The lower traces correspond to executions using the Multi-Constraints Temporal-Level strategy, labelled as `MC_TL`. Whether we refer to Figure 9a or Figure 9b, both comparative diagrams provide a clear visual representation of *an acceleration factor of 2* in execution time by applying the new `MC_TL` strategy.

A more in-depth analysis of the execution can be done by analyzing the repartition of cells according to their temporal level in each domain. This is presented in Figure 10a. We can observe that the `MC_TL` strategy is able to evenly distribute each cell type among the processes. Moreover, this results in a completely balanced workload for each subiteration (see Figure 10b). This leads to having all the processes active throughout the execution which was the targeted objective. This analysis presented in Figure 10 should be compared with Figure 7 to see the impact of the `MC_TL` strategy.



(a) CYLINDER (b) CUBE

Fig. 9: Execution trace with FLUSIM, which simulates 16 MPI processes (32 cores per process). Top trace represents SC_OC, bottom trace MC_TL.



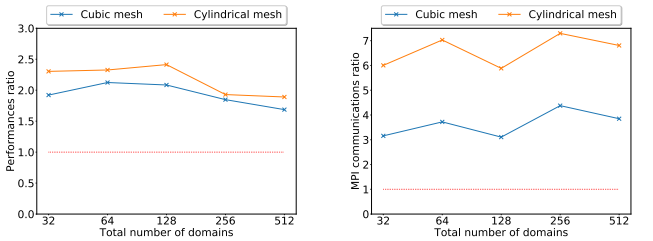

(a) Distribution of the total operating costs by temporal level among MPI processes

(b) Cumulative computation time by subiteration among MPI processes

Fig. 10: Domain characteristics for the execution of the CYLINDER case using 16 MPI processes (32 cores per process), using MC_TL.

An interesting property of the MC_TL approach is that it produces fine-grained tasks in comparison with the initial approach SC_OC. A domain produced through MC_TL includes internal cells with $\tau$ values ranging from 0 to 3. The solver algorithm generates separate tasks to process $\tau = 0$ cells, $\tau = 1$ cells, and so forth. *By taking advantage of the differentiation of each $\tau$ value in the task graph generation algorithm*, the computations are broken down into smaller tasks in these MC_TL domains, which leads to better occupancy on the computing nodes. Finally, it is important to remind that the total amount of work is independent of partitioning strategy. Both SC_OC and MC_TL perform the same amount of operations.

We now present the results illustrating the comparative behavior of both partitioning techniques with respect to the number of partitions targeted. Figure 11a presents the performance ratio of MC_TL with respect to SC_OC for different domain numbers. We can observe in Figure 11a that MC_TL outperforms SC_OC for both test cases, no matter the targeted domain count. Finally, we can see that the ratio decreases for larger domain counts. This is mainly due to the fact that larger domain numbers means smaller domains and finer granularity. By reducing task granularity, pipelining can be improved, which in turn overcomes load imbalances at each subiteration, especially in the SC_OC partitioning case.




(a) Performance ratio

(b) Estimated MPI communications ratio

Fig. 11: Comparing CYLINDER and CUBE executions under different partitioning strategies using 16 processes having 32 cores each. The blue curve is CYLINDER executions. The orange curve is CUBE executions.

Requesting the partitioner to create more contiguous domains while also aiming to balance the temporal levels is contradictory. The problem arises from the gradual distribution of temporal levels, originating from highly concentrated and dense mesh points. When using MC_TL, the partitioning phase often fails to guarantee the connexity of the domains while respecting all the constraints. This may induce an increase in communication volume, which can be verified in Figure 11b. Although FLUSIM does not provide any relative information about this topic, the amount of interprocess communication can be estimated. Indeed, a communication is considered to be an edge of the task graph connecting two nodes *whose domains are distributed across two different processes*. We expect most of this communication volume to be overlapped by FLUSEPA thanks to its use of the task-based programming model.

Finally, Figure 12 provides a runtime comparison within FLUSIM, where the traces highlight the influence of MC_TL and SC_OC. Consistent with earlier findings, the MC_TL strategy improves execution for the PPRIME_NOZZLE mesh. Its more intricate structure produces a slightly smaller, but still considerable, improvement of around 20%. This precise configuration (see Figure 5) serves as a validation support and comparison source when integrating the new implementation into FLUSEPA.

## VII. Experimental validation using FLUSEPA

In this section, we demonstrate the effectiveness of MC_TL when applied directly to the production environment where real-world simulations are conducted, such as the *PPRIME* jet nozzle experiment.

Validating the approach described in this paper involves achieving temporal level balance among the MPI processes in a *real-life execution of the FLUSEPA solver*. The final trace incorporates PPRIME_NOZZLE, which encapsulates the entirety of the simulated nozzle jet, making

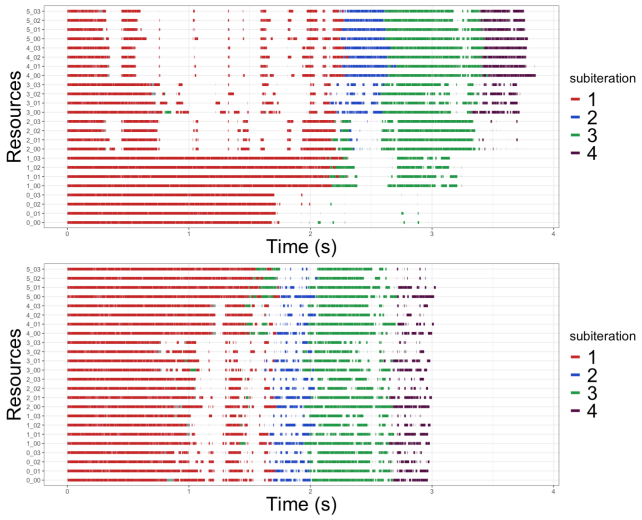



Fig. 12: Comparative traces within FLUSIM run with the same configuration and mesh as in Figure 5. `MC_TL` is depicted in the bottom trace, while `SC_OC` is depicted in the top one. Tasks are color-coded according to their subiteration.

it the most substantial representation. The configuration remains identical to both the prior trace comparisons presented in Figure 5 and 12. We present in Figure 13 the impact of the `MC_TL` strategy when used in the context of the FLUSEPA application. We can observe that using `MC_TL` saves about 20% of execution time when compared to the `SC_OC` strategy. The key point is that this performance gain is achieved *within the production code itself*, with *all the overhead and communication* that goes with it.

Continuous blocks of inactivity exacerbated by the original `SC_OC` strategy are easily seen on the top trace. The first half of the processes only work during the first and third subiteration. These mostly contain $\tau = 2$ and $\tau = 1$ cells, and the density of the inserted tasks appears quite dense. The other half, those with the $\tau = 0$-cells, are globally active in each subiteration. In contrast, the *best-performing trace* clearly shows a *well-balanced distribution* of tasks across both *processes* and *each subiteration*, filling in the periods of greatest inactivity. This trace reveals a significantly improved workload distribution between the resources than it was before.

A perspective for these works is proposed as a variation of the temporal-level strategy. The aim is to minimize the remaining communication and overhead in FLUSEPA, while maintaining reasonable performance. The number of domains defined by the user depends on both the resources of the target platform and the calculation granularity. We propose that these two aspects be decoupled into two phases of partitioning. The first balances the temporal levels (`MC_TL`) where a process is assigned to a single domain. To achieve efficient granularity with minimal communication, a second phase of partitioning is performed within each domain using an operational cost balancing strategy (`SC_OC`). Preliminary results suggest that this dual-phase multi-criteria partitioning is able to find a favorable compromise between performance improvement and communication overhead management.

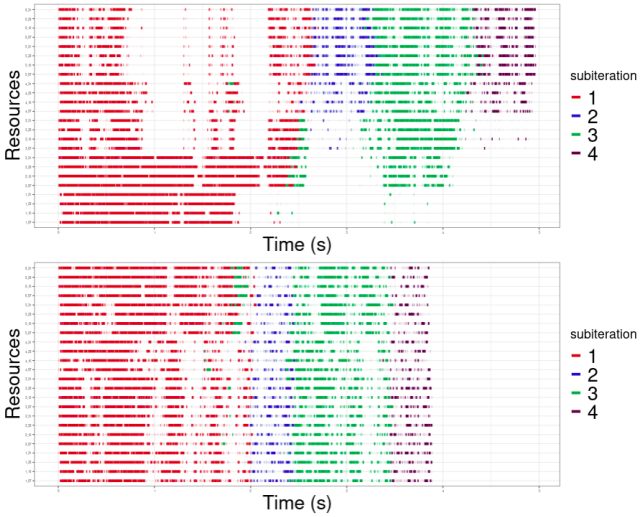



Fig. 13: The relevance of the temporal partitioning strategy is illustrated by the comparison of the FLUSEPA outputs, which were run with the same configuration and mesh as in Figure 5. Tasks are color-coded according to their subiteration. The temporal level strategy corresponds to the bottom trace, while the operating cost strategy corresponds to the top graph. A 20% gain can be observed within the production code itself.

## VIII. Related work

A multi-constraint approach is formulated in this paper. The graph partitioner of choice for this study is Metis, which allows defining a vector of constraints that directly influence the initial partitioning of the multilevel algorithm [11], [17]. Partitioners offering geometric approaches, such as Zoltan [10] or KaHIP [16], can also be used to obtain a multi-criteria approach directly. These are based directly on the physical coordinates of the mesh data, without taking into account the connectivity of the elements.

Another approach is to extend single-criteria partitioners with additional constraints. Some works propose new implementations of the various steps of the multi-level algorithm (contraction, initial partitioning and refinement) integrated in the Scotch [14] graph partitioning library. In [5], the authors introduce a hierarchical partitioning that tries to minimize the number of edge cuts while balancing the workload across the processes and across cores within each process. In [3], [4], the authors focus on obtaining domains that strictly respect the equilibrium tolerances. The goal is to achieve the most efficient solution with the minimum number of communications.

Finally, the work that comes closest to the approach proposed in this article is discussed in [18]. The authors present mesh partitioning strategies with direct application to multiphase numerical simulations with behavior similar to the one in the article. Indeed, these simulations present several phases, separated by synchronizing steps, that interpolate a highly disparate amount of calculation between them. Two approaches are suggested to achieve an individual balance of the phases. The first one is to carry out an independent partitioning for each of the phases. The data is then redistributed between the phases, which leads to considerable data having to transfer. The same article suggests a more convincing approach of balancing the work of these phases simultaneously in a single partitioning step. In one example, an equal number of active elements from each phase are distributed between the domains.

## IX. Conclusion

Fluid dynamics explicit solvers used in aeronautics operate on unstructured meshes of very large size that need to be distributed over the underlying parallel machine. To ensure the accuracy of computations, the mesh cells are sorted in distinct classes which define their maximum allowable time step. To avoid penalizing the simulation by calculating all cells with the maximum frequency of the smallest cells, solvers use an adaptive time-stepping scheme. As a result, the iterations are divided into subiterations in which only a subset of cells is computed.

In this paper, we point out that the parallel task-based implementation of the FLUSEPA solver suffers from unexpected load-balancing issues, even though the same amount of computation is assigned to each MPI process. Furthermore, we demonstrate that these problems are not due to implementation flaws or task scheduling issues, but arise directly from the imbalanced structure of the task graph.

We propose a new partitioning strategy that seeks to evenly distribute temporal-level classes across domains through the task graph generation algorithm. Our implementation of this strategy is based on a multi-criteria partitioning performed by the Metis tool. We validate our approach by applying it to three production meshes, each of which has its own unique set of characteristics. In the end, we obtain a 20% increased performance when running FLUSEPA.

This work opens up several perspectives. We are currently exploring ways to automatically determine the best domain granularity with respect to the target machine's number of cores. We also intend to develop post-processing techniques to minimize the artifacts produced by partitioners when constrained by many criteria [1]. Indeed, they tend to create disconnected subdomains that increase the number of domain borders and, thus, the number of communications and tasks.

## References


[1] Michael Aftosmis, Marsha Berger, and Scott Murman. Applications of space-filling-curves to cartesian methods for cfd. 06 2003.

[2] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[3] Remi Barat. *Load Balancing of Multi-physics Simulation by Multi-criteria Graph Partitioning.* Theses, Université de Bordeaux, December 2017.

[4] Remi Barat, Cédric Chevalier, and François Pellegrini. Multi-criteria Graph Partitioning with Scotch. In *SIAM Workshop on Combinatorial Scientific Computing*, pages 66–75, Bergen, Norway, June 2018.

[5] Astrid Casadei, Pierre Ramet, and Jean Roman. An improved recursive graph bipartitioning algorithm for well balanced domain decomposition. In *IEEE International Conference on High Performance Computing (HiPC 2014)*, pages 1–10, Goa, India, December 2014.

[6] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917, June 2014.

[7] Jean Marie Couteyen Carpaye. *Contribution à la parallélisation et au passage à l'échelle du code FLUSEPA*. Theses, Université de Bordeaux, September 2016.

[8] Jean Marie Couteyen Carpaye, Jean Roman, and Pierre Brenner. Towards an efficient Task-based Parallelization over a Runtime System of an Explicit Finite-Volume CFD Code with Adaptive Time Stepping. In *International Parallel and Distributed Processing Symposium*, PDSEC'2016 workshop of IPDPS, page 10, Chicago, IL, United States, May 2016.

[9] Jean Marie Couteyen Carpaye, Jean Roman, and Pierre Brenner. Design and Analysis of a Task-based Parallelization over a Runtime System of an Explicit Finite-Volume CFD Code with Adaptive Time Stepping. *International Journal of Computational Science and Engineering*, pages 1 – 22, 2017.

[10] Devine, Karen and Boman, Erik and Riesen, Lee and Catalyurek, Umit and Chevalier, Cédric. *Zoltan : Parallel toolkit for combinatorial scientific computing: dynamic partitioning, graph coloring and ordering*, 2022.

[11] George Karypis and Vipin Kumar. Multilevel algorithms for multi-constraint graph partitioning. pages 28– 28, 12 1998.

[12] Karypis, George and Kumar, Vipin. *METIS : A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices*, 2022.

[13] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*, June 2021.

[14] Pellegrini, François. *SCOTCH : Static mapping, graph partitioning, and sparse matrix block ordering package*, December 2021.

[15] Grégoire Pont, Jérôme Huber, Jean-Paul Roméo, and Pierre Brenner. Installed jet noise simulation in industrial framework. 03 2020.

[16] Peter Sanders and Christian Schulz. Think locally, act globally: Highly balanced graph partitioning. 7933:164–175, 2013.

[17] Kirk Schloegel, George Karypis, and Vipin Kumar. Parallel multilevel algorithms for multi-constraint graph partitioning (distinguished paper). pages 296–310, 01 2000.

[18] Kirk Schloegel, George Karypis, Vipin Kumar, Jack Dongarra, Ian Foster, Geoffrey Fox, K. Kennedy, A. White, and Morgan Kaufmann. Graph partitioning for high performance scientific simulations. 06 2000.