Int. J. Numer. Meth. Eng.

MPI+X: task-based parallelization and dynamic load balance of finite element assembly

Marta Garcia-Gasulla*,[1] Guillaume Houzeaux,[1] Roger Ferrer,[1] Antoni Artigues,[1] Victor López,[1] Jesús Labarta,[1] and Mariano Vázquez[1]

[1] *Barcelona Supercomputing Center (BSC), Edificio NEXUS II, c) Jordi Girona 29, 08034 Barcelona (Spain)*

Correspondence: *Marta Garcia-Gasulla, Email: marta.garcia@bsc.es

Present Address: Barcelona Supercomputing Center (BSC), Edificio NEXUS II, c) Jordi Girona 29, 08034 Barcelona (Spain)

Received -; Revised -; Accepted -

Summary

The main computing tasks of a finite element code(FE) for solving partial differential equations (PDE's) are the algebraic system assembly and the iterative solver. This work focuses on the first task, in the context of a hybrid MPI+X paradigm. Although we will describe algorithms in the FE context, a similar strategy can be straightforwardly applied to other discretization methods, like the finite volume method. The matrix assembly consists of a loop over the elements of the MPI partition to compute element matrices and right-hand sides and their assemblies in the local system to each MPI partition. In a MPI+X hybrid parallelism context, X has consisted traditionally of loop parallelism using OpenMP. Several strategies have been proposed in the literature to implement this loop parallelism, like coloring or substructuring techniques to circumvent the race condition that appears when assembling the element system into the local system. The main drawback of the first technique is the decrease of the IPC due to bad spatial locality. The second technique avoids this issue but requires extensive changes in the implementation, which can be cumbersome when several element loops should be treated. We propose an alternative, based on the task parallelism of the element loop using some extensions to the OpenMP programming model. The taskification of the assembly solves both aforementioned problems. In addition, dynamic load balance will be applied using the DLB library, especially efficient in the presence of hybrid meshes, where the relative costs of the different elements is impossible to estimate a priori. This paper presents the proposed methodology, its implementation and its validation through the solution of large computational mechanics problems up to 16k cores.

Keywords: CFD, solid mechanics, finite element, dynamic load balance, task parallelism, MPI, OpenMP, hybrid parallelization, runtime

1 Introduction

The two most intensive computing tasks of computational mechanics codes for unstructured meshes are the algebraic system assembly and the iterative solver to solve it. In this paper we will focus on improving the performance and execution of the first task, the algebraic system assembly.

The algebraic system assembly consists of a loop over elements, in the Finite Element (FE) context, and faces or cells in the Finite Volume (FV) context. Although this work will focus on the first family of methods, all the strategies described here can be applied to the second one.

For each element, the system assembly consists of two main steps:

- Compute the element matrix and right-hand side.
- Assembly the element system into the local algebraic system of each MPI partition.

The element loop is local to each MPI partition and does not involve any communication. It is thus well-suited for shared memory parallelism. In an MPI+X hybrid parallelism context, X has consisted traditionally of loop parallelism using OpenMP. However, assembling the element system into the local one involves an update of a shared variable which limits drastically the efficiency of the straightforward use of OpenMP pragmas.

Several strategies have been proposed in the literature to circumvent this weakness, like the coloring or substructuring techniques to avoid the race condition appearing in the assembly of the element system into the local system. The main drawback of the first technique is the drop of the number of instructions per cycle (IPC) due to the bad spatial locality inherent to the coloring. The second technique solves this issue but requires intensive recoding, which can be cumbersome when several element loops should be treated. These techniques will be summarized in Section 3.

We propose an alternative, based on the task parallelism of the element loop using an extension to the OpenMP programming model and implemented in the OmpSs model(9)(4). The taskification of the assembly that we propose solves both aforementioned problems. The technique will be described in Section 3.2.2.

In addition, in the context of MPI parallelization load imbalance is an issue that can degrade the performance and does not have a straightforward solution. The main issue when load balancing an MPI application comes from the fact that the data is not shared among the different MPI processes. Consequently, application developers put a lot of effort at obtaining a well balanced data partition (23) (18) (29).

Unfortunately a well balanced partition is not always easy to obtain as we will see in Section 4. And, even, if a well balanced partition is achieved it does not imply a well balanced execution. In some cases the load can change during the execution, i.e. particles moving or a dam breaking. In this case, a runtime solution is necessary. One of the solutions proposed in the literature is to repartition the mesh during the execution to obtain a better balanced distribution (30). This kind of solutions implies a redistribution of data and cannot be applied each timestep because of the overhead they introduce. Moreover, they cannot react to punctual load changes or load imbalance introduced by system noise. We will apply a dynamic load balance that does not require to modify the application neither to redistribute data.

Finally, in Section 5, the efficiency of the proposed taskifying strategy will be compared to classical loop parallelism with OpenMP using an element coloring strategy. In this section we will also present the performance evaluation of the load balancing library. And we will demonstrate that both mechanisms can be useful to scale a finite element code up to 16386 cores.

2 Fluid and structure dynamics

In this work we consider two different sets of partial differential equations (PDE's), modeling incompressible flows and large deformations of structures. We will put more emphasis on the first set of equations, as the numerical modeling and system solution are more complex. Apart from the sets of equations to be solved, we will introduce as well the case examples selected to carry out the proposed optimizations. In the case of the Navier-Stokes equations we will consider the airflow in the respiratory system, while for structure mechanics, we will consider a fusion reactor.

2.1 Fluid solver

The high performance computational mechanics code used in this work is Alya (27), developed at BSC-CNS, and part of the Unified European Application Benchmark Suite (UEABS) (6). This suite provides a set of scalable, currently relevant and publically available codes and datasets, of a size which can realistically be run on large systems, and maintained into the future. In this section, will briefly describe the CFD module of Alya and its parallelization.

2.1.1 Physical and Numerical models

The equations governing the dynamics of an incompressible fluid are the so-called incompressible Navier-Stokes equations. They express the Newton's second law for a fluid continuous medium, whose unknowns are the velocity $\boldsymbol{u}$ and the pressure p of the fluid. Two physical properties are involved, namely μ be the viscosity, and ρ the density. At the continuous level, the problem is stated as follows: find the velocity $\boldsymbol{u}$ and pressure p in a domain Ω such that they satisfy in a

given time interval

$$\rho\frac{\partial \boldsymbol{u}}{\partial t} + \rho(\boldsymbol{u}\cdot\nabla)\boldsymbol{u} - \nabla\cdot[2\mu\boldsymbol{\varepsilon}(\boldsymbol{u})] + \nabla p = \boldsymbol{0},$$
$$\nabla\cdot\boldsymbol{u} = 0,$$

together with initial and boundary conditions. The velocity strain rate is defined as $\boldsymbol{\varepsilon}(\boldsymbol{u}) := \frac{1}{2}(\nabla\boldsymbol{u} + \nabla\boldsymbol{u}^t)$.

The variational multiscale (VMS) method is applied to discretize this set of equations, as extensively described in (15). In addition, the velocity subgrid scale is tracked in convection and time. This means that apart from solving for the previous unknowns $\boldsymbol{u}$ and p, an additional equation is solved to obtain the subgrid scale $\tilde{\boldsymbol{u}}$. A typical assembly for the grid scale equations consists in a loop over the elements of the mesh, as shown in Algorithm 1.

Algorithm 1 Assembly of a generic matrix $\mathbf{A}$ and vector $\mathbf{b}$.

1: **for** elements e **do**
2: Compute element matrix and RHS: $\mathbf{A}^e$, $\mathbf{b}^e$
3: Assemble matrix $\mathbf{A}^e$ into $\mathbf{A}$
4: Assemble RHS $\mathbf{b}^e$ into $\mathbf{b}$
5: **end for**

2.1.2 Algebraic system solution

After the assembly step, the following monolithic algebraic system for the grid scale unknowns, velocity $\mathbf{u}$ and pressure $\mathbf{p}$, is obtained:

$$\begin{bmatrix} \mathbf{A}_{uu} & \mathbf{A}_{up} \\ \mathbf{A}_{pu} & \mathbf{A}_{pp} \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \mathbf{p} \end{bmatrix} = \begin{bmatrix} \mathbf{b}_u \\ \mathbf{b}_p \end{bmatrix}. \tag{1}$$

This system can be solved directly using a Krylov solver and efficient preconditioner (24). However, an algebraic-split approach is used instead in this work. We extract the pressure Schur complement of the pressure unknown $\mathbf{p}$ and solve it with the Orthomin(1) method, as detailed in (12). The resulting algorithm is shown in Algorithm 2. In the algorithm, matrix $\mathbf{Q}$ is the pressure

Algorithm 2 Algebraic solver: Orthomin(1) method for the pressure Schur complement.

1: Solve momentum eqn $\mathbf{A}_{uu}\mathbf{u}^{k+1} = \mathbf{b}_u - \mathbf{A}_{up}\mathbf{p}^k$
2: Compute Schur complement residual $\mathbf{r}^k = [\mathbf{b}_p - \mathbf{A}_{pu}\mathbf{u}^{k+1}] - \mathbf{A}_{pp}\mathbf{p}^k$
3: Solve continuity eqn $\mathbf{Q}\mathbf{z} = \mathbf{r}^k$
4: Solve momentum eqn $\mathbf{A}_{uu}\mathbf{v} = \mathbf{A}_{up}\mathbf{z}$
5: Compute $\mathbf{x} = \mathbf{A}_{pp}\mathbf{z} - \mathbf{A}_{pu}\mathbf{v}$
6: Compute $\alpha = (\mathbf{r}^k, \mathbf{x})/(\mathbf{x}, \mathbf{x})$
7: Update velocity and pressure

$$\begin{cases} \mathbf{p}^{k+1} = \mathbf{p}^k + \alpha\mathbf{z} \\ \mathbf{u}^{k+2} = \mathbf{u}^{k+1} - \alpha\mathbf{v} \end{cases}$$

Schur complement preconditioner, computed here as an algebraic approximation of the Uzawa operator, and as explained (12). On the one hand, the momentum equation is solved with the GMRES method and diagonal preconditioning in steps 1 and 4 of the algorithm. On the other

hand, the continuity equation is solved with the Deflated Conjugate Gradient (DCG) method (19) with linelet preconditioning (25) in step 3 of the algorithm. The deflation provides a low frequency damping across the domain, especially very efficient for the case study considered in this work, where the geometry is elongated. The linelet preconditioner consists of a tridiagonal preconditioner applied in the normal direction to the boundary layer mesh near the walls.

At each time step, this system is solved until convergence is achieved. Convergence is necessary because the original equation is non-linear (the convective term makes matrix $\mathbf{A}_{uu}$ depend on $\mathbf{u}$ itself). For any information concerning the parallel solution system (1) on distributed memory supercomputers, see (12, 16). Only a brief description will be given herein in Section 3.1.

2.1.3 Subgrid scale

Once the velocity $\mathbf{u}$ and pressure $\mathbf{p}$ are obtained on the nodes of the mesh, the velocity subgrid scale vector is obtained through a general equation of the form

$$\tilde{\mathbf{u}} = \tau^{-1} \left(\mathbf{R}\mathbf{u} + \mathbf{b}_{\tilde{\mathbf{u}}} \right), \tag{2}$$

where τ is the so-called stabilization diagonal matrix and $\mathbf{R}$ is the residual rectangular matrix, as the subgrid scale is obtained element-wise and not node-wise. Both τ and $\mathbf{R}$ may depend on $\tilde{\mathbf{u}}$ and thus Equation (2) can be non-linear. Note finally that in practice, $\tilde{\mathbf{u}}$ is obtained via a simple loop over the elements of the mesh and the system of the equation does not need to be explicitly formed.

2.1.4 Solution strategy

In practice, the iterations of the Orthomin(1) iterative solver to solve for the pressure Schur complement are coupled to the non-linearity iterations of the Navier-Stokes equations, which include not only the convective term but also the subgrid scale. The resulting workflow is shown in Algorithm 3.

Algorithm 3 Solution strategy for solving Navier-Stokes equations.

```
1: for time steps do
2:     while until convergence do
3:         Assemble global matrices and RHS of Equation (1) using Algorithm 1
4:         Solve momentum equation with GMRES, step 1 of Algorithm 2
5:         Solve continuity equation with DCG, step 3 of Algorithm 2
6:         Solve momentum equation with GMRES, step 4 of Algorithm 2
7:         Compute subgrid scale using Equation (2)
8:     end while
9: end for
```

The workflow consists of three main computational kernels. The *assembly* which carries out operations on the elements of the mesh in order to construct the algebraic system; the *algebraic solver*, that is the algorithm for the pressure Schur complement, which consists in solving twice the momentum equation and once the continuity equation; finally, the *subgrid scale* calculation which is computed on the elements of the mesh and thus involves a loop over the elements.

2.1.5 Case example: respiratory system

For the evaluation of the different techniques described in the following sections we will consider the case of the respiratory system, similar to that described in (7). The mesh is hybrid and composed of 17.7 million elements: prisms to resolve accurately the boundary layer; tetrahedra in the core flow; pyramids to enable the transition from prism quadrilateral faces to tetrahedra. This

kind of mesh is quite representative in fluid dynamics, as most of the fluid problems of interest involve boundary layers and a core flow. Figure 1 shows some details of the mesh, and in particular the prisms in the boundary layer. We will see in Section 4.1 how the presence of different types of elements makes difficult the control of the load balance when using the mesh partitioner METIS.

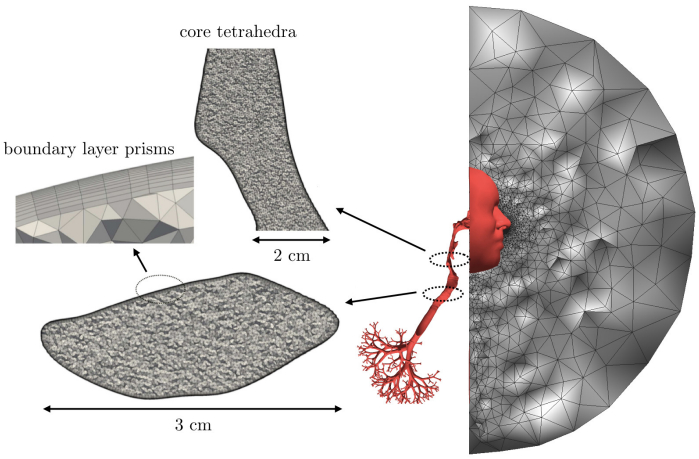


Figure 1: Respiratory system, details of the mesh.

2.2 Structure solver

The structure mechanics solver is extensively described in (8). For the sake of completeness, we will only briefly describe the set of equations to be solved.

2.2.1 Physical and Numerical models

The equation of balance of momentum with respect to the reference configuration can be written as

$$\rho_0 \frac{\partial^2 \boldsymbol{u}}{\partial t^2} - \nabla_0 \cdot \boldsymbol{P} = \boldsymbol{b}_0, \tag{3}$$

where ρ_0 is the mass density (with respect to the reference volume) and $\nabla_0 \cdot$ is the divergence operator with respect to the reference configuration. Tensor $\boldsymbol{P}$ and vector $\boldsymbol{b}_0$ stand for the first Piola-Kirchhoff stress and the distributed body force on the undeformed body, respectively. Equation (3) must be supplied with initial and boundary conditions.

To discretize this equation, the Galerkin method is used in space and the Newmark method (5) in time. A Newton-Raphson method is used to solve the linearized system. For each time step, and until convergence, one has to assemble the algebraic system using Algorithm 1 (where $\mathbf{A}$ is the Jacobian and $\mathbf{b}$ is the residual of the equation) and then solve the corresponding algebraic system for the displacement unknown. According to the characteristic of this system, the GMRES or the DCG methods are considered.

2.2.2 Case example: Iter

The mesh is a slice of a torus shaped chamber, representing the center part of a nuclear fusion reactor called the vacuum vessel. In Figure 2 we can see the representation of the mesh made of 31.5 million hexahedra, prisms, pyramids and tetrahedra elements.

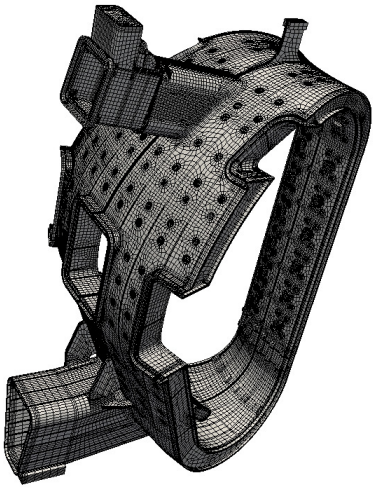


Figure 2: Iter, mesh.

3 Parallelization of Algebraic System Assembly

For the sake of completeness, this section describes briefly the classical parallelization techniques of a finite element assembly in an HPC environment.

3.1 Distributed Memory Parallelization Using MPI

In the finite element context, two options are available when defining a distributed memory parallelization, which we refer to as partial-row and full-row methods (20). In the finite difference context or in the finite volume context using a cell centered discretization scheme, the second method is generally considered, by construction. (Guillaume) (Do we mean by "considered by construction" mean something like "emerges naturally"? If so, then I don't understand the paragraph below about the "partial-row" being quite natural in the finite element community) (Roger)

In the finite element community, the partial-row method is quite natural when partitioning the original element set (mesh) into disjoint subsets of elements (subdomains). In this case, nodes belonging to several subdomains (interface nodes) are duplicated. As the matrix coefficients come from element integrations, the coefficients of the edges involving interface nodes are never fully assembled. However, if one considers iterative solvers to solve the resulting algebraic system (which is the case in this work), the main operation consists of Sparse Matrix-Vector Products (SpMV) $\mathbf{y} = \mathbf{A}\mathbf{x}$. Thus, there is no need for obtaining these global coefficients, as the result $\mathbf{y}$ can be computed locally on each subdomain and then assembled later on interface nodes using MPI send and receive messages (by associativity of the multiplication operation).

The full-row approach consists in assigning nodes exclusively to one subdomain. Two options are thus possible. A first option in which the global matrix coefficients of the interface nodes can be obtained by summing up the contributions coming from neighboring subdomains, thus obtaining full rows. This can be achieved through MPI communications.

A second option in which the mesh is partitioned into disjoint subsets of nodes. The full rows of the matrix are thus obtained by assigning all the necessary elements to the subdomains in order to get all the element contributions. In this case, interface elements must be duplicated (halo elements), leading to duplication of the work during the assembly process on these halo elements. As far as SpMV is concerned, MPI communications are performed before the product on the multiplicand $\mathbf{x}$.

The advantage of the partial-row approach is that the load balance of the assembly can be controlled, in principle, when partitioning the mesh. With partitioners like METIS (17), the number of elements per subdomain can in addition be constrained with the minimization of the interface sizes. The main drawback is that while balancing the number of elements per subdomain,

one loses control on the number of nodes, which dictates the balance of SpMV. On the other hand, the presence of halo elements in the full-row approach limits the scalability, due to the duplicated work on them and to the lack of control on the number of elements per subdomain.

When considering hybrid meshes, an additional difficulty arises, as one has to estimate the relative weights of the different element types in order to balance the total weight per subdomain, as we will show in Section 5. Thus, no matter if full-row or partial-row is ultimately chosen, load imbalance will occur before starting the simulation. In this work, the partial-row approach is considered and described in (27), while load imbalance will be treated in Section 4.

3.2 Shared Memory Parallelization Using OpenMP

3.2.1 Loop parallelism

During the last decade, the predominance of general purpose clusters have obliged parallel code designers to devise distributed memory techniques, mainly based on MPI, as briefly described in last subsection. Then, while the number of CPUs has been multiplied, the number of subdomains has been increasing. The side effect is the increase of communication which limits the strong scalability, and the increase of number of subdomains (Guillaume)these are subdomains of the partial-row option described above, right? Perhaps reminding the reader here would help.(Roger), which limits the weak scalability. Nowadays, supercomputers offer a great variety of architectures, with many cores on nodes (e.g. Xeon Phi). Thus, shared memory parallelism is gaining more and more attention as a it offers more flexibility to parallel programming. This parallelism has traditionally been based on OpenMP, a programming model enabling a straightforward parallelization through simple pragmas. Finite element assembly consists in computing element matrices and right-hand sides ($\mathbf{A}^{(e)}$ and $\mathbf{b}^{(e)}$) for each element e, and assembling them into the local matrices and RHS of each MPI process, namely $\mathbf{A}$ and $\mathbf{b}$, as shown in Algorithm 1. This assembly has been treated using mainly three techniques, as illustrated in Figure 3.

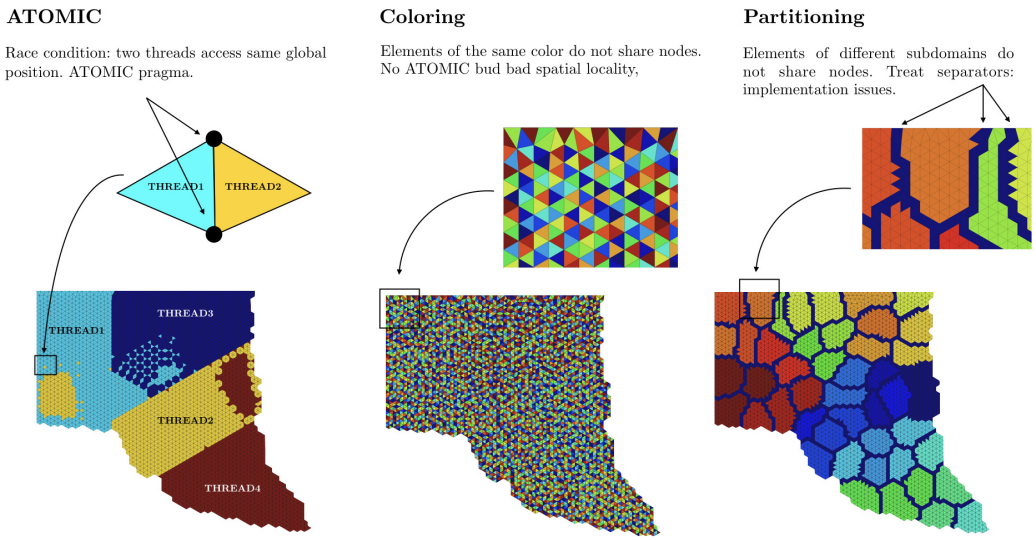


Figure 3: Shared memory parallelism techniques using OpenMP. (a) ATOMIC pragma. (b) Element coloring. (c) Local partitioning.

All of these techniques are based on loop parallelism, each of them offering different advantages and drawbacks. The main issue is the race condition appearing in the assembly scattering the element arrays $\mathbf{A}^{(e)}$ and $\mathbf{b}^{(e)}$ into the local ones, $\mathbf{A}$ and $\mathbf{b}$. The first method consists in avoiding

the race condition using ATOMIC pragmas to protect these shared variables (Figure 3 (left)). The cost of the ATOMIC limits the scalability of the assembly. This strategy is shown in Algorithm 4.

Algorithm 4 Matrix Assembly without coloring in each MPI partition

```
1: !$OMP PARALLEL DO SCHEDULE (DYNAMIC,Chunk_Size) &
2: !$OMP PRIVATE (...) &
3: !$OMP SHARED  (...)
4: for elements e do
5:     Compute element matrix and RHS: A^e, b^e
6:     !$OMP ATOMIC
7:     Assemble matrix A^e into A
8:     !$OMP ATOMIC
9:     Assemble RHS b^e into b
10: end for
```

In the context of vectorization, the element coloring technique has been proposed (10, 21). By coloring elements such that elements with the same color do not share nodes, no ATOMIC is required to protect $\mathbf{A}$ and $\mathbf{b}$. The main drawback of this method is that any spatial locality of data is lost which implies a low IPC (instructions per cycle). In the performance evaluation based in hardware counters included in Section 5, we will show that the loss in IPC is up to 100% due to the ATOMIC pragma, while it is of 50% using the coloring technique respect using a pure MPI version. This strategy is shown in Algorithm 5. To unify the terminology with other techniques, we define a subdomain as a set of elements of the same color, and $nsubd$ the total number of subdomains, that is the total number of colors.

Algorithm 5 Matrix assembly with coloring in each MPI partition

```
1: Partition local mesh in nsubd subdomains using a coloring strategy
2: for isubd = 1, nsubd do
3:     !$OMP PARALLEL DO SCHEDULE (DYNAMIC,Chunk_Size) &
4:     !$OMP PRIVATE (...) &
5:     !$OMP SHARED  (...)
6:     for elements e in isubd do
7:         Compute element matrix and RHS: A^e, b^e
8:         Assemble matrix A^e into A
9:         Assemble RHS b^e into b
10:    end for
11:    !$OMP END PARALLEL DO
12: end for
```

To circumvent these two inconveniences, local partitioning techniques (in each MPI partition independently) have been proposed (2, 26). Here, classical partitioners like METIS or Space Filling Curve based partitioners can be used. Elements are assigned to subdomains, and subdomains are unconnected through separators (layer of elements) such that elements of neighboring subdomains do not share nodes. By assigning elements to a subdomain, the loop over elements is substituted by the parallelization of the loop over subdomains. This techniques guarantees spatial locality and avoids the race condition. However, this force us to treat the separators differently (e.g. by re-decomposition) and makes its implementation more complex. The algorithm is shown in Algorithm 6. We can observe the similitude between the loops of the coloring and local partitioning techniques. The differences are in the way the subdomains are obtained (coloring *vs* METIS) and the existence of separators in the local partitioning technique.

Algorithm 6 Matrix assembly with local partitioning in each MPI partition

```
1: Partition local mesh in nsubd subdomains using METIS
2: for isubd = 1, nsubd do
3:     !$OMP PARALLEL DO SCHEDULE (DYNAMIC,Chunk_Size) &
4:     !$OMP PRIVATE (...) &
5:     !$OMP SHARED  (...)
6:     for elements e in isubd do
7:         Compute element matrix and RHS: A^e, b^e
8:         Assemble matrix A^e into A
9:         Assemble RHS b^e into b
10:    end for
11:    !$OMP END PARALLEL DO
12: end for
13: Treat separators
```

3.2.2 Task parallelism

It is possible to implement another strategy by forgoing the loop parallelism approaches shown above and using a task parallelism approach instead.

As of OpenMP 3.0 a new tasking model was introduced which allows the OpenMP programmer to parallelize a set of problems with irregular parallelism. When a thread of the OpenMP program encounters a `TASK` construct it creates a task which is then run by one of the threads of the parallel region. In principle the order, i.e. the schedule, in which the tasks are run is not determined by the creation order. OpenMP 4.0 allows constraining the schedule by adding the possibility of defining dependences between tasks. This way the OpenMP programmer can use a data-flow style for irregular parallelism.

While intuitive, the dependency approach based on input and output dependences is too strict for a problem like the finite element assembly. It forces the runtime to determine a particular order (necessarily influenced by the task creation order) that fulfills the dependences when executing the tasks.

The research in the OmpSs programming model (28) led to the proposal of a new kind of dependency between tasks called `COMMUTATIVE`. This new dependency kind means that two tasks cannot be run concurrently if they refer to the same data object but does not impose any other restriction in the particular order in which such exclusive execution happens. This kind of dependency is suitable for our problem as, in principle, we do not really care which subdomain is processed first as long as two subdomains that share a border are not processed concurrently.

A further complication exists, though, for the current dependency support in OpenMP 4.0 implies that the number of dependences is statically defined at compile time. This is inconvenient as each subdomain may have a variable number of neighbors. To address this we use the multidependences extensions in which each task may have a variable number of dependences (28).

In this way, the tasking parallelization is possible by first computing the adjacency list of each subdomain. Figure 4 depicts this idea, where the subdomain 3 has 5 neighbors (including itself). Given that adjacency list, it is then possible to use a `COMMUTATIVE` multidependence on the neighbors. This causes the runtime to run as many subdomains as possible in parallel that do not share any node.

As subdomains are processed, less of them will remain and the parallelism available will decrease. It is possible to get higher concurrency levels if, of all the non-neighboring subdomains, we first process those with a bigger number of neighbors: intuitively this potentially can free more subdomains that are not neighbors. To this end, we prioritize subdomains with a higher neighbor count. We can achieve this using the `PRIORITY` clause.

Algorithm 7 shows the final task parallelization. For each subdomain: we create a task (step 5); then we declare a commutative dependency with all its neighbors (step 6); we prioritize tasks with a higher number of neighbors (step 7). These tasks are created by a single thread inside of

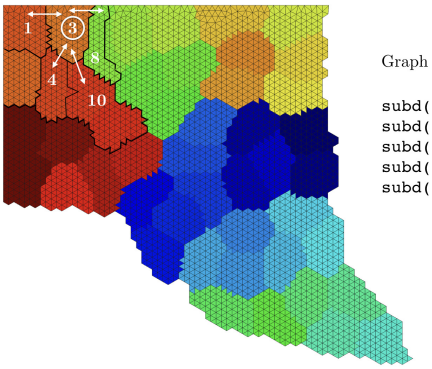


Figure 4: Task parallelism using multidependences: neighbors of subdomain 3 (including itself).

a parallel region (not shown in the listing) and the thread will not proceed until all of them have been run by itself or other threads of the team (step 17).

Algorithm 7 Matrix assembly with commutative multidependences in each MPI partition

1: Partition local mesh in *nsubd* subdomains of size `Chunk_Size`
2: Store connectivity graph of subdomains in structure *subd*
3: **for** $isubd = 1, nsubd$ **do**
4: $nneig = SIZE(subd(isubd)\%lneig)$
5: `!$OMP TASK &`
6: `!$OMP COMMUTATIVE ([subd(subd%lneig(i)), i=1,nneig]) &`
7: `!$OMP PRIORITY (nneig) &`
8: `!$OMP PRIVATE (...) &`
9: `!$OMP SHARED (subd, ...)`
10: **for** Elements e in *isubd* **do**
11: Compute element matrix and RHS: $\mathbf{A}^e$, $\mathbf{b}^e$
12: Assemble matrix $\mathbf{A}^e$ into $\mathbf{A}$
13: Assemble RHS $\mathbf{b}^e$ into $\mathbf{b}$
14: **end for**
15: `!$OMP END TASK`
16: **end for**
17: `!$OMP TASKWAIT`

4 Dynamic Load Balancing

4.1 MPI Load Imbalance

As we explained in Section 3.1, using an MPI parallelization implies partitioning the original mesh into n subdomains. The essential characteristic of MPI is that it works on distributed memory, thus, each MPI process will work on the data in its subdomain. This fact makes the mesh partitioning crucial, as it will determine the load balance of the execution. Although there are techniques to redistribute or repartition the mesh during the execution, these are expensive as they require to move data between processes and to modify the code the code to repartition when necessary.

The mesh partitioning software provides, in general, load balancing features which necessarily are based on optimizing a metric. As we discussed in the introduction, the main computational

tasks of a CFD and structure mechanics codes are the algebraic system assembly and the iterative solver.

We want to obtain a partition that ensures load balance in the matrix assembly. But in hybrid meshes the number of elements might not be a good metric to measure the load balance, as their relative weights in constructing the matrix may be different.

In this paper we focus on the assembly phase, for this reason we want to obtain a partition that achieves a well balanced distribution of elements among the different MPI processes. In a hybrid mesh, the computational loads of different elements are not the same. As a intuitive guess, we will assign as a weight to each element the number of Gauss points used in the matrix assembly.

We define Load Balance as the percentage of time that the computational resources are doing useful computation:

$$\text{Load Balance} = \frac{\text{Useful CPU time}}{\text{Total CPU time}} = \frac{\text{average}_{i=1}^{n_{\text{MPI}}} time_i}{\max_{i=1}^{n_{\text{MPI}}} time_i}$$

Let us define w_e the weight of element e and n_e^i the number of elements of partition i. We define two theoretical load balance (LB) measures: the *non-weighted load balance* which is the ratio of the average number of elements to the maximum number of elements, as well as the *weighted load balance*, which represents the same including the weights given to METIS (that is what METIS load balances). We also introduce the *measured load balance* obtained by measuring the elapsed time (time_i) of each MPI task in the assembly or subgrid scale loop and dividing the average by the maximum of the elapsed times.

$$\begin{aligned}
\text{Theoretical weighted LB} &= \frac{(\sum_i^{n_{\text{MPI}}} \sum_e^{n_e^i} w_e)/n_{\text{MPI}}}{\max_i(\sum_e^{n_e^i} w_e)}, \\
\text{Theoretical non-weighted LB} &= \frac{(\sum_i^{n_{\text{MPI}}} n_e^i)/n_{\text{MPI}}}{\max_i n_e^i}, \\
\text{Measured LB} &= \frac{(\sum_i^{n_{\text{MPI}}} time_i)/n_{\text{MPI}}}{\max_i(time_i)}.
\end{aligned}$$

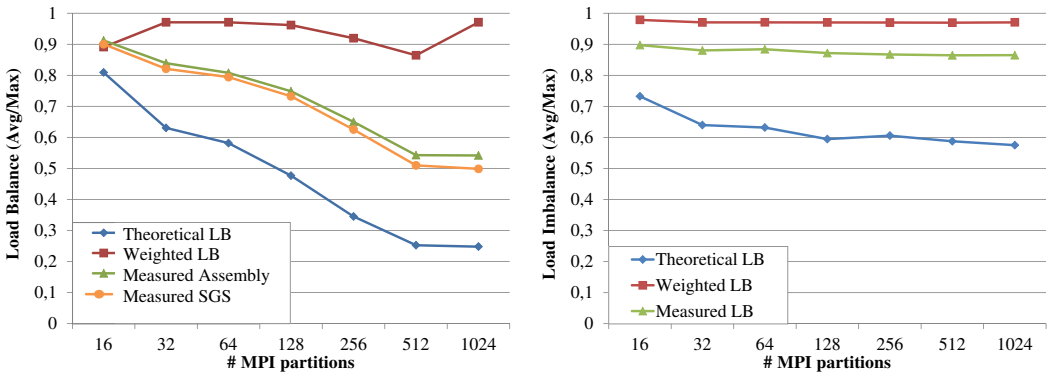


Figure 5: Load Balance for the Respiratory system (Left) and Iter (Right) simulations.

Figure 5 shows the load balance measured for the two use cases presented in the previous section, on the left hand side, the vacuum vessel, and on the right hand the respiratory system. The X axis represents the number of MPI partitions used for each simulation. The different values have been computed using the formulas presented above and the *Measured LB* has been measured from an execution of the simulation and averaged over 10 time steps.

We first observe that METIS provides a fairly good load balance based on the heuristic provided for both meshes (weighted LB), even though an imbalance up to 14% is observed for 512 partitions in the respiratory system. We can say that the theoretical load balance achieved by METIS depends on the mesh structure and the number of partitions, more partitions tend to higher load imbalances, we can observe this specially in the case of the respiratory system.

But if we compare this theoretical result with the measured one, we can see a significant difference. In practice the load imbalance increases with the number of partitions, specially for the Respiratory system, for both the assembly and for the subgrid scale element loops (which are quite similar). We conclude that the number of Gauss points is absolutely not a good measure of the work per element. Indeed, the measured load balance follows the non-weighted load balance. On the contrary, the load balance of the Iter simulation seems to follow the weighted load balance. This means that the same heuristic cannot be used to obtain a well balanced partition of different problems.

Finally, let us take a look at typical partitions and traces. Figures 6(a) and 7(a) shows some statistics of the partitions for the fluid and solid problems using 256 CPUs. The nodes are placed at the centers of gravity of the MPI subdomains, while the edges represent neighboring relations. We observe that in the case of the respiratory system, METIS happens partitions some MPI subdomains into non connected parts (identified by the arrow). We also observe that one subdomain has much more elements than the others (identified with an arrow in the middle figure). This is the subdomain located at front of the face (see Figure 1), which is exclusively composed of tetrahedra. Tetrahedra have less Gauss points and thus METIS admits more elements than in average.

The associated traces are shown in Figures 6(b) and 7(b). In the case of the Respirartory system, we can easily identify the subdomains responsible for the load imbalance, near the bottom of the trace. These are the subdomains mentioned previously, in the front face region. This is because the element weight based on the Gauss points given to METIS is not a good metric.

In the case of the Iter simulation, we can observe that we have some subdomains with much less work than others. Once more, this indicates that the weights based on Gauss points is not a good heuristic for load, although it affects much less the load balance than for the fluid simulation.

4.2 DLB library

Load imbalance is a concern that has been targeted since the beginning of parallel programming. In the literature, we can see that it has been attacked from very different points of view (data partition, data redistribution, resource migration, etc.).

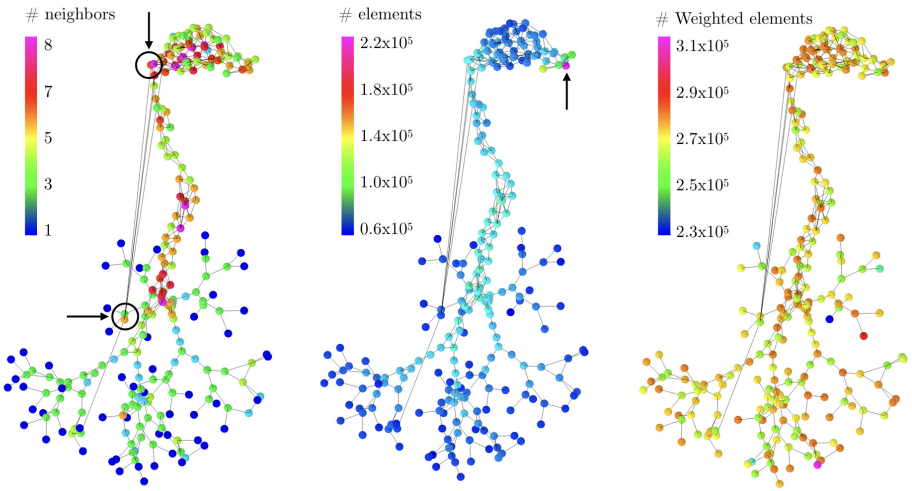
In this case we are using METIS to partition the mesh and obtain a balanced distribution among MPI tasks. But as we have seen in the previous section, the actual load balance obtained is far from optimal. There are several reasons for this, the geometry of the mesh and the weight of the elements given to METIS.

Additionally, the algorithm or the physics (or both together) could produce very strong work imbalance by increasing the computing needs locally (i.e. particle concentrations (14), solid mechanics fracture, shock in compressible flows, etc.) For these reasons, we opt for a dynamic approach applied at runtime, with no need for an a-priori imbalance analysis.

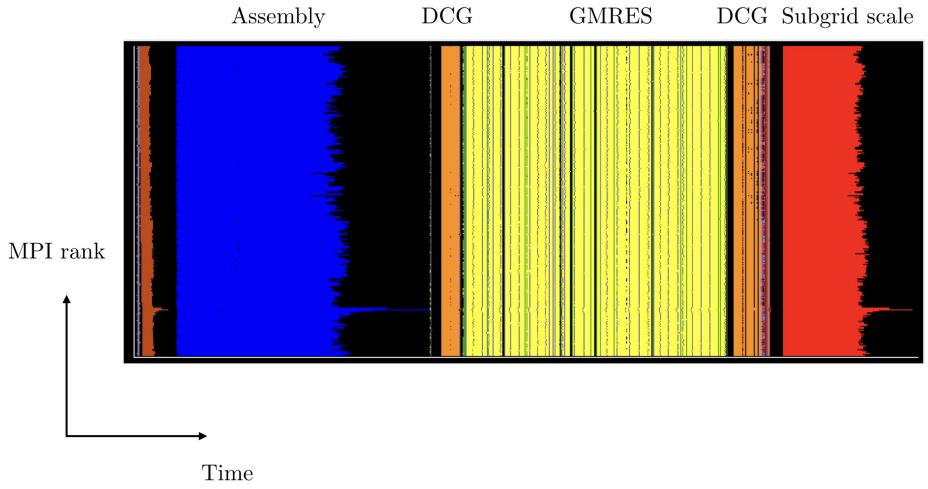
In this work we will use DLB (11) (Dynamic Load Balancing Library). The DLB library aims at balancing MPI applications using a second level of parallelism (i.e. Hybrid parallelization MPI+OpenMP). Currently, the implemented modules balance hybrid MPI + OpenMP and MPI + OmpSs applications, where MPI is the outer level of parallelism and OpenMP or OmpSs are the inner ones.

An important feature of the DLB library is that a runtime interposition technique is used to intercept MPI calls. With this technique we do not need to modify the application, the DLB library is loaded dynamically when running the application to load balance the execution.

The DLB library will reassign the computational resources (i.e. cores) of an MPI process waiting in an MPI blocking call, to another MPI process running on the same node that it is still doing computation.



(a) Partitioning.

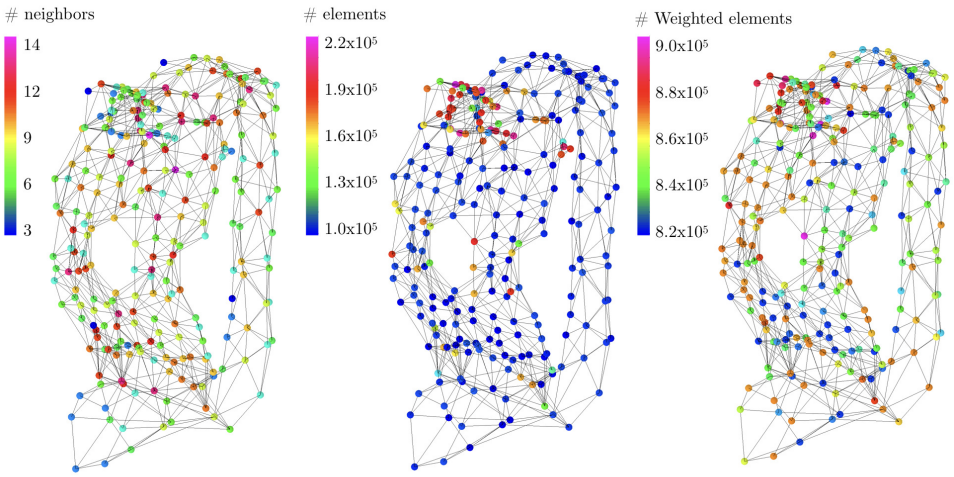


(b) Trace.

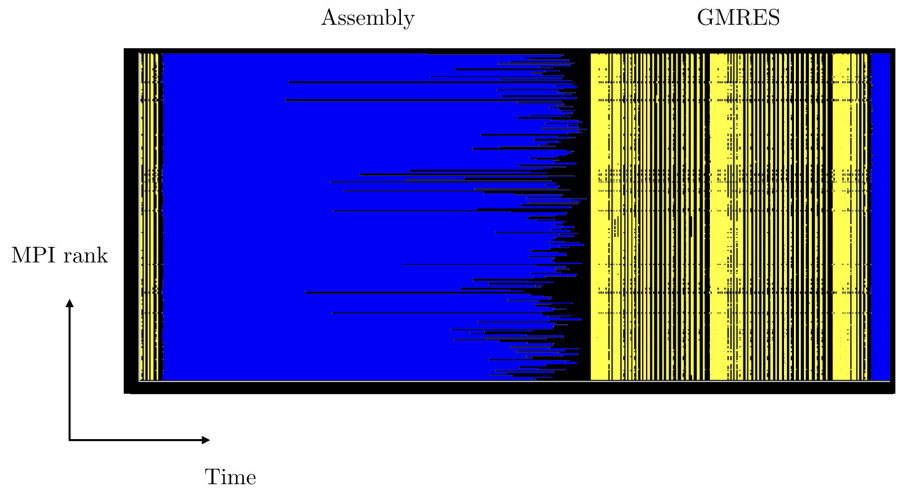
Figure 6: Respiratory system simulaiton on 256 CPUs.

Figure 8 illustrates the load balancing process. In the example, the application is running on a node with 4 cores. Two MPI tasks are started on the same node, and each MPI task spawns 2 OpenMP threads (represented by the wavy lines). Eventually, an MPI blocking operation (in green) synchronizes the execution. Regarding the assembly process, the wavy lines represent the element loops and the MPI call represents the first MPI call in the iterative solvers (namely the initial residual of the algebraic system).

On the one hand, Figure 8 (Left) shows the behavior of an unbalanced application where the excessive work of the threads running on core3 and core4 delays the execution of the MPI call. On the other side, Figure 8 (Right) shows the execution of the same application with the DLB library. We can see that when the MPI task 1 gets into the blocking call it will lend its two cores to the MPI task 2. The second MPI task will use the newly acquired cores and will be able to run with 4 threads. This will allow to finish the remaining computation faster. When the MPI task 1 gets out of the blocking call it retrieves its cores from the MPI task 2 and the execution continues with a core equipartition, until another blocking call is met.



(a) Partitioning.



(b) Trace.

Figure 7: Iter simulation on 256 CPUs.

The fact that DLB relies on the shared memory to load balance the different MPI tasks means that it needs to run more than one MPI task per node. In current many-cores architectures this is the normal trend

The dynamic load balancing algorithm illustrated previously relies on the OpenMP parallelization. One important characteristic of this strategy is that *not* the whole application needs to be fully parallelized, as the second level of parallelism can be introduced only for load balancing purposes, in the main imbalanced loops of the code.

5 Performance Evaluation

5.1 Environment and Methodology

All the experiments presented in Section 2 have been executed on MareNostrum 3 supercomputer. Each node of MareNostrum 3 is composed of two Intel Xeon processors (E5-2670), each of this