Computer-Aided Design 85 (2017) 53–67

Contents lists available at ScienceDirect

Computer-Aided Design

journal homepage: www.elsevier.com/locate/cad

Unique cavity-based operator and hierarchical domain partitioning for fast parallel generation of anisotropic meshes☆

A. Loseille *, F. Alauzet, V. Menier

INRIA Saclay Ile-de-France, Gamma3 Project, 1 rue Honoré d'Estienne d'Orves 91126 Palaiseau, France

ARTICLE INFO

Keywords:
Anisotropic mesh adaptation
Cavity-based primitives
Out-of-core
Parallel meshing
Domain partitioning
Coarse-grained parallelization

ABSTRACT

We devise a strategy in order to generate large-size adapted tetrahedral anisotropic meshes, having $O(10^8–10^9)$ elements, as required in many fields of application in scientific computing. We target moderate scale parallel computational resources as typically found in R&D units where the number of cores ranges in $10^2–10^3$. Both distributed and shared memory architectures are handled. Our strategy is based on typical domain splitting algorithm where the initial mesh is split into parts that are then meshed in parallel while the fictitious boundaries between parts are kept unchanged. Then we iterate the procedure to adapt previously unmodified parts of the domain, *i.e.*, the interface mesh. Both the volume and the surface meshes are adapted simultaneously and the efficiency of the method is independent of the complexity of the geometry. The originality of the method relies on (i) a metric-based static load-balancing, (ii) hierarchical mesh partitioning techniques to (re)split the (complex) interfaces meshes, (iii) a fast, robust and generic sequential cavity-based mesh modification kernel. In order to generate large-size meshes, out-of-core storing of completed parts is used to reduce the memory footprint. We show that we are able to generate (uniform, isotropic and anisotropic) meshes with more than 1 billion tetrahedra in less than 20 minutes on 120 cores. Examples from Computational Fluid Dynamics (CFD) simulations are also discussed.

1. Introduction

Complex numerical simulations (turbulence, noise propagation, etc.) may require billions of degrees of freedom to get a high-fidelity prediction of the physical phenomena. To fit this need, many numerical platforms (numerical solver, solution visualization) have been developed for parallel architectures (distributed or shared-memory). Although few simulations are performed on thousands of processors, recent studies show that many relevant R&D applications run on a daily basis on smaller architectures targeting less than 256 cores [1,2]. In the computational pipeline, mesh generation or adaptation is a critical point as the existence of a mesh (especially with complex geometries) is the necessary condition to start a simulation. In addition, the mesh generation CPU time should be low enough in comparison with the solver CPU time to be actually used in practice. In this paper, we aim at designing an efficient parallel adaptive mesh generation strategy. We target to generate adapted meshes composed of a billion elements in less than 20min on 120 cores. The parallelization of the meshing/remeshing step is a complex problem because it encompasses the following issues: domain partitioning, load balancing, robust surface and volume mesh adaptation.

Parallel mesh generation has been an active field of research [3–6]. Two main frames of parallelization exist: coarse-grained [7,8,3], and fined-grained [6,9–11]. Fine-grained parallelization requires to implement directly in parallel all the mesh modification operators at the lowest level: insertion, collapse, swap etc. This usually implies the use of specific data structures to handle distributed dynamic meshes, especially for adaptive procedures [12]. The second approach consists in the use of a bigger set of operators in parallel. Most of the time a complete sequential mesh generator or mesh optimizer is used. Both approaches have been also extended to adaptive frameworks, see [7,8] for the coarse-grained approach and [9] for the fine-grained. In this paper, we follow the coarse-grained parallelization in an adaptive context within the metric-based framework. In particular, we address the following issues.

Surface–volume problematic. When considering the coarse-grained strategy, parallel mesh generators or parallel local remeshers generally adapt either the surface or the volume mesh. In [13,8,3], the initial fine surface mesh is unchanged during the parallel meshing

☆ This paper has been recommended for acceptance by Chennai Guest Editor.

* Corresponding author.
E-mail address: adrien.loseille@inria.fr (A. Loseille).

http://dx.doi.org/10.1016/j.cad.2016.09.008
0010-4485/

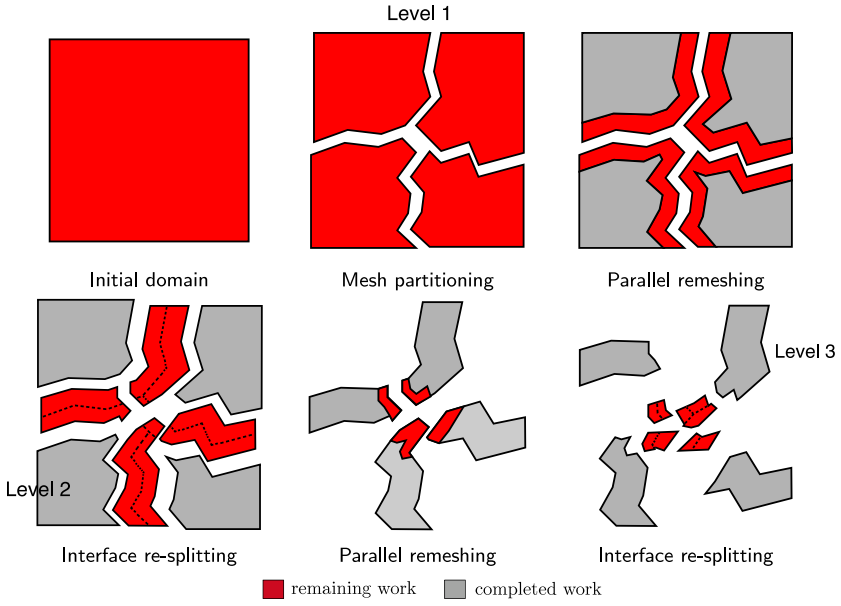

Fig. 1. Sketch of the parallel process remeshing. The red-colored parts represent the part of the domain that remains to be adapted while the gray-colored parts are the final adapted parts that can be stored to disk. 3 levels are depicted. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

process. If this consideration works well for uniform or isotropic meshes, it turns out that it is mandatory to generate the volume and the surface meshes simultaneously when anisotropic meshes are considered. Indeed, the set of methods that have demonstrated a good efficiency and reliability to mesh a given complex surface mesh: advancing front method [14,15], constraint global Delaunay [16–18] or a combination of both [19] are mostly susceptible to fail when an anisotropic surface mesh is provided on input. The frontal methods generally do not succeed to close the front, while the Delaunay-based methods will generally fail during the boundary recovery phase. Consequently, being able to adapt the surface and the volume into a single thread is necessary to gain in robustness [20]. However, adapting both the surface and the volume meshes at the same time implies additional complexity for the load balancing as the costs of the volume or surface operators differ.

Domain partitioning. Domain partitioning is a critical task as each partition should represent an equal level of work [21]. Graph-based techniques [22] tend to minimize the size of the cuts to reduce the communication cost. However, this cost functional is not relevant for adaptative mesh generation, especially if the coarse-grained approach is used where there is no communication at the interface during the remeshing step. For adaptive mesh generation, the cost function (to define the cuts) is related to the amount of work (number of collapse, insertion, optimization steps, etc.) needed on each partition. This becomes even more critical for anisotropic mesh adaptation where refinements have a large variation in the computational domain. Estimating accurately this work *a priori* is also challenging for anisotropic meshing as it strongly depends on the properties of the serial meshing algorithm. Additional developments of graph-based methods are then necessary to work in the anisotropic framework [8]. Domain partitioning represents also one of the main parallel overhead of the method. In particular, general purpose graph-partitioners cannot take into account the different geometrical properties of the sub-domains to be partitioned. Indeed, splitting the initial domain is completely different from splitting the interface mesh, see Fig. 1 (top left and bottom left) and Fig. 4. In addition, there exist additional requirements that the partitioning algorithm should ensure in order to ease the work of the serial mesh generator. The first requirement is to ensure that the parts are connected and the second requirement is to ensure that the number of non-manifold (surface) edges is as minimal as possible.

Partition remeshing. This is the core component of the coarse-grained parallelization. The overall efficiency of the approach is bounded by the limits of the sequential mesh generator. One limit is the speed of the sequential remesher that defines the optimal potential speed in parallel. In addition, as for as the partitioning of interfaces, meshing a partition is different from meshing a standard complete domain. Indeed, the boundary of the partition usually features non-manifold components and constrained boundary faces. In particular, it is necessary to ensure that the speed and robustness of the remesher is guaranteed on interface meshes. Consequently, when a black-box mesher is used, achieving good performances in parallel may be difficult. Additional developments and cares are usually needed in the serial meshing algorithm [7]. In addition, estimating the cost of the mesh modification operators of the serial meshing algorithm is required to estimate the required work and to drive accordingly the partitioning algorithm.

Out-of-core. Out-of-core meshing was originally designed to store the parts of the mesh that were completed on disk to reduce the memory footprint [13]. Despite the high increase of memory (in terms of storage and speeds with solid state drives), coupling out-of-core meshing with a parallel strategy may be advantageously used. On multi-socket shared memory machines (with 40–200 cores), if the memory used by a thread is bigger than the memory of a socket, then the memory exchange between neighboring sockets implies a huge overhead on the sequential time (when running the procedure with one thread only). For instance, on a DELL PowerEdge R900 with 4 Intel Xeon E7 sockets with 10-cores with 1 Tb of RAM, we observe that the speed of the serial meshing algorithm is twice slower when the used memory exceeds 256 Gb, *i.e.*, the RAM of one socket. This drawback is even more critical on NUMA architectures.

Our approach. Our procedure is based on standard coarse-grained parallel strategies [8,23,3] where the initial domain is split into several sub-domains that are meshed in parallel. A sketch of the procedure is depicted in Fig. 1. Note that we can decompose the procedure by level, where 3 levels are shown on the simple example of Fig. 1. At the first level, the initial domain is split and

considered for adaptation. From the set of constrained faces (at interface) of the previous level, a new volume mesh (domain) is deduced for the next level. This new domain is then split and adapted in parallel. This process is then applied until convergence. We will show that a maximum of 5 levels is needed to complete the process.

To address the mesh partitioning issues, we define a hierarchical partitioning technique that depends on the current level of the procedure. For the first level, a fast and parallel Hilbert based partitioning is used while a breadth-first search with restart algorithm is used at the next level. This allows us to take advantage of the geometry of the mesh at the interface in order to minimize and reduce the number of constrained faces at each step. For each level, specific partition corrections are designed to guarantee that each final partition is connected while remaining well-balanced. To handle non uniform refinements (in terms of sizes and directions), a metric-based static load balancing formula is used to *a priori* equilibrate the work on each sub-domain.

For the serial meshing algorithm, we use a unique anisotropic cavity-based operator to perform the mesh adaptation. We show that standard operators (collapse, insertion, swaps) can be recast within this cavity framework. The main advantage is that we obtain a constant speed whatever the considered operator. This feature allows us to derive an accurate metric-based work that is easily deduced from the input metric-field and input mesh only. In addition, we design automatic cavity corrections to ensure that the underlying mesh modification operation will be actually performed. This drastically reduces the number of rejected operations (leading to an negative volume) and thus improves the CPU of the meshing step. This point is particularly useful for surface mesh adaptation when modifying the volume simultaneously as many rejections may occur in the volume mesh when standard operators are used.

Once the remeshing of a sub-domain is completed, two additional sub-domains are created. The first one represents an interface mesh composed of elements that need additional refinement. The second one is the completed part that is stored to disk. To define the interface mesh, mesh modification operators (insertion/collapse) are simulated in order to enlarge the initial interface mesh to perform a quality remeshing in the subsequent levels.

Current state-of-art parallel mesh generation approaches [7] for unstructured (and adapted) meshes require thousands of cores (4092–200 000 cores) to generate meshes with a billion elements. Our scope is to make this size of mesh affordable on cheaper parallel architectures ($\approx$ 120 cores) with an acceptable runtime for an adaptive design process (less than 20 min).

The paper is organized as follows. In Section 2, we describe the anisotropic cavity-based serial meshing algorithm. In Section 3, we describe the domain partitioning methods and the metric-based load balancing. Numerical experiments are used to illustrate the optimality of the partition technique used for the different levels. The procedure to generate a new domain from the set of constrained faces at previous level is explained. Finally, in Section 4, we give some numerical examples of mesh adaptation.

2. Sequential mesh generator and cavity-based operator

We describe the meshing software AMG that is used as the sequential mesh modification operator. It natively allows us to take into account constrained boundary faces (as those defining the interface between parts) and handles non manifold geometries. In addition, the volume and the surface meshes are adapted simultaneously in order to keep a valid 3D mesh throughout the entire process. This guarantees the robustness of the complete remeshing step.

2.1. Metric-based and unit-mesh concept

AMG is a generic purpose adaptive mesh generator dealing with 2D, 3D and surface mesh generation. AMG belongs to the class of metric-based mesh generator [24–28] which aims at generating a unit mesh with respect to a prescribed metric field $\mathcal{M}$. A mesh is said to be unit when composed of almost unit-length edges and unit-volume element. The length of an edge AB in $\mathcal{M}$ is evaluated with:

$$\ell_{\mathcal{M}}(AB) = \int_0^1 \sqrt{{}^t AB\, \mathcal{M}((1-t)A + tB)\, AB}\, \mathrm{d}t,$$

while the volume is given by $|K|_{\mathcal{M}} = \sqrt{\det \mathcal{M}}\, |K|$, where $|K|$ is the Euclidean volume of K. From a practical point of view, the volume and length requirements are combined into a quality function defined by:

$$Q_{\mathcal{M}}(K) = \frac{36}{3^{\frac{1}{3}}} \frac{\sum_{i=1}^{6} \ell^2_{\mathcal{M}}(\mathbf{e}_i)}{|K|^{\frac{2}{3}}_{\mathcal{M}}} \in [1, \infty],$$

where $\{\mathbf{e}_i\}_{i=1,6}$ are the edges of element K. A perfect element has a quality of 1. The generation of a unit mesh is decomposed into two steps that are described below:

1. Generate a unit-mesh : The mesh modification operators are used in the goal to optimize the length of the edges in $\mathcal{M}$.
2. Optimization: The mesh modification operators are used to improve the quality $Q_{\mathcal{M}}$.

2.2. Generation of a unit mesh

The scope of this step is to obtain a mesh where the lengths of the edges are in $[\frac{1}{\sqrt{2}}, \sqrt{2}]$. This procedure is composed of 3 phases: collapse, creation of new points, anisotropic filtering and insertion.

Collapse. For this phase, an iterative procedure is used. The current mesh is iteratively scanned and while there exists an edge with a length lower than $1/\sqrt{2}$, try to collapse the edge. At the end of the process, all edges must have a length greater than $1/\sqrt{2}$. During all the following phases, the collapse is never used again.

Creation of edges. In this phase, we create the set of points that would be needed to decompose all long edges in segments having a length close to one in the metric. As for the collapse, the algorithm consists in scanning the current mesh and while there exists an edge with a length greater than 2, create one or multiple points. During this phase, the topology of the mesh is kept unchanged so that the points are not inserted. Indeed, neighboring edges can generate similar points or points very close to each other, so it is important to filter out the points that are too close (in the metric). For that, we define the anisotropic filtering.

Anisotropic filtering and insertion. In this phase, the length between the points created in the previous phase is checked and only a subset of points are inserted. For the filtering, we use an octree of points. Each octant can contain up to 10 points before being subdivided. Initially, the octree contains the surface points and the volume points remaining from the collapse phase. To validate the insertion of a point, we first check the distances between every points that are in the octant containing the point to be inserted. If no rejection occurs, then the current octant is intersected with the bounding box of the metric. All the intersected octants are checked starting from the octants closer to the point being inserted. Then, each point that is accepted for insertion is inserted in the octree along with its metric. At the end of the filtering, whatever the connectivity generated by the insertor the edges will have an admissible length (as the length was checked in every direction with the octree). This property prevents us from having to perform additional collapses that is the most costly operator.

2.3. Optimization of the mesh

During the phase, only the topology of the mesh is modified by using edges or faces swaps, see [29] for the details of these operators. The only constraint is to make sure that the quality in the metric is strictly improved at each application of a swap.

For all the previous phases, standard operators can be used [29]. However, in our approach, we use the cavity-based version for each of them. We show in this next section that this choice speeds up the CPU time of the remeshing by minimizing the number of rejections of each operator. We show also that one call of a cavity operator may be equivalent to a combination of several simple operators due to the use of cavity correction algorithm.

2.4. Surface approximation

During the generation of the adapted surface mesh, it is necessary to maintain a sufficient level of fidelity of the geometry. To do so, two different options, a discrete and continuous, are used to control the geometry approximation.

For the discrete approach, a fine and fixed discrete mesh is used as a background support. The surface points along with their normals at the points are computed on this support. In addition, a surface-based metric is recovered and intersected with the current computational metric in order to control the required level of fidelity. We refer to [20] for a detailed description of the process. When a continuous description of the geometry is provided, as a CAD data, the newly created surface points are projected onto the continuous position by querying the CAD. In what follows, we use the discrete approach for all the numerical examples.

2.5. Cavity-based operators

A complete mesh generation or mesh adaptation process usually requires a large number of operators: Delaunay insertion, edge-face-element point insertion, edge collapse, point smoothing, face/edge swaps, etc. Independently of the complexity of the geometry, the more operators are involved in a remeshing process, the less robust the process may become. Consequently, the multiplication of operators implies additional difficulties in maintaining, improving and parallelizing a code. In [30], a unique cavity-based operator has been introduced which embeds all the aforementioned operators. This unique operator is used at each step of the process for surface and volume remeshing.

The cavity-based operator is inspired from incremental Delaunay methods [31–33] where the current mesh $\mathcal{H}_k$ is modified iteratively through sequences of point insertion. The insertion of a point P can be written:

$$\mathcal{H}_{k+1} = \mathcal{H}_k - \mathcal{C}_P + \mathcal{B}_P, \tag{1}$$

where, for the Delaunay insertion, the cavity $\mathcal{C}_P$ is the set of elements of $\mathcal{H}_k$ such that P is contained in their circumsphere and $\mathcal{B}_P$ is the ball of P, i.e., the set of new elements having P as vertex. These elements are created by connecting P to the set of the boundary faces of $\mathcal{C}_P$.

In [30], each meshing operator is equivalent to a node (re)insertion inside a cavity. For each operator, we just have to define judiciously which node P to (re)insert and which set of volume and surface elements will form the cavity $\mathcal{C}$ where point P will be reconnected with $\mathcal{R}_P$:

$$\mathcal{H}_{k+1} = \mathcal{H}_k - \mathcal{C} + \mathcal{R}_P. \tag{2}$$

Note that if $\mathcal{H}_k$ is a valid mesh (only composed of elements of positive volume) then $\mathcal{H}_{k+1}$ will be valid if and only if $\mathcal{C}$ is connected (through internal faces of tetrahedron) and $\mathcal{R}_P$ generates only valid elements. In Fig. 2, we list the initial cavity choice along with the point to (re)insert for the collapse, insertion and swap. As it, the cavity operators are equivalent to their standard counterparts. However, using the cavity formalism allows to easily modify the cavity to enforce automatically the operator. The cavity enlargement correction is one example of such correction and is given in algorithm 1. The basic idea is to enlarge the cavity to make sure that C_P becomes valid. To illustrate this feature, we consider a simple 2D example where we want to relocate a point A to a new position A_{new}, see Fig. 3. Given the initial configuration, we see that a collapse, then a swap and finally a point-smoothing is needed to actually move A to A_{new}. To do this, 4 volumes are computed for the collapse, 2 for the swap and finally 7 for the point-smoothing. Then, if we use the cavity version, the initial cavity has 2 negatives faces (in red in Fig. 3, bottom). Using the cavity enlargement algorithm 1, a valid cavity is found in 3 enlargement iterations. To build the final C_P, 4 volumes are computed with the initial cavity, 4 for the first iterations, 2 for the second and 2 for the third. The cost of using the cavity moving is 12 volumes computations whereas 13 volumes are needed with the standard operators. The most interesting feature is that the cavity operator creates automatically the combination of simple operators without the need to know in practice the sequence. From a practical point of view, only one operator is used for the meshing operators.

The use of the previous cavity-based operators allows us to design a remeshing algorithm that has a linear complexity in time with respect to the required work (sum of the number of collapses and insertions). On a typical laptop computer Intel Core I7 at 2.7 GHz, the speed for the (cavity-based) collapse is around 20 000 points removed per second and the speed for the insertion is also around 20 000 points or equivalently 120 000 elements inserted per second. Both estimates hold in an anisotropic context [34].

Algorithm 1 Cavity enlargement for (re)insertion of P

Volume Part:
For each K in $\mathcal{C}_P$

  For each face $[A, B, C]$ such that $P \notin [A, B, C]$:
    if $volume(A, B, C, P) < 0$, then
      if P is a surface point then **reject**
      else add neighboring tetrahedron to $\mathcal{C}_P$
      endif
    endif
  EndFor

EndFor

if $\mathcal{C}_P$ is modified goto **Volume Part**.

2.6. Additional features of the serial remesher

In addition to the core algorithms to adapt a mesh to a metric, several components have to handle with care to make sure the linear complexity of the cavity operator is maintained whatever the size of the mesh is. We list some of them in this section.

As the mesh is dynamically modified, the data structure needs to be compressed in order to remove destroyed entities (points, elements). It is important to verify that the complexity to compress the mesh depends on the number of destroyed entities (and not on the number of current entities). More generally, a rule of thumb is then to make sure that each loop on entities used in the remeshing phase has a complexity proportional to the required work rather than a complexity proportional to the size of the current mesh. In many cases, it is sufficient to replace the initial complete loop over the entities with a loop on a front of entities; this front being updated dynamically during the underlying process. If these

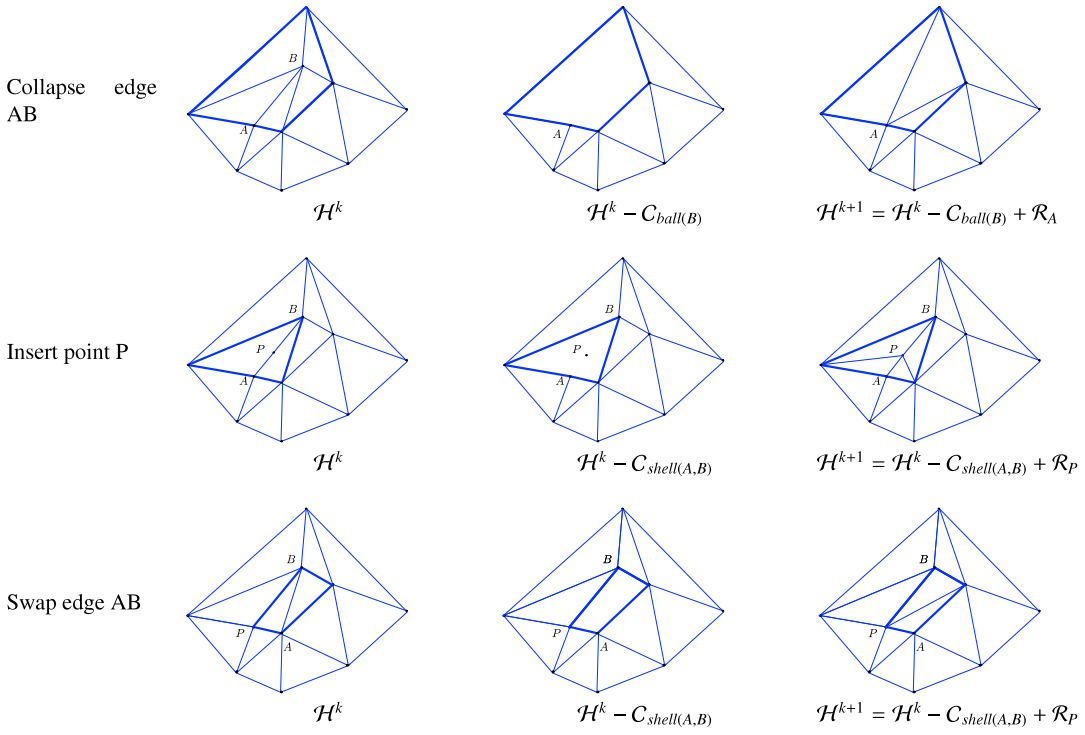

Fig. 2. Three 2D meshing operators reinterpreted as a cavity-based operator with an appropriate choice of the point to (re)insert and cavity to remesh. From top to bottom, the collapse, insertion and swap operators.

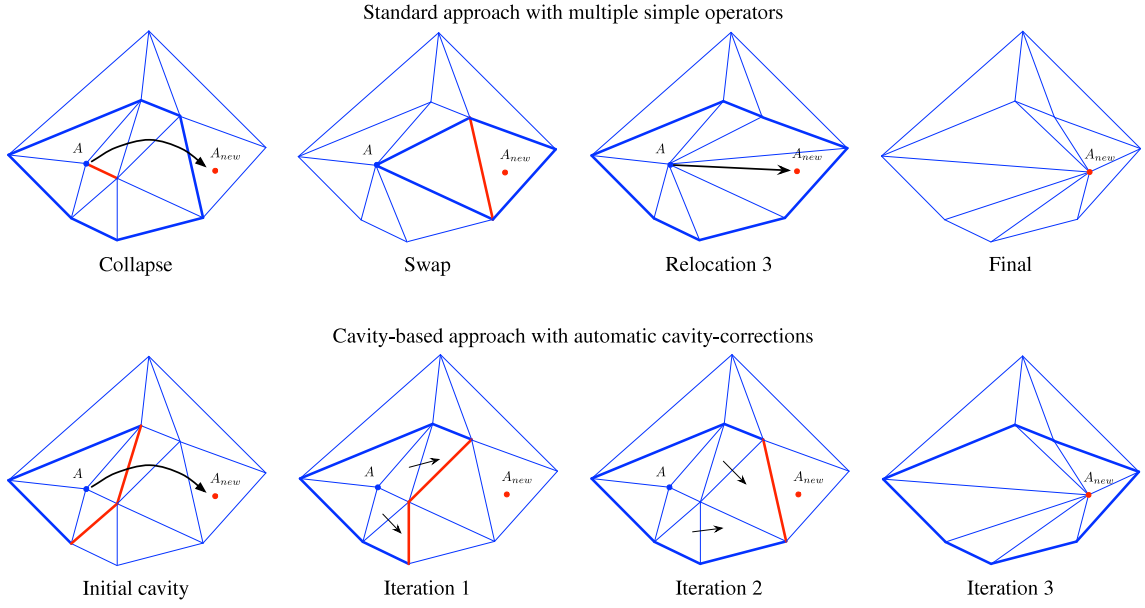

Fig. 3. Illustration of the relocation of point A to new position A_{new}. Top, if standard operators are used, the following sequence has to be applied: collapse, swap, relocation. Bottom, with the cavity enlargement, 3 enlargement iterations are needed to perform the operation.

modifications have a little impact on medium size meshes, they appear to be a drastic bottleneck for very large meshes or when the process is run in parallel with a *a priori* metric-based static load balancing.

3. Hierarchical Domain partitioning

In the context of parallel remeshing, the domain partitioning method must be fast, low memory, able to handle domains with many connected components and effective to balance the remeshing work. Moreover, we should have efficient partitioning method for several hierarchical levels of partitions. In particular, the method should be such that the size of the interface between the partitions converges toward zero when the partitioning level increases in order to have a converging parallel algorithm. More precisely, we first – level 1 – split the domain volume. Level 2, we split the interface of the partitions of level 1; the interface volume domain being formed by all the elements having at least one vertex sharing several sub-domains. Level 3, we split the interface of the partitions of level 2, and so on. The different levels for the hierarchical decomposition of a cubic domain into 32 partitions are shown in Fig. 4. In this example, we observe that the domain topology varies

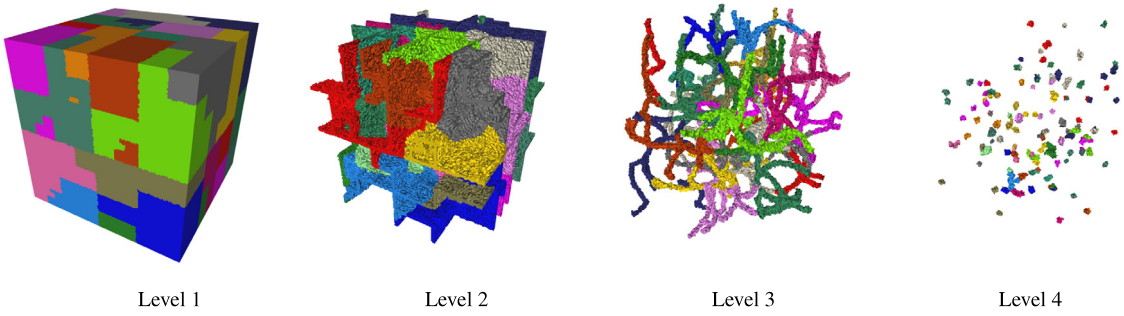

Fig. 4. Hierarchical partitioning into 32 sub-domains of a cubic domain for a constant work per element. From left to right, levels 1, 2, 3 and 4 of partitioning. We observe that the domain topology varies drastically with the level, and the size of interface meshes decreases at each level and converges toward zero.

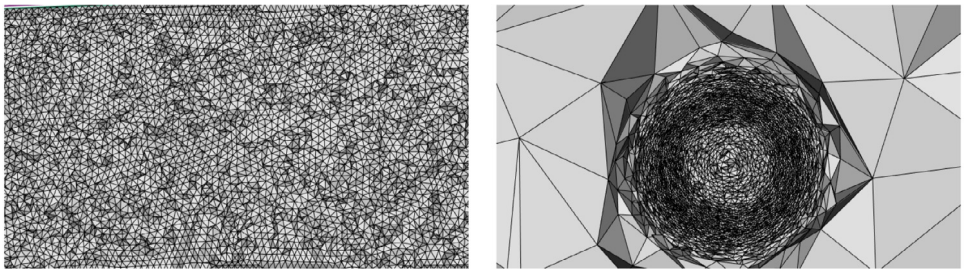

Fig. 5. Blast example to assess the hierarchical partitioning techniques and anisotropic work prediction: initial uniform mesh (left) and final adapted mesh (right). The serial adaptation takes 36 s.

drastically with the level. We also observe that the size of interface meshes decreases at each level and converges toward zero.

To emphasize the choices made in this work for the hierarchical partitioning method, in this section, we will always compare the proposed methodology on the same example. The considered example is the adaptation of an initial uniform mesh to the numerical solution (at one time step) of a spherical blast problem. We will refer to it as the blast example. The initial uniform mesh is composed of 821 373 vertices and 4 767 431 tetrahedra while the adapted resulting mesh is composed of 82 418 vertices and 511 998 tetrahedra. These two meshes are shown in Fig. 5. It takes 36 s to generate that adapted mesh in serial.

This example is very interesting because the positions of the spherical shock waves imply that a large number of insertions and collapses are needed while being non uniformly distributed in the domain. Hence, the amount of work and the kind of meshing operation vary drastically in the domain.

3.1. Element work evaluation

An effective domain partitioning strategy should balance the work which is going to be done by the local remesher on each partition, knowing that each partition is meshed independently. Thus, there is no communication between partitions and the partition interfaces are constrained (they are not remeshed). The work to be performed depends on the used mesh operations (insertion, collapse, swap, smoothing), the given metric field $(\mathcal{M}(\mathbf{x}))_{\mathbf{x}\in\Omega}$, and the initial mesh $\mathcal{H}$ natural metric field $(\mathcal{M}_{\mathcal{H}}(\mathbf{x}))_{\mathbf{x}\in\Omega}$, where Ω is the domain to be remeshed. Indeed, if the initial mesh already satisfies the given metric, *i.e.*, $(\mathcal{M}_{\mathcal{H}}(\mathbf{x}))_{\mathbf{x}\in\Omega} = (\mathcal{M}(\mathbf{x}))_{\mathbf{x}\in\Omega}$, then nothing has to be done. It is convenient to define the work at the elements because it is the elements that are uniquely distributed to each partition. We recall that the natural metric of an element K is the unique metric tensor $\mathcal{M}_K$ such that all edges of K are of length 1 for $\mathcal{M}_K$. It is obtained by solving a simple linear system [35]. And, metric field $(\mathcal{M}_{\mathcal{H}}(\mathbf{x}))_{\mathbf{x}\in\Omega}$ is the union of the element metrics $\mathcal{M}_K$.

To define the remesher work per element and the total work, we use a continuous approach – similarly to the error estimate theory [35,36] – because the initial mesh and the targeted final adapted mesh are represented by their respective metric fields $(\mathcal{M}_{\mathcal{H}}(\mathbf{x}))_{\mathbf{x}\in\Omega}$ and $(\mathcal{M}(\mathbf{x}))_{\mathbf{x}\in\Omega}$. We recall that, for a given metric field $(\mathcal{M}(\mathbf{x}))_{\mathbf{x}\in\Omega}$, also called continuous mesh, the point-wise mesh density is given by $d_{\mathcal{M}}(\mathbf{x}) = \sqrt{\det \mathcal{M}(\mathbf{x})}$ and the continuous mesh complexity is

$$\mathcal{N} = \int_{\mathbf{x}\in\Omega} \sqrt{\det \mathcal{M}(\mathbf{x})}\, \mathrm{d}\mathbf{x} = \int_{\mathbf{x}\in\Omega} d_{\mathcal{M}}(\mathbf{x})\, \mathrm{d}\mathbf{x}.$$

The continuous mesh complexity $\mathcal{N}$ is the dual of the mesh number of vertices N in the continuous mesh framework [36]. As the work to be done by the local remesher is clearly proportional to the mesh size, in the continuous approach, the work is thus proportional to the integral of the mesh density.

We propose to define the work per element, the total work of the remesher being the sum of the mesh element works. Each element K of initial mesh $\mathcal{H}$ is supplied with its natural metric $\mathcal{M}_K$ and the given metric $\mathcal{M}$. The given metric at the element is obtained by averaging (in the log-Euclidean framework) the metric at its vertices:

$$\mathcal{M} = \exp\left(\sum_{i=1}^{k} \frac{1}{k}\, \ln(\mathcal{M}(\mathbf{x}_i))\right) \quad \text{where the } \mathbf{x}_i \text{ are the vertices of } K,$$

as the given metric field $(\mathcal{M}(\mathbf{x}))_{\mathbf{x}\in\Omega}$ is generally point-wise. We also compute the intersection [37] of these two metrics: $\mathcal{M}_\cap = \mathcal{M}\cap\mathcal{M}_K$. We denote by $d_{\mathcal{M}}$, d_K and $d_\cap$ the density of metric $\mathcal{M}$, $\mathcal{M}_K$ and $\mathcal{M}_\cap$, respectively. Now, we analyze the work depending on specific remeshing case and, then, we propose a work for the general case.

Insertion case. Assuming the initial mesh is only going to be refined (first case in Fig. 6), the density of points to be inserted is $d_{\mathcal{M}\setminus\mathcal{M}_K}$ then the work per element is

$$wrk(K) = \alpha\,|K|\,(d_{\mathcal{M}} - d_K) = \alpha\,|K|\,(d_\cap - d_K),$$

where $|K|$ is the element volume, α is the coefficient defining the cost of the insertion operator, and in that case we have $\mathcal{M}_\cap = \mathcal{M}$. Note that the metric density is inversely proportional to the ellipse

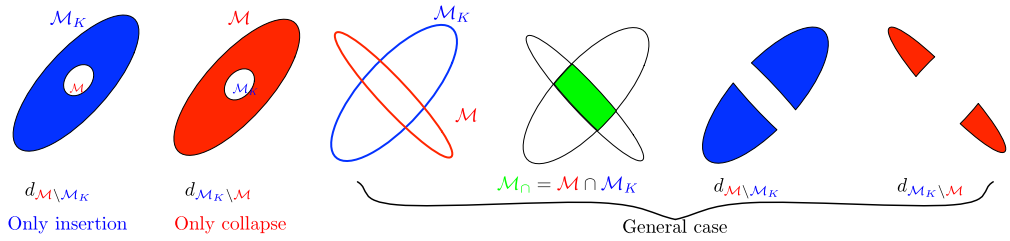

Fig. 6. Illustration of the continuous work in the metric-based framework. Blue regions represent insertion work. Red regions represent collapse work. Note that the metric density d is inversely proportional to the ellipse area in the graphic representation. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

area in the graphic representation. If we assume that $d_{\mathcal{M}} = d_\cap \gg d_K$, then $wrk(K) \approx \alpha\,|K| d_{\mathcal{M}}$. Thus, the total work will be:

$$wrk(\mathcal{H}) = \sum_{K\in\mathcal{H}} \alpha\,|K|\,d_{\mathcal{M}} = \alpha \int_{\mathbf{x}\in\Omega} d_{\mathcal{M}}(\mathbf{x})\,\mathrm{d}\mathbf{x} = \alpha\,\mathcal{N}^{new},$$

which is logical because in that case the work is effectively proportional to the size of the final mesh. Let us give a concrete example in three dimensions. We have a uniform isotropic mesh of size $\mathcal{N}$ with length size h thus $\mathcal{M}_K = h^{-2}\,\mathcal{I}_3$ for all elements K. We want to generate a mesh at $h/2$, thus $\mathcal{M} = 4\,h^{-2}\,\mathcal{I}_3$. This new mesh will have a size of $8\,\mathcal{N}$. The work per element is $wrk(K) = \alpha\,|K|\,7\,h^{-3}$ leading to a total work of $wrk(\mathcal{H}) = \alpha\,7\,\mathcal{N}$. This result is the expected answer as $7\,\mathcal{N}$ vertices will be inserted to generate the new mesh.

Collapse case. Assuming the initial mesh is only going to be collapsed (second case in Fig. 6), the density of points to be removed is $d_{\mathcal{M}_K\setminus\mathcal{M}}$ then the work per element is

$$wrk(K) = \beta\,|K|\,(d_K - d_{\mathcal{M}}) = \beta\,|K|\,(d_\cap - d_{\mathcal{M}}),$$

where β is the coefficient defining the cost of the collapse operator, and in that case we have $\mathcal{M}_\cap = \mathcal{M}_K$. If we assume that $d_K = d_\cap \gg d_{\mathcal{M}}$, then $wrk(K) \approx \beta\,|K| d_K$. Thus, the total work will be:

$$wrk(\mathcal{H}) = \sum_{K\in\mathcal{H}} \beta\,|K|\,d_K = \beta \int_{\mathbf{x}\in\Omega} d_{\mathcal{H}}(\mathbf{x})\,\mathrm{d}\mathbf{x} = \alpha\,\mathcal{N},$$

which is logical because in that case the work is effectively proportional to the size of the initial mesh.

Optimization case. At the end of the remeshing process, an optimization phase is applied to improve mesh quality. Thus, the work due to the optimization is proportional to the size of the final mesh:

$$wrk(K) = \gamma\,|K|\,d_{\mathcal{M}},$$

where γ is the coefficient defining the cost of the optimization operator.

General case. In the general case, we may have to insert and collapse locally because the anisotropic information may be contradictory depending on the considered direction. For instance, two metrics may have the same density but opposite directions hence in one direction we should refine the mesh and in the other direction we should coarsen the mesh (third case in Fig. 6). The general case gathers all the previous cases and the density of the intersected metric represents the common ground. The work per element is:

$$wrk(K) = |K|\,\left(\alpha\,(d_\cap - d_K) + \beta\,(d_\cap - d_{\mathcal{M}}) + \gamma\,d_{\mathcal{M}}\right).$$

The constants depend on the underlying remesher properties. In our case, the local remeshing strategy uses a unique cavity operator for all mesh modifications (see Section 2), therefore all mesh modifications have exactly the same cost. We thus set: $\alpha = \beta = \gamma = 1$, the work per element becomes:

$$wrk(K) = |K|\,\left(2\,d_\cap - d_K - d_{\mathcal{M}} + d_{\mathcal{M}}\right), \tag{3}$$

or if no optimization is performed $\gamma = 0$, thus:

$$wrk(K) = |K|\,\left(2\,d_\cap - d_K - d_{\mathcal{M}}\right).$$

Remark 3.1. If we are in the case where only refinement is performed, at a first order approximation, we can assume that $d_{\mathcal{M}} = d_\cap \gg d_K$ and the work per element is:

$$wrk(K) \approx |K|\,(2\,d_\cap + (\gamma - 1)d_{\mathcal{M}}) = |K|\,(1+\gamma)\,d_{\mathcal{M}}.$$

This means that the work without optimization is proportional to the work with optimization. In that case, it changes nothing to take into account the work due to the optimization to define the work per element.

If we are in a case where only collapse is performed, at a first order approximation, we can assume that $d_K = d_\cap \gg d_{\mathcal{M}}$, thus the element work is:

$$wrk(K) \approx |K|\,(2\,d_\cap - d_K) \;=\; |K|\,d_K.$$

Thus, the work without optimization is equal to the work with optimization and it changes nothing to take into account the work due to the optimization to define the work per element.

This simple analysis shows why it is very important to choose simple example that involves the general case (and not only coarsening or refinement) to validate the obtained work function. Indeed, in the general case, it is primordial to take into account the cost of the mesh optimization to have well-balanced partitions. This is why the blast example has been chosen.

Surface work. When the surface is remeshed with the volume, the work to remesh the surface is added to the work to remesh the volume. Therefore, the same formula is used to estimate the work per face by taking into account the surface metric and the surface density. Then, the work of each face is added to the work of the element sharing that face.

Blast example. We first illustrate on the blast example why it is crucial to take into account the metric of the initial mesh and the given metric to define the remesher work (as we have done in this section), and not to use – as frequently observed – only the given metric (which is valid if only insertion is done, see previous remark). The results obtained on 8 processors with the anisotropic work given by Relation (3) and the work only based on the density of the given metric $\mathcal{M}$ are given in Table 1. In both cases, the Hilbert partitioning method is used. The anisotropic work leads to a quasi-uniform CPU time for each of the 8 partitions whatever the number of collapses or insertions. On the contrary, considering only the given metric density to compute the work to balance the partitions leads to a completely non uniform CPU time. In fact, we have balanced the number of insertions on each partition ($\approx$8600 per partitions) which is expected if only $d_{\mathcal{M}}$ is used. But, the collapses have not been taken into account. It results in a larger maximal time and a large overhead in waiting for the end of the remeshing of the most time-consuming partition.

Table 1
Remeshing statistics on 8 processors for the blast problem test case (Fig. 5). The total CPU time, the number of collapses and the number of insertions for each partition are presented. Top and bottom tables show the statistics for a mesh partitioning based on the anisotropic work (Relation (3)) and on the given metric density-based work, respectively. In both cases, the Hilbert partitioning method is used.

Anisotropic work	Statistics for each partition								Waiting time
CPU time (s)	5.1	4.8	4.7	4.6	4.6	4.7	4.6	4.6	0.5
Number of collapses	78 844	95 615	96 457	81 437	91 787	85 754	96 020	83 763	
Number of insertions	14 029	4952	4299	12 667	7121	10 058	4586	11 103	
Density-based work	**Statistics for each partition**								**Waiting time**
CPU time (s)	3.9	4.2	6.7	2.4	7.0	0.9	9.6	3.4	8.7
Number of collapses	63 047	73 404	133 349	33 764	138 473	2801	205 677	59 218	
Number of insertions	8626	8530	8526	8614	8605	8604	8585	8598	

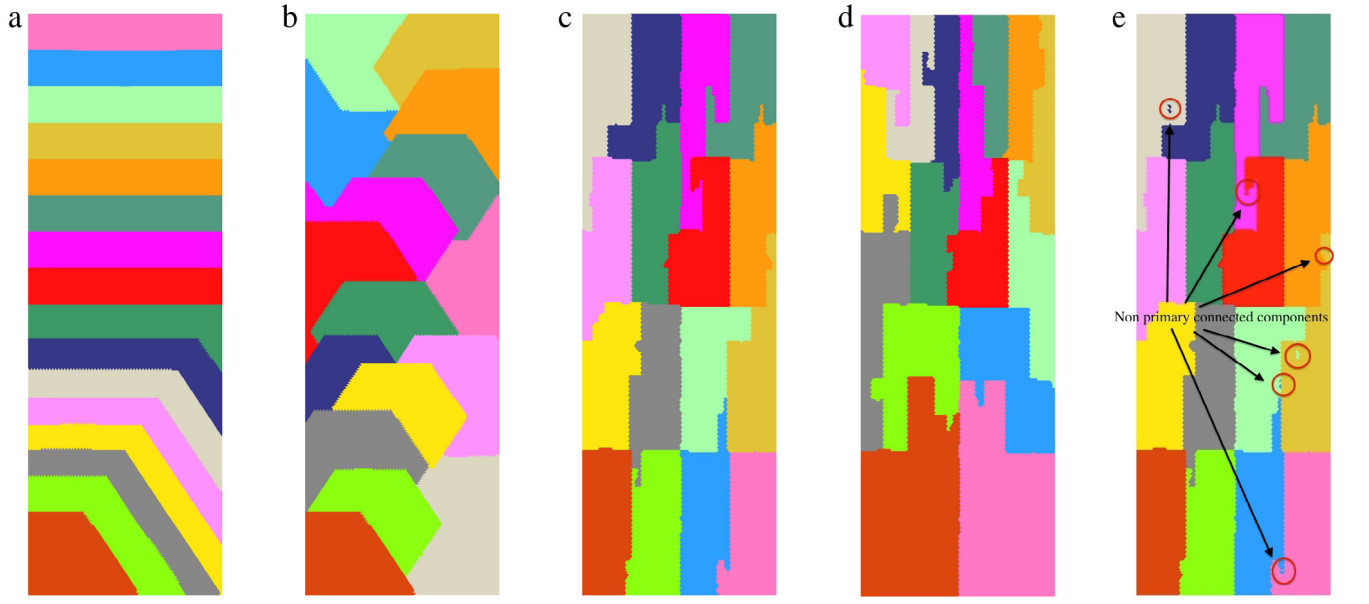

Fig. 7. Partitioning into 16 sub-domains of a – level 1 – rectangular domain for a constant work per element with the BFS (a), BFS with restart (b) and Hilbert (c) methods. Picture (d) shows the Hilbert partitioning with a linear work function (the work per element increase with y) which has to be compared with picture (c) for a constant work per element. Picture (e) shows the Hilbert partitioning before the connected components correction. Several isolated connected components appear. The result after the correction is shown in picture (c).

3.2. Partitioning methods

Before using any of the partitioning methods presented below, the mesh vertices are first renumbered using a Hilbert space filling curve based reordering [38]. A Hilbert index (the position on the curve) is associated with each vertex according to its position in space. This operation has a linear complexity and is straightforward to parallelize as there is no dependency. Then, the vertices renumbering is deduced from the vertices Hilbert indices. Vertices are sorted using the standard C-library `quicksort`.

The domain partitioning problem can be viewed as a renumbering problem of the elements. In that case, the first partition is composed of the elements from 1 to N_1 such that the sum of these elements work is equal to the total mesh work divided by the total number of partitions. Then, the second partition is composed of the elements from $N_1 + 1$ to N_2 such that the sum of these elements work is equal to the total mesh work divided by the total number of partitions. And so on. The difference between all strategies lies on the choice of the renumbering strategy. Note that, for efficiency purposes, the elements are not explicitly reordered but they are only assigned an index or a partition index on the fly.

Now, assuming the vertices have been renumbered, we propose three methods to split the mesh: Hilbert based, breadth-first search (BFS) or frontal approach, and BFS with restart.

Hilbert partitioning. It consists in ordering the elements list according to the element minimal vertex index. In other words, we first list the elements sharing vertex 1 (the elements ball of vertex 1), then we list the elements sharing vertex 2 (the elements ball of vertex 2 not already assigned), etc. This splitting of the domain is based on the Hilbert renumbering of the vertices. For level 1 domain (initial domain splitting), it results in block-shaped partitions which are very convenient for sub-domain remeshing (see Fig. 7(c)). But, it may lead to partitions with several connected components on complex geometry due to domain holes not seen by the Hilbert curve. For level 2 or more domains, it is not effective because it will reproduce the previous level result and thus it will not gather the interfaces of different sub-domains. The size of the interface mesh will not decrease at each level.

Breadth-first search (BFS) partitioning. This method starts from an element root – generally, element 1 – and adds the neighbor elements of the root first, *i.e.*, the neighbors are the next elements in the renumbered list. Then, we move to the next level of neighbors, in other words, we add the neighbors of the neighbors not already assigned. And so on. This splitting of the domain progresses by front. Indeed, each time an element is assigned, its non-assigned neighbors are added to a stack. The elements in this stack represent the current front. For level 1 domain, it results in layered partitions which contain only one connected component (see Fig. 7(a)) except the last one(s) which could be multi-connected. But, it results in several unconnected interface domains at level 2 which is not appropriate here. For level 2 or more domains, this method is able to gather the interfaces of

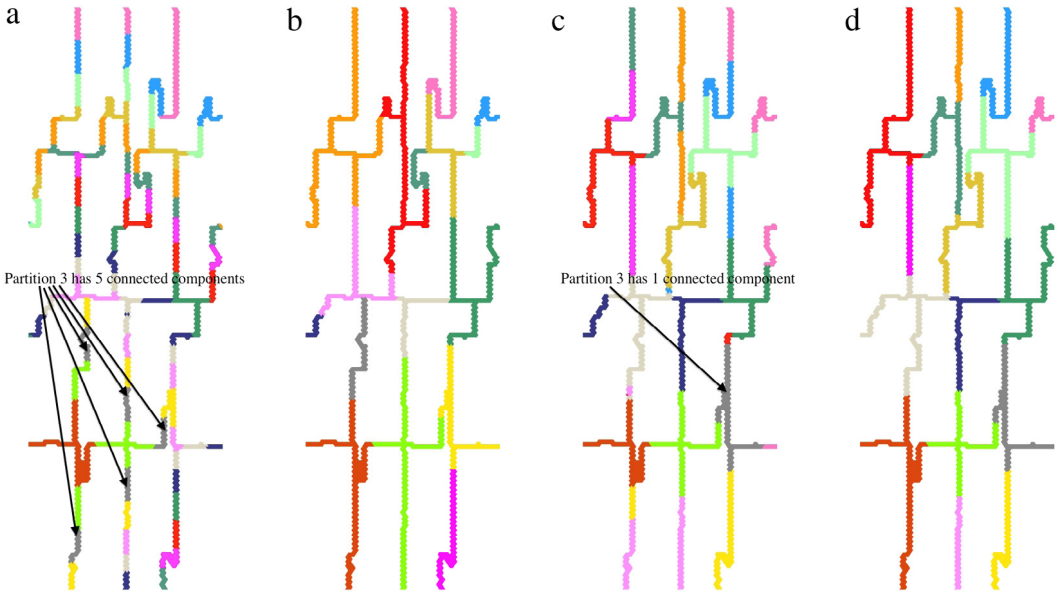

Fig. 8. Partitioning into 16 sub-domains of an – level 2 – interface mesh of a rectangular domain for a constant work per element. The interface mesh results from the Hilbert partitioning of the level 1 domain. Partitions obtained with the BFS method before and after correction are shown in pictures (a) and (b), respectively. Many connected components are created for each partition (a) due to the bifurcations resulting in an unbalanced domain decomposition after correction (b). Partitions obtained with the BFS method before and after correction are shown in pictures (c) and (d), respectively. Just a few isolated small connected components are created leading to a balanced domain decomposition after correction.

different sub-domains but, as the stack is always growing, the number of connected components grows each time a bifurcation is encountered (see Fig. 8(a)). This leads to very unbalanced sub-domains after the connected component correction presented below. Therefore, we prefer to consider the modified BFS method described hereafter.

Breadth-first search (BFS) with restart partitioning. In the previous BFS algorithm, the splitting progresses by front, and generally this front grows until it reaches the diameter of the domain. During the splitting of interface domains (level 2 or more), this is problematic because the resulting partitions are multi-connected, cf. Fig. 8(a). One easy way to solve this issue is to reset the stack each time we deal with a new partition. The root of the new partition is the first element of the present stack, all the other elements are removed from the stack. For level 1 domain, it results in more circular (spherical) partitions (see Fig. 7(b)). For level 2 or more domains, this method is able to gather the interfaces of different sub-domains and also to obtain one connected component for each partition expect the last one(s), see Fig. 8(c). Therefore, this method is very efficient to deal with the level 2 or higher domains of the hierarchical partitioning. Moreover, we observe in Fig. 4 that the size of the partition interface meshes reduces at each level.

Connected components correction. As the interface is constrained and not remeshed, the number of connected components per sub-domain should be minimized to maximize the work done by the remeshing strategy. In other words, each partition should have only one connected component if it is possible. All elements of the same connected component are linked by at least a neighboring face.

After the domain splitting, a correction is applied to merge isolated connected components, see Fig. 7(e). First, for each sub-domain, the number of connected components is computed and the primary connected component (the one with the most work) of each partition is flagged. Second, we compute the neighboring connected components of each non-primary connected component. Then, iteratively, we merge each non-primary connected component with a neighboring primary connected component. If several choices occur, we pick the primary connected component with the smallest work. The impact of this correction is illustrated in Fig. 7 from (e) to (c).

Remark 3.2. We may end-up with non-manifold (but connected) partitions, *i.e.*, elements are linked by a vertex or an edge. As the local remeshing strategy is able to take care of such configurations, no correction is applied. Otherwise, such configurations should be detected and corrected.

Blast example. We compare, on the blast example, the efficiency of the proposed partitioning methods on the level 1 and level 2 meshes.

For the level 1 partitioning, we first analyze the size of the interface provided by each method. Indeed, the best method should minimize the number of interface faces as these faces are constraint and prevent the remesher to work. The result is given in Table 2. Clearly, the Hilbert method minimizes the number of interface faces (by a factor two w.r.t. the BFS method), and it also somehow balances the number of interface faces between the partitions. This should have an impact on the efficiency. Second, we analyze the CPU time and the number of operations done with each method, the result is presented in Table 3. The Hilbert method achieves the lowest maximal CPU time and minimizes the waiting time between the partitions. It leads to a quasi-uniform CPU times for each of the 8 partitions whatever the number of collapses or insertions.

Now, we analyze the results for the level 2 partitioning where the level 1 partitioning has been done with the Hilbert method. The size of the interface provided by each method is given in Table 4. As previously mentioned, the Hilbert method does not reduce the interface on the subsequent level and it is thus not appropriate. We notice that the BFS with restart is clearly better than the BFS, and the number of interface faces drops from 153 542 on the level 1 to 25 682 on the level 2. As regards the efficiency, the BFS with restart achieves the best CPU time and minimizes the waiting time, see Table 5.

In conclusion. The following strategy is thus proposed for the hierarchical mesh partitioning. The Hilbert method is used to partition the initial volume mesh, *i.e.,* the level 1 partitioning,

Table 2
Interface size between level 1 partitions on 8 processors for the blast problem test case (Fig. 5): the number of interface faces – which are constrained faces for the remesher – for each level 1 partition are given. The three partitioning methods are compared.

Method	Number of interface faces for level 1 partitions								Total	Max difference
Hilbert	18 043	19 408	21 870	16 849	19 580	19 268	22 256	16 270	153 542	5 407
BFS	14 550	34 115	41 550	45 883	50 885	56 579	47 350	17 758	308 670	42 029
BFS restart	14 532	26 615	23 710	11 344	37 407	33 773	30 290	23 683	201 334	26 063

Table 3
Remeshing statistics on 8 processors for the blast problem test case (Fig. 5) on the level 1 mesh. The total CPU time, the number of collapses and the number of insertions for each partition are presented. From top to bottom, tables show the statistics for the Hilbert, the BFS, and the BFS with restart methods.

Hilbert method	Statistics for each level 1 partition								Waiting time
CPU time (s)	5.1	4.8	4.7	4.6	4.6	4.7	4.6	4.6	0.5
Number of collapses	78 844	95 615	96 457	81 437	91 787	85 754	96 020	83 763	
Number of insertions	14 029	4952	4299	12 667	7121	10 058	4586	11 103	
BFS method	Statistics for each level 1 partition								Waiting time
CPU time (s)	4.6	5.0	5.1	5.4	5.9	5.9	5.5	4.8	1.3
Number of collapses	106 496	100 976	50 380	42 465	73 217	98 160	97 906	102 908	
Number of insertions	1	582	25 726	29 666	12 360	0	0	1	
BFS restart method	Statistics for each level 1 partition								Waiting time
CPU time (s)	5.2	5.5	5.4	4.6	5.3	5.0	5.4	4.8	0.9
Number of collapses	106 508	103 028	93 868	3702	92 508	103 594	101 370	94 954	
Number of insertions	3	515	5474	54 978	7331	1	0	0	

Table 4
Interface size between level 2 partitions on 8 processors for the blast problem test case (Fig. 5): the number of interface faces – which are constraint faces for the remesher – for each level 1 partition are given. The three partitioning methods are compared.

Method	Number of interface faces for level 2 partitions								Total	Max difference
Hilbert	18 934	21 583	22 378	16 956	19 843	19 766	23 040	16 300	158 794	6084
BFS	3013	5476	6129	8422	9492	8563	4737	910	46 742	8582
BFS restart	3002	4585	5288	5602	2036	1390	1408	2373	25 682	4212

Table 5
Remeshing statistics on 8 processors for the blast problem test case (Fig. 5) on the level 2 mesh. The total CPU time, the number of collapses and the number of insertions for each partition are presented. From top to bottom, tables show the statistics for the Hilbert, the BFS, and the BFS with restart methods.

Hilbert method	Statistics for each level 2 partition								Waiting time
CPU time (s)	0.97	0.96	0.95	0.90	0.88	0.94	0.99	0.90	0.11
Number of collapses	0	264	47	84	16	8	118	27	
Number of insertions	2488	1617	1789	2335	2000	2117	1506	2145	
BFS method	Statistics for each level 2 partition								Waiting time
CPU time (s)	0.44	0.54	0.60	0.67	0.70	0.68	0.71	0.41	0.30
Number of collapses	10 074	5817	1860	1568	1659	1357	5026	0	
Number of insertions	0	852	2516	2356	2281	2466	2801	2529	
BFS restart method	Statistics for each level 2 partition								Waiting time
CPU time (s)	0.47	0.57	0.54	0.51	0.55	0.52	0.53	0.53	0.10
Number of collapses	10 074	6122	4763	9160	603	2	0	76	
Number of insertions	0	876	1398	17	3099	3616	3950	3100	

as it is very efficient, it ends-up with well balanced partitions, and it minimizes the size of the interface. Moreover, the Hilbert partitioning of the initial volume mesh provides a nice continuous network-shaped domain (see Figs. 4 and 7) for the interface mesh which is very convenient for the subsequent partitioning levels.

Then, the BFS with restart method is used for the level 2 or more partitioning because it minimizes the number of created connected components and thus minimizes the size of the interface. Moreover, it allows the size of the interface to be reduced at each level, thus having a converging hierarchical mesh partitioning.

3.3. Partitions balancing optimization by migration

On some complex configurations, the connected components correction leads to unbalanced partitions because the size of the non-primary connected components is non-negligible. The partitions balancing is then optimized by migrating elements between neighboring partitions. To this end, each element is analyzed and if it has a neighboring element on another partition which has a lower total work, then this element is migrated to that partition. This optimization phase improves the partitions balancing but may create new connected components for partitions, thus the correction presented in the previous section is again applied.

This process is iterated until the partitions are well-balanced with respect to the given work.

3.4. Efficiency of the method

The presented domain partitioning methods minimize the memory requirement as the data structures they use are only: the elements list, the elements' neighbors list, the elements' partitions indices list and a stack.

They are efficient in CPU because the elements assignment to a sub-domain is done in one loop over the elements. Then, the

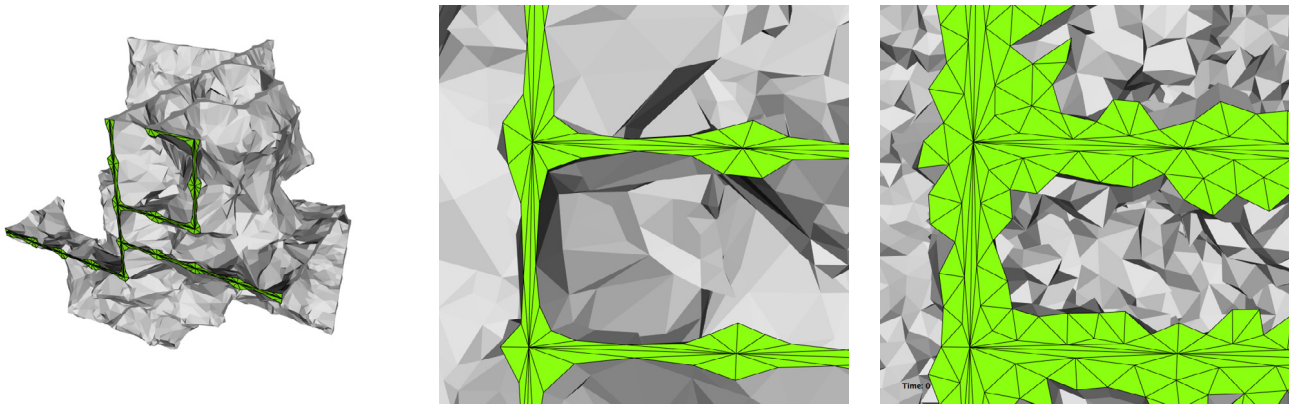

Fig. 9. Definition of the interface mesh on a cube example. Global view of interface geometry (left), interface defined by the balls of the vertices belonging to the interface (middle), and interface mesh defined by predicting the set of elements needed to perform the remeshing operation (insertion or collapse).

connected components correction requires only a few loops over the partitions. For instance, let us consider the domain partitioning of a cubic domain composed of 10 million tetrahedra into 64 sub-domains. In serial on an Intel Core i7 at 2.7 GHz, it takes 0.52, 0.24 and 0.24 s for the partitioning of the level 1, 2 and 3 domains, respectively, where the Hilbert partitioning has been used for level 1 domain and the BFS with restart partitioning has been used for the level 2 and 3 domains.

3.5. Definition of the interface mesh

During the remeshing phase, the set of elements that surrounds the constrained faces (defining the boundary of the current partition) is not adapted. It is then necessary to define a set of elements that needs to be adapted at the next iteration (or level). An initial choice consists in introducing all the elements having at least one node on the boundary of the interface. This choice is illustrated in Fig. 9 (middle). Despite its simplicity, this choice is appropriate only when the size of the elements of the interface is of the same order as the size imposed elsewhere. However, when large size variation occurs, additional elements need to be part of the new interface volume. Optimally, a sufficient number of elements needs to be added to make sure that underlying local modification will be possible at the next level. An automatic way to find these elements is to add the relevant set of elements of the cavity [34] for each operator. Two situations occur. When an edge of the interface is too short, a collapse will be needed at the next level. Consequently, for all interfaces sharing this edge, the ball of the two end-points edges is added. When an edge is too long, a point will be inserted at the next level, consequently, the Delaunay cavity of the mid-point edge is added. Note that these modifications are done in parallel at the end of the remeshing step, thus limiting the overhead of this correction. The modification of the set of elements defining the interface is illustrated in Fig. 9 where a cube domain is refined from a size h to $h/4$. If we select only the balls of the interface vertices, then the remeshing process is much more constrained, see Fig. 9 (middle). Including additional elements based on the cavity defining the relevant mesh modification operator (collapse or insertion) gives additional room to the mesh generator to perform a quality modification 9 (right).

4. Numerical results

Several examples are illustrated in this section. For each case, the parallel mesh generation converges in 5 iterations. The number of cores is chosen to ensure that at least 100 000 tetrahedra per core will be inserted. Consequently, the number of cores is reduced when the remaining work decreases. All the examples are run on a cluster composed of 40 nodes with 48 Gb of memory, composed of two-chip Intel Xeon X56650 with 12 cores. A high-speed internal network InfiniBand (40 Gb/s) connects these nodes. For each example, we report the complete CPU time including the IOs, the initial partitioning and gathering along with the parallel remeshing time. To evaluate the overheads and efficiency of the parallel method, serial meshes are generated on a super-node having 1 Tb of memory.

Vortical flows on the F117 geometry. This case is part of an unsteady adaptive simulation to accurately capture vortices generated by the delta-shaped wings of the F117 geometry, see Fig. 10. The final adapted mesh of the simulation is depicted in Fig. 10. The final adapted mesh is composed of 83 752 358 vertices, 539 658 triangles and 520 073 940 tetrahedra. The initial background mesh is composed of 1 619 947 vertices, 45 740 triangles and 9 710 771 tetrahedra. The complete CPU time (including initial domain partitioning and final gathering) is 12 min on 120 cores. The parallel mesh adaptation of the process takes 8 min 50 s. The parallel procedure inserts 10^6 vertices/min or equivalently 6.10^6 tetrahedra/min, see Table 7. The maximal memory used per core is 1.25 Gb. The same example on 480 cores is reported in Table 8, the CPU for the parallel mesh generation part is 3 min 36 s while the maximal memory used per core is 0.6 Gb. The speed up from 120 to 480 cores is limited to 1.5 (4 optimally), this is due to the large increase of the interfaces in the mesh, see Table 9 (left). For a partition, the typical time to create its interface mesh using the anisotropic Delaunay cavity is less than 10% of the meshing time.

The quality of the mesh along with the histogram of the length of edges is reported in Table 6. More than 94% of edges have a unit length computed in the metric. The serial CPU time for this case is 4 h with an equivalent distribution of lengths and qualities. The mesh generated in serial is composed of 81 920 668 vertices, 510 052 triangles and 493 948 440 tetrahedra. For the complete process, we obtain a speed-up of 20 on 120 cores.

Blast simulation on the tower bridge

The example consists in computing a blast propagation on the London Tower Bridge. The geometry is the 23rd IMR meshing contest geometry, see Fig. 11. The initial mesh is composed of 3 837 269 vertices 477 852 triangles and 22 782 603 tetrahedra while the final mesh is composed of 174 628 779 vertices 4 860 384 triangles and 1 090 324 952 tetrahedra. From the previous example, the surface geometry and mesh adaptation is much more complex as many shock waves impact the bridge. The time to generate the adapted mesh on 120 cores is 22 min 30 s

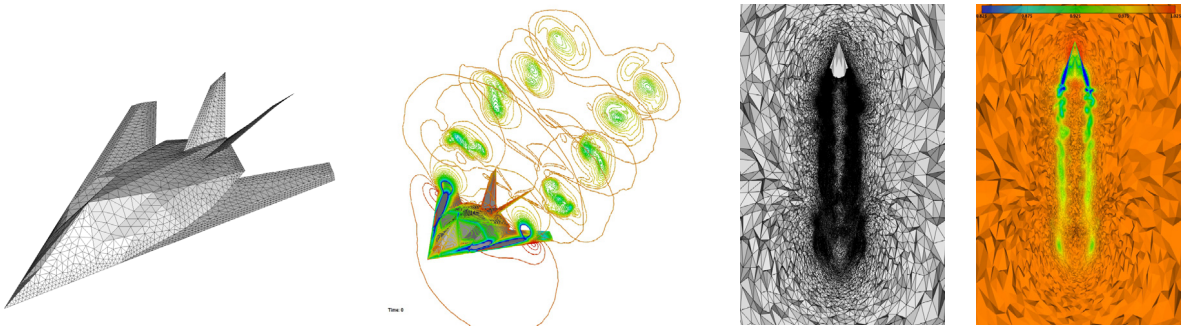

Fig. 10. F117 test case. From left to right, geometry of the f117 aircraft, representation of the vortical flow, top view of the mesh adapted to the local Mach number and local Mach number iso-values.

Table 6
F117 test case on 120 cores: Distribution of the length of edges in the metric, histograms of the quality of surface and volume elements in the metric.

Distribution of $\ell_{\mathcal{M}}$ for edges				
0.00	$< \ell_{\mathcal{M}} <$	0.20	24 815	0.00%
0.20	$< \ell_{\mathcal{M}} <$	0.50	555 487	0.09%
0.50	$< \ell_{\mathcal{M}} <$	0.71	2 577 068	0.41%
0.71	$< \ell_{\mathcal{M}} <$	0.90	181 600 710	**28.63**%
0.90	$< \ell_{\mathcal{M}} <$	1.11	224 256 106	**35.35**%
1.11	$< \ell_{\mathcal{M}} <$	1.41	195 459 095	**30.81**%
1.41	$< \ell_{\mathcal{M}} <$	2.00	25 649 625	4.04%
2.00	$< \ell_{\mathcal{M}} <$	5.00	4 141 743	0.65%
5.00	$< \ell_{\mathcal{M}}$		68 602	0.01%

Distribution of $Q_{\mathcal{M}}$ for triangles				
1	$< Q_{\mathcal{M}} <$	2	495 271	**99.45**%
2	$< Q_{\mathcal{M}} <$	3	1 557	0.31%
3	$< Q_{\mathcal{M}} <$	4	535	0.11%
4	$< Q_{\mathcal{M}} <$	5	299	0.06%
5	$< Q_{\mathcal{M}} <$	6	152	0.03%
6	$< Q_{\mathcal{M}} <$	7	73	0.01%
7	$< Q_{\mathcal{M}} <$	8	55	0.01%
8	$< Q_{\mathcal{M}} <$	9	24	0.00%
9	$< Q_{\mathcal{M}} <$	10	11	0.00%
10	$< Q_{\mathcal{M}} <$	100	39	0.01%

Distribution of $Q_{\mathcal{M}}$ for tetrahedra				
1	$< Q_{\mathcal{M}} <$	2	467 446 828	**94.15**%
2	$< Q_{\mathcal{M}} <$	3	14 064 980	2.83%
3	$< Q_{\mathcal{M}} <$	4	4 054 558	0.82%
4	$< Q_{\mathcal{M}} <$	5	2 333 665	0.47%
5	$< Q_{\mathcal{M}} <$	6	1 556 518	0.31%
6	$< Q_{\mathcal{M}} <$	7	1 118 945	0.23%
7	$< Q_{\mathcal{M}} <$	8	842 774	0.17%
8	$< Q_{\mathcal{M}} <$	9	657 686	0.13%
9	$< Q_{\mathcal{M}} <$	10	524 793	0.11%
10	$< Q_{\mathcal{M}} <$	100	3 773 750	0.76%
100	$< Q_{\mathcal{M}} <$	1 000	130 948	0.03%
1000	$< Q_{\mathcal{M}} <$	10 000	2 256	0.00%

Table 7
F117 test case on 120 cores. Table gathering the size of the interface, the number of inserted tetrahedra and the CPU time for each iteration.

Iteration	% done	# of tets in interface	# of tets inserted	CPU time (s)	# of cores	elt/sec	elt/sec/core
1	84%	69 195 431	433 495 495	180.8	120	2.410^6	19 980
2	96%	1 692 739	502 706 732	95.0	120	7.210^5	6 071
3	99%	1 231 868	518 850 149	35.9	91	4.610^5	5 068
4	99%	6 459	520 067 586	7.5	7	1.610^5	2 318
5	100%	0	520 073 940	1.7	1	3.710^3	3 737

Table 8
F117 test case on 480 cores. Table gathering the size of the interface, the number of inserted tetrahedra and the CPU time for each iteration.

Iteration	% done	# of tets in interface	# of tets inserted	CPU time (s)	# of cores	elt/sec	elt/sec/core
1	76%	109 269 782	389 476 861	109.9	480	3.510^6	7 383
2	91%	42 836 303	486 695 293	67.0	480	1.410^6	1 440
3	98%	5 567 744	525 073 846	28.1	228	1.310^6	6 011
4	99%	32 292	530 573 260	8.9	30	6.110^5	20 597
5	100%	0	530 605 308	2.3	1	1.410^4	13 933

and 28 min for the total CPU time including the initial splitting, final gathering and IOs. On 480 cores, the time to generate the mesh reduces to 16 min 30 s. The maximal memory used on 120 cores is 1.8 Gb and reduces to 1 Gb on 480 cores. We report in Table 9 (right), 10 and 11, the convergence of the process. This example exemplifies the robustness of this approach with complex geometries.

The quality of the mesh along with the histogram of the length of edges is reported in Table 12. As for as the f117 test case, more than 94% of edges have a unit length computed in the metric. The serial CPU time for this case is 10 h 40 min with an equivalent distribution of length and qualities. The mesh generated in serial is composed of 183 761 201 vertices, 4 572 302 triangles and 1 102 880 450 tetrahedra. For the complete process, we obtain

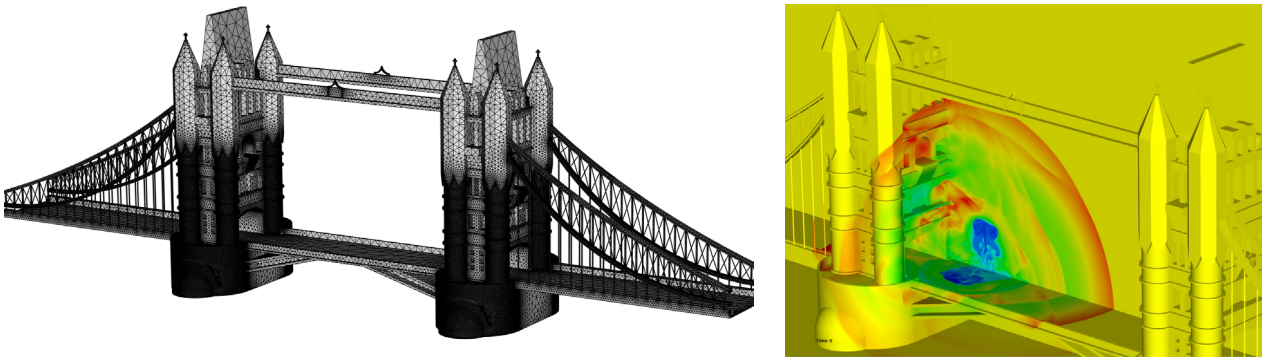

Fig. 11. Tower-bridge test case. Initial mesh and geometry (left) and density iso-values of the blast on an adapted mesh (right).

Table 9
Number of faces at the interfaces at each iteration when running on 120 and 480 cores for the F117 (left) and the tower-bridge (right) test cases.

Iteration	120 cores	480 cores	Iteration	120 cores	480 cores
1	590 038	954 166	1	1 081 246	1 627 846
2	1 711 512	4 306 256	2	2 416 840	5 265 939
3	130 262	589 532	3	132 659	451 355
4	869	4 018	4	488	3 230
5	0	0	5	0	0

a speed-up of 30 on 120 cores. In comparison with the f117 test case, the improvement of the speed-up is related to the serial approach that tends to be less efficient when the size of the meshes increases. The use of the parallel out-of-core approach tends to minimize this memory effect as the complete mesh is never stored on memory.

Landing gear geometry mesh refinement. This geometry is designed for the study of the propagation of the noise generated by a landing gear. This simulation requires large isotropic surface and volume meshes to capture the complex flow field which is used for aeroacoustic analysis. The initial background mesh is composed of 2 658 753 vertices 844 768 triangles and 14 731 068 tetrahedra while the adapted mesh is composed of 184 608 096 vertices 14 431 356 triangles and 1 123 490 929 tetrahedra. The parallel remeshing time is 15 min 18 s and the total CPU time is 24 min 57 s (with the initial splitting and the final gathering). This example illustrates the stability of this strategy when the surface mesh contains most of the refinement. Indeed, the surface mesh is composed of more than 7.2 million vertices and 14.4 million triangles. Table 13 gathers all the data per iteration on this case. The geometry and closer view on the surface mesh are depicted in Fig. 12.

The quality of the mesh along with the histogram of the length of edges is reported in Table 14. More than 95% of edges have a unit length computed in the metric. The serial CPU time for this case is 14 h 5 min with an equivalent distribution of length and qualities. The mesh generated in serial is composed of 182 103 059 vertices, 14 348 710 triangles and 1 077 433 606 tetrahedra. For the complete process, we obtain a speed-up of 33 on 120 cores.

5. Conclusion and future works

An efficient coarse-grained parallel strategy is proposed to generate large-size adaptive meshes. Both uniform, isotropic and anisotropic refinements are handled. The volume and the surface meshes are adapted simultaneously and a valid mesh is kept throughout the process. The parallel resources are used to remove the memory impediment of the serial meshing software. Even if the remeshing is the only part of the process completely done in parallel, we still achieve reasonable CPU times. The CPU time for the meshing part ranges from 15 to 30 min to generate 1 billion tetrahedra adapted meshes. The key components of the process are:

- a fast sequential cavity-based remesher that can handle constrained surfaces and non-manifold geometries during the remeshing,
- specific splitting of the interface mesh ensuring that the number of faces defining the interfaces tends to zero,

Table 10
Tower-bridge test case on 120 cores. Table gathering the size of the interface, the number of inserted tetrahedra and the CPU time for each iteration.

Iteration	% done	# of tets in interface	# of tets inserted	CPU time (s)	# of cores	elt/sec	elt/sec/core
1	84%	89 577 773	919 345 377	577.3	120	1.510^6	13 277
2	95%	14 290 245	1 062 994 802	280.7	120	5.110^5	4 264
3	97%	1 290 855	1 089 035 610	56.3	120	4.610^5	3 854
4	97%	3 636	1 090 321 352	8.0	7	1.610^5	22 959
5	100%	0	1 090 324 952	2.1	1	1.710^3	1 714

Table 11
Tower-bridge test case on 480 cores. Table gathering the size of the interface, the number of inserted tetrahedra and the CPU time for each iteration.

Iteration	% done	# of tets in interface	# of tets inserted	CPU time (s)	# of cores	elt/sec	elt/sec/core
1	79%	193 529 057	922 145 088	255.8	480	3.610^6	7510
2	93%	52 837 674	1 115 428 211	106.7	379	1.810^6	4779
3	96%	4 258 411	1 165 096 167	34.6	282	1.410^6	5090
4	97%	27 095	1 169 283 585	23.0	23	1.810^5	7915
5	100%	0	1 169 310 260	3.9	1	6.810^3	6839

Table 12
Tower bridge test case on 120 cores: Distribution of the length of edges in the metric, histograms of the quality of surface and volume elements in the metric.

Distribution of $\ell_{\mathcal{M}}$ for edges				
0.00	$< \ell_{\mathcal{M}} <$	0.20	120 421	0.01%
0.20	$< \ell_{\mathcal{M}} <$	0.50	1 489 489	0.11%
0.50	$< \ell_{\mathcal{M}} <$	0.71	5 406 707	0.38%
0.71	$< \ell_{\mathcal{M}} <$	0.90	394 284 285	**27.80**%
0.90	$< \ell_{\mathcal{M}} <$	1.11	496 597 258	**35.02**%
1.11	$< \ell_{\mathcal{M}} <$	1.41	444 060 651	**31.31**%
1.41	$< \ell_{\mathcal{M}} <$	2.00	66 636 395	4.70%
2.00	$< \ell_{\mathcal{M}} <$	5.00	9 496 734	0.67%
5.00	$< \ell_{\mathcal{M}}$		110 093	0.01%

Distribution of $Q_{\mathcal{M}}$ for triangles				
1	$< Q_{\mathcal{M}} <$	2	4 519 374	**98.95**%
2	$< Q_{\mathcal{M}} <$	3	19 900	0.44%
3	$< Q_{\mathcal{M}} <$	4	7 661	0.17%
4	$< Q_{\mathcal{M}} <$	5	4 693	0.10%
5	$< Q_{\mathcal{M}} <$	6	3 209	0.07%
6	$< Q_{\mathcal{M}} <$	7	2 169	0.05%
7	$< Q_{\mathcal{M}} <$	8	1 648	0.04%
8	$< Q_{\mathcal{M}} <$	9	1 276	0.03%
9	$< Q_{\mathcal{M}} <$	10	976	0.02%
10	$< Q_{\mathcal{M}} <$	100	6 352	0.14%
100	$< Q_{\mathcal{M}} <$	1000	48	0.00%

Distribution of $Q_{\mathcal{M}}$ for tetrahedra				
1	$< Q_{\mathcal{M}} <$	2	1 040 413 376	**93.86**%
2	$< Q_{\mathcal{M}} <$	3	32 540 901	2.94%
3	$< Q_{\mathcal{M}} <$	4	9 077 486	0.82%
4	$< Q_{\mathcal{M}} <$	5	5 316 550	0.48%
5	$< Q_{\mathcal{M}} <$	6	3 618 423	0.33%
6	$< Q_{\mathcal{M}} <$	7	2 643 492	0.24%
7	$< Q_{\mathcal{M}} <$	8	2 021 664	0.18%
8	$< Q_{\mathcal{M}} <$	9	1 597 095	0.14%
9	$< Q_{\mathcal{M}} <$	10	1 294 143	0.12%
10	$< Q_{\mathcal{M}} <$	100	9 691 925	0.87%
100	$< Q_{\mathcal{M}} <$	1 000	297 559	0.03%
1000	$< Q_{\mathcal{M}} <$	10 000	2 950	0.00%

Table 13
Landing gear test case on 120 cores. Table gathering the size of the interface, the number of inserted tetrahedra and the CPU time for each iteration.

Iteration	% done	# of tets in interface	# of tets inserted	CPU time (s)	# of cores	elt/sec	elt/sec/core
1	84%	89 718 245	1 009 783 723	487.5	120	2.010^6	17 261
2	91%	16 368 313	1 107 015 758	126.7	120	7.610^5	6 395
3	92%	645 035	1 122 857 778	36.6	87	4.310^5	4 975
4	97%	2 351	1 123 488 597	5.6	4	1.110^5	28 161
5	100%	0	1 123 490 929	1.7	1	1.310^3	1 371

Table 14
Landing gear test case on 120 cores: Distribution of the length of edges in the metric, histograms of the quality of surface and volume elements in the metric.

Distribution of $\ell_{\mathcal{M}}$ for edges				
0.00	$< \ell_{\mathcal{M}} <$	0.20	5 1384	0.00%
0.20	$< \ell_{\mathcal{M}} <$	0.50	71 7381	0.05%
0.50	$< \ell_{\mathcal{M}} <$	0.71	3 156 440	0.23%
0.71	$< \ell_{\mathcal{M}} <$	0.90	395 251 255	**28.25**%
0.90	$< \ell_{\mathcal{M}} <$	1.11	507 393 290	**36.27**%
1.11	$< \ell_{\mathcal{M}} <$	1.41	435 996 200	**31.16**%
1.41	$< \ell_{\mathcal{M}} <$	2.00	49 278 188	3.52%
2.00	$< \ell_{\mathcal{M}} <$	5.00	7 153 341	0.51%
5.00	$< \ell_{\mathcal{M}}$		38 377	0.00%

Distribution of $Q_{\mathcal{M}}$ for triangles				
1	$< Q_{\mathcal{M}} <$	2	14 173 311	**99.37**%
2	$< Q_{\mathcal{M}} <$	3	41 197	0.29%
3	$< Q_{\mathcal{M}} <$	4	22 200	0.16%
4	$< Q_{\mathcal{M}} <$	5	14 378	0.10%
5	$< Q_{\mathcal{M}} <$	6	7 188	0.05%
6	$< Q_{\mathcal{M}} <$	7	2 839	0.02%
7	$< Q_{\mathcal{M}} <$	8	1 364	0.01%
8	$< Q_{\mathcal{M}} <$	9	649	0.00%
9	$< Q_{\mathcal{M}} <$	10	257	0.00%
10	$< Q_{\mathcal{M}} <$	100	349	0.00%

Distribution of $Q_{\mathcal{M}}$ for tetrahedra				
1	$< Q_{\mathcal{M}} <$	2	1 032 582 415	**95.46**%
2	$< Q_{\mathcal{M}} <$	3	23 777 126	2.20%
3	$< Q_{\mathcal{M}} <$	4	6 604 689	0.61%
4	$< Q_{\mathcal{M}} <$	5	3 973 735	0.37%
5	$< Q_{\mathcal{M}} <$	6	2 736 791	0.25%
6	$< Q_{\mathcal{M}} <$	7	2 010 912	0.19%
7	$< Q_{\mathcal{M}} <$	8	1 540 367	0.14%
8	$< Q_{\mathcal{M}} <$	9	1 208 878	0.11%
9	$< Q_{\mathcal{M}} <$	10	966 202	0.09%
10	$< Q_{\mathcal{M}} <$	100	6 281 111	0.58%
100	$< Q_{\mathcal{M}} <$	1 000	51 584	0.00%
1000	$< Q_{\mathcal{M}} <$	10 000	43	0.00%

- a cavity-based correction of the interface mesh to ensure that enough elements are included in order to favor the success of the needed mesh modification operator at the next iteration.

Additional developments are needed to still reduce the total CPU time. The current work is directed at recovering the IOs with the remeshing. Indeed, as we use an out-of-core strategy, the final gathering can be partially done at the same time. Then, the partitioning techniques of the interfaces are also currently extending to work efficiently as well in a parallel environment. For the cavity algorithm, additional developments are needed to speed-up the insertion process when working on interface meshes to make sure that the optimal serial time is kept as constant as possible for all the levels of the parallel remeshing process.

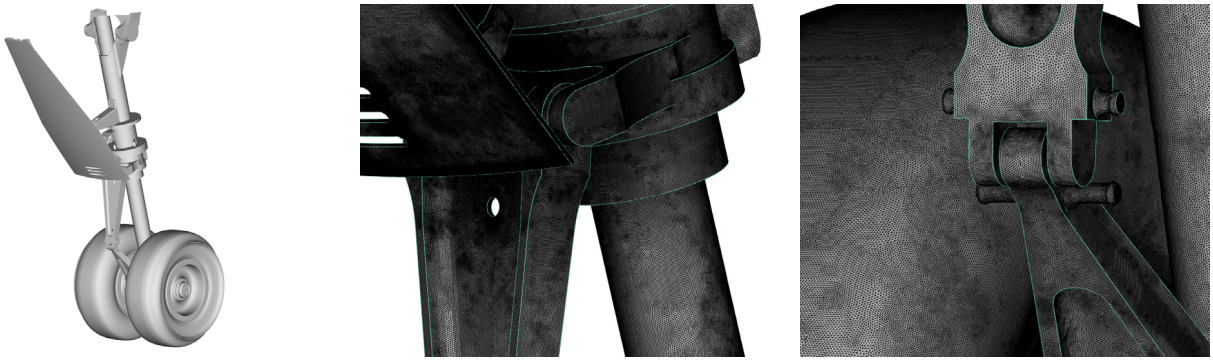

Fig. 12. Landing gear test case. Geometry of the landing gear (left) and closer view of the surface mesh around some geometrical details (middle and right).

References

[1] The ubercloud hpc experiment: Compendium of case studies, 2013.
[2] Dongarra J. Toward a new metric for ranking high performance computing systems. Sandia Report. 2013.
[3] Löhner R. A 2nd generation parallel advancing front grid generator. In: Jiao Xiangmin, Weill Jean-Christophe, editors. Proceedings of the 21st international meshing roundtable. Springer Berlin, Heidelberg; 2013. p. 457–74.
[4] Tremel U, Sørensen KA, Hitzel S, Rieger H, Hassan O, Weatherill NP. Parallel remeshing of unstructured volume grids for cfd applications. Internat J Numer Methods Fluids 2007;53(8):1361–79.
[5] Ito Y, Shih AM, Erukala AK, Soni BK, Chernikov AN, Chrisochoides NP, Nakahashi K. Parallel unstructured mesh generation by an advancing front method. Math Comput Simul 2007;75(5–6):200–9.
[6] Foteinos P, Chrisochoides NP. Dynamic parallel 3D delaunay triangulation. In: Quadros WilliamRoshan, editor. Proceedings of the 20th international meshing roundtable. Springer Berlin, Heidelberg; 2012. p. 3–20.
[7] Digonnet Hugues, Silva Luisa, Coupez Thierry. Massively parallel computation on anisotropic meshes. In: 6th international conference on adaptive modeling and simulation. Lisbon (Portugal): International Center for Numerical Methods in Engineering; 2013. p. 199–211.
[8] Lachat C, Dobrzynski C, Pellegrini F. Parallel mesh adaptation using parallel graph partitioning. In: 5th European conference on computational mechanics (ECCM V). Minisymposia in the frame of ECCM V, vol. 3. Barcelone (Spain): IACM & ECCOMAS, CIMNE - International Center for Numerical Methods in Engineering; 2014. p. 2612–23.
[9] Shephard MS, Smith C, Kolb JE. Bringing hpc to engineering innovation. Comput Sci Eng 2013;15(1):16–25.
[10] Chernikov AN, Chrisochoides NP. A template for developing next generation parallel delaunay refinement methods. Finite Elem Anal Des 2010;46(12):96–113. Mesh Generation - Applications and Adaptation.
[11] Özturan C, deCougny HL, Shephard MS, Flaherty JE. Parallel adaptive mesh refinement and redistribution on distributed memory computers. Comput Methods Appl Mech Engrg 1994;119(12):123–37.
[12] Alauzet F, Li X, Seegyoung Seol E, Shephard MS. Parallel anisotropic 3D mesh adaptation by mesh modification. Eng Comput 2006;21(3):247–58.
[13] Alleaume A, Francez L, Loriot M, Maman N. Automatic tetrahedral out-of-core meshing. In: Brewer MichaelL, Marcum David, editors. Proceedings of the 16th international meshing roundtable. Springer Berlin, Heidelberg; 2008. p. 461–76.
[14] Löhner R, Parikh P. Three-dimensional grid generation by the advancing front method. Internat J Numer Methods Fluids 1988;9:1135–49.
[15] Mavriplis DJ. An advancing front delaunay triangulation algorithm designed for robustness. J Comput Phys 1995;117:90–101.
[16] Baker TJ. Three-dimensional mesh generation by triangulation of arbitrary point sets. In: 8th AIAA computational fluid dynamics conference, AIAA Paper1987-1124, Jun 1987.
[17] George PL, Borouchaki H. Delaunay triangulation and meshing: application to finite elements. Paris (Oxford): Hermès Science; 1998.
[18] George PL, Hecht F, Saltel E. Fully automatic mesh generator for 3D domains of any shape. Impact Comput Sci Eng 1990;2(3):187–218.
[19] Marcum DL. Efficient generation of high-quality unstructured surface and volume grids. Eng Comput 2001;17:211–33.
[20] Loseille A, Löhner R. On 3D anisotropic local remeshing for surface, volume and boundary layers. In: Proceedings of the 18th international meshing roundtable. Springer; 2009. p. 611–30.
[21] De Cougny HL, Shephard MS. Parallel refinement and coarsening of tetrahedral meshes. Internat J Numer Methods Engrg 1999;46(7):1101–25.
[22] Karypis G, Kumar V. A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J Sci Comput 1998;20(1):359–92.
[23] Löhner R, Camberos J, Merriam M. Parallel unstructured grid generation. Comput Methods Appl Mech Engrg 1992;95(3):343–57.
[24] Coupez T. Génération de maillages et adaptation de maillage par optimisation locale. Rev Eur Élém Finis 2000;9:403–23.
[25] Dobrzynski C, Frey PJ. Anisotropic Delaunay mesh adaptation for unsteady simulations. In: Proceedings of the 17th international meshing roundtable. Springer; 2008. p. 177–94.
[26] Li X, Shephard MS, Beal MW. 3D anisotropic mesh adaptation by mesh modification. Comput Methods Appl Mech Engrg 2005;194(48–49):4915–50.
[27] Loseille A, Löhner R. Adaptive anisotropic simulations in aerodynamics. In: 48th AIAA aerospace sciences meeting, AIAA Paper2010-169, Orlando, FL, USA, Jan. 2010.
[28] Michal T, Krakos J. Anisotropic mesh adaptation through edge primitive operations. In: 50th AIAA aerospace sciences meeting, Jan. 2012.
[29] Frey PJ, George PL. Mesh generation. Application to finite elements. 2nd ed. ISTE Ltd. and John Wiley & Sons; 2008.
[30] Loseille A, Menier V. Serial and parallel mesh modification through a unique cavity-based primitive. In: Proceedings of the 22th international meshing roundtable. Springer; 2013. p. 541–58.
[31] Bowyer A. Computing dirichlet tessellations. Comput J 1981;24(2):162–6.
[32] Watson DF. Computing the n-dimensional Delaunay tessellation with application to voronoi polytopes. Comput J 1981;24(2):167–72.
[33] Hermeline F. Triangulation automatique d'un polyèdre en dimension *n*. RAIRO - Anal Numér 1982;16(3):211–42.
[34] Loseille A. Metric-orthogonal anisotropic mesh generation. In: Proceedings of the 23th international meshing roundtable, procedia engineering, vol. 82. 2014, p. 403–15.
[35] Loseille A, Alauzet F. Continuous mesh framework. Part I: well-posed continuous interpolation error. SIAM J Numer Anal 2011;49(1):38–60.
[36] Loseille A, Alauzet F. Continuous mesh framework. Part II: validations and applications. SIAM J Numer Anal 2011;49(1):61–86.
[37] Frey PJ, Alauzet F. Anisotropic mesh adaptation for CFD computations. Comput Methods Appl Mech Engrg 2005;194(48–49):5068–82.
[38] Alauzet F, Loseille A. On the use of space filling curves for parallel anisotropic mesh adaptation. In: Proceedings of the 18th international meshing roundtable. Springer; 2009. p. 337–57.