Parallel mesh adaptation using parallel graph partitioning

C Lachat, C Dobrzynski, F Pellegrini

11th World Congress on Computational Mechanics (WCCM XI)
5th European Conference on Computational Mechanics (ECCM V)
6th European Conference on Computational Fluid Dynamics (ECFD VI)
E. Oñate, J. Oliver and A. Huerta (Eds)

PARALLEL MESH ADAPTATION USING PARALLEL GRAPH PARTITIONING

C. Lachat[1], C. Dobrzynski[2] and F. Pellegrini[3]

[1] Inria Bordeaux Sud-Ouest, 200 av. de la Vieille Tour, 33405 Talence, France
cedric.lachat@inria.fr
[2] Institut Polytechnique de Bordeaux & Inria Bordeaux Sud-Ouest, 351 cours de la Libération, 33405 Talence, France cecile.dobrzynski@inria.fr
[3] Université de Bordeaux, LaBRI & Inria Bordeaux Sud-Ouest, 351, cours de la Libération, 33405 Talence, France francois.pellegrini@inria.fr

Key words: high performance computing, parallel remeshing, distributed memory, anisotropic mesh, mesh adaptation, parallel graph partitioning

Abstract. This paper presents a parallel remeshing algorithm for distributed-memory architectures. It is an iterative parallel algorithm that divides the areas to be remeshed into multiple pieces which can be distributed to as many processing elements as possible, in order for these pieces to be remeshed concurrently by a third-party sequential remesher. Then, remeshed pieces are reintegrated into the distributed mesh, and this process is iterated until all relevant areas of the mesh have been remeshed. Any sequential remesher can be used, provided it allows some of the mesh elements not to be modified, so as to preserve interfaces between pieces. Our method, which has been implemented in the PaMPA library, is validated by a set of experiments involving both isotropic and anisotropic meshes.

1 Introduction

Today's large scale simulations can only be run in parallel, because many meshes are now too big to fit in the memory of a single computer. Since shared-memory architectures are subject to memory bottlenecks, scalability can only be achieved by using distributed-memory architectures such as workstation clusters. Therefore, a prerequisite for such simulations is to be able to generate huge meshes in a parallel, distributed-memory fashion. Moreover, in the case where users would like to perform mesh adaptation, the latter must also be performed in parallel.

In this work, we propose a parallel mesh adaptation procedure that relies on existing sequential mesh adaptation software. The only requirement of this sequential software is that it should not modify prescribed mesh elements. The advantage of this approach is

its ease of use, through the ability to use existing sequential software to generate huge meshes in reasonable time.

This paper is organized as follows. After presenting the state of the art in parallel remeshing, we will briefly discuss the features of the PaMPA [25] library that we designed to handle distributed meshes, and then present out parallel remeshing algorithm. We will then show results of parallel remeshing that we achieve with the current version of PaMPA, combined with the MMG3D sequential tetrahedral remesher.

2 State of the art in parallel remeshing

The need for remeshing during a numerical simulation may arise from two different reasons: either to reduce the numerical inaccuracy induced by an inadequate mesh (most often because it is too coarse; yet, too fine meshes may induce numerical error and waste too much compute time), or because of a changing geometry (moving bodies, deformations, etc.).

Sequential remeshers are confronted to physical limitations which hinder their practical use. The hardest limit is the maximum available memory per processing element (PE), but run time also becomes a strong concern for very large meshes. This is why many authors explored the path of parallel remeshing.

Shared-memory parallelism is a straightforward way to reduce run time, at the expense of only small modifications to existing sequential remeshing algorithms, by adding locking mechanisms that prevent multiple PEs to process simultaneously the same elements. However, shared-memory parallelism cannot scale to a large number of processing elements, because of memory bottlenecks. Moreover, all large-scale parallel architectures implement a distributed-memory model, so that remeshing on the fly the distributed meshes used in large-scale simulations requires distributed-memory parallel remeshing methods. In the case where the physical nodes of the parallel cluster comprise several cores, shared-memory parallelism can be used at the node level, but explicit message passing still has to be used across machine nodes.

There exist two main families of remeshing algorithms. The first one, which may seem the most natural, consists in parallelizing existing sequential remeshing algorithms. However, in a distributed-memory context, this is not an easy task, because of the synchronization costs across PEs. This is why a second family of methods has been investigated, which uses existing sequential remeshers as black boxes within an iterative framework.

2.1 Parallelization of sequential algorithms

In order to parallelize sequential remeshing techniques, one has to adapt to the distributed-memory framework, elementary mesh transformations such as Delaunay triangulation and edge subdivision.

The first works on parallel remeshing were carried out in the 1990's on 2D meshes,

by Castaños and Laemmer. In 1996, Castaños and Savage [4] proposed a distributed-memory remeshing method for 2D meshes. Their technique allows one to adapt the meshes several times, while preserving the correspondance between elements across successive remeshings. In the case of triangles, mesh refinement is performed by cutting in two the longest edge, so as to yield two smaller triangles. In order to avoid inconsistencies on the frontier across two subdomains, a synchronization procedure is applied to frontier edges, so that, when some PE wants to split an edge, the same node is created on all copies of this edge possessed by neighboring PEs.

The parallel remeshing of homogeneous 3D meshes has been studied by Oliker, followed by the works of Chrisochoides, Casagrande and Cavallo. In 2000, Oliker *et al.* [20] presented both some parallel remeshing techniques, which they implemented in the 3D_TAG software, and a load balancing library called PLUM [19]. In 2003, Chrisochoides [6] proposed a parallel version of the Delaunay triangulation algorithm. This method is based on querying neighboring PEs about the frontier face for which the Delaunay cavity has to be extended.

In the method proposed by Casagrande [3] in 2005, it is not necessary to keep the correspondance between the remeshed and the original meshes, thus allowing for better flexibility in the remeshing and higher quality of the adaptation. While edge and face flipping can be performed easily at the interface between two or more PEs, structural changes such as node addition and removal are much more difficult to achieve, because of the need to synchronize all involved PEs.

Also, parallel remeshing has been implemented for mixed meshes by Lawlor [18], in the ParFUM library [18].

As one can see, the parallelization of sequential remeshing algorithms requires to perform some synchronization across several PEs when the elements to be remeshed belong to the frontier between two or more subdomains. While locking and synchronization primitives are reasonably cheap in shared memory, they incur very high latency in a distributed memory context. This is why methods of a higher granularity of synchronization have been investigated. They amount to run concurrently several instances of existing sequential remeshers, on independent pieces of the mesh.

2.2 Re-use of existing sequential remeshers

The re-use of sequential algorithms in a parallel context started with the works of Coupez [8], followed by those of Dobrzynski [13] and Tremel [23].

In 2000, Coupez [8] proposed a parallel remeshing method based on an iterative process. At each step, sequential remeshing techniques are applied to the interior of the subdomains possessed by every PE, without changing their frontiers. Then, the mesh is migrated so that former frontier areas reside in the interior of the new subdomains, so that they can be remeshed during the next iteration.

This technique has been extended by Dobrzynski [13], using the MMG3D sequential

remesher [14]. It allowed her to remesh isotropic as well as anisotropic meshes, in 2D and in 3D. Meshes comprising more than 2 millions of tetraedra have been remeshed on 32 PEs.

In 2005, Cavallo *et al.* [5] presented the adaptation method they implemented in the CRUNCH CFD code [16]. Remeshing is applied to mixed unstructured meshes by means of edge deletion, Delaunay triangulation and smoothing techniques [2] for tetraedra of low quality (that is, tetraedra one angle of which is greater than 120 degrees). Every subdomain performs its remeshing without modifying the frontiers. Then, interfaces are migrated between pairs of PEs, so that PEs receive the other side of one of their interfaces and process them alone.

In all of the above methods, balancing the load of remeshing tasks is a concern, as some PEs may have much more work to do than others. This is why, in the method we propose, we will over-decompose the areas to be remeshed so as to leave as few idle PEs as possible.

Another aspect of parallel remeshing is the ability to project the values of the variables from the original mesh to the remeshed mesh. In 2007, Digonnet *et al.* [10] proposed both a parallel remeshing method based on interface migration, and a parallel hierarchical migration method to copy the values of the original mesh to the remeshed mesh.

Also, some numerical simulations may require several correlated meshes, each of them having their specificity. Ramadan [22] extends the parallel remeshing method to such meshes, in the case of incremental industrial processes like wire drawing and metal rolling. A fine mesh is used to store the results and perform thermal computations, while a coarser mesh is used for structural mechanics computations. This method has been implemented in the Cimlib library [11], which uses the MTC mesh generator [7] as a remesher.

3 The PaMPA library

Remeshing is not the only feature that is needed by writers of numerical solvers that operate on unstructured meshes. More generally, the numerical resolution of problems arising in, *e.g.*, fluid dynamics, requires to represent adequately the unstructured meshes that discretize the simulation domain and bear the simulation data, and then to write the numerical scheme so as to compute efficiently the solution of the numerical problem on this mesh. Consequently, most of the code of such parallel numerical solvers does not relate to the numerical scheme itself, but to the handling of data structures and communication, which is indeed not the core business of applied mathematicians. Consequently, huge amounts of time are spent writing again and again blocks of code that perform the same tasks. The common set of functional needs that have to be addressed by solver writers is the following.

First, software should be able to represent adequately distributed meshes in memory, according to the distributed-memory paradigm. This imposes to distribute mesh data across the PEs of a parallel architecture, so as to evenly balance workload. It amounts to

performing a subdomain decomposition of the mesh, each subdomain being assigned to a different PE. For the sake of usability, meshes should consist of any kind of element, and the software should be able to handle mixed and/or non-conformant meshes.

In order to perform the desired computations, data has to be attached to the different entities that make up the mesh: elements, faces, edges, nodes, etc. These data may differ according to some sub-classification of the mesh entities, *e.g.* numerical boundary condition values attached only to boundary faces. Since computations on every PE involve data that belongs to other PEs, primitives must allow one to perform data exchange across neighboring entities, taking into account the different amount and type of data borne by entities and sub-entities. Data exchanges should be asynchronous, so as to overlap communication with computation: computations can be performed on mesh entities that belong to the interior of the subdomains, while data borne by subdomain frontier entities are exchanged asynchronously.

Writing the numerical schemes equates to iterating over all the members of some entity in an outer loop, and then to iterating over members of some other entity that are adjacent to each member of the entity considered in the outer loop. Hence, one should benefit from efficient iterator methods that allow him/her to iterate, on each PE, over any kind of entity (*e.g.* local elements, local faces, etc.) or sub-entity (*e.g.* local regular faces, local boundary faces, etc.), potentially distinguishing between internal or frontier entities so as to overlap communication with computation (*e.g.* local internal regular elements, local frontier boundary faces, etc.). In order to speed-up computations, mesh data must be placed in memory so as to maximize cache effects. This amounts to performing a reordering of all mesh entities on each PE, such that data that are topologically close (and thus will be accessed together) are placed close to each other in memory. This reordering can advantageously be performed while distributing mesh data across the PEs.

Then, in some cases, one may want to dynamically modify the mesh structure, or the computations attached to it (*e.g.* when switching from an elastic deformation model to more expensive plasticity computations in some parts of a mesh). This imposes to perform a dynamic redistribution of the mesh entities, so as to restore load balance. All relevant numerical data associated with the migrated entities has to be migrated, and may also benefit from the aforementioned data reordering.

Finally, this dynamic modification can be delegated to a third-party remesher. Since almost all remeshing software is sequential, one has to run multiple times this sequential software on different pieces of the mesh, and to handle cases when areas to remesh span across several subdomains.

As one can see, implementing all the aforementioned features represents a huge amount of work, which is all the more wasteful as it is done independently by every solver writer.

PaMPA [17] (for "*Parallel Mesh Partitioning and Adaptation*") is a software that aims at relieving solver writers from the tedious and error-prone task of writing again and again service routines for mesh handling, data communication and exchange, remeshing,

and data redistribution. PaMPA represents meshes as graphs, whose data is distributed across the processors of the parallel machine. It allows users to define distributed meshes, to declare values attached to the entities of the meshes, to exchange values between overlapping entities located on the frontiers of subdomains assigned to different processors, to iterate over the relations of entities, to remesh the pieces of the mesh that need to be, and to redistribute evenly the remeshed mesh across the processors of the parallel architecture. PaMPA relies on the PT-Scotch library [21] for parallel graph (re)partitioning.

4 Our parallel remeshing algorithm

Our method for parallel remeshing is an iterative algorithm made of two nested loops. The outer loops iterates the remeshing process until all elements that need to be remeshed have been processed. The inner loop, which is depicted in Figure 1, is made of five consecutive steps. It splits the groups of elements to be remeshed into pieces that will be remeshed concurrently on as many PEs as possible by a third-party, sequential remesher. In the following, we will give further details on each of these five steps. A thorough description of all of the sketched algorithms can be found in [17].

Tagging of elements to be remeshed The purpose of the first step is to tag the elements that need to be remeshed. Step $1a$, which represents the initialization phase of the inner loop, just considers the tags that have been computed at the beginning of the outer loop. In subsequent inner iterations, corresponding to step $1b$, tags are removed from the elements that have been successfully remeshed. The elements that belong to the skin of the pieces are left tagged, as they cannot be remeshed (see below).

Computation of independent pieces This step aims at computing pieces of the mesh that can be remeshed independently from each other. Three criteria must be enforced: (i) pieces must contain mostly elements that have been flagged. However, because remeshers have to modify the neighboring elements of an element being refined, a piece may contain additional un-tagged elements that form a "skin" around the tagged elements; (ii) the size of the pieces must be big enough to leave room for work to the sequential remesher. Yet, there must be enough pieces not to leave any PE idle. Hence, the size of the pieces may depend on the number of PEs, and has also to be bounded by the available remaining memory on the PEs; (iii) the size of the skin must be as small as possible, so that the aspect ratio of the piece to remesh is as high as possible, hence constraining the remesher as little as possible.

During our research, we have experimented several algorithms to compute the pieces. The most efficient one, with respect to the above criteria, is to use graph partitioning on the dual graph. The desired number of parts is determined by estimating the number of nodes that will be created (or removed) by the remeshing process. Each of the tagged elements is assigned a weight that represents an estimation of the amount of work that the

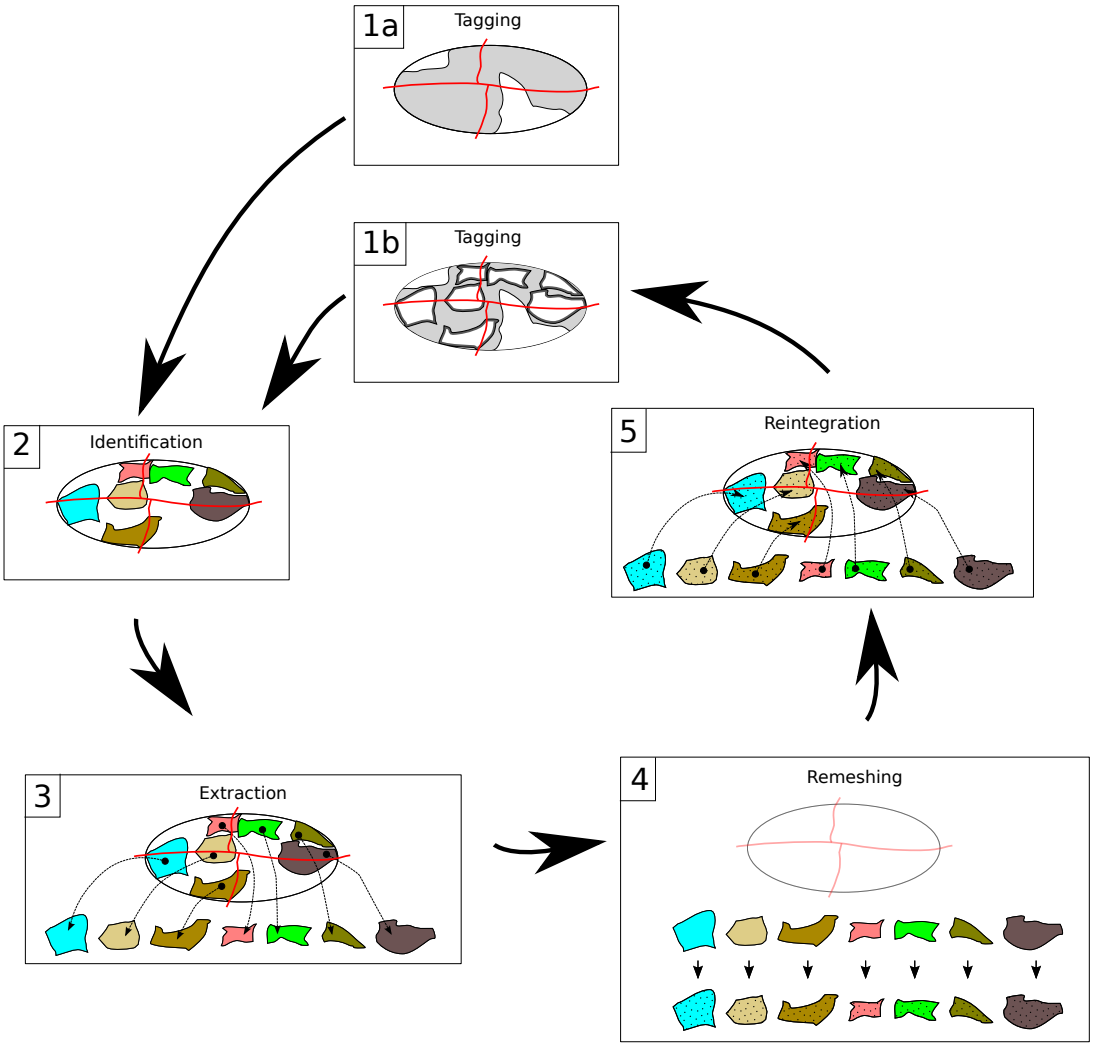

Figure 1: Sketch representing the inner loop of our algorithm, composed of five steps. The first one ($1a$, $1b$) tags the elements that still need to be remeshed (the gray area). The second one computes independent pieces. The third one extracts the pieces and gives them to PEs so as to balance the workload of the fourth step, which is the sequential remeshing phase. In the last step, remeshed pieces are reintegrated into the distributed mesh. This process iterates until no tagged element remains.

remesher will spend processing this element. Then the parallel partitioner PT-Scotch is called to partition this weighted dual graph into the desired number of pieces.

Extraction of the pieces to be remeshed The pieces that have been computed at the previous step may span across multiple PEs. The goal of this step is therefore to gather all the fragments of a given piece to the same PE, so that PEs can work on

complete, centralized instances of the pieces that they have to remesh. All the entities adjacent to an element that belongs to some piece are copied to the destination PE for this piece. Consequently, some entities (*e.g.* nodes) may be duplicated on several PEs, if they are adjacent to elements that belong to different pieces and thus are to be processed by different PEs.

The assignment of pieces to PEs is also performed by using the Scotch partitioner. The purpose of this partition is to balance the estimated remeshing workload of the pieces among the PEs, while migrating as little data as possible across PEs, by favoring the assignment of pieces to PEs that already hold most, or at least some, of their data.

Remeshing of the pieces Every piece, now on the form of a centralized submesh, is processed concurrently by an instance of the third-party, sequential remesher. In order to be able to reintegrate the remeshed pieces into the distributed mesh, the outermost layer of elements that belong to the skin of the piece must never be modified by the remesher.

Reintegration of the remeshed pieces Once the pieces have been remeshed, they have to be reintegrated into the unmodified part of the distributed mesh. This includes merging the new topological information, as well as the new values associated with the new mesh entities. Once this step is complete, a new iteration of the inner loop can take place.

After all the tagged elements have been remeshed, the new distributed mesh may be repartitioned and redistributed, so as to rebalance workload across the PEs.

Our parallel remeshing method has been implemented in the PaMPA parallel library. An interface has been defined, which allows one to use any sequential remesher that allows some of the elements of the mesh that is passed to it not to be modified.

5 Experimental validation

In order to show the genericity of our method, we applied it to several test cases, both isotropic and anisotropic. For that purpose, we combined PaMPA with the MMG3D sequential tetrahedral isotropic and anisotropic remesher [12]. Our test machine is Avakas [24], a cluster of Intel® Xeon® x5675's running at 3 GHz.

5.1 Isotropic mesh generation

On 240 processors, we have been able to remesh an isotropic mesh with 27 millions of elements into a refined mesh comprising more than 600 millions of elements, in less than 35 minutes. Table 1 evidences the quality of the produced mesh. We could not compare our method to its sequential counterpart, as the sequential remesher ran out of memory on our largest system, which has 200 Gib of memory.

Initial number of elements	27 044 943
Final number of elements	609 671 387
Elapsed time	00h34m59s
Elapsed time × number of PEs	139h56m
Smallest edge length	0.2911
Largest edge length	8.3451
Worst element quality	335.7041
% element quality between 1 and 2	98.92%
% edge length between 0.71 and 1.41	97.20%

Table 1: Execution time and mesh quality of the remeshing of a link rod composed of isotropic tetrahedra, on 240 processors, with PaMPA-MMG3D.

5.2 Anisotropic mesh generation

In a second test case, we generated an anisotropic mesh with the aim to capture an interface. In [9], the authors propose a geometric approximation control of interfaces using an anisotropic metric. They rely on the generation and the adaptation of an anisotropic triangulation to a metric tensor field related to the intrinsic properties of the manifold. We adapted this technique to generate meshes under the constraint that the distance between a point and the surface is limited. Thanks to a proper definition of the metric tensor, we can generate anisotropic meshes adapted to an interface defined as a level-set.

In our example, the surface is an ellipse (sizes: $(0.5, 0.1, 0.2)$) centered in a sphere (of radius $r = 10$), we approximate the surface with a precision of $\varepsilon = 0.0003$, and we impose a minimal edge size equal to 0.0003. The results concerning this test case are summed up in Table 2. The quality of the mesh after remeshing is good enough, while execution time could be improved.

Initial number of elements	715 791
Final number of elements	29 389 210
Elapsed time	00h34m
Elapsed time × number of PEs	27h12m
Smallest edge length	0.1116
Largest edge length	8.2191
Worst element quality	14.8259
% element quality between 1 and 2	99.61%
% edge length between 0.71 and 1.41	93.65%

Table 2: Execution time and mesh quality of the remeshing of a sphere composed of anisotropic tetrahedra, on 48 processors, with PaMPA-MMG3D.

5.3 Mesh adaptation to a physical solution

Finally, we used our method within an adaptation loop to solve a transsonic Eulerian flow around a M6 wing. The system of equations is discretized using a residual distribution scheme [1]. Mesh adaptation is guided by a metric based on an *a posteriori* error estimate of the interpolation error [15].

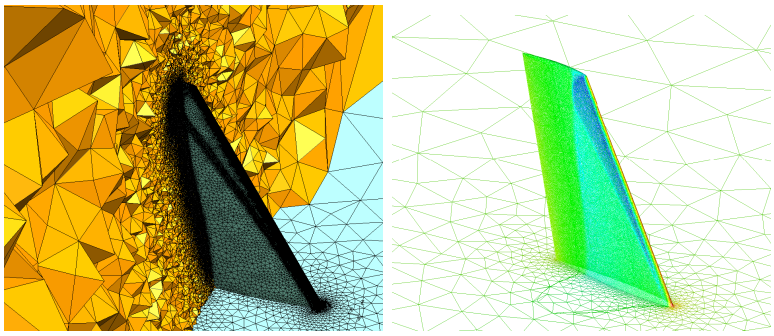

Figure 2: On the left: volumic cut on the adapted mesh; on the right: density field.

6 Conclusion

We have presented a parallel method for remeshing distributed meshes on distributed-memory machines. This method allows one to use any sequential remeshing software, provided the latter can be instructed not to modify some prescribed elements. This method, which has been implemented in the PaMPA library, already allows us to produce meshes of several hundred millions of elements in reasonable time. Yet, smaller run times could be achieved with a better distribution of the remeshing workload across PEs. PaMPA [25] and MMG3D [14] are available from the Inria GForge, under the GPL license.

REFERENCES

[1] R. Abgrall, A. Larat, and M. Ricchiuto. Construction of very high order residual distribution schemes for steady inviscid flow problems on hybrid unstructured meshes. *Journal of Computational Physics*, 230(11):4103–4136, 2011.

[2] T. J. Baker and J. C. Vassberg. Tetrahedral mesh generation and optimization. In *Proc. of the 6th International Conference on Numerical Grid Generation*, pages 337–349, 1998.

[3] A. Casagrande, P. Leyland, L. Formaggia, and M. Sala. Parallel mesh adaptation. *Series on Advances in Mathematics for Applied Sciences*, 69:201, 2005.

[4] J. G. Castaños and J. E. Savage. The dynamic adaptation of parallel mesh-based computation. In *SIAM 7th Symposium on Parallel and Scientific Computation*, 1996.

[5] P. A. Cavallo, N. Sinha, and G. M. Feldman. Parallel unstructured mesh adaptation method for moving body applications. *AIAA journal*, 43(9):1937–1945, 2005.

[6] N. Chrisochoides and D. Nave. Parallel delaunay mesh generation kernel. *International Journal for Numerical Methods in Engineering*, 58(2):161–176, 2003.

[7] T. Coupez. A mesh improvement method for 3d automatic remeshing. *Numerical Grid Generation in Computational Fluid Dynamics and Related Fields*, pages 615–626, 1994.

[8] T. Coupez, H. Digonnet, and R. Ducloux. Parallel meshing and remeshing. *Applied Mathematical Modelling*, 25(2):153–175, 2000.

[9] C. Dapogny and P. Frey. Computation of the signed distance function to a discrete contour on adapted triangulation. *Calcolo*, 49(3):193–219, 2012.

[10] H. Digonnet, M. Bernacki, L. Silva, and T. Coupez. Adaptation de maillage en parallèle, application à la simulation de la mise en forme des matériaux. In *Congrès Français de Mécanique Grenoble-CFM 2007*, page 6 pages, Grenoble, France, 2007. http://hdl.handle.net/2042/16046.

[11] H. Digonnet, T. Coupez, et al. Object-oriented programming for "fast and easy" development of parallel applications in forming processes simulation. *Computational Fluid And Solid Mechanics 2003, Vols 1 and 2, Proceedings*, 2003.

[12] C. Dobrzynski and P. Frey. Anisotropic delaunay mesh adaptation for unsteady simulations. In *Proceedings of the 17th international Meshing Roundtable*, pages 177–194. Springer, 2008.

[13] C. Dobrzynski and J.F. Remacle. Parallel mesh adaptation. *International Journal for Numerical Methods in Engineering*, 2007.

[14] MMG3D: Anisotropic tetrahedral remesher/moving mesh generation. `http://www.math.u-bordeaux1.fr/~cdobrzyn/logiciels/mmg3d.php`.

[15] P.-J. Frey and F. Alauzet. Anisotropic mesh adaptation for cfd computations. *Computer methods in applied mechanics and engineering*, 194(48):5068–5082, 2005.

[16] A. Hosangadi, R.A. Lee, P.A. Cavallo, N. Sinha, and B.J. York. Hybrid, viscous, unstructured mesh solver for propulsive applications. *AIAA paper*, pages 98–3153, 1998.

[17] C. Lachat. *Conception et validation d'algorithmes de remaillage parallèles à mémoire distribuée basés sur un remailleur séquentiel*. PhD thesis, Université de Nice Sophia Antipolis, 2013.

[18] O. Lawlor, S. Chakravorty, T. Wilmarth, N. Choudhury, Is. Dooley, G. Zheng, and L. Kalé. Parfum: a parallel framework for unstructured meshes for scalable dynamic physics applications. *Engineering with Computers*, 22:215–235, 2006. 10.1007/s00366-006-0039-5.

[19] L. Oliker and R. Biswas. Plum: Parallel load balancing for adaptive unstructured meshes. *Journal of Parallel and Distributed Computing*, 52(2):150–177, 1998.

[20] L. Oliker, R. Biswas, and H. N. Gabow. Parallel tetrahedral mesh adaptation with dynamic load balancing. *Parallel Computing*, 26(12):1583–1608, 2000.

[21] F. Pellegrini. Scotch and PT-Scotch Graph Partitioning Software: An Overview. *Combinatorial Scientific Computing*, pages 373–406, 2012.

[22] M. Ramadan, L. Fourment, and H. Digonnet. A parallel two mesh method for speeding-up processes with localized deformations: application to cogging. *International Journal of Material Forming*, 2:581–584, 2009. 10.1007/s12289-009-0440-x.

[23] U. Tremel, K. A. Sørensen, S. Hitzel, H. Rieger, O. Hassan, and N. P. Weatherill. Parallel remeshing of unstructured volume grids for cfd applications. *International journal for numerical methods in fluids*, 53(8):1361–1379, 2007.

[24] Cluster avakas. `http://redmine.mcia.univ-bordeaux.fr/projects/cluster-avakas`.

[25] PaMPA: Parallel Mesh Partitioning and Adaptation. `http://pampa.bordeaux.inria.fr`.