Parallel Algorithm Oriented Mesh Database *

Jean-François Remacle   Ottmar Klaas
Joseph E. Flaherty   Mark S. Shephard

Scientific Computation Research Center,
Rensselaer Polytechnic Institute,
Troy, New York, USA.
Corresponding author: remacle@scorec.rpi.edu

Abstract

In this paper, we present a new point of view for efficiently managing general parallel mesh representations. Taking as a starting point the Algorithm Oriented Mesh Database (AOMD) of [12] we extend the concepts to a parallel mesh representation. The important aspects of parallel adaptivity and dynamic load balancing are discussed. We finally show how AOMD can be effectively interfaced with mesh adaptive partial differential equation solvers. Results of the calculation of an elasticity problem and of a transient fluid dynamics problem involving thousands of mesh refinements, and load balancings are finally presented.

Introduction

In a recent paper [12], we presented a new approach to manage the topological relationships needed from a mesh data structure to meet the needs of multiple applications. We called this new approach the Algorithm Oriented Mesh Database (AOMD) because we have the ability to shape AOMD to

*This work was supported by the ASCI Flash Center at the University of Chicago, under contract B341495, by the U.S. Army Research Office through grant DAAG55-98-1-0200, and by the National Science Foundation through grant DMS-0074174.

the needs of the algorithms. In [12], we discussed both construction and implementation of AOMD.

In this paper, we will discuss some more specific features of AOMD. First, we will explain how AOMD is extended to manage distributed meshes. Based on the distributed data management paradigm described in [14], we will show how AOMD is able to manage interprocess communications independently of the mesh representation. Then, we will show how AOMD can minimize load balancing cost in parallel adaptive mesh refinement by using the minimum mesh representation presented in [12]. For the load balancing algorithms, AOMD uses the capabilities of the Zoltan library developed at Sandia [3]. Finally, we will show how AOMD is able to perform parallel adaptive computations including dynamic load balancing, non-conforming refinement and coarsenings. As examples, we will present an elasticity problem solved with an object oriented framework that has been coupled with AOMD, and a large scale three dimensional computational fluid dynamics example where AOMD is used as mesh library.

1 Algorithm Oriented Mesh Database

A mesh is a discretization of a geometrical domain consisting of mesh entities of controlled size and distribution with simple topology (hexahedron, tetrahedron...). The topology of a mesh is described with adjacencies between mesh entities [1]. Meshes are used for scientific computation. Physical parameters, i.e. material properties and boundary conditions, are to be prescribed on the geometrical model which is the most natural representation of the domain [13] and are then related to the mesh during the analysis process.

Data structures have been published for specific algorithms: mesh generation [9, 6], mesh refinement [4] or solution process [8]. The aim of AOMD is to be a mesh management library (or database) that is able to provide a variety of services for mesh users with an optimal mesh representation: the user is able to shape AOMD to its own algorithms. In [12], we presented this adaptive approach that was able to deal with any representation. For that, we made a certain number of hypotheses. These hypotheses are readily extendable to distributed meshes.

- In order to ensure a valid discretization of the solid model, we maintain a direct link between every mesh entity i of dimension d, refered to as M_i^d, and the geometrical entity G_j^q (with $q \geq d$) it is discretizing. We call this association a *classification* [1] of a mesh entity to a geometrical entity and we note it $M_i^d \sqsubset G_j^q$.

- A unique `iD` is associated with each vertex M_i^0. In this paper, we extend this definition to distributed meshes. We suppose that there is a unique labeling of vertices for a distributed mesh. It does require some care on partition boundaries. One easy selection is a unique numbering which is discussed more with implementation details.

- Any higher dimensional mesh entity M_i^d, $1 < d \leq 3$ is defined using one set of ordered lower order mesh entities. This assumption allows us to access the set of ordered vertices for any mesh entity.

- Two mesh entities are equal if they have the same vertices. Due to the previous hypothesis, we always have access to mesh entity vertices so that it is always possible to compare mesh entities independently of their representation. This definition of course scales to distributed meshes because of the unique global vertex `iD`'s.

We have seen in [12] that the given hypotheses allow us to build efficient algorithms to search the mesh database. We have shown that efficient hashing functions for mesh entity insertion, deletion and searching operations have constant complexity which means that their costs do not depend on the size of the mesh database. These properties remain applicable to distributed meshes because every partition is treated as a serial mesh.

A second interesting feature of AOMD is that it is possible to define a minimum representation of a mesh that allows generation of any other representation using integer operations (no geometrical tests). A *sufficient minimum set* of data is a set in which each mesh entity classified on an equally dimensioned geometrical entity has to be present in the representation. It means that all entities M_i^d for which we have $M_i^d \sqsubset G_j^d$ are to be present and classified if we want the ability to construct a representation. Note that all vertices are to be present in the representation but only those that are classified on model vertices need to be classified. Given that minimum set all the other classifications can be determined. This was outlined in [12]. The mimimum representation does not need to be modified for distributed meshes.

2 Distributed mesh representation

We consider a mesh divided in partitions for distribution to the various processors. Each partition is a mesh that does not differ from a serial mesh. The only AOMD requirement for distributed meshes is that vertices have a global labeling through all partitions.

Mesh partitions have mesh entities in common. In Figure 1, we show an example of a distributed mesh with three partitions. We consider partition boundaries as artificial model entities that represent connections between partitions. The minimum mesh representation in parallel takes these new model entities into account: mesh entities equally classified on partition boundaries must be present as well in the representation. This is a natural extension of the serial definition.

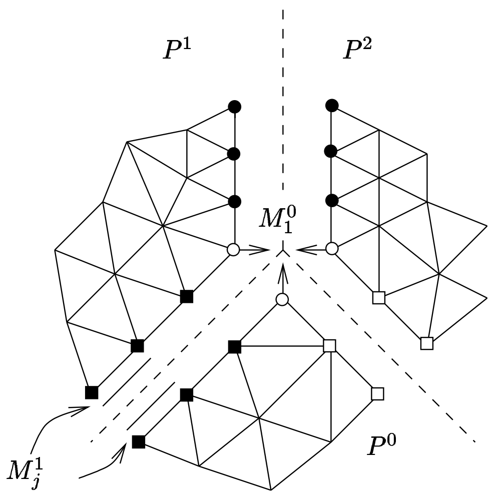

Figure 1: Distributed mesh, three partitions P^1, P^2 and P^3. Vertex M_1^0 is common to all partition and is then classified to a partition vertex. On each partition, several mesh edges like M_j^1 are common to two partitions and so is classified to a partition edge separating two partitions.

In order to make partitions aware of remote entities that are present in other partitions, we use the algorithmic capabilities of AOMD. For each mesh entity M_i^d classified on a partition boundary G_j^q, we send a message to each remote partition. This message contains

- The local adress of M_i^d,

- The list of M_i^d vertices `iD`'s.

With vertices `iD`'s, we are able to find the counterpart of M_i^d on every remote partition of G_j^q. After this round of communication, each remote partition is aware of all remote entities present on other partitions with the local adress of each copy. This simple procedure is called every time the mesh is modified.

3 Parallel mesh adaptation

AOMD being a database, it is possible to add and remove mesh entities in an efficient manner. We have developed mesh refinement procedures that produce adaptive meshes in parallel. Our goal was to provide a mesh adaptation procedure which is sufficiently efficient to be applied thousands of times in one computation. A second requirement is the effective support of non-conformal mesh refinements which are effective for use with specific discretization methods (Figure 2). Mesh modifications consist of refinements

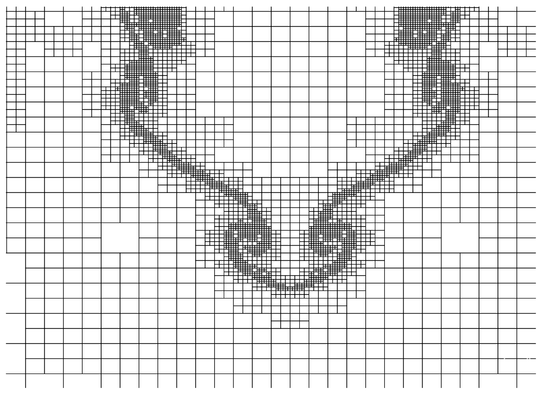

Figure 2: Adaptive non-conformal quad mesh.

and coarsenings. For reasons of efficiency, we conserve the history of mesh refinement in a tree structure for the applications presented here. For that, we use the adjacency list of equivalent dimension in mesh entities. The current mesh adaptation procedure is not the most general in the sense that it can not produce coarser meshes than the initial mesh. The functions needed to support the more general case, including curved domain issues, are given in [5].

In mesh refinement, some entities are split. We use the concept of templates described in [12] which allow us to manage hybrid meshes. For each distinct topological entity (triangle, tetrahedron, wedge,...), we provide one splitting pattern that returns a set of sub-entities. Typically a quadrilateral is split into four quadrilaterals, one tetrahedron is split into eight tetrahedron, etc.

One round of communications is needed to account for the new mesh entities created by the splitting process. This round of communictaions allows:

- Entities that have remote copies must be be split on all partitions boundaries to allow connection between partitions.
- Entities that have remote copies must be split while using the same vertex labels as required to conserve the consistency of the mesh database.

Figure 3 illustrates how parallelism affects mesh adaptation. We have split some triangles into four. When this refinement is done, vertex M_n^0 has to be created on processor P^0 because it has a counterpart M_m^0 on processor P^1. Vertices M_n^0 and M_m^0 must share the same `iD`. On the other hand some edges may be split on several processors (in 3-D, this can be more than 2). In this case, we have to ensure that, for example, vertices M_i^0 and M_j^0 have the same `iD` in order to fulfill consistency requirements of AOMD. For that purpose, we first assign predictor `iD`'s to all entities that have to be split. With one round of communication, we chose as the corrected `iD` the smallest `iD` for all processors: $\mathtt{iD}(M_{i,j}^0) = \min(\mathtt{iD}(M_i^0), \mathtt{iD}(M_j^0))$. Then, the mesh refinement procedures can be applied in serial. Finally we re-connect inter-processor entities using procedure described in §2.

For the parallel implementation of mesh coarsening, we have followed principles presented in [5]. If a cavity has to be coarsened, all elements of this cavity are first migrated to one partition. Then, the coarsening algorithm can be performed without taking care of parallelism.

4 Dynamic load balancing and mesh migration

Mesh refinement introduces load unbalance, i.e. after the refinement procedures finishes some partitions may have significantly more or less mesh entities than other partitions. This unbalance is not acceptable since it prevents the algorithms using the mesh in parallel to scale. Dynamic re-partitioning

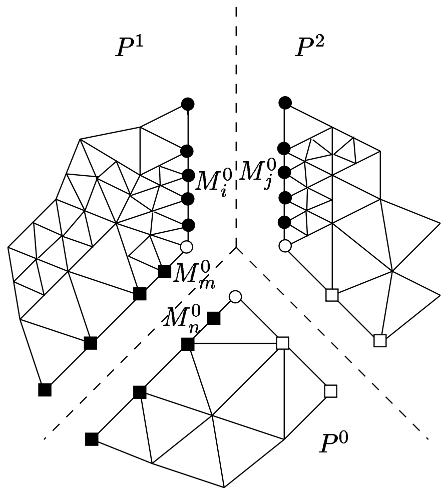

Figure 3: Non-conforming parallel mesh adaptation.

of the mesh can eliminate the unbalance. Several load balancing techniques are available. The Zoltan Dynamic Load-Balancing Library [3] provides critical capability to a number of parallel applications. Zoltan includes a suite of algorithms for dynamically computing partitions of problems over sets of processors; geometric, tree-based and graph-based algorithms are included.

The load balancer takes as input a representation of the parallel mesh, usually a weighted graph. In a mesh, we have different levels of adjacencies that produce different graphs. An illustration can be found on Figure 4: two different graphs are created for the same mesh using different sets of adjacencies. The choice of a graph will depend on what the mesh is used for. In case of a flux-based method (finite volumes, discontinuous finite elements), information is passed through edges (in two dimensions) so that the relevant graph would be the one build with edge-face adjacencies. Typically, a re-partitioning algorithms provides as output a vector containing the redistribution information for the mesh entities that will restore the load balance. With that information at hand, the dynamic re-partitioning step is completed by moving the appropriate entities from one partition to another. One of the effects of the re-partitioning is that interprocessor boundaries are changing. The parallel topology of the distributed mesh can change dramatically during one computation: some new inter-processor boundaries may be created and some other ones may disappear. Entities that were classified

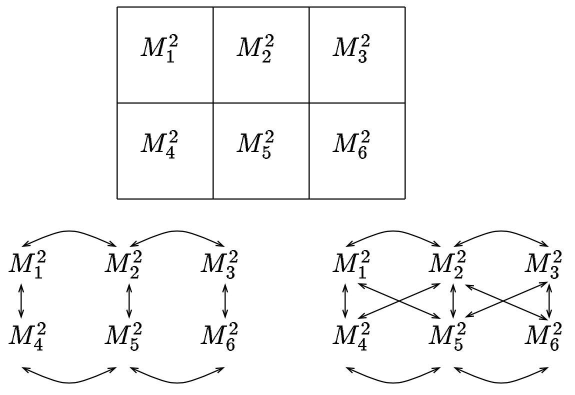

Figure 4: Mesh composed of 6 quadrilaterals and two different graphs for this mesh corresponding to edge-face adjacencies (left) and vertex-face adjacencies (right).

on inter-processor boundaries may be re-classified interior and conversely. Before doing any migration, we first have to re-classify mesh entities in order to be able to restore inter-proccessor links. As shown in Figure 5, the re-classification procedure involves two steps.

- The result of the application of a load balancing algorithm to a distributed mesh is, for each element of the mesh, a partition `iD` which represents the destination of the element. This is represented in Figure 5. We first look inside the present partition for destinations of all vertices. In Figure 5, vertices like M_k^0 will be shared by partitions P^1 and P^2. Vertex M_i^0 will be shared by the three partitions.

- Then we perform a round of communications in order to update the list of destinations with the list of destinations of all remote copies of vertices. In Figure 5, partition P^2 thinks that vertex M_i^0 will only be in P^2 after load balancing. After one round of communication, partition P^0 will communicate to partition P^2 that M_i^0 will also be on P^0.

It is clear that any other entity will know its destination by taking the intersection of vertices destinations. The edge connecting vertices M_i^0 and M_j^0 (Figure 5) that was internat to partition P^1 will be, after load balancing, in the interface between partitions P^0 and P^2. The concept of AOMD allows a straightforward algorithm to perform mesh migration. Because there exists a known minimal representation in AOMD, we first reduce the mesh

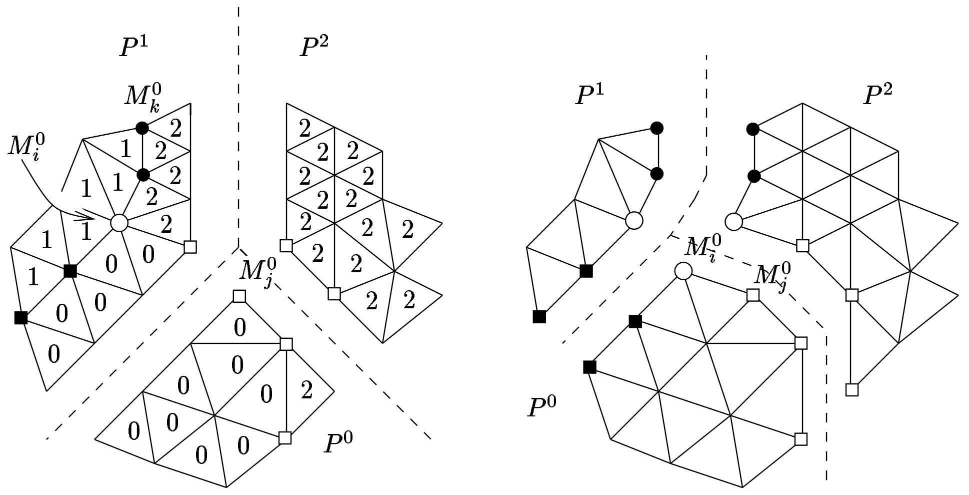

Figure 5: Illustration of a load balancing. On the left, triangles are tagged with their destination. On the right, final configuration after load balancing

representation to its minimum in order to minimize the amount of mesh entity data migrated. Then, entities are migrated. Vertices are migrated first, with their coordinates, `iD`'s and classification. Then, other mesh entities are migrated, with their vertices `iD`'s and classification. Then, one can rebuild any given mesh representation locally and efficiently using AOMD template operators [12] in serial. Finally, it is necessary to re-construct inter-processor adjacencies. For that, we simply call a general procedure that is described in a further section §5.1.

5 User interface, callbacks

The idea of a parallel mesh database is to provide some services to mesh users that are not experts in parallel programming. There are three services that involve parallelism. We describe here a `C++` callback-type interface for those services. The application of this interface does not require any parallel calls or refer to any particular parallel protocol like `MPI` or `OpenMP`. Therefore, applications can be developed with no consideration of parallel implementation issues.

5.1 Inter-processor communications

Inter-processor communications are done through partition boundaries. A partition boundary G knows the list of processors $P^k,\ k = 1, ..., N$ it is connected to. On processor P, if a mesh entity M_i^d is classified on a inter-processor boundary G, AOMD sends to all processors P^k the list of vertices labels of this entity together with a pointer $ptr(P, M_i^d)$ to the entity itself. Then, each processor P^k responds with the adress $ptr(P^k, M_i^d)$ of the mesh entity on its side. AOMD will then store these remote pointers $ptr(P^k, M_i^d),\ k = 1, .., N$ so that user can send efficiently a message to all its remote copies. This procedure of setting up inter-processor communication is done at startup and each time the mesh is modified. The user is then

```
class AOMD_RndOfComm
{
public:
 virtual void *sendBuffer (const meshEntity &localME,
                           int destProc, int &bufsize) = 0;
 virtual void *receiveBuffer (const meshEntity &remoteME,
                           int sourceProc, void *buffer) = 0;
};
```

Figure 6: Round of communication callback description.

able to pass messages to all it remote copies of a given mesh entity using the callback procedure of Figure 6. The user creates a particular round of communications by subclassing the abstract class
`AOMD_RndOfComm` from AOMD. In this, the user sends a buffer of size `bufsize` the remote copy of the mesh entity `localME` on processor `destProc`. If several remote copies of `localME` exist, one message per remote copy is sent. If user returns `0`, no message is sent. Then, the remote copy `remoteME` on processor `destProc` recieves the message `buffer` from processor `sourceProc`. Typically, one round of communications will look like Figure 7.

5.2 Mesh adaptation

AOMD provides another callback procedure for mesh adaptation (Figure 8). In this callback, the user must tell if a given mesh entity is too big, too small

```
AOMD_Mesh *theMesh;
...                        // load the mesh
theMesh->bdryLinkSetup(); // set up communications
class My_RndOfComm : public AOMD_RndOfComm // subclassing
{ ... };                                   // user code
theMesh->RndOfComm(My_RndOfComnc());
```

Figure 7: Typical set of instructions for doing a round of communications with AOMD.

or of good dimensions. Mesh adaptation consists basically in some mesh modifications which consequences are the replacement of a set of element (a cavity) by another one. The user may provide some actions to be performed when the mesh modification occurs. This can be for example a projection of the solution defined in the non-modified mesh into the modified mesh. The AOMD mesh adaptation callback is implemented as: If the callback

```
class AOMD_ADAPT_Callback
{
public:
 virtual int operator () (const meshEntity &e) const = 0;
 virtual void action (std::list<meshEntity*> & before,
                      std::list<meshEntity*> & after) const;
};
```

Figure 8: Mesh adaptation callback description.

`operator(const meshEntity &e)` returns 1, mesh entity `e` has to be refined, if it returns -1, the mesh entity has to be coarsened and if it returns 0, the mesh entity does not have to be modified. AOMD provides default behavior for the class member *action*: simply not doing anything at all. One adaptive refinement step with AOMD will look like Figure 9. As we have already mentioned in §3, a mesh adaptation algorithm based on non conformal splitting has been implemented. This procedure is an algorithm that operates on the top of AOMD kernel.

```
AOMD_Mesh *theMesh;
...
class My_ADAPT : public AOMD_ADAPT_Callback // subclassing
{ ... };                                     // user code
Non_Conformal_Adaptation(theMesh,My_ADAPT());
```

Figure 9: Typical set of instructions for doing a mesh adaptation with AOMD.

5.3 Load Balancing

In order to perform load balancing, we define a model for computational load. We mentioned in §4 that load balancers need a weighted graph. The load of a given processor P is defined as the number of elements in its partition multiplied by a weight that can be determined based, for example, on the elements computational demands. If no weight information is provided, AOMD sets all weights to 1 by default so that the load is simply proportional to the number of elements. The aim of load balancing is to balance loads between processors while minimizing inter-processor communications i.e. size of partition boundaries. The user can also provide weights for partition boundaries in order to take into account differing communication costs. Finally, user defined data can be migrated as well as mesh entities. The interface for load balancing is described in Figure 10. The interface looks very much the same

```
class AOMD_LB_Callback
{
public:
 virtual void *sendBuffer (const meshEntity &localME,
                           int destProc, int &bufsize) = 0;
 virtual void *receiveBuffer (const meshEntity &remoteME,
                           int sourceProc, void *buffer) = 0;
 virtual int vWeight (const meshEntity&);
 virtual int eWeight (const meshEntity&);
};
```

Figure 10: Load balancing callback description.

as the one for round of communications. Some default values are provided for weight functions as we discussed above so that base class members are not pure virtual functions. One load balancing with AOMD will look like Figure 11.

```
AOMD_Mesh *theMesh;
...
class My_LB : public AOMD_LBCallback // subclasssing
{ ... };                             // user code
theMesh->LB(My_LB());
```

Figure 11: Typical set of instructions for doing a dynamic load balancing with AOMD.

5.4 Message Packing

An important remark concerns the strategy for message passing. In all AOMD callback procedures, it appears that we send messages one by one which is usually not efficient because of the inherent latency of inter-processor communications. Messages are not sent one by one but are packed in order to optimize message size. The choice of message sizes strongly depends on the network architecture. We use *autopack*, a message packing library developed at Argonne by Ray Loy [10]. Note that this allows unexperienced users to use their hardware optimally without having to deal with packet size optimizations. For finding the optimum packet size, the AOMD user can easily tune this size parameter by doing some simple test cases.

6 Results

6.1 AOMD and Trellis

Trellis [2] is the computational framework developed at the Scientific Computation Research Center at Rensselaer. We present here the example of an elliptic problem computed with the higher-order finite element capabilities of Trellis. AOMD was shaped for each of these problems: linear tetrahedron only require element-node adjacencies while second order element require edge modes. Third order elements require also face modes. With AOMD, we were able to shape the database for each of the different computations.

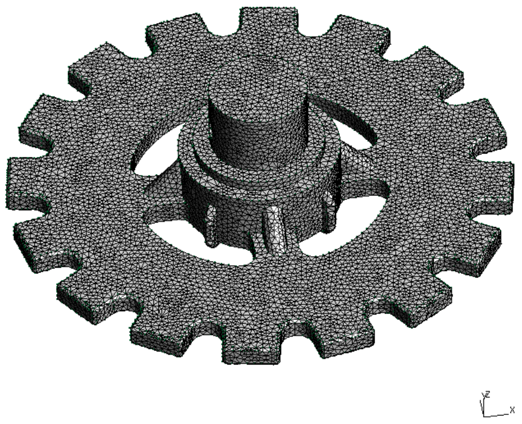

Figure 12: Mesh of a mechanical part

We have solved a linear elasticity problem on the $121,792$ tetrahedra mesh of figure 6.1. We held the shaft steady and applied a constant force distribution to one of the teeth. The resulting displacement distribution can be seen in figure 6.1. Memory requirements were as follows: 38 MB for the minimum representation, 66 MB with edges created and tet-edge adjacencies and 83 MB with faces created and tet-faces adjacencies. First order element computations were done on the minimum representation. For the computations using higher order elements the representation containing the edges and faces have been used, respectively.

6.2 A stand alone example of parallel AOMD

In this example, we take as input a two dimensional triangular mesh (Figure 14). After an initial partitioning into four partitions, we apply adaptive refinements and coarsening using the following rule. Let us consider a circular level set function

$$f(x,y,z,t) = (x-x_c)^2 + (y-y_c)^2 + (z-z_c)^2 - t^2.$$

We create a mesh refinement callback that asks for splitting all elements that have an intersection with $f = 0$. After each refinement, we apply load balancing to the refined mesh and change t to $t + dt$. This artificial problem

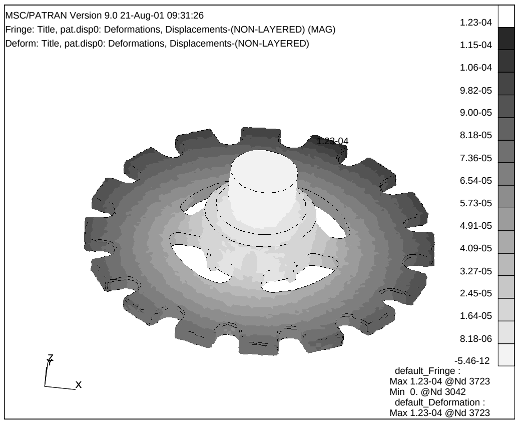

Figure 13: Deformation of a mechanical part

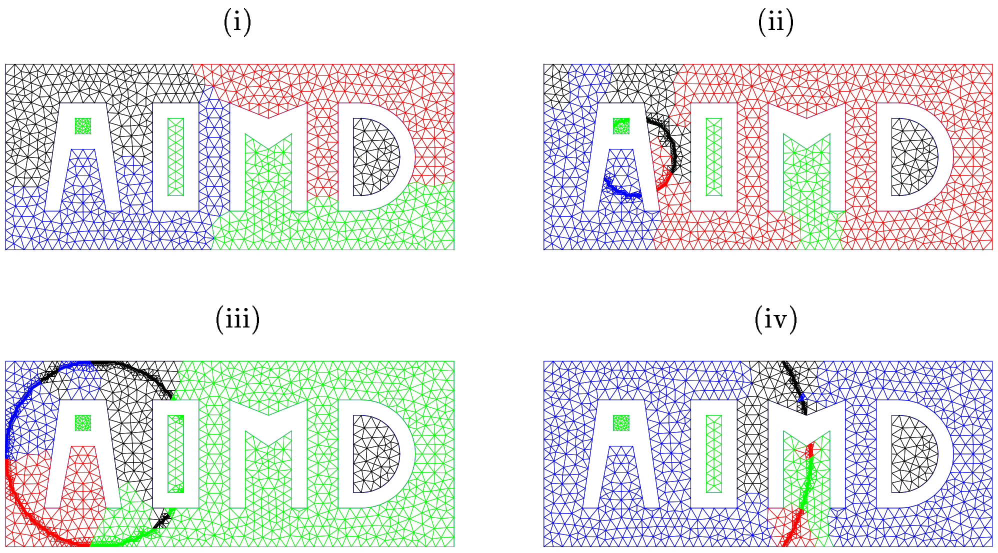

Figure 14: Mesh evolution for the artificial problem. Initial mesh (i) and refined meshes after 10 (ii) , 24 (iii) and 45 (iv) adaptive steps. Elements colors (black, blue, green and red) refer to mesh partitions.

may be seen as a model for the propagation of a cylindrical discontinuity in a medium. The source code for this example can be found on the AOMD web site at `http://www.scorec.rpi.edu/AOMD/LB.html`. Figure 14 shows resulting meshes and partitions for this artificial problem at four different time steps. This example shows how load balancing can affect topology of inter-connections between processors. We see that black and green partitions that were not topologically connected at stratup have a common boundary at iteration 24. At startup, there is one vertex that is common to 3 partitions (red, blue and black). It is then classified on an inter-processor model vertex. After 10 iterations, there are no inter-processor model vertex anymore. In this problem, we have used a graph-based load balancer. This is obvious by looking how the load balancing algorithm has dealt with islands inside letters `A`, `O` and `D` that are not topologically connected with the rest of the mesh.

Figure 15 shows a plot of the average load on all processors divided by the maximal load on all processors as a function of the adaptation step. Load balancing greatly improves this factor which is a good measure of the scalability of the computation.

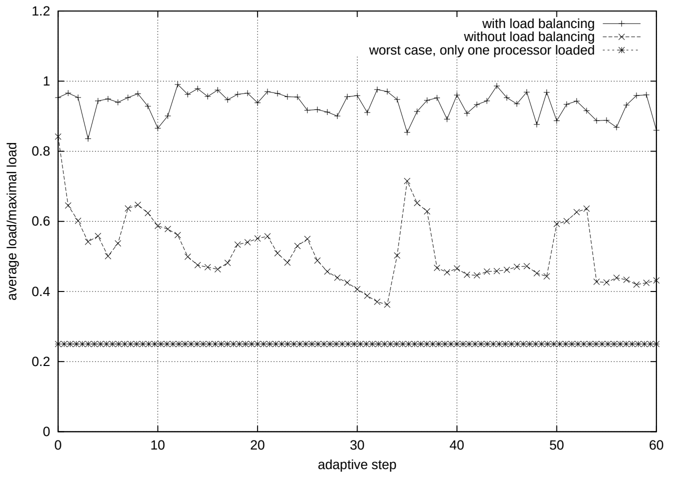

Figure 15: Comparison of mesh adaptation with and without load balancing.

6.3 Coupling AOMD with an adaptive solver

In [11], we have developped an adaptive discontinuous Galerkin method (DGM) for solving non-linear conservation laws. We have used AOMD as the mesh tool for our DGM. We present here some results for a 3-D Rayleigh Taylor instability. A Rayleigh-Taylor instability involves a heavy (cold) fluid overlying a light (warm) fluid [15, 7]. We consider two inviscid fluids initially in hydrostatic (unstable) equilibrium in a cavity. The upper half of the cavity is filled with a fluid of density two while the lower part is filled with a fluid of unit density. The initial pressure corresponds to hydrostatic equilibrium. An initial perturbation of the velocity initiates the instability. The perturbation is made of one Fourier mode which wave length is the width of the cavity. The flow is governed by the Euler equations of gas dynamics. The mesh refinement callbacks are based on the error indicator described in [11]. The initial mesh is a regular hexahedron mesh.

Our first set of results were computed on a 4 processor machine powered with 700 MHz Intel Pentium 3 processors. The computation started with 10^6 degrees of freedom and reached 6.5 10^6 degrees of freedom after 1.3 seconds of computation (Figure 16). Figure 17 shows the mesh after 6720 time steps

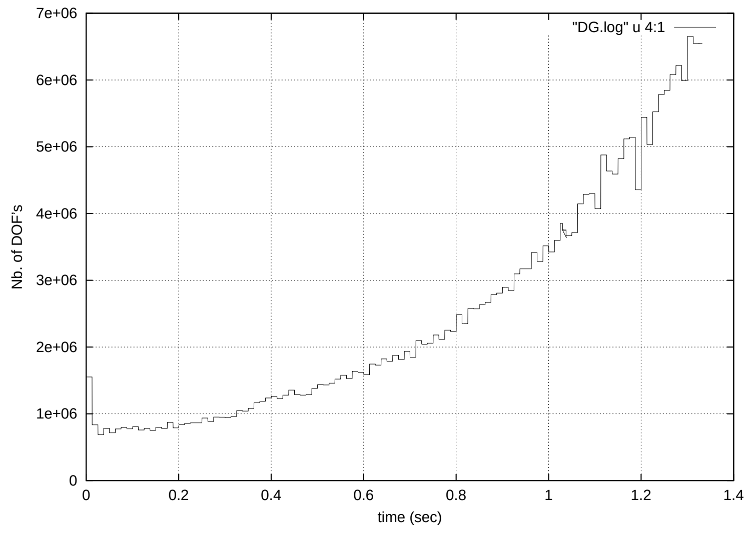

Figure 16: Number of degrees of freedom vs. time for the Rayleigh Taylor problem

and 104 refinement steps. In figure 18, (i) shows the part of the domain where the fluid density $\rho < 1.5$ and (ii) shows the part of the domain where the fluid density $\rho > 1.5$. Other computations were performed on 64 and 256 processors of Blue Horizon, an IBM supercomputer based at the San-Diego supercomputing center. On 64 processors, we have kept the single mode perturbation. On the 256 processors, we have used a random perturbation.

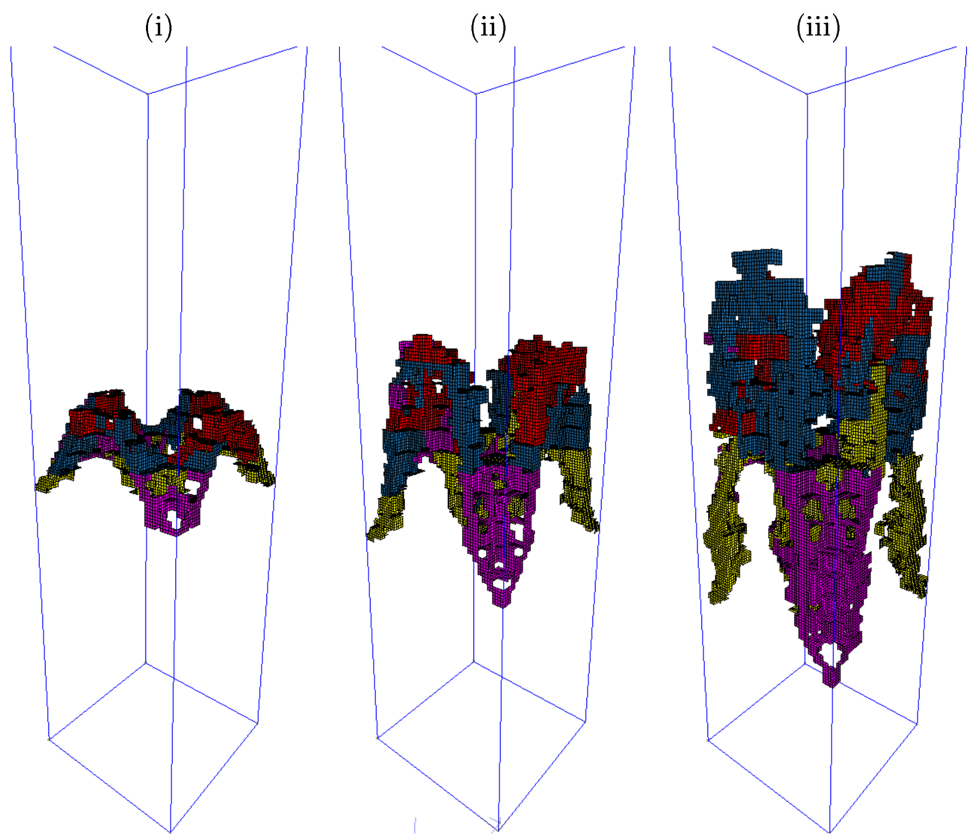

Figure 17: Pictures show the mostly refined elements in the mesh for the Rayleigh Taylor instability problem. Picture (i) shows mesh after 24 refinements, picture (ii) shows the mesh after 72 refinements and picture (iii) shows the mesh after 104 refinements. Element colors are for partitions `iD`'s.

Figuer 19 show some pictures of the two computations. On the 64 processors, we have reached 3×10^7 degrees of freedon and on 256 processors, we have reached 1.18×10^8 degrees of freedom, both after 10 hours of wall clock time.

7 Conclusions

The concept of AOMD proposed in [12] has been extended to distributed meshes. We have shown that the hypotheses we have made on mesh rep-

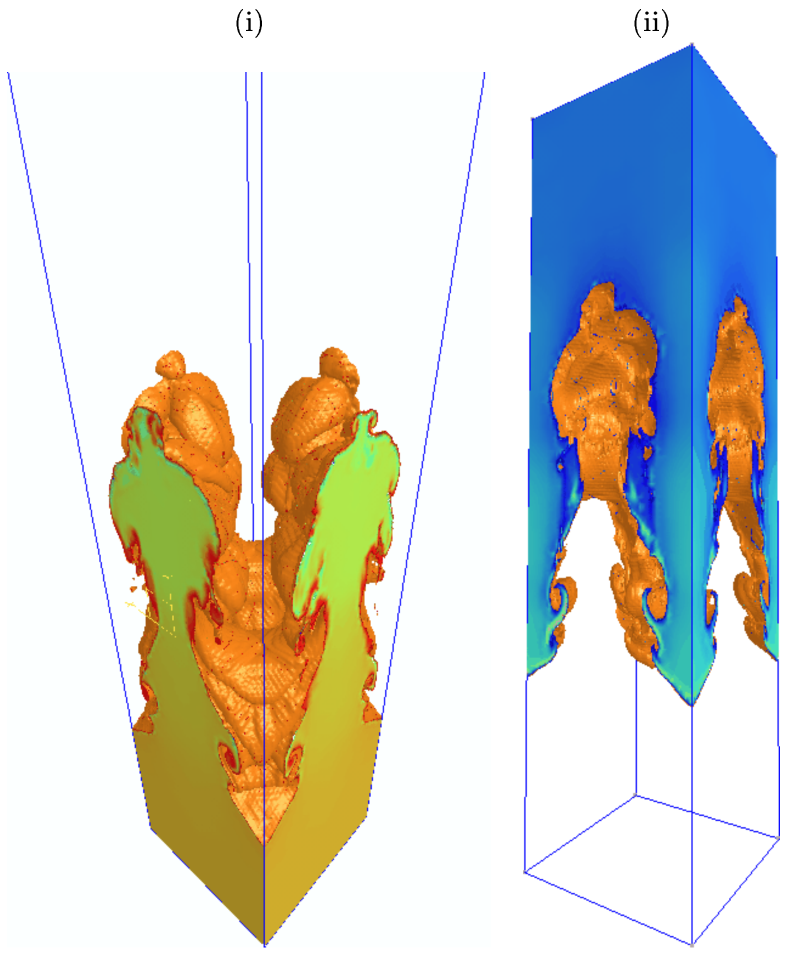

Figure 18: Representation of the fluid density ρ for the Rayleigh Taylor instability. Figure (i) shows the part of the domain where $\rho < 1.5$ and Figure (ii) shows the part of the domain where $\rho > 1.5$

resentations were applicable to a distributed mesh. AOMD and its parallel extensions is open source. It can be downloaded from the Internet at `http://www.scorec.rpi.edu/AOMD/`.

Currently we are working on predictive load balancing. The load model that we are using now is based on weights provided by the user. It is difficult for the user to take into account parameters like network heterogenity, relative speeds of processors or relative amount of cache memory available on different nodes. Supercomputers are usually made of a set of SMP boxes

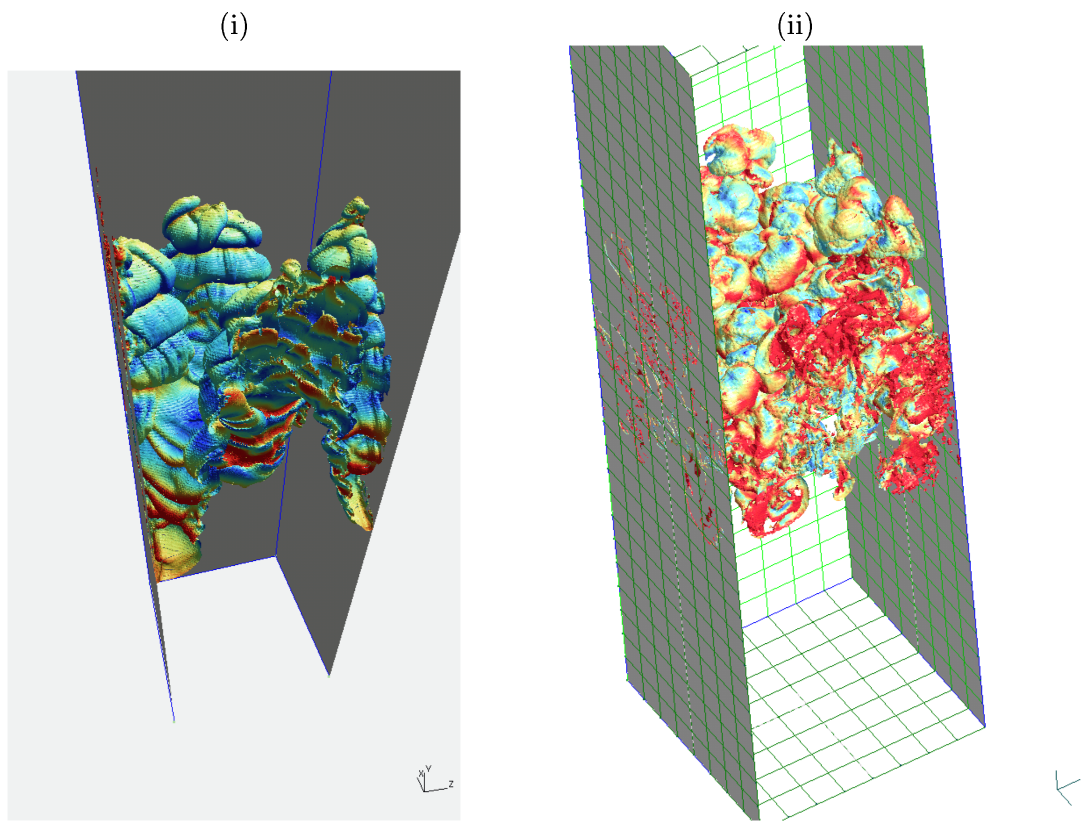

Figure 19: Iso-surface of the fluid density. Values of the modulus of the velocity are used for colors.

connected together with a fast network. The load balancer should be able to take into account that fact and create neighboring partitions for processors located in the same box, for example. We think that *a priori* models are not sufficient. Automatic agents based, for example, on time passed at barriers, should be added to the system. Alerts could be thrown to processors to ask for some load balancing when the system detects an unbalance.

References

[1] Beall, M. W. ; Shephard, M. S.: A General Topology-Based Mesh Data Structure. In: *International Journal for Numerical Methods in Engineering* 40 (1997), S. 1573–1596

[2] Beall, M. W. ; Shephard, M. S.: An Object-Oriented Framework for Reliable Numerical Simulations. In: *Engineering With Computers* 15 (1999), S. 61–72

[3] Boman, E. ; Devine, K. D. ; Hendrickson, B. ; Mitchell, W. F. ; John, M. S. ; Vaughan, C. : *Zoltan: A Dynamic Load-Balancing Library For Parallel Applications.* Sandia National Labs., 2001. – Zoltan's User Guide version 1.23, *available at* `http://www.cs.sandia.gov/Zoltan/`

[4] Carey, G. F. ; Sharma, M. ; Wang, K. C.: A class of data structures for 2-D and 3-D adaptive mesh refienement. In: *International Journal for Numerical Methods in Engineering* 26 (1997), S. 2607–2622

[5] de Cougny, H. L. ; Shephard, M. S.: Parallel Refinement and Coarsening of Tetrahedral Meshes. In: *International Journal for Numerical Methods in Engineering* 46 (1999), S. 1101–1125

[6] Dannelongue, H. ; Tanguy, P. : Efficient Data Structure for adaptive remeshing with FEM. In: *Journal of Computational Physics* 91 (1990), S. 94–109

[7] Glimm, J. ; Grove, J. ; Li, X. ; Oh, W. ; Tan, D. C.: The dynamics of bubble growth for Rayleigh-Taylor unstabe interfaces. In: *Phys. Fluids* 31 (1988), S. 447–465

[8] Hawken, D. ; Townsend, P. ; Webster, M. : The use of dynamic data structures in finite elememt applications. In: *International Journal for Numerical Methods in Engineering* 33 (1992), S. 1795–811

[9] Löhner, R. : Some useful data structures for the generation of unstructured grids. In: *Comm. appl. numer. methods* 4 (1997), S. 123–135

[10] Loy, R. : *AUTOPACK User Manual.* Science Division, Argonne National Laboratory, 2000. – Technical Memorandum ANL/MCS-TM-241, Mathematics and Computer

[11] Remacle, J.-F. ; Flaherty, J. E. ; Shephard, M. S.: An efficient local time stepping scheme in adaptive transient computations. In: *Submitted to SIAM J. Sci. Comput.* (2000)

[12] Remacle, J.-F. ; Karamete, B. K. ; Shephard, M. S.: Algorithm Oriented Mesh Database. In: *Ninth International Meshing Roundtable*, 2000

[13] Shephard, M. S.: The specification of physical attribute information for engineering analysis. In: *Engineering With Computers* 4 (1988), S. 145–155

[14] Teresco, J. D. ; Beall, M. W. ; Flaherty, J. E. ; Shephard, M. S.: A hierarchical partition model for adaptive finite element computations. In: *Computer Methods in Applied Mechanics and Engineering* 184(4) (2000), S. 269–285

[15] Young, Y.-N. ; Tufo, H. ; Dubey, A. ; Rosner, R. : On the Miscible Rayleigh-Taylor Instability. In: *Under consideration for publication in J. Fluid Mech.* (2000)