PARALLEL ALGORITHM ORIENTED MESH DATABASE

Jean-François Remacle [1], Joseph E. Flaherty and Mark S. Shephard

Scientific Computation Research Center,
Rensselaer Polytechnic Institute,
Troy, New York, USA.
Coresponding 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 [1] 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 a transient fluid dynamics problem involving thousand of mesh refinements, load balancings are finally presented.

Keywords: Parallel, mesh database, adaptivity, load balancing

INTRODUCTION

In a recent paper [1], 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 the AOMD to the needs of the algorithms. In [1], we discussed both construction and implementation of the 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 [2], 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 [1]. 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 an example, we will present a large scale three dimensional computational fluid dynamics example where AOMD is used as mesh library.

[1] This work was supported 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.

1. ALGORITHM ORIENTED MESH DATABASE

A mesh is a discretization of a geometrical domain, that consists of mesh entities of controlled size and distribution that have simple topology (hexahedron, tetrahedron...). The topology of a mesh is described with adjacencies between mesh entities. Meshes are used for scientific computa-

tion. 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 [4].

The aim of the Algorithm Oriented Mesh Database (AOMD) is to be a mesh management library (or database) that is able to provide a variety of services for mesh users. Note that the optimal form of the mesh representation is application dependant with different applications requiring different sets of mesh adjacencies. In [1], we presented an approach that was able to deal with any representation. For that, we made a certain number of hypothesis. These hypothesis are readily extendable to distributed meshes.

- To each vertex M_i^0 is associated a unique `iD`. On this paper, we extend this definition to distributed meshes. We suppose that there is a unique numbering of vertices for a whole distributed mesh. Note that this definition does not scale in the sense that the maximum vertex `iD` is not bounded when the number of partitions tends to infinity. If we choose to represent `iD`'s with unsigned long integers, the total number of vertices may not exceed $4.2\ 10^9$ for 32 bytes architectures and $1.8\ 10^{19}$ for 64 bytes architectures. In both cases, the limit seems to be sufficient for a majority of applications.

- 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 previous hypothesis, we always have access to mesh entity vertices so that it's 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.

- In order to maintain both representations of a model (solid model and the mesh), we have to maintain a direct link between every mesh entity M_i^d and the geometrical entity G_j^q (with $q \geq d$) it is discretizing. We call this association a *classification* of a mesh entity to a geometrical entity and we note it $M_i^d \sqsubset G_j^q$.

We have seen in [1] that those hypothesis allow us to build some efficient algorithms to search into the mesh database. We have shown that it was possible to construct efficient hashing functions for mesh entities so that insertion, deletion and searching operations into the mesh database were showing constant complexity behavior which means that their cost do not depend on the size of the mesh database. These properties of course remains applicable to distributed meshes because every partition is as a serial mesh.

A second interesting feature of AOMD is that it's possible to define a minimum representation of a mesh that allows generation of any other representation in a deterministic way i.e. only using integer operations (no geometrical tests). A *sufficient minimum* of data is that any mesh entity "equally classified" has to be present in the representation. It means that all entities M_i^d for which we have $M_i^d \sqsubset G_j^q$ 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 ones that are classified on model vertices need to be classified. With the minimum of information we have defined, all the other ones may be classified deterministically. We have presented in [1] an algorithm to complete any representation from the minimum one. The mimimum representation does not need to be modified for distributed meshes.

2. DISTRIBUTED MESH REPRESENTATION

We consider a mesh divided in pieces or partitions. 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 numbering 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 partitions 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 patrition boundaries must be present as well in the representation. This is a natural extension of the serial definition.

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.

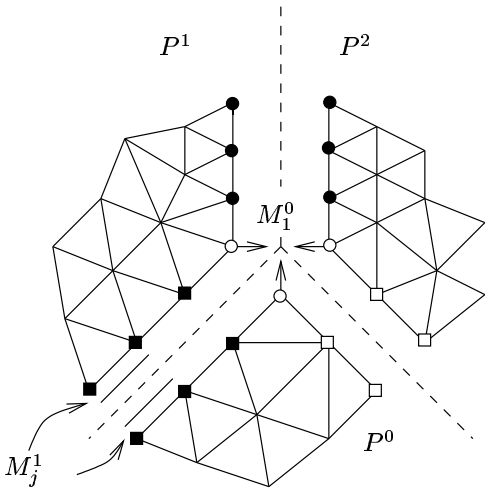

Figure 1. Distributed mesh, three partitions P^1, P^2 and P^3. Vertex M_1^0 is common to all partition and is then classified to a partition vertex. On each partition, several mesh edges like M_j^1 are common to two partitions and so is classified to a partition edge separating two partitions.

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 copies of itself 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 some mesh entities. We have developped 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 (Figure 2).

Mesh refinement consists of refinements and coarsenings. In mesh refinement, some entities are split. We use the concept of templates described in [1] which allow us to manage hybrid meshes. Splitting templates sends a request to the database for getting some new vertices `iD`'s. One round of communications is needed so that

- Entities that have remote copies must be split on all partitions in order to allow connection between partitions,

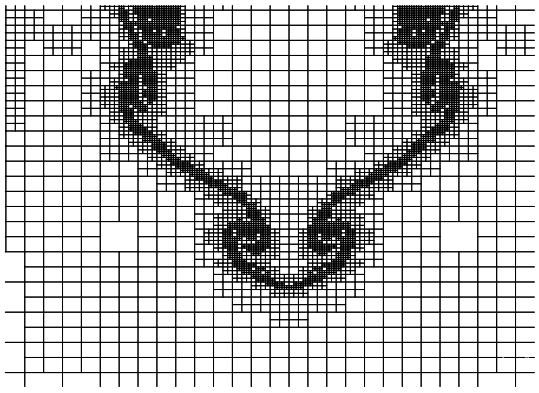

Figure 2. Adaptive non conformal quad mesh.

- Entities that have remote copies must be spilt while using the same vertices `iD`'s. This is the requirement to conserve the consistence of the mesh database.

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. For reasons of efficiency, we conserve the history of mesh refinement in a tree structure. 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.

4. DYNAMIC LOAD BALANCING AND MESH MIGRATION

Mesh refinement introduces load unbalance in partitions. This unbalance is not acceptable if one wants to achieve scalable parallel software. The solution to this unbalance is to dynamically repartition the mesh. Some load balancing libraries are available on the web. The library Zoltan [3] from Sandia is a package which includes four load balancing libraries based on both graph and octree partitioning. Classically, the balancer takes as input a representation of the parallel mesh (octree or partitioned graph) and provides as output a partition vector telling on which partition a given mesh entity has to be in order to restore the load balance. The completion of dynamic re-partitioning consists of dynamically moving the appropriate entities from one partition to another.

The concept of AOMD allows a straightforward al-

gorithm to perform mesh migration and recover a consistent distributed mesh representation. When each mesh entity knows its destination, it's possible to re-classify mesh entities. For that, we have to know in which partitions vertices will be present after migration. As shown in Figure 3, 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 3. We first look inside the present partition for destinations of all vertices. In Figure 3, 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 3, 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. Edge connecting vertices M_i^0 and M_j^0 (Figure 3) will be in the interface between partitions P^1 and P^2. Because there exist a known minimal representation in AOMD, we first reduce the mesh 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. Finally, one can rebuild any given mesh representation locally and efficiently. It is of course necessary to re-construct inter-processor adjacencies. Because each vertex knows in which partition it has remote copies, it is possible to know that information for all mesh entities. For that, we simply call the procedure described in §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 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 developped with no consideration of parallel implementation issues.

5.1 Inter-processor communications

Inter processor communications are done through partition boundaries. The user is able to pass messages to its remote copies.

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

The user can send a buffer of size `bufsize` to all its remote copies of the mesh entity `localME`. If several remote copies of `localME` exist, one message per remote copy is sent. This allows sending different messages depending on the destination processor `destProc`. Then, the remote copy `remoteME` on processor `destProc` recieves the message `buffer` from processor `sourceProc`. Typically, one round of communications will look like

```
AOMD_Mesh *theMesh;
...
class My_RndOfComnc :
 public AOMD_RndOfComnc
{
...
My_RndOfComnc foo;
theMesh->RndOfComnc(foo);
```

5.2 Mesh adaptation

AOMD provides simple callback procedure for mesh adaptation. The user has to provide one callback that will tell if a given mesh entity is too big, too small 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

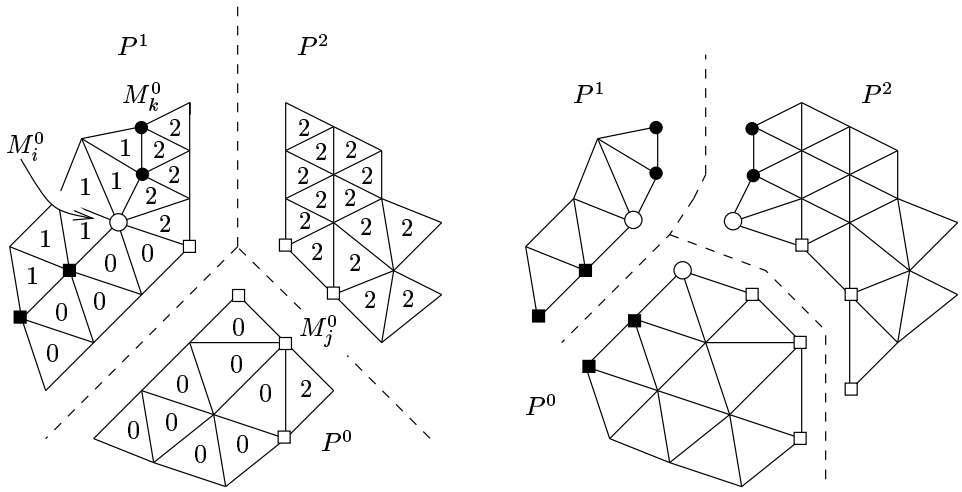

Figure 3. Illustration of a load balancing. On the left, triangles are tagged with their destination. On the right, final configuration after load balancing

AOMD mesh adaptation callback is implemented as:

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

If the function operator () returns 1, mesh entity has to be refined, if it returns -1, mesh entity has to be coarsened and if it returns 0, mesh entity does not have to be modified. AOMD provides default behavior for the class member *action*: simply not doing at all. One adaptive refienement step with AOMD will look like:

```
AOMD_Mesh *theMesh;
...
class My_ADAPT :
  public AOMD_ADAPT_Callback
{
...
My_ADAPT foo;
theMesh->ADAPT(foo);
```

5.3 Load Balancing

In order to perform load balancing, we define a model for computational load. The load is defined as the number of elements in a partition multiplied by a weight that can be determined based, for example, on the elements computational demands. All weights are 1 if the user does not provide anything. The aim of load balancing is to balance loads between processors while minimizing inter-processor communications i.e. size of partition boundaries. User can also provide weights for partition boundaries in order to take into account differing communication costs. Finally, user data's can be migrated as well as mesh entities. The interface for load balancing is then as follow:

```
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&);
};
```

The interface looks very much the same 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.

An important remark concerns the strategy for message passing. In all cases, it can look like we send messages one by one which is usually not efficient because of the inherent latency of inter-processor communications. Messages are packed

in order to optimize message size which depends on the network architecture. For that purpose, we use *a*utopack, a library developped at Argonne by Ray Loy [6]. Note that this allow unexperimented users to use its hardware optimally without having to deal with packet sizes optimizations.

One load balancing with AOMD will look like:

```
AOMD_Mesh *theMesh;
...
class My_LB : public AOMD_LBCallback
{
...
My_LB foo;
theMesh->LB(foo);
```

6. RESULTS

6.1 A stand alone example of parallel AOMD

In this example, we take as input a two dimensional triangular mesh (Figure 4). After an initial partitioning into four partitions, we apply adaptive refinements and coarsening using the following rule. Let us consider a circular levelset 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 will ask 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 may be seen as a model for the propagation of a cylindrical discontinuity in a medium. Setting up this problem required less that 100 lines of code with no specific parallel calls. Figure 4 shows resulting meshes and partitions for this artificial problem. Figure 5 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.

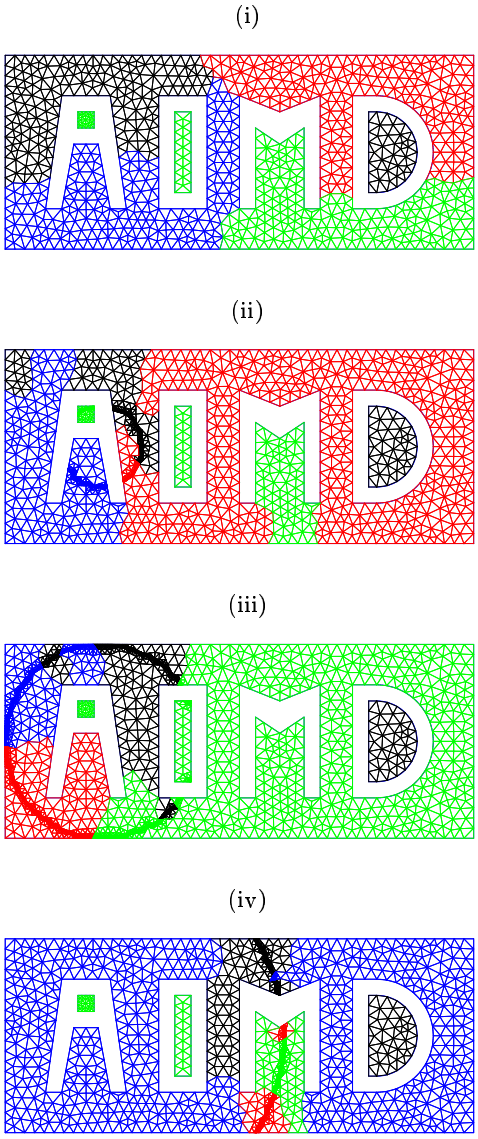

Figure 4. Mesh evolution for the artificial problem. Initial mesh (i) and refined meshes after 10 (ii) , 24 (iii) and 45 (iv) adaptive steps. Elements colors (black, blue, green and red) refer to mesh partitions.

6.2 Coupling AOMD with an adaptive solver

In [7], 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. The mesh refinement callbacks are based on the error indicator described in [7]. The initial mesh is a regular hexahedron mesh. 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 6). Figure 7 shows the mesh after 6720 time steps and 104 refinement steps. In figure 8, (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 $\rho > 1.5$.

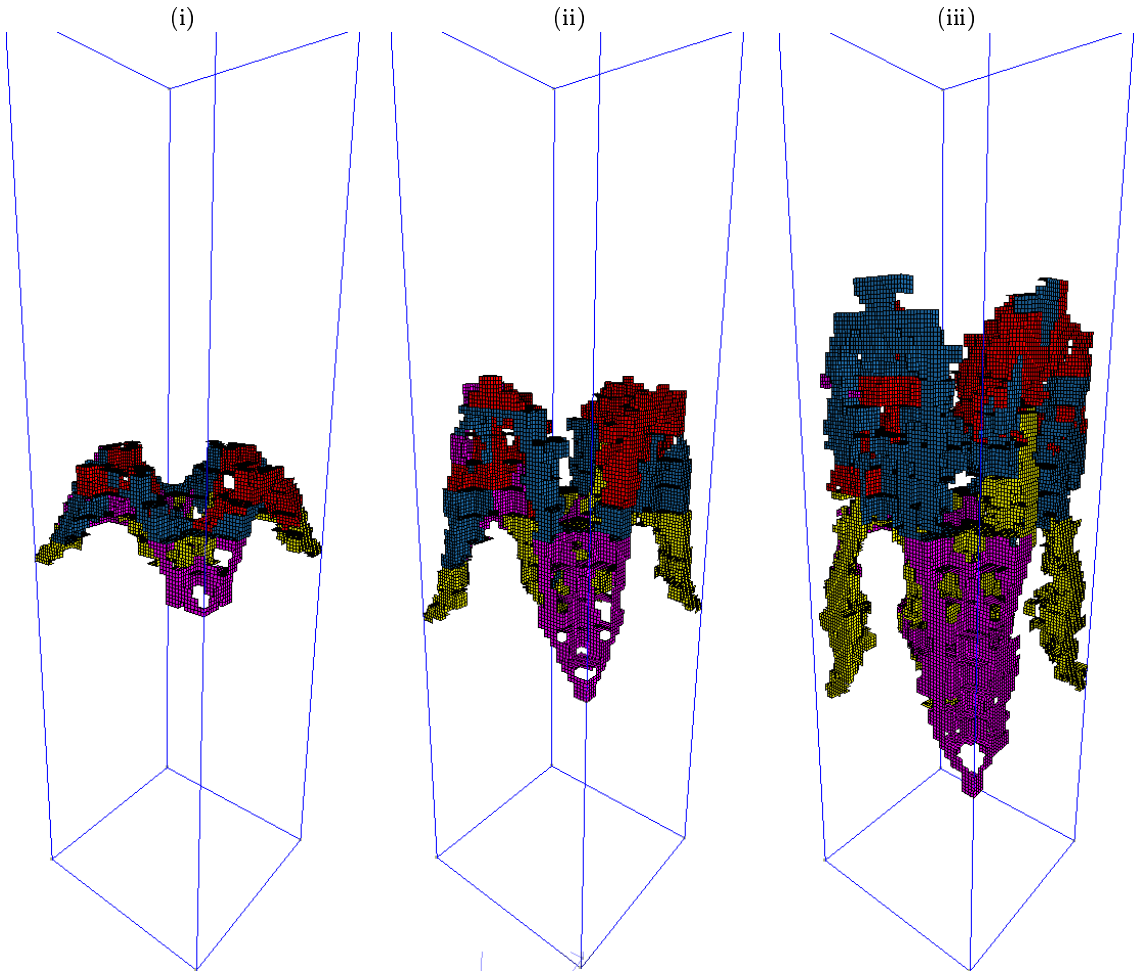

Figure 7. Pictures show the mostly refined elements in the mesh for the Rayleigh Taylor instability problem. Picture (i) shows mesh after 24 refinements, picture (ii) shows the mesh after 72 refinements and picture (iii) shows the mesh after 104 refinements. Element colors are for partitions `iD`'s.

7. CONCLUSIONS

The concept of AOMD proposed in [1] has been extended to distributed meshes. We have shown that the hypothesis we have made on mesh representations were applicable to a distributed mesh.

We are now working actively on predictive load balancing. We want to take into account the heterogeneity of the network in our calculations. Supercomputers are usually made of a set of SMP boxes connected together with a fast network. The load balancer should be able to take into account that fact and create neighboring partitions for processors into the same box, for example. Other issues like processor speed must also be taken into account.

Another interesting aspect concerns the design of the database itself. We use some advanced `C++` concepts for AOMD like `Iterators`, `Singletons`, `Mementos` and `Visitors` [8]. We are also interrested in the advanced dynamic allocation and thread `Policies` proposed in [9]. Some thread `Policies` enables locking procedures for simultaneous accesses to one mesh database. Because polymorphism always involves manipulation of arrays of pointers, it is impossible to allocate memory in big blocks in modern `C++` software design. `C` allocation (`malloc`) is fast and optimum for large chunks of memory but is inefficient when it is used for allocating lots of small objects. Reference [9] describes an allocation `Policy` that allows smart allocation for programs that manipulates small objects. We can also imagine a cache aware allocation `Policy` that trys to preserve mesh locality

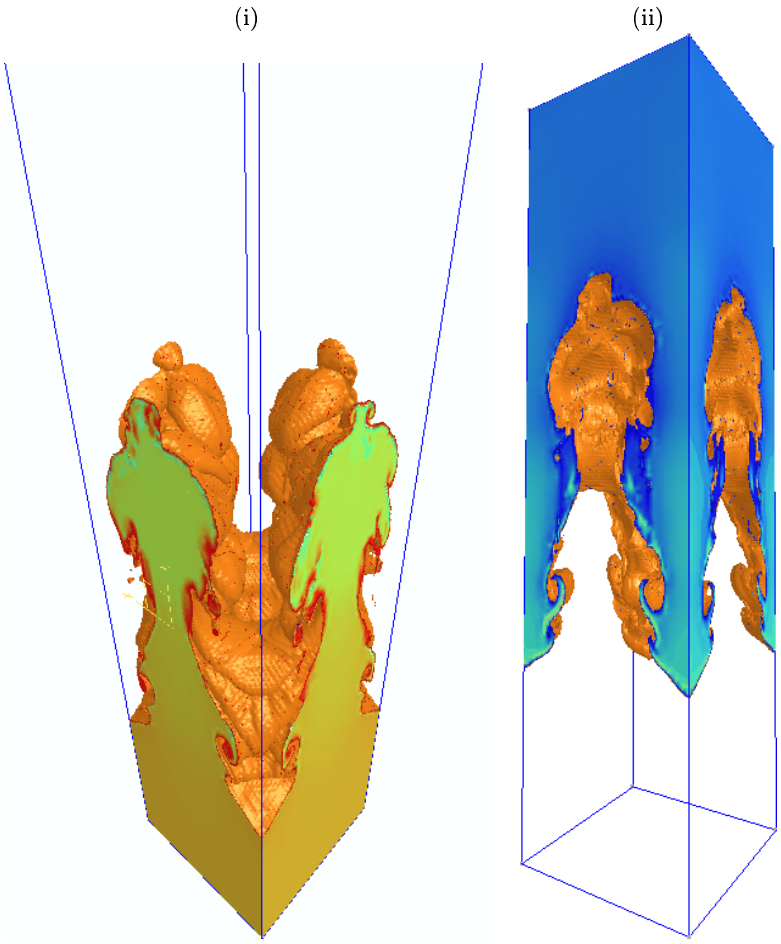

Figure 8. Representation of the fluid density ρ for the Rayleigh Taylor instability. Figure (i) shows the part of the domain where $\rho < 1.5$ and Figure (ii) shows the part of the domain where $\rho > 1.5$

in memory. Using the same idea's, AOMD user could be able to provide its own algorithms for refinement and coarsening. For example, conformal adaptation could be an alternate adaptation `Policy`. These advanced patterns will improve AOMD by making it more general and more efficient.

REFERENCES

[1] J.-F. Remacle, B.K. Karamete and M.S. Shephard, Algorithm Oriented Mesh Database, *Proceedings of the 9th international meshing roundtable*, SAND2000-2207, pp 349-359, New-Orleans, 2000.

[2] J.D. Teresco, M.W. Beall, J.E. Flaherty and M.S. Shephard, A hierarchical partition model for adaptive finite element computations, *Comp. Meth. Appl. Mech. Engng.*, 184(2-4):269-285, 2000.

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

[4] M.S. Shephard, The specification of physical attribute information for engineering analysis, *Engineering with computers.*, 4:145-155, 1988.

[5] H.L. de Cougny and M.S. Shephard, Parallel Refinement and Coarsening of Tetrahedral Meshes, *Int. J. Numer. Meth. Engng.*, 46:1101-1125, 1999.

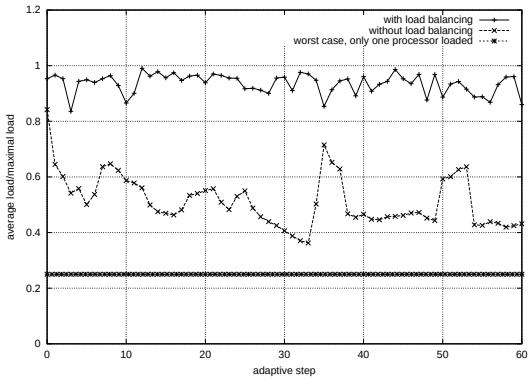

Figure 5. Comparison of mesh adaptation with and without load balancing.

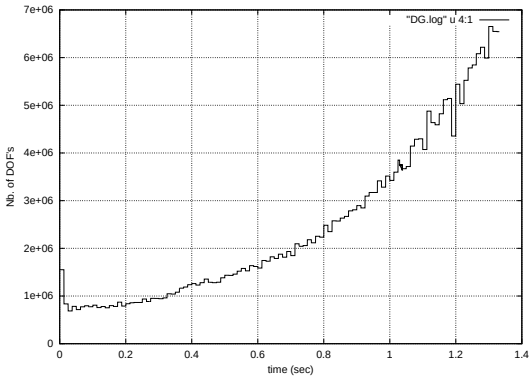

Figure 6. Number of degrees of freedom vs. time for the Rayleigh Taylor problem

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

[7] J.-F. Remacle, J.E. Flaherty and M.S. Shephard, An Adaptive Discontinuous Galerkin Technique with an Orthogonal Basis Applied to Rayleigh-Taylor Flow Instabilities, submitted to *SIAM Journal on Scientific Computing*, 2000.

[8] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns, Elements of Reusable Object-Oriented Software, Adison-Wesley Professional Computer Series, 1994.

[9] A. Alexandrescu, Modern C++ Design, C++ In-Depth Series, Adison-Wesley, 2001.