Parallel Volume Meshing using Face Removals and Hierarchical Repartitioning

H. L. de Cougny and M. S. Shephard
Scientific Computation Research Center
Rensselaer Polytechnic Institute, Troy, NY 12180-3590, USA

February 16, 1998

Abstract

Parallel unstructured three-dimensional mesh generation is a challenging problem for many reasons, the most obvious coming from the complexity of "partitioning" the problem such that meshing is load balanced at all times. In this paper, a parallel volume meshing procedure whose input is a surface mesh (distributed or not) is presented. In order to help driving the parallel execution, a distributed octree is built considering the surface mesh and meshing size attributes. The tree is partitioned in parallel using the Recursive Bisection methodology and "portions of space" are handed out to processors for meshing. Volume meshing operates on two techniques: (i) octant template meshing for interior octants and (ii) face removals (advancing front) to fill the space in between the surface mesh and the "templated" octants. Due to the mesh being distributed, domains corresponding to the interfaces of the initial partitioning are left unmeshed. To complete volume meshing, it is necessary to repartition the interface domains. A *hierarchical repartitioning* procedure has been developed to effectively mesh face, edge, end vertex interfaces. It is the focus of this paper.

1 Introduction

Scalable parallel computers have enabled researchers to apply finite element and finite volume analysis techniques to larger and larger problems. As problem sizes have grown into millions of grid points, the task of meshing models on a serial machine has become a bottleneck for two reasons: (i) it will take too much time to generate meshes and (ii) meshes will not fit in the memory of a single machine.

Parallel mesh generation is difficult because it requires the ability to decompose the domain to be meshed into sub-domains that can be handed out to processors. This is referred to as partitioning. Partitioning in the context of parallel mesh generation is hard because it has to be done with an input that is either a geometric model or a surface mesh. This means one is trying to partition a 3-d domain when only having the knowledge of its boundary, at least, initially. In contrast, it is much easier to partition a 3-d mesh, which is what finite element or finite volume parallel solvers typically do. Proper evaluation of the work load is also a challenge in parallel mesh generation. It is problematic to accurately predict the number of elements to be generated in a given sub-domain, or how much computation per element will be required. This leads to difficulties in maintaining good load balance at all times. Complexities also arise in the design phase due to the requirement that every single algorithm and data structure used should scale (with the number of processors). This is an issue that is often forgotten in the literature.

Parallel unstructured mesh generators presented to date all employ the concept of domain partitioning. Typically, a processor will be given the task to mesh a sub-domain. What differentiates the various approaches is how they treat the interfaces between sub-domains. Three classes of parallel unstructured mesh generators have been proposed:

1. those that mesh interfaces as they mesh the sub-domains,
2. those that pre-mesh the interfaces, and
3. those that post-mesh the interfaces.

For a more in-depth review of parallel mesh generation, the interested reader is referred to [4].

The first class of parallel mesh generators refers to those that neither pre-mesh nor post-mesh interfaces. Interfaces are meshed at the same time as

sub-domains. The parallel implementation of the Boyer-Watson algorithm [2, 20] by Chrisochoides and Sukup [3], and Okusanya and Peraire [12] are examples of meshing interfaces at the same time as sub-domains. The parallel Boyer-Watson algorithm as described by Chrisochoides and Sukup [3] delays the insertion of a point until the information needed to insert it (which has been requested) has been brought on processor.

In the second class, objects are partitioned in such a way that sub-domain meshing requires no communication. This is possible by meshing interfaces before the sub-domains. This class has been further subdivided into three sub-classes depending on how the partitioning into sub-domains is performed:

1. partitioning of an initial "coarse" mesh,
2. partitioning of a background tree, and
3. direct partitioning (pre-partitioning) of the input surface mesh.

In the context of initial "coarse" mesh partitioning, a commonly used method [8, 21] will (i) generate a coarse initial mesh, (ii) partition that coarse mesh into n_p sub-domains, (iii) refine interface edges of coarse mesh to proper sizes, (iv) distribute the sub-domains to the n_p processors, and (v) mesh sub-domains, Saxena and Perucchio [14] considers the interaction between a background octree and the model to be meshed to define a "discrete" representation (of the model) [16]. After the tree is partitioned, no communication is needed to mesh the octants which are either interior or boundary (interacting with the model's boundary). Triangulation conformity on shared octant faces can be guaranteed using the Delaunay criterion on planes. Galtier and George [9] pre-partition a surface mesh by triangulating appropriately placed separators. A separator cuts a domain into two parts. Given a surface mesh and a separator (say, a plane), the triangulation of the separator is such that (i) it separates, without modification, the initial surface mesh into two sub-surface meshes, and (ii) sizes of mesh entities on the separator are consistent with imposed sizes. The separator is not triangulated in the usual sense. The geometry of the separator is used to guide the meshing of the domain, defined by the input surface mesh, in the vicinity of the separator. The triangulation associated with the separator is made of triangles. In other words, a separator and its associated triangulation have same dimension. The input surface mesh is a priori "subdivided" into as many sub-domains as there are processors.

In the third class, sub-domains are meshed and interfaces are left out for later processing. This technique was used first by Shostko and Löhner [18]. Given an input surface mesh, a background grid is built serially on one (host) processor. Its primary role is to control parallel execution. After the grid has been partitioned (by the host processor), tasks are handed out by the dedicated host processor. On a given processor, the advancing front method is used to mesh the sub-domain defined by the background grid elements that have been handed by the host. To prevent overlapping of sub-meshes coming from different processors, a mesh region will not be created if it crosses the sub-domain's boundary. After the processors have created the mesh regions within their respective sub-domains, the space in between the meshed sub-domains remains to be meshed. The "skeleton" of this empty space is made up of the interfaces between the sub-domains. In 3-d problems, there are three types of interfaces. They are, in this paper, referred to as (i) face, (ii) edge, and (iii) vertex interfaces, depending upon dimensionality. Repartitioning is necessary to complete meshing. The parallel mesh generator presented by de Cougny et al. [5] also uses an advancing front method to mesh the volume in between a surface mesh and template-meshed interior octants. Given a distributed surface mesh, this procedure builds a "background" octree that satisfies meshing size requirements on one processor which is then duplicated on all other processors. Once interior terminal octants (far away enough from the boundary) have been meshed, the domain between the input surface mesh and the meshed octants is meshed in parallel by applying face removals. A face removal is the basic operation in the advancing front method which, given a front face, creates a mesh region. A front face is not removed if the tree neighborhood from which target vertices are drawn is not fully present on processor. Face removals are applied until there is no front face that can be removed. At that point, the tree is repartitioned only considering the terminal octants that interact with the current front. The process of applying face removals and repartitioning the tree continues until the front is empty.

The aim of this paper is to present a complete parallel scalable volume meshing procedure. The presented mesh generator (i) uses a tree as a parallel driver and (ii) post-meshes interfaces. It builds upon the work by Shostko and Löhner [18] and de Cougny et al. [5]. Focus is given to the meshing of interfaces that result from the initial partitioning, since it is believed to be the "weak" part of an "advancing front-based" mesh generator. It is assumed a *Distributed Memory* parallel architecture is being used, communication

being handled by a *Message Passing Library* like the standardized *Message Passing Interface* [11]. Note that the *Distributed Memory* architecture can be emulated by a *Shared Memory* parallel computer. Section 2 describes the (serial) volume mesh generation approach that has been parallelized. Section 3 explains the distributed octree structure and briefly presents a parallel tree building procedure. Section 4 describes the parallel version of the volume meshing procedure. The interface repartitioning strategy, referred to as "hierarchical repartitioning", is presented in section 5. Results are grouped in section 6. Finally, section 7 offers some concluding remarks.

2 Volume Mesh Generation

2.1 Introduction

Figure 1 graphically depicts the basics of the current mesh generator. The first step in meshing a model is to develop a variable level octree consistent with the triangulation on the boundary of the model and any other a priori mesh control information. Octants containing mesh entities classified on the boundary of the model are approximately of the same size as the mesh entities they contain. A one level difference on octants sharing an octant edge is enforced to control smoothness of the mesh gradations. Once the octree is generated, terminal octants which intersect boundary mesh entities are classified boundary. Terminal octants which are "too close" to boundary mesh entities are classified boundary as well in order to avoid the creation of interior mesh entities, coming from the application of pre-defined meshing templates, in close proximity of the model boundary. This is of concern since having mesh entities "too close" to each other may lead to the creation of poorly shaped elements in that neighborhood. A terminal octant is "too close" to a boundary mesh entity if the distance between the two is below the size of the mesh entity times some threshold smaller than unity. The remaining terminal octants are classified interior or outside. Those classified outside receive no further consideration. Interior terminal octants are meshed using templates [15]. Face removal procedures are then used to connect the boundary triangulation to the interior octants. Figure 2 graphically describes a face removal in a two-dimensional setting. Face removals are the basic operation in the commonly known advancing front method [10]. The input to the presented volume mesher is a non self-intersecting (within some tolerance) boundary mesh (initial front). As long as the boundary mesh defines one or more volumes without any ambiguity, the volume mesher can function correctly. In the general case, the boundary mesh consists of mesh faces, disconnected mesh edges, and disconnected mesh vertices. The term "disconnected" means here "not connected to any higher order mesh entity". Disconnected mesh edges may come from (i) "dangling" model edges (not connected to model faces) or (ii) the user himself who has imposed them. Disconnected mesh vertices may come from (i) "floating" model vertices (not connected to anything else) or (ii) the user himself. Note that some boundary mesh faces may also have been imposed by the user. Details about the "serial" volume meshing procedure can be found in [4]. Since focus is given

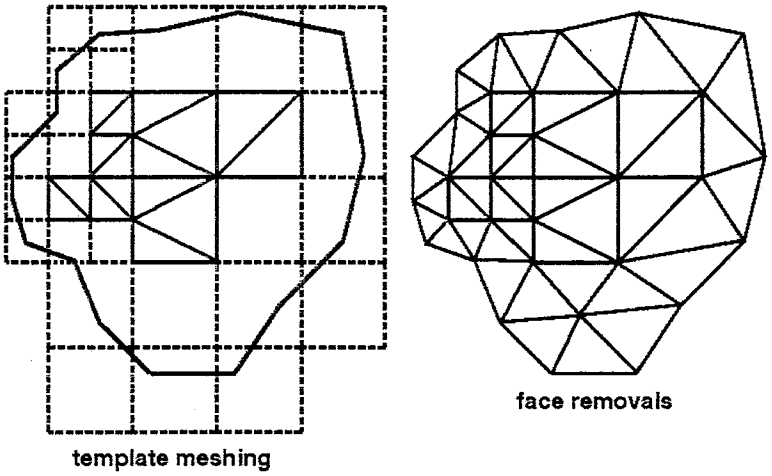

Figure 1: Basics of the presented mesh generator.

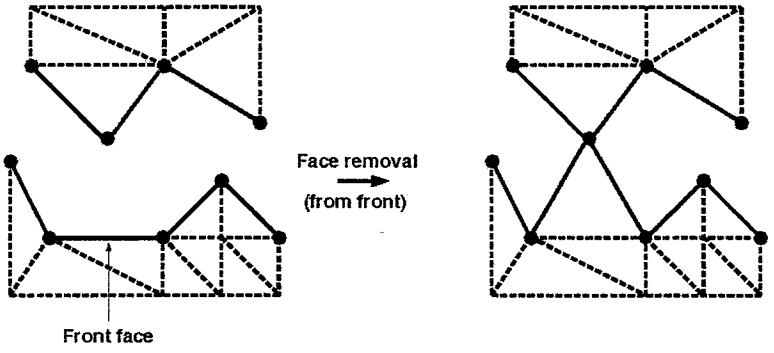

Figure 2: Face removal.

in this paper on the repartitioning of interfaces, only relevant procedures are described. The next sub-sections describe how the tree is built and how face removals are performed in a serial setting. Parallel versions for both tree building and face removals are presented in sections 3 and 4, respectively.

2.2 Tree Building

An octree is built for two main purposes (in the serial case): (i) data localization during face removals, and (ii) template meshing of interior terminal octants. The tree in its distributed version (described in section 3) will also drive the parallel execution of the volume meshing procedure. The input to

the tree building procedure is the surface mesh which represents the initial front. The tree building procedure is described below in pseudo-code form.

```
buildTree()
Get bounding box for front
Build root octant
for each front vertex do
   Get average length of connected edges
   Transform length into a tree level
   Find terminal octant that contains the point
    associated with the vertex
   refineOctant(octant,point,level)
endfor
```

```
refineOctant(octant,point,level)
get octant's level octantLevel
if octantLevel < level then
   subdivide octant into 8 child octants
   find child octant childOctant that contains the point
   refineOctant(childOctant,point,level)
endif
```

The bounding box for the front is obtained by querying the geometric modeler about the bounding box of the full model under consideration. In case some front edges on the model's boundary need to be split, the front is guaranteed to always be within the octree. For simplicity, the root octant is assumed to be cubic. The level of a terminal octant is defined as its depth in the tree. The root is at level 0. Given a length d, the associated tree level is given by:

$$level = \log_2(D/d)$$

where D is the dimension of the root octant.

The maximum 2:1 level of difference rule [16] is applied to the tree in order to: (i) ensure a smooth gradation between the input surface mesh and the mesh resulting from the templates, and (ii) be able to apply meshing templates to interior terminal octants which assume this particular state of the tree. This rule is enforced by making sure that, for any terminal

octant, for any octant edge, the level difference between the octant under consideration and any terminal octant sharing that edge does not exceed one. In practice, if a neighboring octant is at a level which is "too low" (differ by more than one with the octant under consideration), it is subdivided. Initially, all terminal octants are put into a processing queue. Terminal octants are dequeued and processed in turn. When a terminal octant is subdivided, the resulting child octants are pushed into the queue to make sure they will be processed. This procedure is described in detail by Yerry and Shephard [22].

2.3 Tree-supported Face Removals

2.3.1 General Algorithm

A face removal (from the front) consists in connecting a front mesh face, referred to as a base face, to a target mesh vertex, creating a mesh region. The face becomes fully connected, assuming it needs only one connection, and is therefore removed from the front. It is the basic operation in the advancing front method [10]. Possible target vertices for a base face are drawn from a "neighborhood". This neighborhood can be defined as a region of space in the vicinity of the base face. Once a neighborhood is defined, the underlying octree is used to efficiently localize the front vertices that should be considered as potential targets. Once the targets have been identified, they are ordered with respect to some criterion. Any target that would lead to the creation of a region with negative volume (upside-down) is immediately discarded. Targets are then processed in order until one, if any, satisfies the face removal validity checks. It should be noted that, at this point, additional vertices are not added into the triangulation when removing faces. It is assumed that, a priori, all interior vertices have been provided by template meshing. Face removals are performed in no particular order until the front is empty or all remaining front faces cannot be removed. The following pseudo-code describes the tree-supported face removal of a given front mesh face.

faceRemoval(face)
Define neighborhood from which target vertices are drawn
Use tree to gather possible target vertices

```
Order target vertices with respect to some criterion
for each target vertex do
  if new region does not intersect front then
    Create mesh region connecting face to vertex
    Update front
    end
  endif
endfor
```

Given a front face to remove (base face) and a target vertex, the new region which connects the base face to the vertex has to be checked for interference with the current front. The creation of a new region involves the creation of up to three new edges and up to three new faces. A new region should not be created if (i) it contains an existing front vertex, or (ii) any bounding new edge intersects an existing front face, or (iii) any bounding new face intersects an existing front edge. Note that existing front entities, the new region should be checked against, are obtained using the tree as a localization structure. Details about face removal validity checks can be found in [4] and [10].

Upon completion of the tree-supported phase of face removals, what remains to be meshed are "small" disconnected sub-domains. These "small" sub-domains can be meshed without the support of a tree since there is no need for localization. Given a sub-domain defined by its bounding mesh faces (sub-front), this phase adds to the previously discussed face removal procedure the possibility to create mesh vertices, delete previously created mesh regions, and split front mesh edges. Reomovals in the context of "small" sub-domain meshing is referred to as "tree-less" face removals. This is a critical part since it involves domains that are "difficult" to mesh. Details about "tree-less" face removals can be found in [4].

Figure 3 shows the main steps of the presented volume meshing procedure on a "real" example:

a. input surface mesh (sphere refined at equator).

b. front after template meshing of interior octants (only half of the sphere is shown). The region of space remaining to be meshed is between the surface mesh and the "templated" octants.

c. front after tree-supported face removals have been applied. What remains to be meshed are "small" domains.

d. mesh after the "small" domains have been triangulated (only half of the sphere is shown).

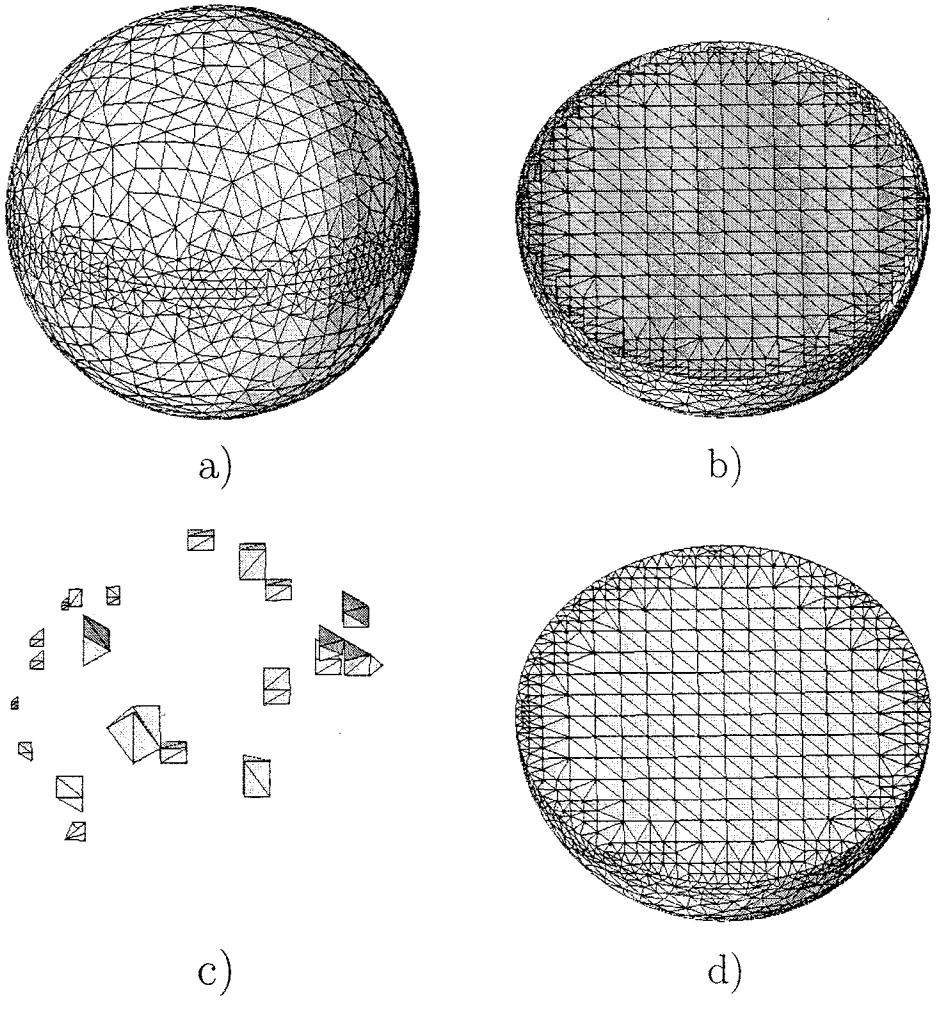

Figure 3: Steps of presented volume mesh generator on a real example.

3 Distributed Tree Building

3.1 Introduction

In order for the presented parallel volume mesher to be memory scalable, the mesh and octree data structures should be distributed. The parallel mesh data structure builds upon (i) a serial mesh data structure where regions, faces, edges, and vertices are represented in a hierarchical manner [1] and (ii) a methodology to "link" together mesh entities on partition boundaries that belong to more than one processor [13, 17]. In the past [5], a serial octree data structure (duplicated over all processors) was used. This is not a scalable construct. A distributed octree data structure has been developed to provide memory scalability to the parallel volume meshing procedure [19]. A distributed data structure complicates the process of tree building. The complication comes from the fact that the tree building procedure needs to be time scalable. The following sub-sections present the distributed tree data structure, and briefly describes a scalable tree building procedure.

3.2 Distributed Tree

A distributed tree requires some added data to be stored past that of a sequential tree. If the child or parent of an octant is not on processor, the octant's downward or upward link, respectively, contains a pointer to an off-processor octant along with the appropriate processor id. A link to an off-processor octant is referred to as an off-processor link. If an octant's parent is not on processor, a structure known as a "local root" exists. The "local root" is an octant with no parent on that processor. Any "local root" has knowledge of its origin and level. Local roots are stored in a linked list. Figure 4 shows a distributed binary tree on two processors. Nodes which are on processor are represented by bullets. Local roots are represented by squares. A downward or upward link is shown dashed when the child node or parent node, respectively, is not on processor. Details about the parallel octree data structures and related algorithms can be found in [19].

The tree building procedure presented here is based upon the *Divide and Conquer* paradigm which calls for the recursive subdivision of a problem. Figure 5 shows a typical application of this paradigm.

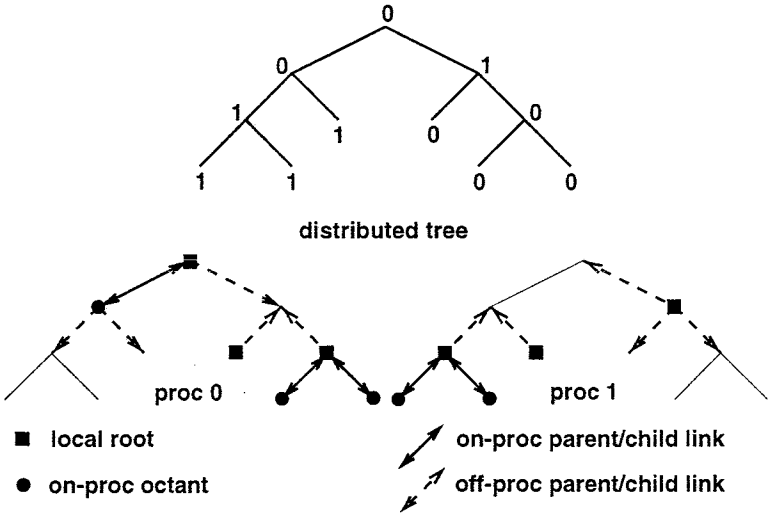

Figure 4: Distributed tree.

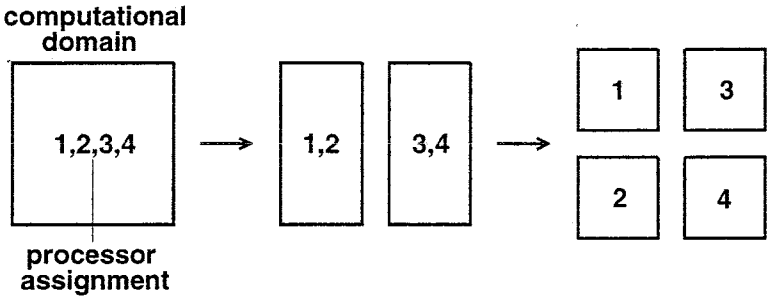

Figure 5: The *Divide and Conquer* paradigm.

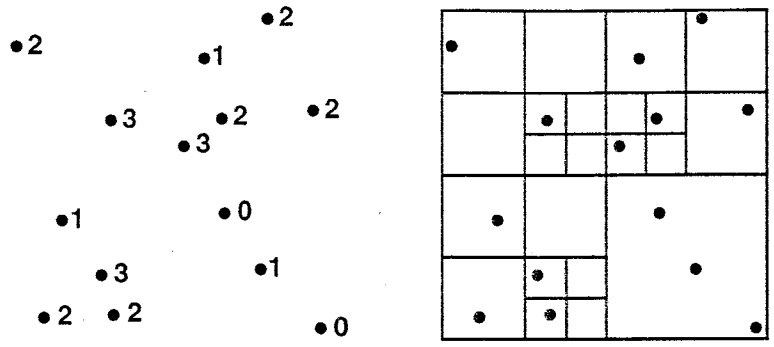

Figure 6: Tree building.

3.3 Algorithm

The input for the tree building procedure is a set of locations in 3-d space each associated with a desired tree level. This input is referred to as the *(point,level)* array. When considering the "serial" tree building procedure of section 2, each front vertex location along with the tree level is an entry of that array. It is assumed to be distributed, that is, each processor has its own sub-array. The resulting tree is distributed. It is such that, for any input point, the level of any terminal octant that contains it is greater than or equal to the level associated with the point. This is a general procedure that can be used in other contexts than volume meshing. Figure 6 shows a two-dimensional example for tree building given an input *(point,level)* array. In this figure, each input point is associated with a tree level.

Initially, the root octant is built on all processors available. Assuming it needs to be subdivided (due to at least one level greater than zero), each processor subdivides once the root octant. The set of available processors is split into sub-sets and each sub-set is given the task to further subdivide (if needed) one or more of the current terminal octants. Each processor sub-set can work independently of each other. Because each sub-set can be split further, sub-sets are simply referred to as sets for clarity. Given a processor set, current terminal octants are subdivided once (on each processor in the set) if their levels are too low. The processor set is then split into sub-sets and the current terminal octants are associated with the sub-sets for possible further subdivision. This process continues until all processor sets are of cardinality one. It should be well understood that this "coarse" tree building phase is performed in parallel (processor sets work independently of

each other), key to its scalability. How processor sets are split into sub-sets and how current terminal octants are assigned to processor sub-sets depend upon the "load" associated with a current terminal octant. The load of a terminal octant is defined as the sum of the levels associated with points within the octant. This load is strictly with respect to tree building. In other words, one considers this load definition in order to balance the tree building process. Once all processor sets are reduced to single processors, each processor stores a "coarse" tree. The top portion of figure 7 shows the tree (in this example, binary) stored by each processor (among the four) once all processor sets have been reduced to single processors. Note that a processor set reduced to a single processor can now be simply referred to as a processor. The "bulleted" node shown on each processor's tree represents the terminal node assigned to the processor. In this simple example, only one terminal node has been assigned to a processor. Given a processor, the terminal octants under its control are the roots of soon to be refined sub-trees. The *(point,level)* array stored by a given processor is such that any point is contained in a terminal octant under its control. This enables any processor to build its sub-tree(s) using the same methodology as in the serial case without having to communicate. The bottom portion of figure 7 indicates (with triangles) the sub-trees that have been built on each processor (among the four). Once the sub-trees have been built, the "global" tree is fully distributed. Details about this procedure can be found in [4].

Figure 8 shows the two stages of tree building on a simple "real" example. The left picture corresponds to the tree building phase which operates on processors sets. The right picture corresponds to the second phase, when each processor builds sub-trees without communication. The octant sizes correspond to the surface mesh edge sizes. This particular run was performed on four processors. One can see on the left picture that each processor ends up being in control of two terminal octants once the processor sets have been reduced to single processors. Each processor then builds two sub-trees rooted at those octants.

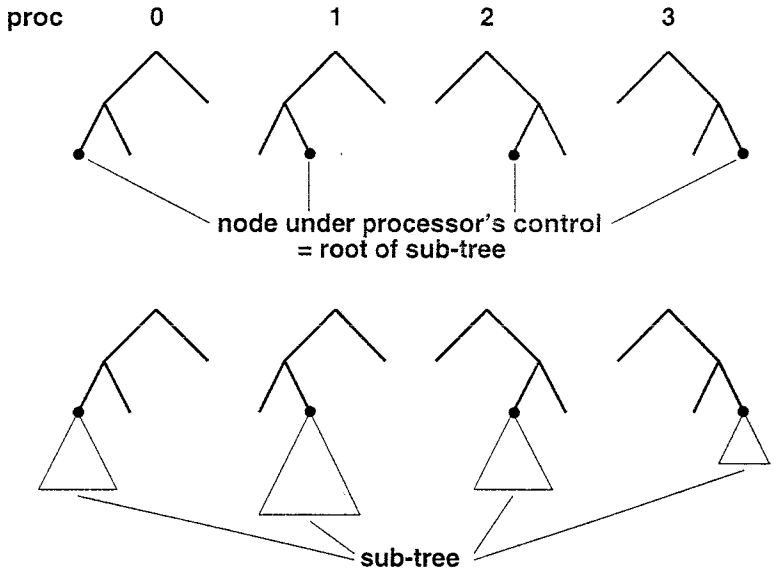

Figure 7: Distributed tree building.

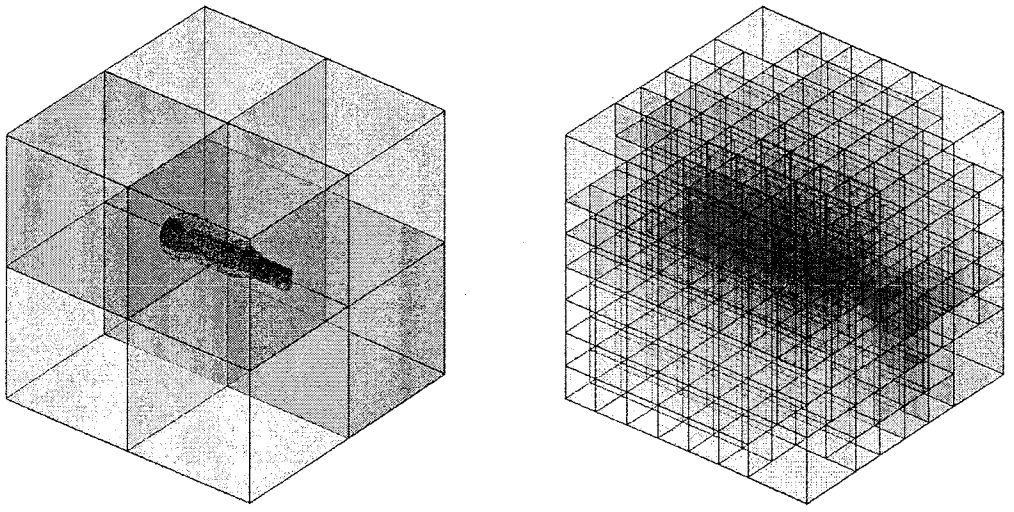

Figure 8: Tree building example.

4 Parallel Volume Meshing

4.1 Introduction

The input to the parallel volume mesher is a surface mesh. This input is either already distributed among the processors (e.g., when using the volume mesher after a parallel surface mesher [4]), or read from a surface mesh file (when using the volume mesher standalone). The complete parallel volume procedure can be decomposed into several main steps:

1. build distributed tree,
2. apply templates to interior terminal octants,
3. partition boundary terminal octants,
4. apply tree-supported face removals,
5. repartition face interfaces,
6. apply tree-supported face removals,
7. repartition edge interfaces,
8. apply tree-supported face removals,
9. repartition vertex interfaces,
10. apply tree-less face removals,
11. repartition mesh regions, and
12. optimize distributed mesh with respect to quality.

The partitioning of octants enables the domain to be decomposed into sub-domains for parallel processing. The partitioning methodology used in the construction of the initial partition is a parallel Recursive Bisection method. Three types of Recursive Bisection method can be used here:

- Recursive Orthogonal Bisection (ROB) [6, 7],
- Recursive Inertial Bisection (RIB) [6, 7], and
- Modified Recursive Orthogonal Bisection (MROB).

With ROB, the cutting axes are the coordinate frame axes. The directions (for the axes) are switched at each step, that is, axes are chosen in the order $(x, y, z, x, \ldots)$. With RIB, the cutting axis is the principal axis of inertia for the domain under consideration. MROB comes in between ROB and RIB. With MROB, the cutting axis is chosen to be the coordinate frame axis "closest" (with respect to direction) to the axis of inertia. Note that, implementation-wise, ROB and MROB are variants of the RIB procedure. Details about a parallel implementation of RIB can be found in [4]. In this paper, favor has been given to the MROB procedure since it produces boxed partitions which work well with an underlying octree structure. ROB should not be used since this methodology never takes into account the geometry of the domain to partition. This partitioning strategy using Recursive Bisection can be applied on octants as well as mesh regions. In practice, upon completion of the volume meshing procedure, mesh regions regions are repartitioned using the RIB methodology. Template meshing is briefly discussed in the next sub-section. Parallel tree-supported face removals are described in the third sub-section. Repartitioning of face, edge, and vertex interfaces is discussed in details in section 5. Parallel mesh optimization is not discussed here, but details can be found in [4]. Other tree-related processes like front insertion and octant classification are discussed in [4].

4.2 Template Meshing

Meshing templates have been written such that the triangulation of an octant face does not depend upon the template used to mesh the (interior terminal) octants sharing the face [15]. This means that the triangulations of an octant face shared by octants on different processor are guaranteed to match. Interior terminal octants can be meshed without communication. Once they are meshed, "duplicate" mesh vertices, edges, and faces are linked together in order to satisfy the requirement of the parallel mesh database [13, 17]. Figure 9 shows the result of template meshing on four processors. The figure on the left represents the surface mesh after tree building. Details about parallel template meshing can be found in [19].

4.3 Tree-supported Face Removals

The term "tree-supported" means that a tree is used for localization. In a parallel distributed environment, it actually means a distributed tree is

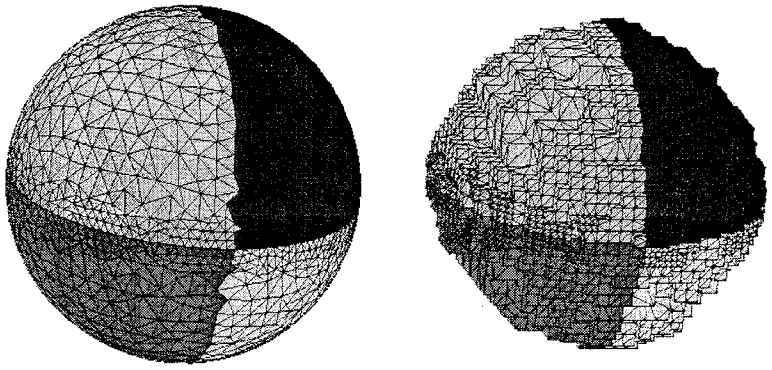

Figure 9: Template meshing on 4 processors.

used. Given a base face to remove, target vertices are considered within some neighborhood, typically, a box whose size is comparable to the size of the face. The terminal octants intersecting with the box are gathered. If not all of them are residing on processor, the face is not attempted to be removed. If not all terminal octants are on-processor, the front face is near a partition boundary. Proper removal of the face would require communication with the processors owning the missing octants. Allowing communication to happen during face removals would be prohibitive in terms of cost, and is therefore not done here. Repartitioning will allow these "unremovable" faces (due to lack of information) to be removed later on. In terms of face removal validity checks, off-processor front entities known by the terminal octants involved in these checks must be considered.

Results for the parallel volume meshing procedure follow the discussion of the hierarchical repartitioning scheme of section 5. They are presented in section 6.

Figures 10, and 11 show the application of parallel tree-supported face removals and interface repartitioning for simple models on 8 processors. The various pictures represent:

a. the front after octant partitioning,

b. the front after face removals,

c. the front after face interface repartitioning and face removals, and

d. the front after edge interface repartitioning and face removals.

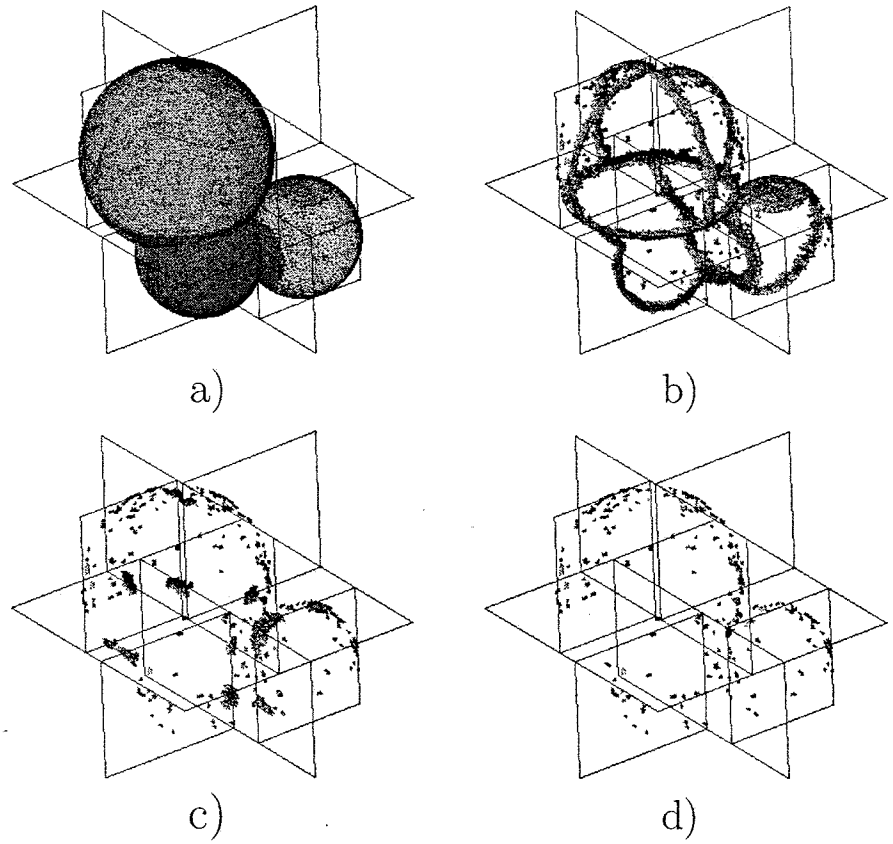

Figure 10: Parallel tree-supported face removals and interface repartitioning on 8 processors for *g44* model.

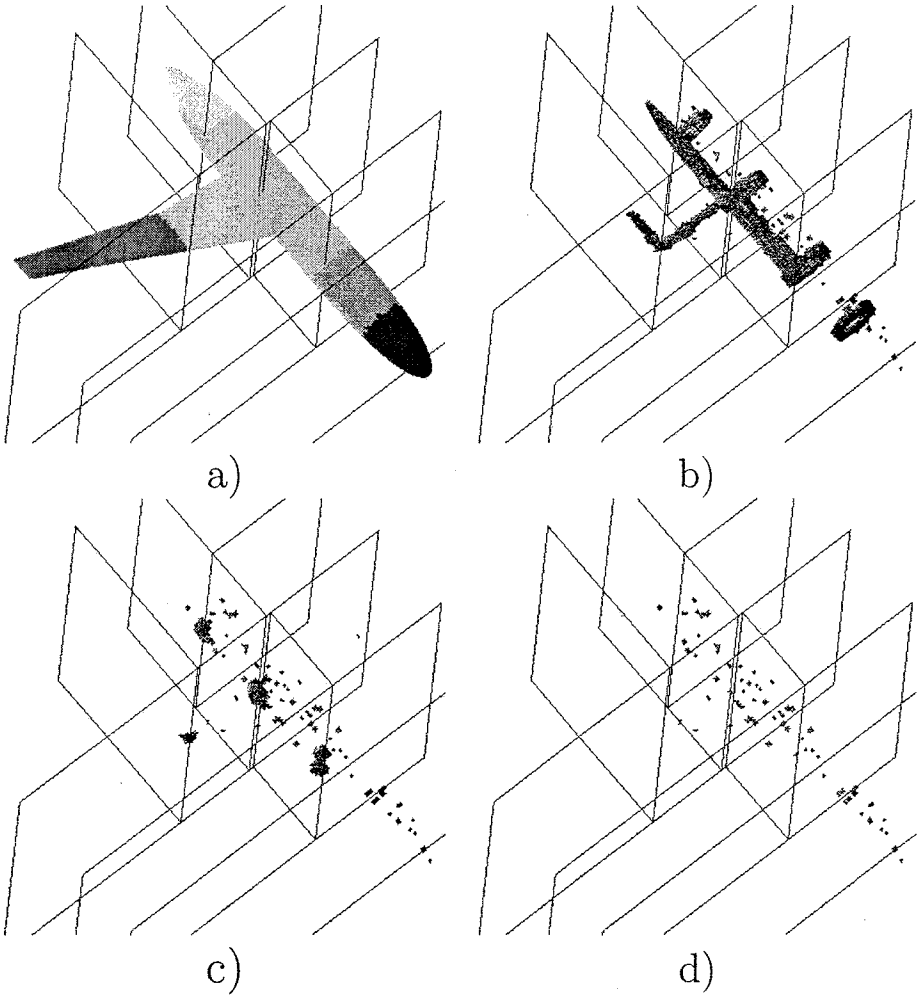

Figure 11: Parallel tree-supported face removals and interface repartitioning on 8 processors for *Airplane* model.

5 Hierarchical Repartitioning

5.1 Introduction

Three levels of repartitioning are used in parallel volume meshing:

1. face interface repartitioning,
2. edge interface repartitioning, and
3. vertex interface repartitioning.

The idea behind face and edge interface repartitioning is to redistribute terminal octants along with the data they are associated with, that is, front entities, such that face removals can be applied again in a balanced fashion, if possible. Vertex interface repartitioning does not operate on the distributed tree, but rather on the remaining front entities themselves. The objective is however the same. To make these hierarchical repartitioning techniques general, it is assumed that mesh regions are created only using face removals. The advantage of interior octant template meshing will be pointed out in the last sub-section.

Once sub-domain meshing is complete, that is, no face removal can be applied using only on-processor information, the domain remaining to be meshed corresponds to the interfaces of the partition. Figure 12 shows the front after face removals have been applied within a partition. This partition is bounded by two planar face interfaces and one linear edge interface. Face interfaces need to be repartitioned so that face removals can be applied in the next face removal phase. Face interface repartitioning is described in the next sub-section.

5.2 Face Interface Repartitioning

It is assumed a Recursive Bisection procedure was used to initially partition the domain (at the octant level) into sub-domains. Because the separators (between partitions) are hyperplanes (each defined by an axis and the median on that axis), face interfaces are planar. The only assumption that is made here is that the separators are planar. The following face interface procedure can be used with any partitioning scheme that uses planar separators. As a starting point, it is assumed that each processor has knowledge of the separators bounding the sub-domain and the neighboring processors.

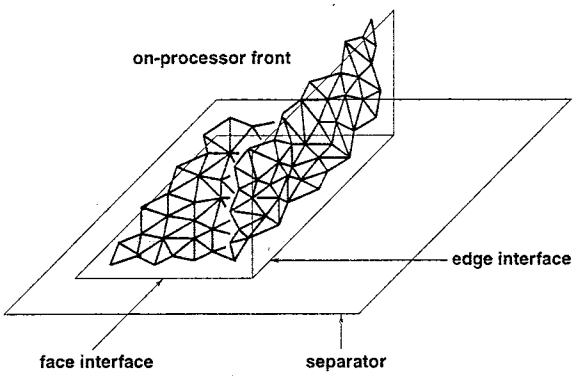

Figure 12: Front (on one processor) after face removals have been applied.

Note that sub-domain always refers to the region of space coming from the initial partitioning of the domain. In the following, face interfaces are not initially associated with any boundaries, they are just considered equivalent to planar separators. As the procedure to repartition face interfaces evolves, face interfaces are more explicitly defined in terms of their boundaries. Only boundary terminal octants are considered when repartitioning. It should be noted that, in the face removal phase, any boundary terminal octant that has been "filled in" is reclassified as "done". Such a terminal octant will not participate in any subsequent repartitioning. This minimizes data movement as octants are actually migrated according to the repartitioning. The procedure to repartition face interfaces *repartitionFaceInterfaces* is given below in pseudo-code form.

```
repartitionFaceInterfaces()
assignOctantsToInterfaces()
for each interface do
  for each octant associated with interface do
    Add octant's centroid and size to data array
     to be sent to processors sharing interface
  endfor
endfor
Send and receive messages
for each received message do
  processMsg(msg)
```

```
endfor
assignInterfacesToProcs()
Assign destination processors to octants
 according to face interface assignment
Migrate octants according to their destinations

assignOctantsToInterfaces()
for each octant do
   Get interface closest to octant
   Assign octant to that interface
endfor
```

The distance from an octant to an interface is computed by projecting the octant's centroid onto the interface. Figure 13 shows the assignment, on one processor, of current boundary terminal octants (octants near interfaces) to face interfaces (numbered 0, 1, and 2). Due to the nature of the initial partitioning, a processor will only know about terminal octants whose centroids fall within its sub-domain. The cost of assigning terminal octants to interfaces is equal to the number of boundary terminal octants times the number of face interfaces on a given processor. Since boundary terminal octants are reclassified as "done" as they are "filled" during the face removal phase, the number of boundary terminal octants, at this point, represents the size of the partition boundary of the given processor. If the number of processors increases at the same rate as the problem size, the partition boundary for any given processor can be assumed to remain constant. The number of face interfaces can also be assumed to remain constant. These two observations are key to guaranteeing scalability. Each processor sends information regarding its face interfaces to the neighboring processors sharing those interfaces. This communication step is required to "bring together" terminal octants on either side of a face interface. The cost of setting up the messages is equal to the number of face interfaces times the number of boundary terminal octants on a given processor. The cost of sending the messages is equal to the number of face interfaces times the maximum number of processors sharing a face interface. One can observe that if the number of processors increases at the same rate as the problem size, the number of neighboring processors for a given processor is constant. Again, this is key to scalability.

After messages have been sent, they are received by the appropriate processors. Each message contains information regarding the face interface plane

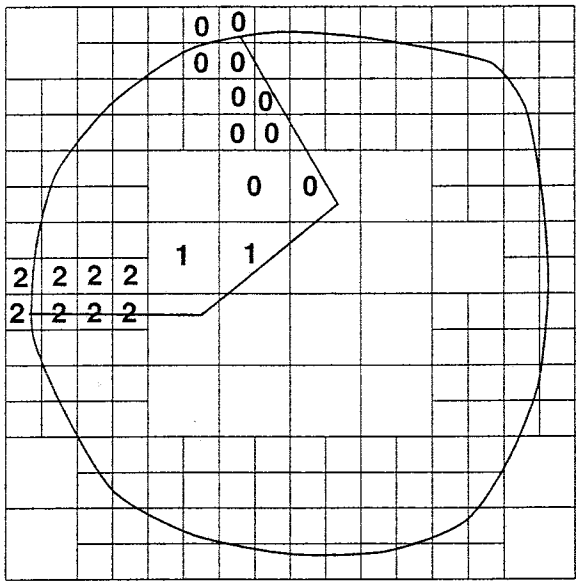

Figure 13: Assignment of octants to interfaces.

(planar separator) and the octants associated with the face interface (on the sending processor). The procedure to process a received message is given below in pseudo-code form.

```
processMsg(msg)
Get originating processor
Get on-processor interface the data is for
for each entry do
  Get octant's centroid and size
  Project onto interface
endfor
/* Projected centroids define the off-processor interface */
for each octant do
  if not associated with interface continue
  Project onto interface
  if projection within off-processor interface then
    Assign octant to off-processor interface
  endif
endfor
```

The goal of this face interface repartitioning procedure is to associate terminal octants on either side of a given interface so that they can be migrated as a block to a destination processor. The projected centroids define the off-processor interface, that is, the actual interface of the sending processor. Considering figure 14, processor x sends away the octants' centroids associated with the interface to processors y and z. Focus is given to what happens next on processor y. Upon reception of the message, processor y finds the interface x is referring to by going through its own interfaces and finding the one that is in the same plane. This is referred to as the on-processor interface. Processor y then projects the centroids on this interface. This defines the off-processor interface. Any octant on processor y assigned to the on-processor interface which, when the centroid is projected onto the interface, falls within the off-processor interface is assigned to the off-processor interface. Because a face interface can be shared by more than two processors, the assignment of boundary terminal octants to interfaces is not unique. Considering figure 14, there are two ways of seeing the interface(s) shared by processors x, y, and z. One can see a single interface on the side of x or two (smaller) interfaces, one on the size of y, the other on the size of z. To consistently break ties, processors on the "low id" side of an interface will have to give up their octants to the processors on the "high id" side. Referring to figure 14, it was assumed $x < y$ and $x < z$. This also means that, in the procedure *processMsg*, any message coming from a processor with higher id than the executing processor is ignored. Figure 15 shows, on the left, the initial assignment of octants (represented as boxes) to interfaces (in bold). The final assignment is shown on the right. Only the interfaces which are still relevant are shown. It is assumed partitions on the left are on the "low id" side while partitions on the right are on the "high id" side. This means that any processor on the left has an id lower than any processor on the right. It should be noted that this property holds as octants are (initially) partitioned using the Recursive Bisection methodology. This is due to the fact that, as the domain is recursively subdivided, "low id" processors always end up on the "left" (with respect to median on cutting axis) sub-domain. This will be further discussed when repartitioning edge interfaces.

The cost of processing the received messages is equal to the partition boundary size times log(partition boundary size) on a given processor. As a reminder, the partition boundary size on a given processor is directly proportional to the number of boundary terminal octants on that processor. This is due to the fact that boundary terminal octants that remain (after reclas-

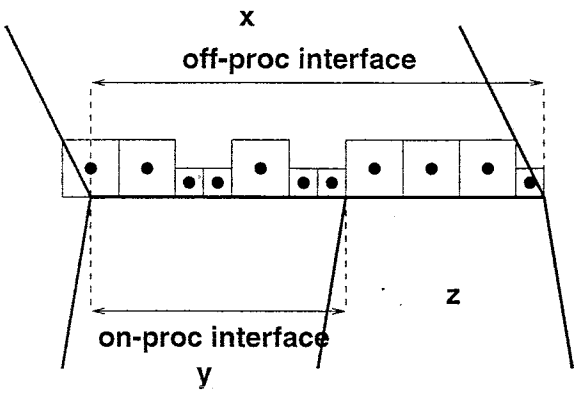

Figure 14: On and off-processor interfaces.

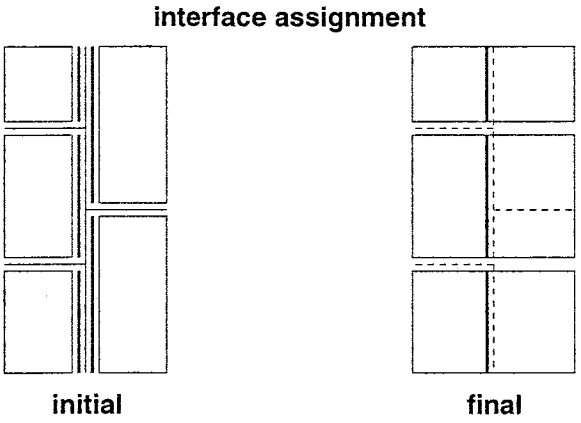

Figure 15: Initial and final assignment of octants to interfaces.

sification from boundary to "done") are "near" the interface, typically, "one layer deep". The log term comes from the quadtree which is being used for localization on the face interface plane. The problem is now to try to assign the face interfaces to processors so that the next face removal phase is load balanced. This is explained next.

If a face interface on a given processor is such that there is no terminal octant that has been associated with it, it is not considered in this face interface distribution phase. The load carried by a given face interface can be estimated by counting the number of terminal octants associated with it. Assignment of face interfaces should be such that the sum of the loads of the face interfaces assigned to a processor is as uniform as possible (across all

processors). Uniform loading is likely not to be obtained in most cases since face interfaces can carry very different loads. The assignment of a given face interface is restricted to one of the processors sharing that interface. This restriction is advantageous because (i) it minimizes data movement, and (ii) it minimizes the local root growth. Given a face interface, the octants associated with that interface reside on the processors sharing the interface. By restricting octant migration to any of these processors, data movement (from one processor to another) is minimized. Octant migration involves not only octants but also the front mesh entities they contain. The procedure to assign face interfaces to processors is given below in pseudo-code form.

assignInterfacesToProcs()
Assign (at random) each interface to 1 processor sharing it
Get load loadP (total load on processor)
Get average load avgLoad
for each interface assigned to executing processor **do**
  Get interface load loadI
  /* Should that load be given away? */
  if $|loadP - loadI - avgLoad| > |loadP - avgLoad|$ **continue**
  /* Yes, it should */
  for each neighboring processor sharing the interface **do**
    Get load loadP2
    /* Does it need the load? */
    if $|loadP2 + loadI - avgLoad| > |loadP2 - avgLoad|$ **continue**
    /* Yes, it does */
    Keep track of processor proc2 with lowest loadP2
  endfor
  if at least 1 processor needs the load **then**
    Assign face interface to proc2
    loadP = loadP - loadI
  endif
endfor

At the start of the procedure, each interface is assigned at random to one of the processors that share it. This is to initialize the assignment. The cost for getting the average load is equal to the partition boundary size (on-processor load) plus $\log(n_p)$ (to obtain the total load). The $\log(n_p)$ term can

be dropped. Any processor with too much load will now give its excess load in an attempt to load balance the processors. This process is completely asynchronous. An interface can be given away if the current load of the processor minus the interface load is closer to the average than the current load. In that case, the interface represents an over-load for the processor. This extra interface can be given away to a neighboring processor sharing the interface if this addition would bring the neighboring processor's load closer to the average. If more than one neighboring processor needs the load, the one with the lowest current load receives the load. Each processor needs to know the loads of the processors that share interfaces. This knowledge can be obtained by communicating with these neighboring processors as a pre-processing step. The cost for obtaining the loads on neighboring processors is equal to the number of neighboring processors. The cost for assigning interfaces to processors is equal to the number of interfaces times the maximum number of neighboring processors sharing an interface on a given processor. When an interface is assigned to a neighboring processor, the load it carried is not updated to avoid any communication and/or synchronization step. This means that it is possible for a processor that was under-loaded to become over-loaded (has been assigned more interfaces than needed). To remedy this problem, the above procedure is iterated until none of the processors is over-loaded. By making sure that a once over-loaded processor cannot be assigned any additional interface, this will converge. In the worst case, the total number of iterations is equal to the number of processors. In theory, this can happen when (i) one processor is over-loaded and all the others are under-loaded and (ii) the load actually propagates from the over-loaded processor to all under-loaded processors (considering the above procedure). In practice, this situation is unlikely to occur because each processor is bounded by interfaces, each of them carrying some load. This means that a situation where one processor is over-loaded and all others are under-loaded is unlikely to happen. The number of iterations is then expected to be constant. Figure 16 shows the initial assignment of face interfaces to processors, the assignment after one iteration of the procedure, and the final assignment. Processor ids have been put in circles and arrows symbolize current face interface assignment. Once face interfaces have been assigned to processors for meshing, terminal octants associated with interfaces are marked with the proper destination processors. They are ready to be migrated by the octant migration procedure. The cost for migrating the octants is equal to the number of octants being sent times the depth of the distributed tree on a given processor. The number of octants

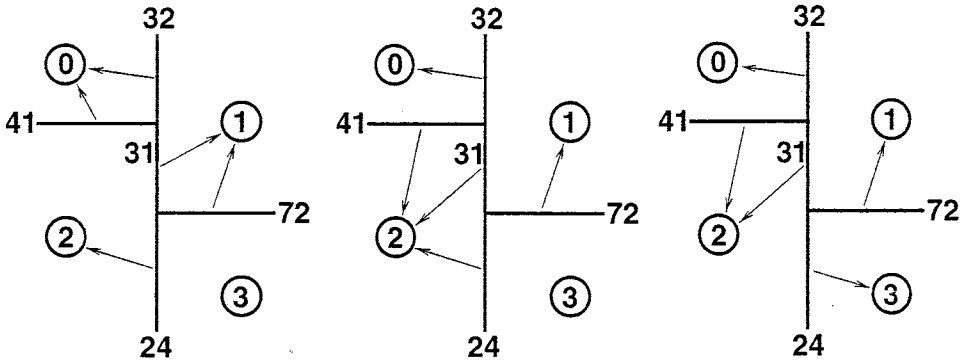

Figure 16: Assignment of interfaces to processors.

to be migrated is at worst equal to the partition boundary size. If n'_o is the number of octants on processor, the depth is $O(\log(n'_o))$. If the tree is well distributed, $n'_o = n_o/n_p$ where n_o is the total number of octants. The total number of octants is proportional to the total number of mesh regions that will be generated. The cost for migrating the front mesh entities associated with the migrated octants is at worst equal to the partition boundary size. The cost for removing the migrated front mesh entities from the front and re-inserting them into the tree is at worst equal to the partition boundary size times $\log(n'_o)$.

All processes used in repartitioning the face interfaces depend on (i) the partition boundary size, and/or (ii) the number of face interfaces, and/or (iii) the maximum number of processors sharing a face interface. These quantities are with respect to a given processor. For example, one looks at the partition boundary size of a given processor, not of all processors. These processes only depend upon "local" or "near" information. The term "local" refers to on-processor information. The term "near" refers to information of neighboring processors. This means face interface repartitioning is scalable.

5.3 Edge Interface Repartitioning

An edge interface results from the intersection of two face interfaces. As discussed previously, face interfaces are assumed to be planar, therefore, edge interfaces are linear. It is assumed that each processor has knowledge of its neighboring processors. It is also assumed that each processor only knows about the planar separators making up the face interfaces bounding the cor-

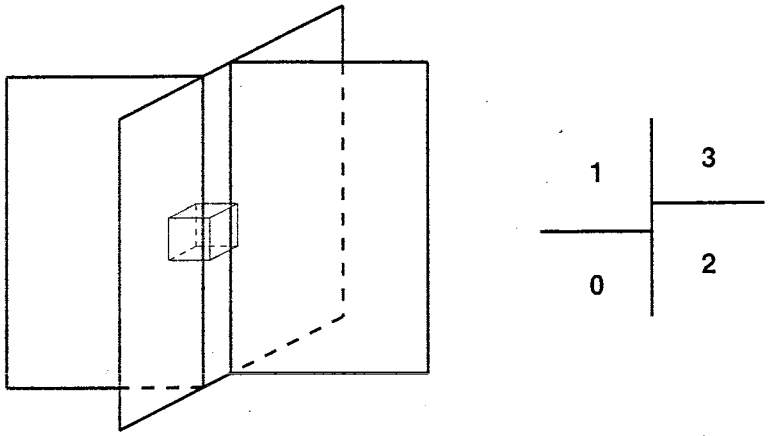

Figure 17: Simple example of edge interface "interference".

responding sub-domain. The difficulty with edge interfaces is that they can "interfere" with each other. In Figure 17, the distance between the two edge interfaces is comparable to the size of boundary terminal octants along the two edge interfaces. In order for the next face removal phase to actually remove faces, it is imperative to consider the two edge interfaces as one. One can either (i) merge the two edge interfaces or (ii) repartition considering one edge interface, apply face removals, repartition considering the other edge interface, and apply face removals. The first approach can be quite challenging especially when edge interfaces are not constrained to run along coordinate frame axes. Figure 18 shows an example where two edge interfaces "interfere" at a point. The two edge interfaces should be split into four defining a vertex interface at the point of contact. The second approach, although less efficient since it needs two iterations of edge interface repartitioning and face removal application, is attractive because there is no need to actually figure out if two edge interfaces "interfere" with each other. This is the approach that has been chosen due to its relative simplicity when compared to the other method.

The basic idea behind edge interface repartitioning is to consider two sets of edge interfaces such that (i) interfaces in one set cannot interfere with each other and (ii) the union of the two sets gives all edge interfaces. The edge interfaces are obtained by intersecting (with each other) the planar separators that make up the sub-domain associated with the processor. The complete set of edge interfaces can be split in two by considering planar separators

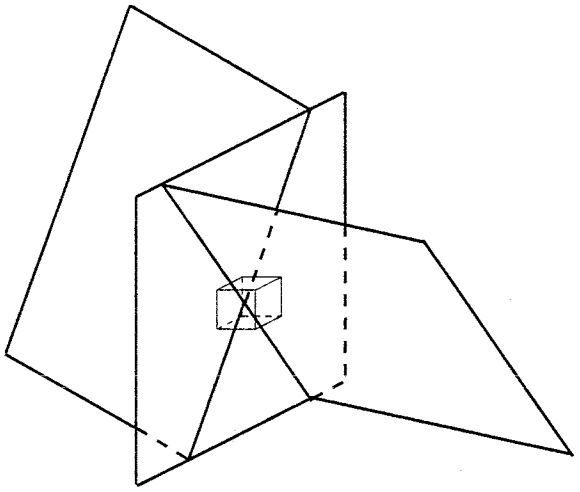

Figure 18: Complex example of edge interface "interference".

only if the processor is on its *low* side for the *low* set of edge interfaces or on its *hgh* side for the *hgh* set of edge interfaces. In the course of performing Recursive Bisection, when a processor set is split (say, in two), the processors with low id's are assigned to the sub-domain on the low ordinate side and the processors with high id's are assigned to the sub-domain on the high ordinate side. Figure 19 gives an example for four processors. On the first cut, the initial processor set (and domain) is split into two. On the second cut, each processor set (among the two) is split further into two.

Proper definition of edge interfaces (for a given side) is a two-step process. The first step consists in defining them by intersecting planar separators with each other. The second step consists of communicating the location of edge interfaces for processors that do not have knowledge of them.

The procedure to extract the edge interfaces from the planar separators is given below in pseudo-code form.

```
defineEdgeInterfaces1(side)
for each planar separator do
  /* Do not consider if processor on other side */
  for each other planar separator do
    /* Do not consider if processor on other side */
    Intersect the 2 planar separators
```

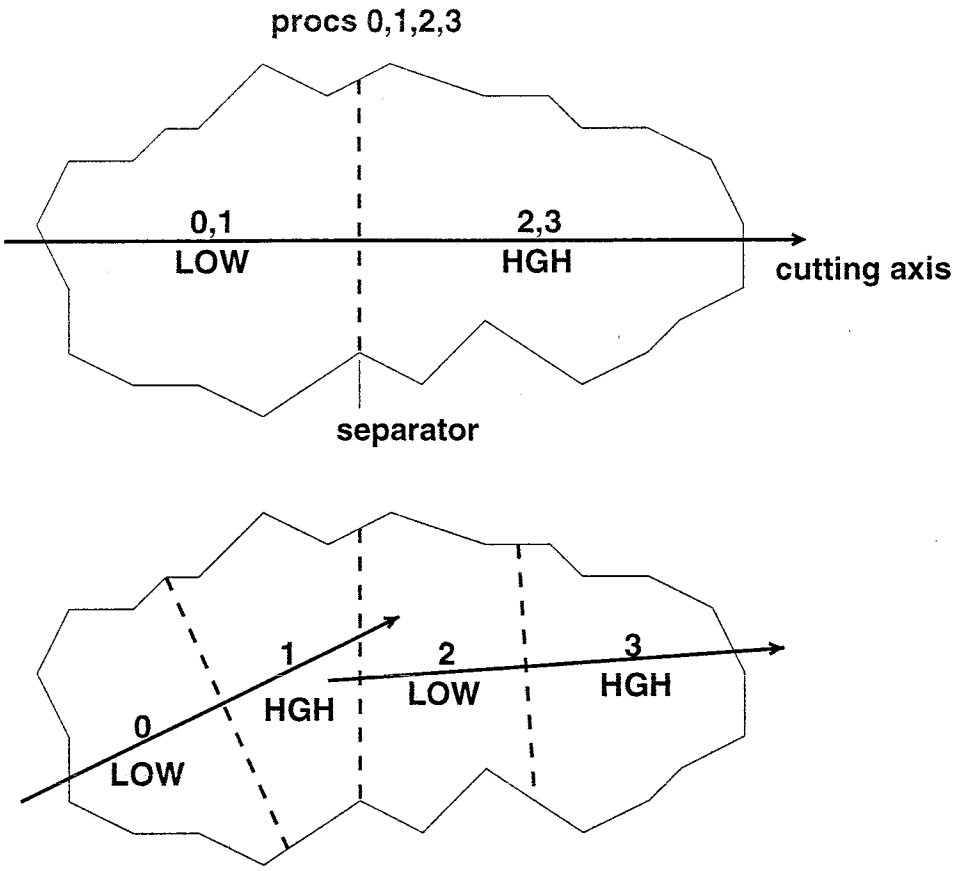

Figure 19: The concept of sides when cutting a domain and associated processor set.

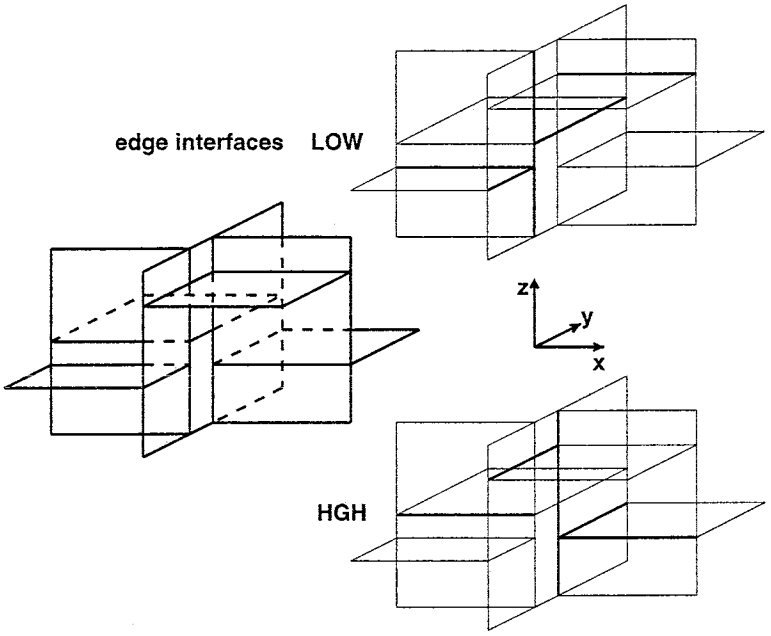

Figure 20: Edge interfaces on *low* and *hgh* sides.

```
    Intersect obtained line with sub-domain to get range
    /* Resulting line segment is an edge interface */
  endfor
endfor
```

A planar separator is considered for intersection only if the executing processor is on the requested side (either *low* or *hgh*). It should be noted that a planar separator is defined by a point and a normal direction. The intersection of two planes gives a line unless the two planes are parallel to each other. The line intersection which can be referred to as a linear separator is defined by a point and a direction vector. The edge interface is fully defined by reducing the a priori infinite linear separator with the polyhedral domain known to the processor (made up of planar separators). The edge interface is then fully defined by considering a range on the linear separator. The cost for *defineEdgeInterfaces1* is equal to the square of the number of planar separators on a given processor. It is a scalable process. Figure 20 gives an example of *low* and *hgh* side edge interfaces.

A processor may not know about defined edge interfaces that are within its sub-domain. This is illustrated in figure 21 assuming there is a perpen-

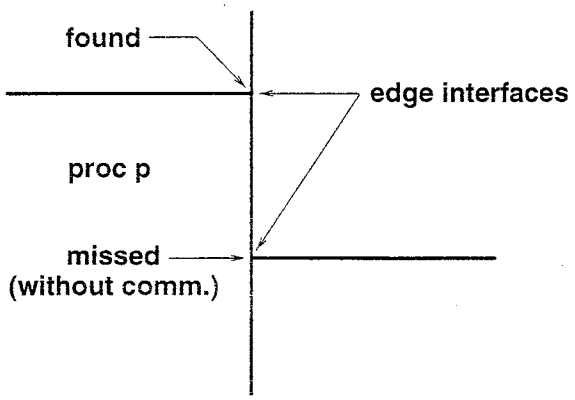

Figure 21: Determination of edge interfaces needs communication.

dicular third dimension. The procedure to update the edge interfaces known to a processor is given below in pseudo-code form.

```
defineEdgeInterfaces2()
for each edge interface do
   Pack info concerning edge interface into msg
    to be sent to each neighboring processor
endfor
Send and receive msgs
for each msg received do
   for each remote edge interface in msg do
      Intersect remote edge interface with polyhedral domain
      /* Resulting line segment is a new edge interface */
   endfor
endfor
```

The cost for *defineEdgeInterfaces2* depends upon the number of edge interfaces and the number of neighbors on a given processor. It is therefore scalable.

The procedure to distribute the edge interfaces to processors so that the next face removal phase is balanced (as much as possible) is similar to the one used for face interface repartitioning.

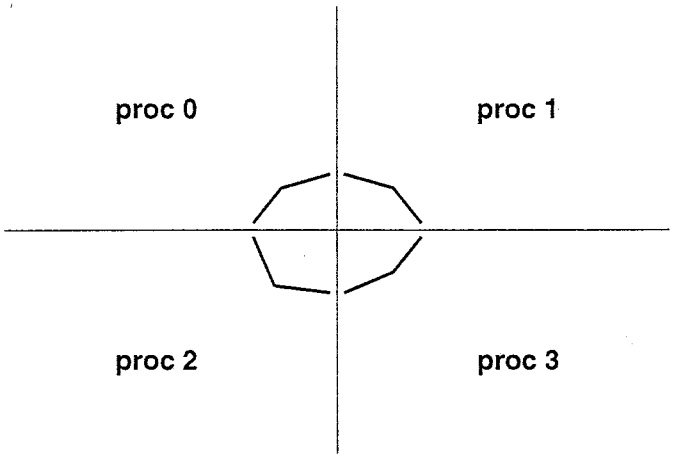

Figure 22: Example of domain to be localized.

5.4 Vertex Interface Repartitioning

Domains remaining to be meshed are either fully on processor or are shared (in terms of bounding front mesh faces) by several processors. Domains which are fully on processor have nothing to do with the fact that data is partitioned. They are the result of unremoved faces due to lack of a proper target in the tree-supported face removal phase. Domains shared by several processors correspond to the vertex interfaces. Vertex interfaces are defined as intersections of edge interfaces. Because the vertex interfaces represent an isolated subset of the domains remaining to be meshed, vertex interface repartitioning can be generalized so that it actually repartition any domain still to be meshed whether it is actually a vertex interface or not. The generalized vertex interface repartitioning is therefore considered a domain repartitioning procedure. The first task is to localize all domains, that is, bring any domain shared by several neighboring processors on a single processor (one of the neighboring processors). Figure 22 shows a domain "spanning" over processors 0, 1, 2, and 3. This domain needs to be localized by bringing the faces making up the domain to one processor, say processor 0. At this point, it does not matter if some processors end up with more domains than others since these domains will be redistributed for load balance in view of the final face removal phase.

Once all domains have been localized, they have to be distributed to processors such that the next face removal step is as balanced as possible. The

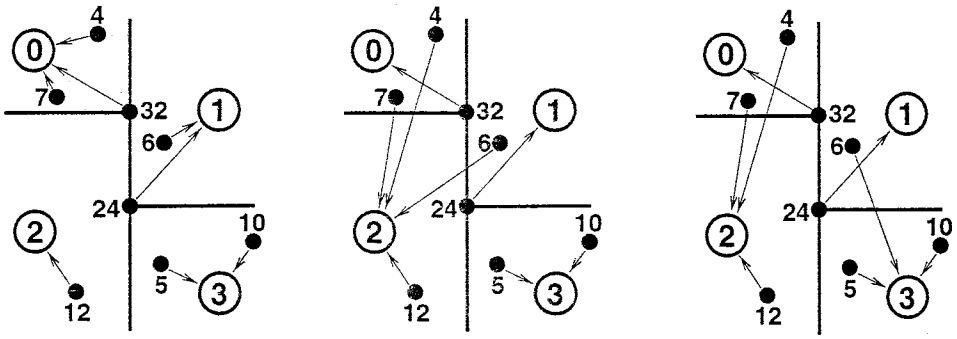

Figure 23: Assignment of domains to processors.

load carried by a given domains can be estimated by counting the number of boundary mesh faces. A procedure similar to the one used for face interface repartitioning can be used here. Each domain is initially assigned to the processor it is in. Domains are considered instead of face interfaces. An overloaded processor can potentially give away any extra load to any neighboring processor. The neighboring processor with the lowest load actually gets the extra load. Figure 23 shows the initial assignment of domains to processors, the assignment after one iteration of the procedure, and the final assignment. Processor ids have been put in circles and arrows symbolize current domain assignment. Once domains have been assigned to processors for meshing, mesh entities are migrated, and the final face removal step begins. Because vertex interface repartitioning depends only on "local" or "near" information, it is a scalable process.

5.5 Remarks

The above hierarchical repartitioning technique is general. This sub-section discusses hierarchical repartitioning in the context of the presented volume mesh generator, in particular, the use of templates to mesh interior terminal octants. Templates reduce the domain to be meshed by face removals. This means only parts of face and edge interfaces need to be actually meshed. Figure 24 shows the area on the face interface (between the templates and the model's boundary) that defines a domain still to be meshed. Figure 25 shows the segments on the edge interface (between the templates and the model's boundary) that correspond to an area still to be meshed.

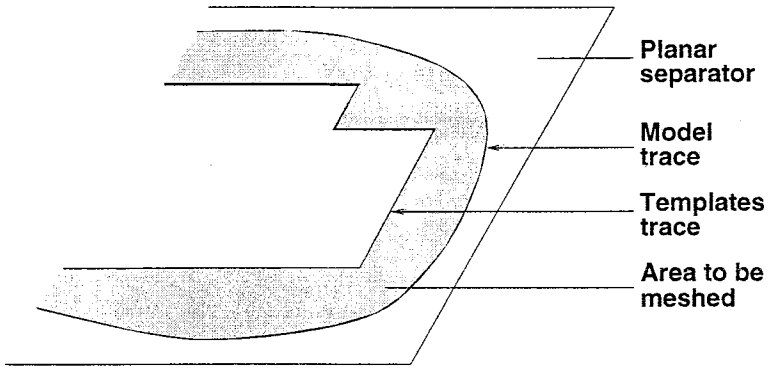

Figure 24: Effect of template meshing on face interfaces.

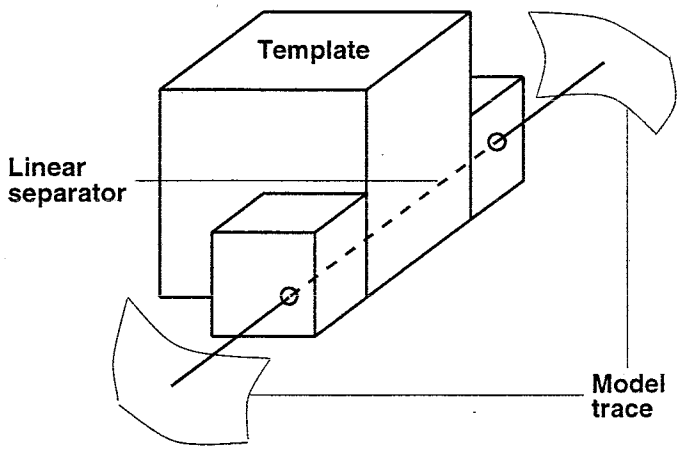

Figure 25: Effect of template meshing on edge interfaces.

6 Parallel Volume Meshing Results

Timing results are given for the models represented in figures 26, 27, and 28. These figures show the initial octant partitioning on 32 processors using the Modified Recursive Orthogonal Bisection (MROB). In order to have a better idea of the sub-domains, the planar separators are also shown.

Tables 1, 2, 3, 4, 5, and 6 give time meshing statistics for the complete parallel volume meshing procedure. All runs have been performed on an *ibm SP-2* with 128 megabytes of core memory (no virtual memory). The time statistics (wall-clock time in seconds) have been broken down into the following main steps:

1. tree building,
2. template meshing,
3. partitioning (initial tree partitioning),
4. face removals (tree-supported and tree-less), and
5. repartitioning (face, edge, and vertex interface).

When runs for a given problem case include a one-processor run, speed-ups and relative speed-ups are given. Otherwise, only relative speed-ups are given. Speed-up on n_p processors is equal to the ratio of the time spent using one processor to the time spent on n_p processors. The relative speed-up is the speed-up obtained when doubling the number of processors. In general, when going from one processor to two processors, speed-up is relatively low. This is mostly due to the fact that there is no need for partitioning and repartitioning when running on one processor. In other words, there is an initial "price to pay" when switching from serial to parallel. Focus is therefore given to relative speed-ups.

Tables 1, 2, and 3 show results for test cases in the $50,000$ regions generated range. Relative speed-up is maintained at around 1.5 but performance drops when using 16 processors mostly due to the fact that few face removals are being performed at each step.

Tables 4, 5, and 6 show results for test cases in the $250,000$ regions generated range. Relative speed-up is maintained at around 1.6 with some expected variations depending upon the model to be meshed.

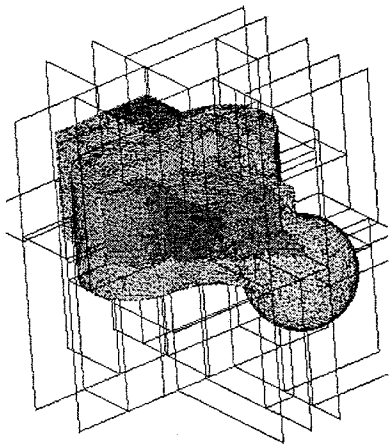

Figure 26: Initial partitioning for *g60* model on 32 processors (213,000 tets).

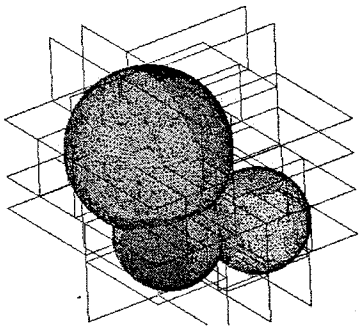

Figure 27: Initial partitioning for *g44* model on 32 processors (264,000 tets).

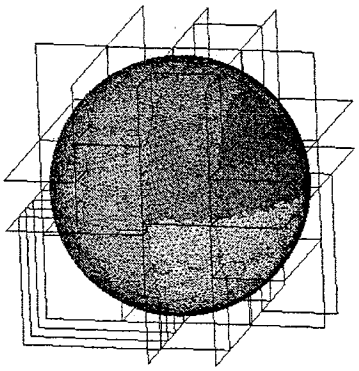

Figure 28: Initial partitioning for *g21* model on 32 processors (309,000 tets).

procs	1	2	4	8	16
tree building	9.8	6.1	3.9	2.7	2.3
template meshing	8.1	5.7	4.4	3.4	3.0
partitioning	0.0	9.5	7.5	5.3	3.4
face removals	106.9	70.3	36.7	24.8	17.6
repartitioning	0.0	16.7	12.4	10.4	10.0
total	124.8	108.3	64.9	46.6	36.3
speed-up		1.15	1.92	2.68	3.44
rel. speed-up		1.15	1.67	1.39	1.28

Table 1: Time statistics for *g60* model (41,000 tets).

procs	1	2	4	8	16
tree building	12.2	7.2	4.8	3.7	2.6
template meshing	11.1	7.3	5.4	3.8	2.7
partitioning	0.0	12.1	13.0	7.7	3.6
face removals	174.9	115.9	76.1	41.3	33.5
repartitioning	0.0	24.7	21.6	16.1	14.9
total	198.2	167.2	120.9	72.6	57.3
speed-up		1.19	1.64	2.73	3.46
rel. speed-up		1.19	1.38	1.67	1.27

Table 2: Time statistics for *g44* model (55,000 tets).

procs	1	2	4	8	16
tree building	16.3	7.5	5.3	3.8	3.2
template meshing	14.2	8.5	5.6	3.6	2.7
partitioning	0.0	19.5	17.7	9.5	4.2
face removals	199.0	123.9	75.0	44.4	26.2
repartitioning	0.0	23.3	17.2	14.3	11.0
total	229.5	182.7	120.8	75.6	47.3
speed-up		1.26	1.90	3.04	4.85
rel. speed-up		1.26	1.51	1.60	1.60

Table 3: Time statistics for *g21* model (69,000 tets).

procs	8	16	32
tree building	10.0	7.1	5.8
template meshing	14.7	11.8	10.0
partitioning	43.1	18.3	9.4
face removals	96.9	58.0	36.0
repartitioning	33.4	23.1	20.2
total	198.1	118.3	81.4
rel. speed-up		1.67	1.45

Table 4: Time statistics for *g60* model (213,000 tets).

procs	8	16	32
tree building	13.3	8.2	7.3
template meshing	14.6	10.3	7.7
partitioning	67.3	22.3	10.6
face removals	162.0	99.1	55.3
repartitioning	47.5	33.8	23.8
total	304.7	173.7	104.7
rel. speed-up		1.75	1.66

Table 5: Time statistics for *g44* model (264,000 tets).

procs	8	16	32
tree building	13.6	11.0	9.1
template meshing	12.1	8.7	7.1
partitioning	62.2	22.1	11.4
face removals	149.1	85.2	51.5
repartitioning	45.2	29.0	21.0
total	282.2	156.0	100.1
rel. speed-up		1.81	1.56

Table 6: Time statistics for *g21* model (309,000 tets).

7 Conclusion

A major step in going toward scalable parallel mesh generation is the introduction of a distributed octree structure. Such a scalable structure was not part of previous efforts by the authors [5]. This structure along with a parallel tree building procedure enable the parallel volume mesher to be truly "memory" scalable with both the octree and mesh structures distributed.

Interface meshing, which has been the focus of this paper, is the most difficult aspect of effective parallel volume meshing when using an advancing front method [18]. Interfaces result from the initial partitioning of the domain to be meshed. Key to efficiency (in terms of load balance) of the complete volume meshing procedure relies upon the interface repartitioning procedure. In this paper, various procedures have been presented as part of the "hierarchical repartitioning" methodology in order to (i) properly define face and edge interfaces, (ii) assign interfaces to processors for load balance, and (iii) repartition the octree and associated front according to interface assignment. The presented interface repartitioning is parallel and scalable. Coupled with "fast" octant and mesh migration procedures, this repartitioning technique makes parallel mesh generation using the advancing front method a viable undertaking. Repartitioning performance can be further enhanced by taking advantage of the fact that octant templates were applied prior to face removals. In most cases, template meshing act as dimension reducer for interfaces.

Promising results have been seen for up to 32 processors. The next step is to benchmark the parallel volume meshing procedure on massively parallel computers with at least 512 processors available.

References

[1] M. W. Beall and M. S. Shephard. A general topology-based mesh data structure. *International Journal for Numerical Methods in Engineering*, 40(9):1573–1596, 1997.

[2] A. Bowyer. Computing dirichlet tessellations. *The Computer Journal*, 24(2):162–166, 1981.

[3] N. Chrisochoides and F. Sukup. Task parallel implementation of the bowyer-watson algorithm. In *Fifth International Conference on Numerical Grid Generation in Computational Field Simulations*, pages 773–782. Mississippi State University, 1996.

[4] H. L. de Cougny. *Parallel Unstructured Distributed Three-Dimensional Mesh Generation.* PhD thesis, Scientific Computation Research Center, Rensselaer Polytechnic Institute, Troy, NY, May 1998.

[5] H. L. de Cougny, M. S. Shephard, and C. Özturan. Parallel three-dimensional mesh generation on distributed memory MIMD computers. *Engineering with Computers*, 12:94–106, 1996.

[6] R. Van Drieeche and D. Roose. Load balancing computational fluid dynamics calculations on unstructured grids. Technical report, Dept. of Comp. Sc., Katholieke Universiteit Leuven, Belgium.

[7] C. Farhat and M. Lesoinne. Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics. *International Journal for Numerical Methods in Engineering*, 36:745–764, 1993.

[8] A. Gaither, D. Marcum, and D. Reese. A paradigm for parallel unstructured grid generation. In *Fifth International Conference on Numerical Grid Generation in Computational Field Simulations*, pages 731–740. Mississippi State University, 1996.

[9] J. Galtier and P.-L. George. Prepartitioning as a way to mesh subdomains in parallel. In *Fifth International Meshing Roundtable, Pittsburgh, PA*, pages 107–121, 1996.

[10] R. Löhner. Progress in grid generation via the advancing front technique. *Engineering with Computers*, 12:186–210, December 1996.

[11] Message passing interface forum MPI: a message-passing interface standard. *International Journal of Supercomputer Applications and High-Performance Computing*, 8(3/4), 1994. Special Issue on MPI.

[12] T. Okusanya and J. Peraire. Parallel unstructured mesh generation. In *Fifth International Conference on Numerical Grid Generation in Computational Field Simulations*, pages 719–729. Mississippi State University, 1996.

[13] C. Özturan. *Distributed Environment and Load Balancing for Adaptive Unstructured Meshes"*. PhD thesis, Computer Science Dept., Rensselaer Polytechnic Institute, Troy, New York, 1995.

[14] M. Saxena and R. Perucchio. Parallel FEM algorithms based on recursive spatial decomposition i. automatic mesh generation. *Computers & Structures*, 45(5-6):817–831, 1992.

[15] W. J. Schroeder and M. S. Shephard. A combined octree/delaunay method for fully automatic 3-d mesh generation. *International Journal for Numerical Methods in Engineering*, 29:37–55, 1990.

[16] M. S. Shephard and M. K. Georges. Automatic three-dimensional mesh generation by the finite octree technique. *International Journal for Numerical Methods in Engineering*, 32(4):709–749, 1991.

[17] M.S. Shephard, J.E. Flaherty, H.L. de Cougny, C. Özturan, C.L. Bottasso, and M.W. Beall. Parallel automated adaptive procedures for unstructured meshes. In *Special Course on Parallel Computing in CFD*, R-807, pages 6.1–6.49. AGARD, 1995.

[18] A. Shostko and R. Löhner. Three-dimensional parallel unstructured grid generation. *International Journal for Numerical Methods in Engineering*, 38:905–925, 1995.

[19] M. L. Simone. *A Distributed Octree Data Structure and Algorithms for Scientific Modeling on Distributed Memory Processors.* PhD thesis, Scientific Computation Research Center, Rensselaer Polytechnic Institute, Troy, NY, May 1998.

[20] D. Watson. Computing the n-dimensional delaunay tessellation with applications to voronoi polytopes. *The Computer Journal*, 24(2):167–172, 1981.

[21] P. Wu and E. N. Houstis. Parallel adaptive mesh generation and decomposition. *Engineering with Computers*, 12:155–167, 1996.

[22] M. A. Yerry and M. S. Shephard. Automatic three-dimensional mesh generation by the modified-octree technique. *International Journal for Numerical Methods in Engineering*, 20:1965–1990, 1984.