Engineering with Computers (1996) 12: 94–106

Engineering with Computers

Parallel Three-Dimensional Mesh Generation on Distributed Memory MIMD Computers

H. L. de Cougny, M. S. Shephard and C. Özturan
Scientific Computation Research Center, Rensselaer Polytechnic Institute, USA

Abstract. *This paper discusses the development of an automatic mesh generation technique designed to operate effectively on multiple instruction multiple data (MIMD) parallel computers. The meshing approach is hierarchical, that is, model entities are meshed after their boundaries have been meshed. Focus is on the region meshing step. An octree is constructed to serve as a localization tool and for efficiency. The tree is also key to the efficient parallelization of the meshing process since it supports the distribution of load to processors. The parallel mesh generation procedure repartitions the domain to be meshed and applies on processor face removals until all face removals with local data have been performed. The portion of the domain to be meshed remaining is dynamically repartitioned at the octant level using an Inertial Recursive Bisection method and local face removals are reperformed. Migration of a terminal octant involves migration of the octant data and the octant's mesh faces and/or mesh regions. Results show relatively good speed-ups for parallel face removals on small numbers of processors. Once the three-dimensional mesh has been generated, mesh regions may be scattered across processors. Therefore, a final dynamic repartitioning step is applied at the region level to produce a partition ready for finite element analysis.*

1. Introduction

The development of automatic mesh generation techniques for complex three-dimensional configurations has been an active area of research for over a decade [1, 2]. The introduction of these mesh generation procedures has removed a major bottleneck in the application of finite element and finite volume analysis techniques. The introduction of scalable parallel computers is allowing the solution of ever larger models. It is now common to see meshes of several million elements solved on these computers, with the ability to solve on meshes of tens or hundreds of millions of elements coming in the very near future. As mesh sizes become this large, the process of mesh generating on a serial computer becomes problematic both in terms of time and storage. This paper discusses efforts to address this problem by the development of a parallel mesh generation procedure that will operate on the same computer, and using similar structures, as the parallel analysis procedures.

With recent advances in the efficiency of automatic mesh generators which create well over two million elements per hour on a workstation [3], one may question the need for the parallel generation of meshes. The obvious answer is that as the problem size grows, the solution process on parallel computers will continue to scale by the addition of more processors. However, mesh generation on a single processor will not scale, therefore becoming the computational bottleneck. A second critical reason for parallel mesh generation is the shortage of memory on a sequential machine when dealing with very large meshes. On a parallel machine, the memory problem is addressed by distributing the mesh over a number of processors, each of which stores its own portion of the mesh.

Efficient parallel algorithms require a balance of work load among the processors while maintaining interprocessor communication at a minimum. Key to determining and distributing the work load and controlling communications is knowledge of the structure of the calculations and communications. In the finite element analysis process, the mesh and its connectivity naturally provide the required structure. The ability to maintain efficiency is compromised when the structure and, therefore, work load and communications is altered as is the case in parallel adaptive finite element analysis [4–7]. Parallel mesh generation is even more complex to control effectively since the only structure known at the start of the

Correspondence and offprint requests to: H. L. de Cougny, Scientific Computation Research Center, Rensselaer Polytechnic Institute, Troy, NY 12180-3590, USA.

process is that of the geometric model which has no discernible relationship to the work load needed to generate the mesh. On the other hand, the more useful structure to discern work load and control communications is the mesh which is only fully known at the end of the process. The lack of initial structure and ability to predict accurately the work load during the meshing process underlies the selection of algorithmic procedures in the parallel mesh generation procedure presented here. In particular, the procedure employs an octree decomposition of the domain to control the meshing process. The octree structure supports the distribution or redistribution of computational effort to processors.

The next section considers efforts to date by others on the parallelization of automatic mesh generation and the general meshing approach developed here. The third section follows with a detailed description of the region (three-dimensional) meshing algorithm in a sequential environment. The fourth section describes the tools that have been developed to support mesh generation in a distributed parallel computing environment. This includes data structures, multiple octant migration, and dynamic partitioning and repartitioning. The fifth section explains how region meshing can be accomplished in parallel along with results showing relatively good speed-ups when the problem size is in accordance with the number of processors. The last section summarizes what has been accomplished and what needs to be developed further.

2. Background and Meshing Approach

To date, there has been limited attention given to parallel automatic mesh generation algorithms. Löhner *et al.* [8] have parallelized a two-dimensional advancing front procedure which starts from a pre-triangulated model boundary. The approach taken is to subdivide (partition) the domain (with the help of a background grid) and distribute the sub-domains to different processors for triangulation. The interior of subdomains are meshed independently. Then, the inter-subdomain regions are meshed using a coloring technique to avoid conflicts. Finally, the 'corners' between more than two processors are meshed following the same basic strategy. After the mesh is generated, the node point positions are smoothed in parallel employing an iterative procedure which positions each node based on local information. The choice of a scheduling scheme based on a master/slave paradigm may be the weak point of the method since it creates bottlenecks when transferring data from the host to the nodes (and vice versa).

Saxena and Perruchio [9] describe a parallel Recursive Spatial Decomposition (RSD) scheme which discretizes the model into a set of octree cells. Interior and boundary cells are meshed by either using templates or element extraction (removal) schemes in parallel. The algorithmic procedure they employ to create these octant level meshes requires no communication between octants. The main difficulty for this meshing approach is to guarantee that a boundary octant can always be meshed regardless of the complexity of the model. Robust loop building algorithms which include possible tree refinement to resolve invalid configurations are in general difficult to parallelize [10]. Parallel results have been simulated on a sequential machine.

The parallel mesh generator presented in this paper builds upon previous work on sequential octree-based mesh generators [10–12], parallel adaptive finite element analysis procedures [4–6], and parallel mesh generation [13]. The present parallel mesh generator meshes three-dimensional nonmanifold objects following the hierarchy of topological entities. That is, the model edges are meshed first, the model faces are meshed second, and the model regions are meshed last. Quadtrees and octrees are used for face and region meshing for the purpose of localization, respectively. This paper will focus on the three-dimensional aspect of mesh generation, that is region meshing with an underlying octree.

Figure 1 graphically depicts the basics of the present mesh generator. The first step for the meshing of a model region is to develop a variable level octree which reflects the mesh control information and is consistent with the triangulation on the boundary of the model region. Octants containing mesh entities classifed on the boundary of the model region to be meshed are approximately of the same size as the mesh entities

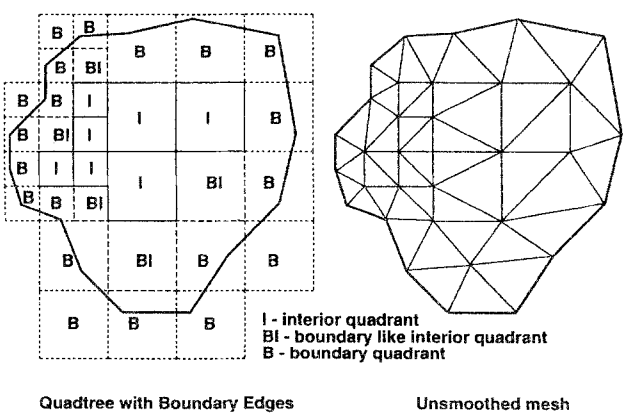

Fig. 1. Graphical depiction of the basics of the peresented mesh generator.

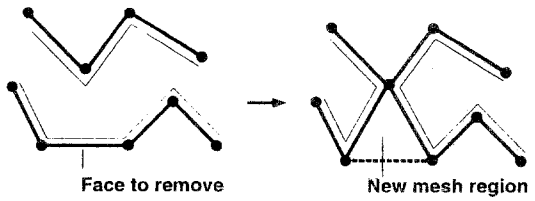

Fig. 2. Face removal (2-d setting).

they contain. A one level difference on octants sharing one or more edges is enforced during this process to control smoothness of the mesh gradations. Once the octree is generated, the octants are classified as *interior*, *outside*, or *boundary*. Those classified as *outside* receive no further consideration. Some *interior* octants are reclassified *boundary* if they are too close to mesh entities classified on the boundary of the model region (*boundary-interior*). The purpose of this reclassification is to avoid the complexities caused when *interior* octant mesh entities (coming from the application of templates) are too close to the boundary and may lead to the creation of poorly shaped elements in that neighborhood. *Interior* octants are meshed using templates. 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. The next section provides more technical details on the region meshing procedure.

3. Sequential Region Meshing

As indicated above, the starting point for the region meshing process is a completely triangulated surface. The surface triangulation must satisfy the conditions of topological compatibility and geometric similarity [14] for the model faces. The region meshing process consists of the three steps of (i) generation of the underlying octree, (ii) template meshing of interior octants, and (iii) face removal to connect the given surface triangulation to the *interior* octants.

Mesh faces to which tetrahedral elements will eventually be connected are referred to as partially connected faces. They are basically missing one connected tetrahedron in the manifold case and up to two in nonmanifold situations. Initially, the mesh faces classified on the model boundary are the partially connected mesh faces. Once templates have been applied, that is, at the start of face removal, the interior mesh faces connected to exactly one tetrahedron are also partially connected mesh faces. In the remainder of this paper, the current set of partially connected mesh faces will be referred to as the front. During face removal, tetrahedra are connected to these faces, therefore eliminating them. Any nonexisting face of a newly created tetrahedra, referred to as a new face, is a partially connected face until it is eliminated. The face removal process is complete when there are no partially connected mesh faces remaining.

3.1. Underlying Octree

The octree is built over the given surface mesh to (i) help in localizing the mesh entities of interest and (ii) provide support for the use of fast octant meshing templates. Proper localization is achieved by having each terminal octant reference any partially connected mesh face which is either totally or partially inside its volume. This information is used efficiently to guarantee the correctness of the face removal technique. The octree building process can be decomposed into: (i) root octant building, (ii) octree building, (iii) level adjustment, (iv) assignment of partially connected mesh faces to terminal octants, and (v) terminal octant classification.

The root octant is such that the given surface mesh is contained within it. It is cubic in order to avoid the creation of unnecessary stretched tetrahedra coming from the application of meshing templates on stretched octants (assuming isotropy is desirable in the resulting mesh).

The terminal octants are constructed to be approximately the same size as any partially connected mesh face associated with them in order to ensure appropriate element sizes and gradations. This is done by visiting each mesh vertex in the initial surface mesh, computing the average size of the connected mesh edges, and refining the octree until any terminal octant around that vertex is at a level corresponding to that average size. The level of the octant is given by:

$$octlev = \log_2\left(\frac{rootlength}{size}\right) \tag{1}$$

where *rootlength* is the length of the root octant and *size* is the size of the mesh entity. It should be noted that this procedure does not theoretically ensure a match in size between every terminal octant and the partially connected mesh faces it knows about. However, since quadtrees are used to tirangulate model faces and constructed following the same rules as the octree, in practice, there is a match.

In order to have a smooth gradation between octant levels, no more than one level of difference is allowed between terminal octants that share an octant edge. Application of this rule can possibly lead to refinement

of some terminal octants past the level that was set by the partially connected mesh faces in their volumes.

Once the tree is completed, partially connected mesh faces are assigned to terminal octants. Given a mesh face, terminal octants that should know about it can be separated into two groups: (i) those that are in the path of each bounding mesh edge (obtained by intersecting line segments with axis aligned solid boxes) and (ii) those whose octant edges are in the path of the mesh face (obtained by intersecting line segments with triangles).

Any terminal octant which knows at least one partially connected mesh face is classified *boundary*. Terminal octants classified *boundary* separate *interior* terminal octants from *outside* terminal octants. At this point, it should be noted that the *interior* of the model can be made of several model regions. One octant corner of a *boundary* terminal octant is then classified either *interior* or *outside* by firing a ray toward a corner of the root octant. Considering the partially connected mesh face closer to the octant corner among the ones that intersect the ray, the classification corresponding to the model region on the side of the mesh face facing the octant corner is given to the octant corner [10]. If there is no intersection, the octant corner is classified *outside*. In case the intersection is on the boundary of the partially connected mesh face, no decision can be taken and a ray to another corner of the root is fired. The classification of the octant corner is then propagated to any neighboring terminal octant (in a recursive way) which has not been classified yet. The process of classifying an octant corner and propagating its classification continues until all terminal octants have been classified.

After the basic octant classification process, *interior* terminal octants can exist which have boundary entities arbitrarily close to surface triangles in *boundary* octants. Since poorly shaped elements can result when these entities are too close, some *interior* terminal octants are reclassified as *boundary*. If an *interior* terminal octant is too close to a partially connected mesh face, it is reclassified *boundary*. Distances between two entities are relative, that is, the actual distance should be divided by the average size of the entities involved. In this case, entities involved are octants (or octant faces) and mesh faces. The threshold for closeness is set to 1.0 to make room for potentially well shaped elements between *interior* terminal octants and surface triangles.

3.2. Template Meshing of Interior Octants

Terminal octants classified *interior* are meshed using (i) meshing templates or (ii) fast meshing procedures

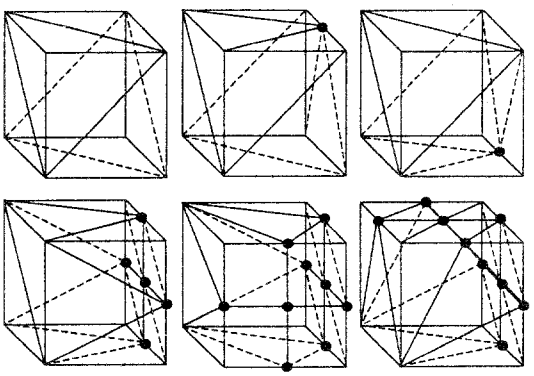

Fig. 3. Terminal octant meshing templates available: one eight vertex case, two nine vertex cases, one thirteen vertex case, and two seventeen vertex cases.

when a template is not available. Examination of the number of templates required for all cases and the distribution of template usage indicates that octants with eight, nine, thirteen, and seventeen vertices cover over 90% of the interior octants. All the nine, thirteen, and seventeen vertex octant configurations can be meshed by six templates (Fig. 3) with the correct rotations applied. The remaining interior octants are then quickly meshed using a fast procedure which accounts for the fact that the octant is a rectangular prism. One very fast option is to create an interior vertex and to create the correct connections to it [15].

3.3. Face Removal

Given a partially connected mesh face, a face removal consists of connecting it to a mesh vertex. Since the volume to be meshed consists of the space between the given surface triangulation and the interior octree, the vertex used is usually an existing one. However, in some situations, it is desirable to create a new vertex. The choice of the target vertex (existing or new) must be such that the created element is of good quality and its creation does not lead to poor (in terms of shape) subsequent face removals in that neighborhood.

The following pseudo-code indicates how the target vertex is selected for a given partially connected mesh face to be removed. Detailed explanation for the key steps is given in the next paragraphs of the section. In this pseudo-code and any other thereafter, **break** forces an exit from a loop, **return** forces an exit from the function or routine (in other words, the function terminates), and text between /* and */ denotes a comment [16].

1. Collect set of potential target vertices from tree neighborhood

2. Reorder target vertices with respect to decreasing shape measure (for the element to be created)
3. Initialize:
   a. *dist_lim* = α
   b. *target_vert* = 0
   c. *max_min_dist* = 0.0
4. **for** each potential target vertex *vert* {
   a. Perform preliminary check on acceptability. If not acceptable, **continue**
   b. If the new element contains any mesh vertex belonging to the front, **continue**
   c. If the new element intersects any existing mesh entity, **continue**
   d. Evaluate how close the new element is to existing mesh entities (compute relative minimum distance *min_dist*)
   e. **if** (*min_dist* $\geq$ *dist_lim*) {
      - *target_vert* = *vert*
      - *max_min_dist* = *min_dist*
      - **break**

      }
   f. **else if** (*min_dist* > *max_min_dist*) {
      - *target_vert* = *vert*
      - *max_min_dist* = *min_dist*

      }
   }
5. **if** (*max_min_dist* $\geq$ *dist_lim*) **return**
6. **if** *target_vert* = 0 {
   a. Create a new vertex *vert* at the best position for the partially connected mesh face to be removed
   b. *target_vert* = *vert*

   }
7. **else** {/* Consider creating a new vertex */
   a. Create a new vertex *vert* at the best position for the partially connected mesh face to be removed
   b. Evaluate closeness of new element to existing mesh entities (*min_dist*)
   c. **if** (*min_dist* > *max_min_dist*) *target_vert* = *vert* /* Better to create a new vertex */

   }

The neighborhood of an entity is defined as a tree neighborhood of a given order. Given a mesh entity, a tree neighborhood of order 0 consists of all terminal octants that know about the entity (have the entity or part of it within their volumes). A tree neighborhood of order $n(n > 0)$ consists of a tree neighborhood of order $n-1$ to which is added all terminal octants that neighbor any octant corner of any terminal octant in the tree neighborhood of level $n-1$. The set of potential target vertices is obtained via the partially connected mesh faces in the tree neighborhood of the appropriate order for the face in consideration. The set of potential target vertices should be as small as possible (for efficiency reasons) but should not be missing the best target (with respect to both shape of new element and closeness to nearby existing mesh entities) assuming all mesh vertices of the front were considered. A tree neighborhood of order 0 is clearly not enough while a tree neighborhood of order 1 is adequate assuming terminal octants have approximately the same sizes as the partially connected mesh faces they know about.

It is of interest to be able to discard potential target vertices as early as possible for purposes of efficiency. A potential target is kept if it satisfies any of the three following conditions (types):

1. connects to a bounding vertex of the face to be removed through a mesh edge of the front. This allows for the removal of partially connected mesh faces other than the face in consideration (not in all cases) and therefore leads to a reduction of the size of the front (guaranteeing convergence of the method).
2. is positioned inside the sphere centered at the best position (with respect to shape) for the fourth vertex of the face to be removed and of radius the size of the face to be removed. This avoids the creation of a stretched element with respect to the face in consideration.
3. any of the three bounding vertices of the face to be removed are positioned inside the sphere of any of the partially connected mesh faces connected to the target vertex. This allows for the creation of a stretched element with respect to the face in consideration which is not stretched with respect to partially connected mesh faces connected to the target.

Figure 4 shows potential target vertices of type 1, 2, and 3 for the face to remove.

Given a potential target vertex, one has to make sure that any new mesh entity (resulting from the

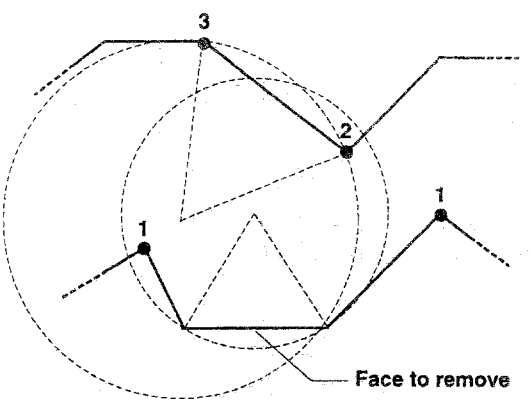

Fig. 4. The three types of potential target vertices (2-d setting).

creation of the new mesh region) does not intersect an existing mesh entity. The creation of a new mesh region may result in the creation of a new mesh vertex, up to three new mesh edges, and up to three new mesh faces. New mesh edges are checked for intersection against nearby partially connected mesh faces. Given a virtual new mesh edge, the nearby partially connected mesh faces are obtained through the tree neighborhood of order 0 (of the new edge). If no intersection is detected, new mesh faces are checked for intersection against nearby front mesh edges. Given a virtual new mesh face, nearby front mesh edges are obtained through the partially connected mesh faces in the tree neighborhood of order 0 (of the new face). Since any terminal octant knows about the partially connected mesh faces in its volume, considering a tree neighborhood of order 0 guarantees that no intersection can be missed.

The closeness of the new mesh region to existing mesh entities is evaluated by considering the minimum relative distance between any new mesh entity and nearby existing mesh entities. The relative distance is defined as the absolute distance divided by the average size of the mesh entities involved. The nearby mesh entities are obtained through a tree neighborhood of order 1 of the new entity being tested. It is important to note that nearby existing mesh entities in a tree neighborhood of order 1 may not be in a tree neighborhood of order 0. On the other hand, nearby existing mesh entities cannot be missed with a tree neighborhood of order 1. Considering the new region to be created, there can be zero or one new vertex, up to three new edges, and up to three new faces. If there is a new vertex, distances between the new vertex and nearby existing partially connected mesh faces are considered. For any new mesh edge, distances between the new edge and nearby existing front mesh edges are considered. If the point on the new edge corresponding to the distance (that is, closest to the nearby existing front mesh edge) corresponds to an existing bounding mesh vertex, the distance is discarded. In that case, it means that the nearby existing front mesh edge is close to another existing mesh entity and not to a new mesh entity. Also, for any new face, distances between the new face and nearby existing front mesh vertices are considered. Again, distances are discarded if the point on the new mesh face corresponding to the distance (that is, closest to the nearby existing front mesh vertex) is actually on an existing bounding mesh vertex or edge. The three different cases are shown in Fig. 5. The threshold α corresponds to what is considered acceptable in terms of closeness when creating a new element. Experimentation led to the use of a value of 0.2 for α.

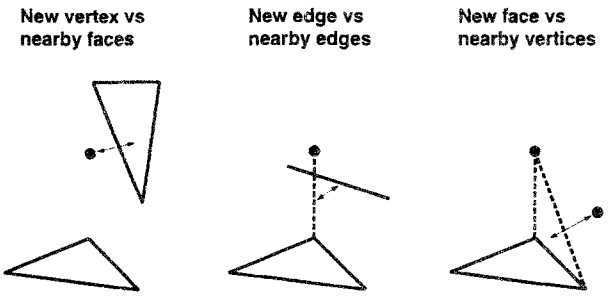

Fig. 5. Evaluation of relative minimum distance between new entities and nearby existing mesh entities.

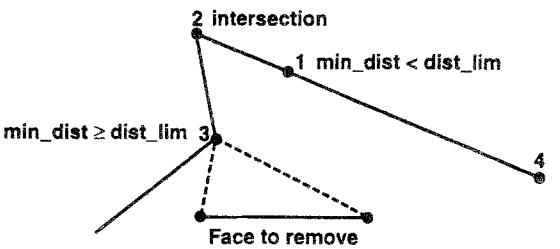

Fig. 6. Potential target vertices and best face removal (2-d setting).

If a new vertex needs to be created, its location must be such that the new element is well-shaped, and neither causes intersection nor is too close to nearby existing mesh entities. The initial location for the new vertex is at the position which creates the best shaped element for the face to be removed. This location is on the perpendicular to the face passing through the centroid. If the current location causes the new element to intersect nearby existing mesh entities, a new location is considered on the normal half-way from the current location, and so on, until a valid location is found. In order not to be too close to existing mesh entities, the final location is considered conservatively half-way from the current location.

Figure 6 graphically depicts a face removal in a two-dimensional setting. There are four target vertices ordered (1, 2, 3, and 4) with respect to increasing shape measure of the element to be created. Target vertex 1 is rejected since the new element is too close to an existing mesh entity (vertex 3). Target vertex 2 is rejected since the new element intersects existing mesh entities. Target vertex 3 is therefore accepted.

4. Parallel Constructs Required

4.1. Octree and Mesh Data Structures

The two main data structures are the octree and mesh data structures. The octree data structure is on top of

the mesh data structure. To gather a tree neighborhood or all terminal octants in the path of a mesh entity (new vertex, edge, or face), any processor must be able to effectively determine to which processor any given terminal octant is assigned. This information is easily available when each processor has full knowledge of the basic octree in terms of structure and processor assignment. This is the approach currently implemented. Although the size of the tree is small compared with that of the mesh and this tree information can easily be copied to each processor for meshes into the millions of elements, this approach does not scale indefinitely. Techniques that maintain only portions of the tree on individual processors are currently under investigation to provide the needed scalability. Any terminal octant stores links to on-processor partially connected mesh faces and off-processor partially connected mesh faces totally or partially within its volume. Octree neighboring information (like finding terminal octants neighboring an octant face, edge, or corner) is obtained through tree traversals. Techniques to reduce and/or avoid tree traversals are under investigation as well [17].

The mesh is stored in a version of Weiler's Radial Edge data structure [18] specifically defined for the efficient storage of meshes [19]. There is a two-way link between mesh entities of consecutive order (between regions and bounding faces, faces and bounding edges, and edges and bounding vertices). From this hierarchy, any entity adjacency relationship can be derived by local traversals. The entities on the partition boundary are augmented with interprocessor links which point to the location of the corresponding entities on neighboring processors [4–7]. Each partition boundary entity can have attached to it either the complete or the minimal set of interprocessor links. In the complete set, all the boundary entities store the location of the entity on the neighboring processor*. Since the lower entities inherit the higher order entity adjacency, it is possible to eliminate the interprocessor links for entities whose adjacency can be derived from higher order entities. This minimal link representation has the advantage of reducing the storage needed to maintain interprocessor link information. However, the minimal representation has the disadvantage of complicating the link update procedures when mesh regions or partially connected mesh faces are migrated. Therefore, a switching mechanism is used to allow for both representations to be used disjointly. Figure 7 shows the minimal representation on a two-dimensional example. The minimal representation can be used when the Finite Element mesh has become fully static, that is, cannot be subject to element migration.

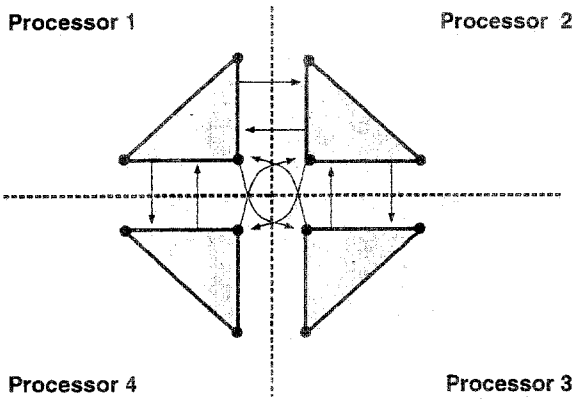

Fig. 7. Interprocessor links (minimal links).

4.2. Multiple Octant Migration

When the mesh generation process comes to a point where no face removal can be applied (face removals are not applied when needed tree neighborhoods are not fully on processor), the tree and associated mesh is repartitioned. The migration of octants is key to repartitioning once decisions concerning new destinations of terminal octants (classified *boundary*) have been made. Multiple octant migration itself relies on the multiple migration of partially connected mesh faces and/or mesh regions, the implementation details of which can be found in Reference 7. Note that multiple mesh region migration is also used in the final repartitioning at the region level once the mesh has been fully generated.

Any processor can send any number of terminal octants to another processor. When a terminal octant is migrated from one processor to another, the partially connected mesh faces not connected to any mesh region (these are the mesh faces remaining from the given surface triangulation) owned by the octant and/or the mesh regions that are bounded by at least one partially connected mesh face owned by the octant are migrated as well. An octant owns a mesh entity when it knows about it (has it within its volume) and has its centroid within its volume. Note that a partially connected mesh face not known by the octant may be migrated as part of a mesh region if that region is bounded by another partially connected mesh face whose owner is the octant. Also, if a mesh region is bounded by more than one partially connected mesh face known to the octant to be migrated (up to four), the ownership is arbitrarily dictated by the first

* In the context of a nonmanifold representation [18], these links represent links to use pairs where the interprocessor boundary is treated in the same manner as a material interface.

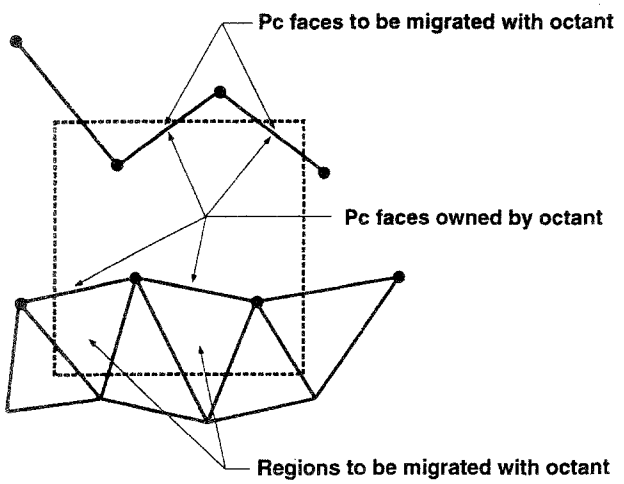

Fig. 8. Octant migration.

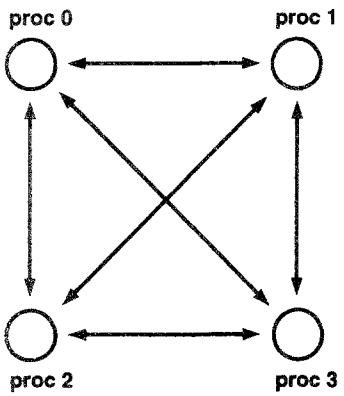

Fig. 9. Example of multiple octant migration.

partially connected mesh face to be processed (from the list of partially connected mesh faces known to the octant). Figure 8 shows an example of the mesh regions to be migrated within an octant. There is no limitation on the octant migration process, in particular, any processor can send and receive at the same time (note that a processor can even send to and receive from the same processor). Figure 9 shows an example of multiple octant migration in a four processor setting when all processors send and receive from all other processors. When the multiple octant migration completes, the processor is informed of the octants it has received. For each received octant, a list of associated mesh entities is also given, basically the partially connected mesh faces and/or mesh regions that were sent.

The primary complexity that rises when migrating octants and associated mesh information is the absence of a global labeling system for the mesh entities. Each processor employs a local labeling for the hierarchy of mesh entities that it is assigned. The interprocessor mesh adjacency links maintain the required knowledge of the adjacent mesh entities on neighboring processors. Although the mesh data for a partially connected face is on one processor, the octants which refer to that face may be on multiple processors. Since the face removal procedure must perform geometric checks on all partially connected faces known to that octant, the time required to perform these operations would be greatly increased if the required information had to be fetched from neighboring processors. To eliminate this requirement, each partially connected face known to an octant will either be a pointer to face when the face is actually on-processor, or a set of three coordinates when the face is stored off-processor. Although this approach avoids interprocessor communications, it complicates the process of updating references to partially connected mesh faces on- and off-processor when octants are migrated. Concerning the update of processor assignment at the octant level, since the tree structure is currently stored on all processors, a broadcast is performed to all processors indicating the fact that octants have been relocated.

4.3. Dynamic Repartitioning

Dynamic repartitioning enables to redistribute the load among processors as evenly as possible at key stages of the mesh generation process. The key stages are:

1. at the beginning of template meshing,
2. at the beginning of each face removal step, and
3. at completion of the mesh generation process.

Repartitioning for stages 1 and 2 is done at the terminal octant level (1 with respect to terminal octants classified *interior* and 2 with respect to terminal octants classified *boundary*). Repartitioning for stage 3 is performed at the mesh region level. The strategy is identical for both cases, only the process of migrating differs. In the following, it is assumed that repartitioning is done at the level of some entity which can be either a terminal octant or a mesh region.

Repartitioning relies on the Inertial Recursive Bisection (IRB) method [20] which is a variation of the more classic Orthogonal Recursive Bisection (ORB) [21]. ORB is a recursive process that bisects a set of entities by considering the median of the set of corresponding centroids with respect to a given coordinate axis. As ORB is recursively called, the choice of coordinate axis is circularly permuted (x, y, z, x, etc.). Unlike ORB, IRB considers the inertial coordinate system (origin is at the center of gravity and the three axes are the principal axes of inertia)

for the set of entities to be bisected. In three dimensions, the determination of the three principal axes of inertia is an eigenvalue problem of order 3. Once the inertial coordinate system is defined, the coordinates of the centroids are transformed and the cut is made at the median with respect to the first coordinate. This first coordinate will be the 'key' the sorting algorithm described later in this section works on.

The main assumption for performing repartitioning in parallel is that the entities are distributed. It is also assumed that there is no reason for the number of entities stored on processor to be uniform across processors. The result of this repartitioning will be an equal number of entities per processor. It should be noted that, in this context, the goal of repartitioning is equivalent to the goal of load balancing [4–7, 22, 23]. The key algorithm in IRB (and ORB) is the determination of the median for a given set of doubles (referred to as 'keys') [24]. With respect to this paper, the 'keys' are the first coordinates of the entities to be bisected. The method used here is to sort the 'keys' and then pick the entry at the middle of the sorted list. In this case, efficiently performing IRB in parallel can be reduced to efficiently sorting in parallel [25]. From the conclusions of a paper by Blelloch *et al.* [26] which compares different parallel sorting algorithms (Batcher's bitonic sort, radix sort, and sample sort), it appears that the sample sort algorithm is the fastest of the three for large data sets. Therefore, a parallel sample sort algorithm has been implemented in order to efficiently support IRB.

Given a set of n 'keys' distributed on p processors ($n \gg p$), a sample sort algorithm consists of three main steps:

1. $p-1$ splitters (or pivots) are chosen among the n 'keys'
2. Each key is routed to the processor corresponding to the bucket the 'key' is in
3. Keys are sorted within each bucket (no communication)

The goal of step 1 is to split the set of 'keys' into p parts (buckets) as evenly as possible and as efficiently as possible. The $p-1$ spliters which are implicitly sorted (say with respect to increasing value) are labeled from 1 to $p-1$. All distributed 'keys' below splitter 1 belong to bucket 0, all distributed 'keys' between splitter i ($0 < i < p-1$) and splitter $i+1$ belong to bucket i, and all distributed 'keys' above splitter $p-1$ belong to bucket $p-1$. Processor i ($0 \leq i < p$) is responsible for the bucket labeled i. In step 2, assuming the $p-1$ splitters have been found and broadcasted to all processors, any distributed 'key' can tell in which bucket it belongs and is rerouted to the processor that is responsible for that bucket. At this point, any processor has knowledge of all 'keys' that belong to the bucket it has been assigned to. Step 3 can be performed using any efficient sequential sorting algorithm, like quicksort [24]. It is clear that the parallel efficiency of the sample sort algorithm depends on the sizes of the buckets. Parallel efficiency is maximal when the sizes of the buckets are near equal. A sampling method is used to obtain 'good' splitters. Given the n input 'keys', ps 'keys' (s is an integer ≥ 1 called the oversampling ratio) are selected at random and sorted typically sequentially. The entries in the sorted list of ranks $s, 2s, \ldots, (p-1)s$ are the $p-1$ splitters. The bound for bucket expansion (ratio of maximum bucket size to average) is given in the paper by Blelloch *et al.* [26]. In practice, the oversampling ratio should be such that the sorting to find the splitters (which is done serially) does not become a bottleneck for the global parallel sample sort algorithm. For the purpose of the presented repartitioning technique, the oversampling ratio is chosen such that ps is of the order of n/p (n/p being of the order of the number of 'keys' to sort in step 3).

The following pseudo-code shows the process of repartitioning using IRB in parallel. It is assumed that the entities are already distributed on processors. A statement of the form **for** $(i = 0;\ i < n;\ i++)\ \{\ldots\}$ indicates a loop which gets executed as long as the loop variable i which begins at 0 ($i = 0$) and is incremented by 1 upon completion of each pass ($i++$) has a value less than n ($i < n$). A statement of the form **while** $(i < n)\ \{\ldots\}$ indicates a loop which gets executed as long as the loop variable i has a value less than n ($i < n$) [16]. Each processor executes the following pseudo-code (MIMD):

1. Associate each entity with a 'key' structure consisting of:
   a. 3 doubles for the coordinates of the entity's centroid with respect to the current inertial coordinate system (initially with respect to original coordinate system)
   b. 1 integer that indicates on which processor the actual entity is stored
   c. 1 pointer to the entity
   d. 1 integer that indicates the current processor destination for the entity (initially 0)
2. **for** $(step = 0;\ step < \log_2 p;\ step++)\ \{$
   a. $to\_pid = 0$
   b. **while** $(to\_pid < p)\ \{$
      - Balance the load such that each proc has approximately the same number of keys with a current destination equal to to_pid and reroute the keys accordingly

- Get center of gravity for the set of keys with a current destination equal to *to_pid*, find the three principal axes of inertia, and apply transformation to all keys with a current destination equal to *to_pid*
- Get p splitters among the keys with a current destination equal to *to_pid*
- Determine in which bucket each key with a current destination equal to *to_pid* goes and reroute the keys accordingly
- Sort on processor (bucket) the keys with a current destination equal to *to_pid*
- Assign a new destination to any key with a current destination equal to *to_pid* that is past the median ($to\_pid + p/2^{step+1}$)
- $to\_pid = to\_pid + 2^{\log_2 p - step}$

}

}

3. Reroute all keys to the orginating processors
4. Migrate entities according to the destination processor stored at the key level

5. Parallel Region Meshing

5.1. Underlying Octree

At this point in time, the octree is built sequentially on a single processor (processor 0). A sequential octree building may become a bottleneck when dealing with very large meshes. It should be noted that the octree building will be accomplished in parallel in the future to allow for scalability.

5.2. Template Meshing of Interior Octants

Once all terminal octants have been properly classified, the terminal octants classified *interior* are partitioned. The parallel application of templates is a strightforward process in which there is no communication required during the process of creating the octant level meshes. It should be noted that the application of templates to octants sharing the same octant face implicitly lead to the same octant face triangulation. The finite elements generated in these octants are loaded into the processor mesh data structure. The interprocessor communication required at the end of this step is for the updating of interprocessor mesh entity links for mesh entities created on the boundaries of interior octants which are on processor boundaries. The cost for the application of templates is small compared with the cost of performing face removals. Therefore, parallel efficiency of parallel region meshing is dictated by the face removal part only.

5.3. Face Removal

Parallel face removal is an iterative process where each iteration consists of three steps:

1. Tree repartitioning at the terminal octant (classified *boundary*) level,
2. Face removal step, and
3. Reclassification of terminal octants from *boundary* to *meaningless*

The goal of step 1 is to make sure that all processors will have an equal amount of work to perform during step 2. It is difficult to predict how much work or, more precisely, how many face removals (step 2) any processor will perform and the total amount of effort for a particular face removal. However, a terminal octant classified *boundary* is a good unit of work load since the set of all terminal octants classified *boundary* approximately corresponds to the domain still to be meshed. The difficulty of performing face removals in parallel resides in the fact that any face removal requires the knowledge of tree neighborhoods. Tree neighborhoods of order 0 or 1 are needed at different steps of the removal of a given mesh face. If, at any point during the face removal, a tree neighborhood is not fully on-processor, the face removal is aborted and the next mesh face is considered for removal. Once all possible face removals have been performed on processor, some terminal octants classified *boundary* which used to know about partially connected mesh faces (on- or off-processor) are reclassified *meaningless*. As those octants do not now cover any portion of the domain still to be meshed, they are now useless (for the purpose of face removals) and will therefore not influence the next repartitioning.

Figure 10 depicts the first iteration on a simplistic example. In the left-side picture, terminal octants classified *boundary* have been partitioned and each of

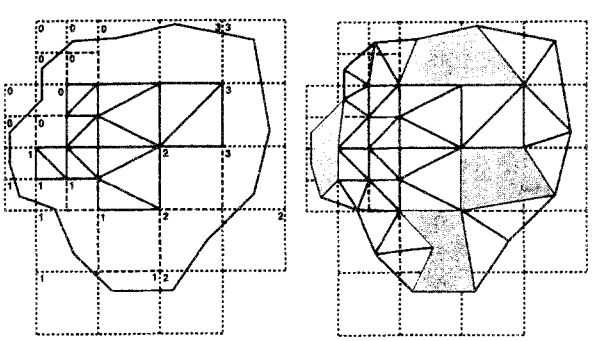

Fig. 10. Parallel face removal (2-d setting).

them is assigned to a processor (0 to 3). The right-hand side picture shows the current mesh after all possible face removals have been performed on processors. Shaded areas represent the domain still to be meshed.

The process of performing face removals and repartitioning the tree continues until there are no more partially connected mesh faces in the mesh. Define the efficiency of the face removal stage as being the ratio of the number of performed face removals to the number of attempted face removals. After a few iterations, the efficiency of the face removal stage can be very low because information required to perform face removals is almost always off-processor. When more than half of the processors have an efficiency below some given threshold (25%), the processor set is reduced (by half).

Since migration of terminal octants only deals with those classified *boundary* and only worries about mesh regions bounded by partially connected mesh faces, it is very likely that the final mesh will be scattered across processors with no real structure. It is therefore necessary to repartition in parallel the distributed mesh using IRB at the mesh region level with the original full set of processors. Figure 11 shows the

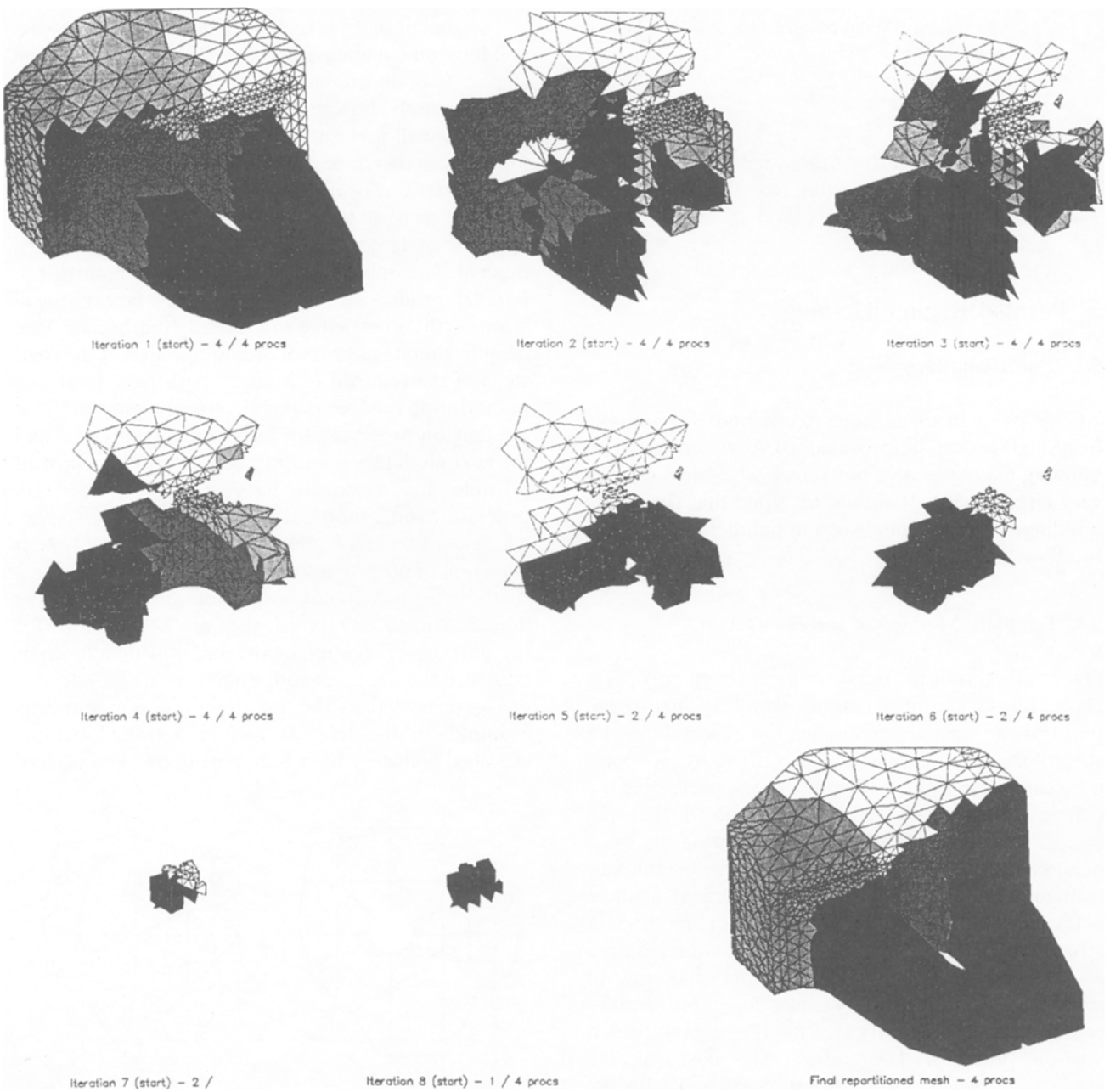

Fig. 11. Successive face removal iterations and final repartitioned mesh for *chicklet*.

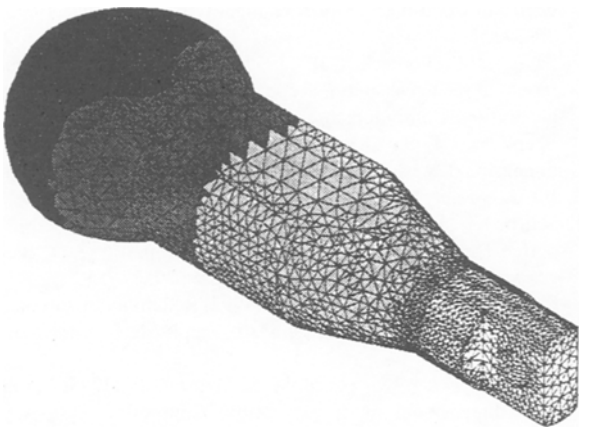

Fig. 12. Final repartitioned mesh for *connecting rod* (4 processors).

Table 1. Face removal statistics for *connecting rod* (initially 21,000 partially connected mesh faces—35,000 mesh regions created).

Procs	1	2	4
Iterations	1	5	7
Face removal speedup	1.0	1.9	3.3
Total speedup	1.0	1.8	2.9

whole process of parallel face removal on four processors. The first 8 pictures display the currently partially connected mesh faces after the terminal octants classified *boundary* have been repartitioned. Note that iterations 1, 2, 3, and 4 use all four processors, iterations 5, 6, and 7 use two processors, and iteration 8 uses one processor. The final picture displays the final three-dimensional repartitioned mesh on four processors.

Table 1 shows speed-ups for up to four processors for a *connecting rod* model (final repartitioned mesh on four processors is shown in Fig. 12). Table 2 shows speed-ups for up to four processors for a *blade* model (final repartitioned mesh on four processors is shown in Fig. 13). The number of mesh regions created indicated in the captions corresponds to parallel face removal only and does not include template meshing. Face removal speed-up indicates speed-up for step 2 of the parallel face removal procedure. Total speed-up indicates speed-up for all steps (1, 2, and 3). In that case, the first repartitioning (iteration 1) is not counted since it can be considered an initial partitioning step. Note that the time taken to perform the first repartitioning depends on the size of the problem and not the number of processors. If the number of processors is one, the speed-up is by definition set to 1. It should be noted that speed-up on p processors

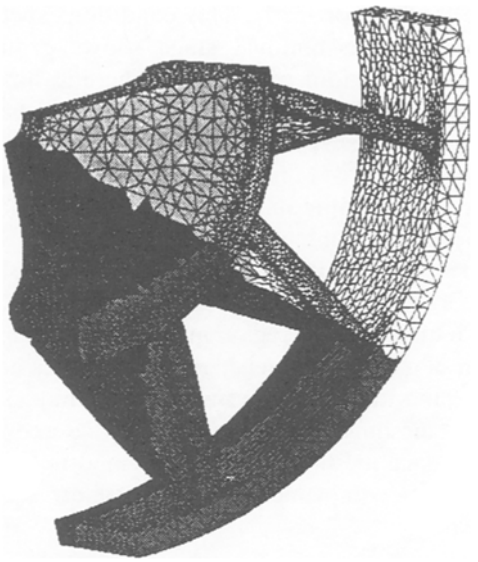

Fig. 13. Final repartitioned mesh for *blade* (4 processors).

Table 2. Face removal statistics for *blade* (initially 35,000 partially connected mesh faces—60,000 mesh regions created).

Procs	1	2	4
Iterations	1	5	8
Face removal speedup	1.0	2.0	3.2
Total speedup	1.0	1.9	2.8

is defined as the time spent by the 'best' sequential version of the algorithm divided by the time spent by the parallel version of the algorithm running on p processors [25]. If there is no or little difference between the sequential algorithm and the parallel algorithm running on 1 processor (which is true here), the speed-up can be simply defined as the time spent by the parallel version of the algorithm running on 1 processor divided by the time spent by the parallel version of the algorithm running on p processors. The procedure demonstrates near-perfect speed-ups for two processors and 'good' speed-ups for four processors.

6. Closing Remarks

This paper has presented a method to mesh objects in three dimensions for operation on MIMD parallel computers. The parallel efficiency of the overall method is dictated by the performance of the parallel face removal stage. It is difficult to predict adequately the amount of work a processor will perform and to

balance the load properly. This conditions speed-up. The parallel face removal stage shows promising speed-ups (up to four processors). There is, however, still work to be done to ensure a better load balance and even better speed-ups. At the moment, the tree structure is stored on all processors. Scalability (memory wise) will require the tree to be distributed. Also, the tree building procedure and related tasks are performed serially. Tree related nonscalable tasks are being worked on and will be solved in the near future to have a fully scalable region meshing algorithm. The problem of meshing model edges and surfaces has not really been addressed here since the techniques used in region meshing can be applied in these contexts. Edge and face meshing will be presented in the near future as a complement to the region meshing procedure.

Acknowledgment

The authors would like to acknowledge the support of NASA Ames Research Center under grants NAG 2-832 and NCC 2-9000, and the National Science Foundation under grant DMS-9318184.

References

1. George, P. L. (1991) Automatic Mesh Generation, Chichester, John Wiley and Sons
2. Shephard, M. S.; Weatherill, N. P. (Editors) (1991) Int. J. Numer. Meth. Engng., Vol 32, Chichester, Wiley-Interscience
3. Weatherill, N. P.; Hassan, O. (1994) Efficient three-dimensional Delaunay triangulation with automatic point creation and imposed boundary constraints, Int. J. Numer. Meth. Engng., 37, 2005–2039
4. de Cougny, H. L.; Devine, K. D.; Flaherty, J. E.; Loy, R. M.; Ozturan, C.; Shephard, M. S. (1994) Load balancing for the parallel solution of partial differential equations, Applied Numerical Mathematics, Vol 16, pp 157–182
5. Ozturan, C., de Cougny, H. L.; Shephard, M. S.; Flaherty, J. E. (1994) Parallel adaptive mesh refinement and redistribution on distributed memory machines, Comp. Meth. Appl. Mech. Engng., 119, 123–137
6. Shephard, M. S., Bottasso, C. L., de Cougny, H. L.; Ozturan, C. (1994) Parallel adaptive finite element analysis of fluid flows on distributed memory computers. In Recent Developments in Finite Element Analysis, pp 205–214, Barcelona, Int. Center for Num. Meth. in Engng.
7. Ozturan, C. (1995) Dynamic load balancing for adaptive finite element methods, PhD Thesis, Rensselaer Polytechnic Institute, Troy NY
8. Löhner, R.; Camberos, J.; Merriam, M.; (1992) Parallel unstructured grid generation, Comp. Meth. Appl. Mech. Engng., 95, 343–357
9. Saxena, M.; Perucchio, R. (1992) Parallel FEM algorithms based on recursive spatial decompositions—I. Automatic mesh generation, Computers and Structures, 45, 817–831
10. Shephard, M. S.; Georges, M. K. (1991) Automatic three-dimensional mesh generation by the Finite Octree technique, Int. J. Numer. Meth. Engng., 32(4); 709–749
11. Schroeder, W. J.; Shephard, M. S. (1990) A combined octree/Delaunay method for fully automatic 3-D mesh generation, Int. J. Numer. Meth. Engng., 29, 37–55
12. Shephard, M. S.; Georges, M. K. (1992) Reliability of automatic 3-D mesh generation, Comp. Meth. Appl. Mech. Engng, 101; 443–462
13. de Cougny, H. L.; Shephard, M. S.; Ozturan, C. (1995) Parallel three-dimensional mesh generation, Computing Systems in Engineering, to appear
14. Schroeder, W. J.; Shephard, M. S. (1991) On rigorous conditions for automatically generated finite element meshes. In Product Modeling for Computer-Aided Design and Manufacturing, Turner J.; Pegna, J.; Wozny, M. (Editors) Amsterdam, North-Holland
15. Yerry, M. A.; Shephard, M. S. (1984) Automatic three-dimensional mesh generation by the modified-octree technique, Int. J. Numer. Meth. Engng. 20, 1965–1990
16. Kernigham, B. W.; Ritchie, D. M. (1990) The C Programming Language, Englewood Cliffs, NJ 07632, Prentice Hall
17. Samet, H. (1984) The quadtree and related hierarchical data structures, Computing Surveys, 16, ACM Comput. Surveys, Vol 16, no 2, June 1984 pp 187–260
18. Weiler, K. J. (1988) The radial-edge structure: A topological representation for non-manifold geometric boundary representations. In Geometric Modeling for CAD Applications, Wozny, M. J.; McLaughlin, H. W.; Encarnacao, J. L. (Editors) pp. 3–36, North-Holland
19. Beall, M. W. (1993) Scorec mesh database users guide, version 2.2—draft, Technical Report SCOREC #26-1993, Scientific Computation Research Center, Rensselaer Polytechnic Institute, Troy, NY
20. Löhner, R.; Ramamurti, R. (1993) A parallelizable load balancing algorithm. In Proc. of the AIAA 31st Aerospace Sciences Meeting and Exhibit
21. Berger, M. J.; Bokhari, S. H. (1987) A partitioning strategy for nonuniform problems on multiprocessors, IEEE Transactions on Computers, C-36(5), 570–580
22. Leiss, E.; Reddy, H. (1989) Distributed load balancing: Design and performance analysis, Technical Report Vol. 5, W. M. Keck Research Computation Laboratory
23. Vidwans, A.; Kallinderis, Y.; Venkatakrishnan V. (1994) Parallel dynamic load-balancing algorithm for three-dimensional adaptive unstructured grids, AIAA Journal, 32(3), 497–505
24. Sedgewick, R. (1990) Algorithms in C, Reading MA, Addison-Wesley Publishing Company
25. JaJa, J. (1992) An Introduction to Parallel Algorithms, Reading MA, Addison-Wesley
26. Blelloch, G.; Leiserson, C.; Maggs, B.; Plaxton, C.; Smith, S.; Zagha, M. (1991) A comparison of sorting algorithms for the connection machine cm-2, ACM, 089791-438-4191, p 3–16