# A Distributed Octree Structure and Its Application to Parallel Mesh Generation


Authors and Affiliations:

Michelle L. Simone (simonm@rpi.edu)

United Technologies Research Center

411 Silver Lane

East Hartford, Connecticut 06108

Raymond M. Loy (loyr@cs.rpi.edu)

Scientific Computation Research Center

Rensselaer Polytechnic Institute

110 8th St.

Troy, NY 12180-3590

Mark. S. Shephard (shephard@scorec.rpi.edu)

Scientific Computation Research Center

Rensselaer Polytechnic Institute

110 8th St.

Troy, NY 12180-3590

Joseph. E. Flaherty (flaherje@cs.rpi.edu)

Scientific Computation Research Center

Rensselaer Polytechnic Institute

110 8th St.

Troy, NY 12180-3590

Running Head: A Distributed Octree Structure and Algorithms

Author to whom proofs should be sent:

Michelle L. Simone

510 Fillmore Avenue

Schenectady, NY 12304

phone: (518) 377-3510

simonm@rpi.edu

Abstract:

A distributed octree data structure was developed to support efforts in parallel mesh generation. The distributed octree augments a basic hierarchical octree structure to include interprocessor links to off processor octants, and also includes lateral links between octants of the same level which share common faces in the octree topology. These lateral links, known as *face neighbor links*, support $O(1)$ neighborhood queries during mesh generation. Two basic algorithms are needed to construct a distributed octree with *face neighbor links*. An octant migration procedure supports an arbitrary redistribution of octants across the processors while maintaining the octree connectivity. An octant refinement algorithm allows octants to be allocated in parallel while maintaining local tree links, and interprocessor tree links between remote octree face neighbors. We describe how the distributed octree and algorithms are implemented inside an octree-based parallel mesh generator. Performance results for all of the algorithms are presented on a 32 processor IBM SP2.

Key Words: octree, mesh generation, neighbor-finding

# A Distributed Octree Structure and Its Application to Parallel Mesh Generation

## 1. Introduction

The introduction of scalable parallel computers have given engineers and scientists the ability to incorporate more complex physical behaviors into their finite element models. The need for meshes consisting of millions of elements is required in order to obtain accurate solutions to many problems. As mesh sizes become this large, mesh generation in a uniprocessor environment becomes problematic in terms of time and storage. This paper describes a distributed octree structure and algorithms which can efficiently support unstructured mesh generation on distributed-memory computers.

Quadtrees and octrees have been successfully used as a spatial data structure for unstructured finite element mesh generation on serial architectures. Work by Yerry and Shephard [28], and later by Baehmann et al. [2] use quadtrees to (i) control the size and gradation of elements in a two-dimensional finite element domain, and (ii) retrieve existing mesh data in specific neighborhoods near the regions being meshed. This work has been extended to the use of octrees for three-dimensional unstructured mesh generation by Yerry and Shephard [29], Schroeder and Shephard [23], Shephard and Georges [25], and Kela [13].

In the distributed-memory environment, the octree is also used to divide a geometric model across a set of processors so that the individual partitions can be meshed in parallel. The distributed octree described in Section 2 augments a basic hierarchical octree structure to include interprocessor links to off processor octants, and also includes lateral links between octants of the same level which share common faces in the octree topology. These lateral links, known as *face neighbor links*, support $O(1)$ neighborhood queries (Section 4) during mesh generation. Two basic algorithms, presented in Section 3, are needed to construct a distributed octree with *face neighbor links*. An octant migration procedure supports an arbitrary redistribution of octants

across the processors while maintaining the octree connectivity. Octant migration and mesh migration procedures [19, 20] are needed to support dynamic load balancing during parallel mesh generation. An octant refinement algorithm allows octants to be allocated in parallel while maintaining local tree links, and interprocessor tree links between remote octree face neighbors.

Section 5 contains an overview of an octree-based parallel mesh generator [5-7, 27], and describes how the distributed algorithms support each of the meshing procedures. In Section 6, scalability experiments on the octant migration and octant refinement are provided on a 32 processor IBM SP2, as well as results comparing traversal-based octree neighbor-finding with the lateral approach to octree neighbor-finding presented in Section 4. Finally, Section 7 contains a summary of the work performed.

## 2. The Octree Data Structure and Notation

An octree is a hierarchical data structure which corresponds to a recursive subdivision of a portion of Euclidean space into eight octants wherever more resolution is required. Octant subdivision can be based on any number of application specific criteria, and easily allows non-uniform refinement. In the case of mesh generation, octant subdivision is based on the desired size of the elements in specific regions of the domain. An illustration of an octree structure and its corresponding octree geometry is shown in figure 1 (a). Specific octants of interest are labeled at the centroid of their front or side face. The root octant $O_0$ represents the largest closed space and is constructed to enclose the geometric domain of interest. Subdividing the root into eight equal child octants gives one level of refinement. Subdividing again, as done for octant $O_8$, gives another level of refinement. The *terminal octants* at the bottom of the hierarchy represent the finest levels of refinement whose union is equal to the space occupied by the root octant $O_0$.

### 2.1. Octree Definitions and Notation

We define the set $O$ as the set of octants $O_i \in O$ describing a given octree rooted by $O_0$.

We define an octant $O_i$ uniquely as

$$O_i = (parent, child\text{-}id) \quad (1)$$

where the subscript $i$ is a unique numerical identification of an octant. The *parent* of $O_i$ is the octant member directly above $O_i$ in the octree hierachy and

$$child\text{-}id = \{n \in \mathbf{N} \mid 0 \le n \le 7\}. \quad (2)$$

the index $n$ is a member of the set of natural numbers and corresponds to the specific order of $O_i$ with respect to its *siblings*. The root $O_0$ is denoted as $(0, 0)$.

We define the set of *children* of octant $O_i$ as $\{O_c \in O \mid O_i = parent(O_c)\}$, and the set of *siblings* of $O_i$ as $\{O_s \in O \mid parent(O_i) = parent(O_s)\}$. We say that $O_a$ is an *ancestor* of $O_i$ iff $O_a = parent(O_i)$ or $O_a = ancestor(parent(O_i))$. We define a *descendent* of $O_i$ as an octant which is a $child(O_i)$, or a $descendent(child(O_i))$. The *level* of $O_i$ is 0 for $O_i = O_0$ and $level(parent(O_i)) + 1$ otherwise.

The recursive subdivision imposes a topology on the subdivided space, where the topology consists of the sets of octant regions, $O^3$, octant faces, $O^2$, octant edges, $O^1$, and octant vertices, $O^0$. A superscript in the notation indicates the dimension of an octant topological entity and was selected to be consistent with notation on finite element mesh data structures in Beall and Shephard [3]. Let

- $\overline{O_i}$ be the closure of octant $O_i$.
- $O_i^3$ be the region of space associated with octant $O_i$.
- $\{O_i^2\}$ be the set of six faces bounding octant $O_i$.
- $\{O_i^2\}_d$ be the $d^{th}$ face bounding octant $O_i$, where $d$ refers to the direction of the normal of the octant face ($+x$, $-x$, $+y$, $-y$, $+z$, $-z$, see Figure 1 (a))

$\{O_i^1\}$ be the set of twelve edges bounding octant $O_i$.

$\{O_i^0\}$ be the set of eight vertices of octant $O_i$.

There are six octant faces, and, therefore, six directions for face normals. It is also convenient to refer to specific edges and vertices of $O_i$. A particular octant edge is identified by the normal directions of the two faces that it bounds. Similarly, a particular octant vertex is identified by the normal directions of the three faces that it bounds.

It is of interest to define sets of octree neighbors which share common octant topological entities and are needed throughout the mesh generation computations. We define the set of *terminal face neighbors* of octant $O_i$ as the set of terminal octants whose associated faces intersect $\{O_i^2\}$ or,

$$\{O_n \in O \mid \{O_n^2\} \cap \{O_i^2\} \neq \varnothing \wedge (children(O_n) = \varnothing) \wedge (O_n \neq descendent(O_i))\}. \quad (3)$$

The set of *terminal face neighbors* of octant $O_i$, in a specific direction $d$, is a subset of those members defined in (3), which intersect $\{O_i^2\}_d$, or

$$\{O_n \in O \mid \{O_n^2\} \cap \{O_i^2\}_d \neq \varnothing \wedge (children(O_n) = \varnothing) \wedge (O_n \neq descendent(O_i))\}. \quad (4)$$

Similarly, the set of *terminal edge neighbors* of octant $O_i$, which intersect a particular edge $\{O_i^1\}_d$ is defined as

$$\{O_n \in O \mid \{O_n^1\} \cap \{O_i^1\}_d \neq \varnothing \wedge (children(O_n) = \varnothing) \wedge (O_n \neq descendent(O_i))\}, \quad (5)$$

while the set of *terminal vertex neighbors* of octant $O_i$, which intersect a particular vertex $\{O_i^0\}_d$ is defined as

$$\{O_n \in O \mid \{O_n^0\} \cap \{O_i^0\}_d \neq \emptyset \wedge (children(O_n) = \emptyset) \wedge (O_n \neq descendent(O_i))\} \quad (6)$$

*2.2. A Distributed Octree Structure and Implementation*

In a distributed-memory environment, we extend the basic definition of an octree to include links across processors. Figure 1 (b) depicts the data representation of the octree in Figure 1 (a) when it is distributed across two processors. An *off processor child* stores the processor id and pointer to a remote child, while an *off processor parent* stores the processor id and pointer to a remote parent. Thus, storage for local links is low, since processor ids are only stored for relatives that are off processor.

Distributed octree structures also need local roots. An octant and its parent may not exist on the same processor, as is the case for $O_8$ on processor $P_1$, in Figure 1. We shall refer to any octant whose parent is stored on a different processor, as a *local root*. Each processor maintains a list of *local roots* which allows all octants on a processor to be accessed, without interprocessor communication, by traversing through the subtrees rooted at the *local root list*. *Local roots lists* and *off processor octants* are set and maintained during octant migration, which is described in Section 3.

In addition to the basic parent and child connectivity in the octree, the storage of a particular octant includes pointers to octants of *equal* or *greater* size sharing each of an octant's faces, referred to as *face neighbor links*. A pointer to an octant of equal size is a lateral link to an octant at the same level in the octree hierarchy. *Equal face neighbors* of $O_i$ are define as

$$\{O_n \in O \mid (\{O_n^2\} \cap \{O_i^2\}) \wedge level(O_i) = level(O_n)\} . \quad (7)$$

When a given *equal face neighbor* of $O_i$ does not exist (due to the varying levels of refinement in the octree hierarchy), $O_i$ *inherits* its parent's face neighbor in the direction $d$. An *inherited face*

*neighbor* of $O_i$ is a pointer to a terminal octant of larger size

$$\{O_n \in O \mid (\{O_n^2\} \cap \{O_i^2\}_d) \wedge (level(O_i) > level(O_n)) \wedge (children(O_n) = \varnothing)\}. \qquad (8)$$

*Face neighbor links* provide a mechanism for lateral movement within the tree structure which supports the lateral approaches to octree neighbor-finding (Section 4).

Figure 2 shows a portion of the octree in Figure 1 with *face neighbor links* indicated by arrows (not all are shown). A *face neighbor link* in the direction $d$ is denoted by the Cartesian symbol which indicates the direction in which it is pointing. For example, octant $O_6$*'s equal face neighbor* in the +*y* direction is octant $O_8$. Conversely, octant $O_8$'s *equal face neighbor* in the -*y* direction is octant $O_6$. Thus, two-way lateral links exist between equal octree face neighbors. *Inherited face neighbor links*, such as those from $O_{10}$ to $O_6$ and from $O_{14}$ to $O_6$ are one-way links to octants of greater size. Therefore, each octant will store six links to either their equal or inherited face neighbors. *Face neighbor links* are created as the tree is generated by the octant refinement operator described in Section 3. Face neighbor links must be maintained as the octree is distributed or redistributed through migration.

The parallel octree also supports the duplication of an octant to multiple processors if desired by the application. This functionality supports procedures which create overlapping neighborhoods, or "ghost" data [8, 9], on octree partition boundaries. Copied octants are either linked to their real parents, children, and face neighbors or to copied instances of them. Thus, an application may selectively visit only regular octants, or both regular octants and any existing copies.

A particular instance of an octant is distinguished from all other instances by its association to a processor model. We use the classification symbol given in [3], to associate an octant $O_i$ in the octree model $O$, with a processor $P_j$ as follows:

$$O_i \sqsubset P_j. \tag{9}$$

Octant copies are not automatically maintained if the application alters the tree either by redistribution or refinement. Typically, if the tree is altered by redistributing the octants, the set of copies from the previous distribution may no longer be valid for the current distribution. Furthermore, the octree design does not allow both an octant and a copy of itself to exist on the same processor. Therefore, each time octants are migrated (or redistributed), all existing copies are freed in order to avoid a situation where an octant and its copy exist on the same processor. An application can update altered octant copies by invoking another octant copy-migration.

## 3. Tools for Parallel Octree Building

This section describes two procedures which distribute and refine an octree with face neighbor links. An octant migration operator supports an arbitrary redistribution of the octants across the processors while maintaining the octree connectivity. An octant refinement algorithm allows octants to be allocated in parallel while maintaining local tree links, and interprocessor tree links between remote octree face neighbors.

### *3.1. An Octant Migration Operator*

Given an initial distribution of an octree $O$ across a set of processors $P$, an application must determine which octants to migrate to which processors to maintain a balanced computation. For convenience, we define the set of migrating octants in the tree as $O_M$. We define the subset $O_{M_s} \subset O_M$ as the set of migrating octants that are stored on source processor $P_s$, i.e.,

$$O_{M_s} = \{O_i \in O_M \mid O_i \sqsubset P_s\}. \tag{10}$$

Clearly

$$\bigcup_{s=1}^{N_p} O_{M_s} = O_M, \tag{11}$$

where $N_p$ is the total number of processors, and

$$O_{M_i} \cap O_{M_j} = 0, \quad i \neq j, \; i, j = 1, \ldots, N_p. \tag{12}$$

We further define the subset $O_{M_d}$ as the set of octants that will be physically stored on the destination processor $P_d$ after migration is completed,

$$O_{M_d} = \{O_i \in O_M \mid O_i \sqsubset P_d\}. \tag{13}$$

The relationships  and (12) also hold for $O_{M_d}$.

Given the set $O_M$ and the set of destination processors $P_D$, octant migration proceeds to send the migrating octants to their destination processors in the four main steps that are within the **FOR ALL** loop of Figure 3. Each step runs concurrently on the processor set $P$. Migrating octants are sent to their destinations using the message passing functionalities of MPI [18]. For ease of illustration, octant migration in a two-dimensional setting (quadrant migration) is provided in Figures 4 and 5. Quadrants targetted for migration to other processors are circled according to the shading of their destination processor in Figure 4 (a). Thus, $O_M$ consists of $O_2 \sqsubset P_0$, the *descendents* of $O_2 \sqsubset P_0$, $O_3 \sqsubset P_0$, and $O_4 \sqsubset P_0$, while $P_D = \{P_1, P_2, P_3\}$, and $P_S = \{P_0\}$.

In the first step of octant migration, each migrating octant must have space allocated for its data on its destination processor. In steps 1.1-1.2, each source processor $P_s$ cycles through each $O_i \in O_{M_s}$ and sends allocation requests to the appropriate destination processors $P_d$. Figure 4 (b) shows an allocation request sent for octant $O_2 \sqsubset P_0$ to its destination $P_1$. In order to avoid synchronizing for each migrating octant, allocation requests are not received by the destination pro-

cessors until each member of $P_S$ has sent out all of its allocation requests. The destination processors must subsequently post an equivalent number of receives to match the number of sends.

In order for a $P_d$ to know the number of allocation requests to receive, each $P_s$ initializes an integer array of length $N_p$, the number of members in $P$. When a $P_s$ processor sends a non-blocking send to a $P_d$ processor, $P_s$ increments by one the element in the array corresponding to the processor member $P_d$. The set of processors then perform an MPI reduction [18] on this processor array in step 1.3. The result of this parallel reduction is that each member of $P$ knows how many messages it must receive from all other members of $P$ at the expense of an $O(N_p \log N_p)$ operation [12].

Once a given destination processor knows the number of MPI receives to post, it subsequently processes each allocation request and responds back to a given $P_s$ with the new address in step 1.4. Figure 4 (b) shows $P_1$ responding back to $P_0$ with the new address of octant $O_2 \sqsubset P_0$. Since a given $P_s$ knows that it will receive $N_{M_s}$ new addresses, where $N_{M_s}$ is the number of members in $O_{M_s}$ on $P_s$, the source processors may immediately begin receiving the new octant addresses in step 1.5 without performing a parallel reduction on the processor array. Thus, each migrating octant will know its new address at the completion of the first phase of migration, in time proportional to approximately $max(N_{M_s}),\ s = 1, \ldots, N_p + O(N_p \log N_p)$. For the mesh generation application, the growth rate of octant migration will be dependent upon $max(N_{M_s})$ since $N_{M_s} \gg N_p$.

Since an application is typically interested in using the tree to provide information regarding the data contained within the octants, it is convenient to keep the migrating octants and their corresponding data on the same processor. In the second phase of migration, the data contained within each member $O_i \in O_{M_s}$ is updated regarding the new address $O_i \sqsubset P_d$. In the mesh generation application, only terminal octants contain direct links to their bounding mesh data; thus, this phase of migration only affects those migrating terminal octants with associated mesh data.

This allows the mesh data to be subsequently migrated by the mesh migration operator [19, 20] along with their bounding octants.

All members of the octree $O$ connected to a migrating octant must also be updated so that their tree links point to the correct octants after the migrating octants are moved to their final destinations. The third stage of octant migration is responsible for updating the affected parent's, children, and face neighbors of each migrating octant on the affected processors $P_A$. In order to update all affected octants, a source processor $P_s$ cycles through each $O_i \in O_{M_s}$ and either updates their relatives and face neighbors locally if they reside on the same processor (as is the case for the quadrants in Figure 4), or sends update messages to each remote relative and face neighbor on $P_A$, using a non-blocking MPI send in step 3.2. After each $P_s$ has sent all of its link updates, the processor set $P$ performs a parallel reduction in the **while** loop on line 3.3 to compute the number of MPI receives that each member of $P_A$ must post to match the number of MPI sends. The affected processors subsequently cycle through all their incoming link updates and inform the appropriate *parent*, *child*, or *face neighbors* connected to a given $O_i \in O_{M_s}$ about its new address $O_i \square P_d$.

A given *face neighbor* pointing to a migrating octant $O_i$ may also need to forward a link update to their *descendents iff* those *descendents* have *inherited face neighbor links* to $O_i \in O_{M_s}$. In this case, each level of the tree is traversed from the *equal face neighbor* down to all of the *equal face neighbors' descendents* with inherited links to $O_i \in O_{M_s}$. If *off processor children* are encountered during this downward traversal, link updates must be forwarded to the appropriate processor $P_a$. Link updates which are forwarded to *off processor children* are received in the next cycle through the **while** loop in step 3. When the tree levels are adjusted such that the level differences between terminal edge neighbors is bound by one (2:1 criteria [2, 7]), as is done in mesh generation, there can only be two iterations through the while loop. This result follows from the fact that octants with inherited face neighbor links to a migrating octant may only be separated by one octree level from their migrating inherited face neighbor. Thus, in the worst case, updating an

inherited neighbor link may require one off processor hop to update the equal neighbor of a migrating octant, and a second off processor hop to update the inherited neighbor links of the equal neighbor's *terminal* children with links to the migrating octant (if the children had descendents, the 2:1 criterion would be violated). The processing which is done during the third phase of migration is proportional to $max(N_{M_s})$. Some of the link updates which are processed for $O_2 \sqsubset P_0$ are illustrated in Figure 5 (a).

Once all the affected relatives and *face neighbors* of the migrating octants have been updated, the data contained within the migrating octants is moved to its final destination in the fourth step of octant migration. The *local root lists* on each processor are also updated to reflect the changes in the distribution of the octree. Octant migration cycles through each $O_i \in O_{M_s}$ and sends all the data associated with $O_i \sqsubset P_s$, to its destination in step 4.1.

Each $O_i \in O_{M_s}$ is subsequently freed on $P_s$ during a regular migration in step 4.2 (they are not freed during a copy-migration). If a member $O_i$ is a *local root* on $P_s$, its reference is also cleared from the *local root list* on $P_s$ in step 4.2. Each octant $O_i$ sent to $P_d$ is then received in steps 4.3, and its connectivity data is set in step 4.4. When an octant's relative or face neighbor resides on another processor, an interprocessor link is set by allocating storage for an *off processor octant* (either an *off processor parent, child,* or *face neighbor*), and the remote link information is inserted into the off processor data structure. In the special case that an octant's parent resides on another processor, the pointer to $O_i$ is inserted into the *local root list* on $P_d$ in step 4.4. Thus, the growth rate of octant migration during the final stage of migration is proportional to $max(N_{M_s})$. Figure 5 (b) depicts the tree after $O_2 \sqsubset P_0$, the children of $O_2 \sqsubset P_0$, $O_3 \sqsubset P_0$, and $O_4 \sqsubset P_0$ are sent to their final destination. Note that $O_2 \sqsubset P_1$, $O_3 \sqsubset P_3$, and $O_4 \sqsubset P_2$ become *local roots* on their respective processors in step 4 since they do not reside on the same processor as their parent, $O_0 \sqsubset P_0$.

*3.2. An Octant Refinement Operator*

The distributed octree can be asynchronously refined, in parallel, to any level via recursive calls to the refinement operator on the local octants of each processor. The refinement operator is divided into two main blocks (see Figure 6). In the first block, *parent-child links*, are set between an octant $O_i$ and its newly allocated children. In the second block, six new *face neighbors links* are set for each new child of $O_i$, and those *face neighbor links* that become outdated as a result of the octant subdivision are reset.

Setting a particular *face neighbor link* in the direction $d$ for a new child $O_c$ begins in step 2.1, by retrieving either the child's *equal face neighbor* or the *inherited face neighbor* if the *equal face neighbor* does not exist at the time $O_c$ is allocated. The existing connectivity of the tree is used to retrieve a face neighbor of $O_c$ in $O(1)$ time. Given the face neighbor $O_d$, a link is set from $O_c$ to $O_d$ in step 2.2. Octree refinement must also reset existing *face neighbor links* which become outdated as a result of the octant subdivision. *Face neighbor links* are reset in steps 2.3-2.5 of the octree refinement algorithm. An equal, *non-sibling* neighbor $O_d$ is reset to point back to its new *equal face neighbor* $O_c$ in steps 2.3-2.4. Thus, two way links between *equal face neighbors* are automatically maintained during octree refinement. Outdated *inherited face neighbor links* are also reset in step 2.5 which visits all the *descendents* of $O_d$ with *inherited face neighbor links* to the $parent(O_c)$, and resets their *–d face neighbors* to the new child $O_c$.

In the parallel environment, the implementation is slightly different when the existing connectivity between a new octant $O_c$ and its *equal face neighbor* $O_d$ is distributed across more than one processor. When step 2.1 cannot retrieve the pointer to a *non-sibling face neighbor* $O_d$, it determines the shortest *path*, or traversal direction, to $O_d$. The computation of the shortest path from a new octant $O_c$ to its remote equal face neighbor $O_d$ in the direction $d$ requires traversing up the octree from $O_c$ until an $ancestor(O_c)$ having an *equal face neighbor link* in the direction $d$ is found. The execution of the path then begins from the *ancestor's face neighbor*, down to the *equal face neighbor* of $O_c$. Steps 2.2 - 2.5 of the octant refinement pseudocode are, therefore, postponed for those octants which have remote connections to their equal face neighbors. Instead,

a *face neighbor* update message containing a computed *path* and the address of the new octant $O_c \sqsubset P_s$ is sent to the next octant $O_n \sqsubset P_a$ (on the affected processor $P_a$), in the path to $O_d \sqsubset P_d$.

Octree refinement allows an application to refine the octree asynchronously on each processor in $O(max(N_{R_s}))$ time, where $N_{R_s}$ is the number of octants allocated by the refinement algorithm on source processor $P_s$. Internally, octant refinement sets *face neighbor links* when possible, and sends *face neighbor* update messages when *face neighbor links* cannot be set. The processors are then synchronized to process the *face neighbor* update messages [26] after the application has finished octree building on each processor.

## 4. A Lateral Approach to Octree Neighbor-Finding

One of the primary motivations for using octrees to support mesh generation computations is that they provide an effective means to localize mesh data. The octree structure becomes a localization tool when each terminal octant stores links to its bounding mesh data. Both de Cougny [5-7] and Lohner [15, 16], for example, advocate the use of trees to localize mesh vertices during their advancing front mesh generation computations. The mesh generator is then able to quickly locate mesh neighborhoods near those regions of space being meshed by retrieving an appropriate set of neighboring terminal octants.

In Section 2, we defined two specific *face neighbors*: *equal face neighbors* and *inherited face neighbors*, which are explicitly stored and maintained by the octant refinement algorithm. In this section, we define a lateral approach to octree neighbor-finding which uses these stored face neighbor links to retrieve the terminal octant neighbors which are needed during mesh generation.

For the purposes of explaining the details of the lateral method, it is convenient to define the procedure, Retrieve_ Face_Neighbor $(O_i, d)$, which returns the stored *equal* or *inherited face neighbor* of $O_i$ in the direction $d$. We also define the procedure, Traverse_to_Terminals $(O_i, d)$,

which returns all the *descendents* of $O_i$ which lie in the direction $d$ with respect to the center of $O_i$. Referring to Figure 7, for example, Traverse_to_Terminals $(O_8, +x)$ returns octants $\{O_{10}, O_{12}, O_{14}, O_{16}\}$ and Traverse_to_Terminals $(O_8, (+x, +y))$ returns octants $\{O_{12}, O_{16}\}$. We further define a boolean operator, $\text{Adjacent}(O_i, d, L)$, which returns true if $\{O_i^2\}_d$ intersects $\{anscestor(O_i)^2\}_d$ (at the level $L$). Adjacent operates in $O(\Delta L)$ time where $\Delta L$ is the level difference between an octant $O_i$ and its *ancestor* at the level $L$, and does not perform any floating point or intersection calculations.

The algorithm, Retrieve_Terminal_Face_Neighbors, in Figure 8 (a) retrieves all the terminal face neighbors which intersect a particular face of $O_i$, or satisfy (4). Finding an octant $O_i$'s terminal face neighbors in the direction $d$ begins by retrieving the *equal* or *inherited face neighbor* $O_d$ from $O_i$'s storage. If $O_d$ is a *non-terminal, equal face neighbor* of $O_i$, the procedure, Traverse_to_Terminals, retrieves all of the terminal $descendents(O_d)$ which intersect $\{O_i\}_d$, in $O(4^{\Delta L + 1})$ time, where $\Delta L$ is the level difference between an octant and its smaller *terminal face neighbors*.

Such a lateral approach to neighbor-finding differs from a top-down method [1] which first computes a point which is known to be in the neighbor(s), and then traverses down the tree from the root octant, performing containment queries at each level in the hierarchy until the *terminal descendents* containing the point are found. It also differs from a bottom-up technique developed by Samet [21, 22] which first traverses up the tree from an octant of interest until an *ancestor* common to both the octant and its terminal neighbor(s) (the nearest common ancestor) is found. The procedure then traverses back down the tree to the desired *terminal face neighbors*. Clearly, both traversal based procedures are $O(\log_8 N)$, where $N$ is the number of octants in the octree.

We define the algorithm, Retrieve_Terminal_Edge_Neighbors in Figure 8 (b), which retrieves all the *terminal edge neighbors* of an octant which intersect a particular edge of $O_i$, or satisfy (5). The algorithm is divided into two main blocks. The first block retrieves $O_i$'s *terminal face neighbors* which satisfy (5). For example, octants $O_{20}$ and $O_4$, in Figure 7, are *terminal face*

*neighbors* which intersect octant edge $\{O^1_{32}\}_d$ where $d = (+z, +y)$. Finding the terminal face neighbors which satisfy (5) consists of retrieving both the $d_1$ and $d_2$ *equal* or *inherited face neighbors* of $O_i$ in steps 1.1 and 1.3 respectively. Given $O_{d_1}$ and $O_{d_2}$, the algorithm Traverse_to_Terminals retrieves their terminal *descendents* which intersect $\{O^1_i\}_d$ in $O(2^{\Delta L + 1})$ time.

The second block of Retrieve_Terminal_Edge_Neighbors retrieves the remaining terminal octants which satisfy (5) and only intersect $O_i$ along its edge $\{O^1_i\}_d$. For example, octant $O_{10}$, in Figure 7, is returned by the second block during a call to Retrieve_Terminal_Edge_Neighbors $(O_{32}, (+z, +y))$. Given an *equal* or *inherited face neighbor* $O_{d_1}$ which intersects edge $\{O^1_i\}_d$, the conditional in step 2.1 determines whether or not, a second lateral hop in the $d_2$ direction, from $O_{d_1}$,will yield an octant which only intersects $O_i$ along its edge in the $(d_1, d_2)$ direction. If $O_{d_1}$, is an *equal face neighbor* of $O_i$, a second lateral hop from $O_{d_1}$ to its *equal* or *inherited face neighbor* in the $d_2$ direction will certainly yield octant $O_{ge}$ which only intersects $O_i$ along its $(d_1, d_2)$ edge. However, if $O_{d_1}$ is an *inherited face neighbor* of $O_i$, the algorithm must determine whether or not the octant face $\{O^1_i\}_{d_2}$ intersects the $d_2$ face of its ancestor at the $level(O_{d_1})$ using the procedure Adjacent. If this intersection criteria is satisfied, then a second lateral hop from $O_{d_1}$ to its equal or inherited face neighbor in the $d_2$ direction yields octant $O_{ge}$ which is only adjacent to $O_i$ along its $(d_1, d_2)$ edge. When the conditional in step 2.1 fails, the *inherited face neighbor* $O_{d_1}$ extends over the octant's $(d_1, d_2)$ edge, and is therefore returned as the *terminal edge neighbor* in step 2.6. If octant $O_{ge}$ in step 2.2 is larger than $O_i$, steps 2.3 and 2.4 search for a $descendent(O_{ge})$ which is an *equal edge neighbor* of $O_i$. Given an equal edge neighbor $O_{ge}$, step 2.5 traverses to the terminal $descendent(O_{ge})$ which intersect the $(d_1, d_2)$ edge of $O_i$ in $O(2^{\Delta L + 1})$ time.

We define an algorithm Retrieve_Terminal_Vertex_Neighbors in Figure 9, which retrieves all the terminal neighbors which intersect a particular vertex $\{O^0_i\}_d$ of an octant $O_i$, or satisfy (6). The algorithm is divided into three main blocks. The first and second blocks retrieve all the *terminal face* and *edge neighbors* which intersect $\{O^0_i\}_d$ in $O(\Delta L)$ time. The third block

retrieves the final terminal octant which only intersects $O_i'$ at its vertex $\{O_i^0\}_d$. Given one of the three *equal* or *inherited edge neighbors* of $O_i$ which intersect the octant's vertex $\{O_i^0\}_d$, the third block retrieves the terminal octant which only intersects $O_i$ on its vertex. For example, octant $O_{14}$ in Figure 7, is returned by the third block during a call to Retrieve_Terminal_Vertex_ Neighbors $(O_{11}, (+z, -y, +x))$.

The conditional in step 3.1 determines if a third lateral hop from one of the three *equal* or *inherited edge neighbors,* $O_{e_1}$, to its stored *face neighbor* in the third direction $d_3$, yields an octant $O_{gv}$, which only intersects the $(d_1, d_2, d_3)$ vertex of octant $O_i$. This conditional is satisfied if $O_{e_1}$ is the same size as $O_i$, or $O_i$ intersects the $d_3$ face of its *ancestor* at the $Level(O_{e_1})$. Steps 3.3-3.5 then traverse down the tree in $O(\Delta L)$ time from $O_{gv}$ until the terminal $descendent(O_{gv})$ which intersects $\{O_i^0\}_d$ is found. When the conditional in step 3.1 fails, the *inherited edge neighbor* $O_{e_1}$ extends over $\{O_i^0\}_d$, and is therefore returned as the *terminal vertex neighbor* of $O_i$ in step 3.6.

Finally, we define the algorithm Retrieve_ All_ Neighbors, in Figure 10, which retrieves all the terminal face, edge, and vertex neighbors of an octant $O_i$. For efficiency reasons, the directions in step 1 are defined to avoid repeated visitations of octants in the tree. For example, the terminal edge and vertex neighbors surrounding the $+z$ face are retrieved when the terminal $+z$ face neighbors are retrieved. The space filling curve [22] in Figure 10 illustrates the way in which octant neighbors are visited by Retrieve_ All_ Neighbors when a terminal octant is surrounded by a set of terminal neighbors at the same level in the octree hierarchy. Retrieve_ All_ Neighbors is needed during the parallel template meshing, which is described in section 5. The growth rate of the lateral neighbor-finding algorithms during mesh generation is $O(1)$ since $\Delta L \le 2$.

*4.1. Overlapping Neighborhoods and Ghost Octants*

During mesh generation, each terminal octant must have access to its *terminal face, edge,* and *vertex neighbors*. In a distributed-memory environment, the octree data required to support

the lateral neighbor-finding techniques may not be available on the processors. One approach is to synchronize the processors during the computations which require neighborhood information and allow them to exchange neighborhood data. However, this approach would significantly degrade the performance of an application, such as mesh generation, which frequently performs neighborhood queries. Another approach is to use ghost octants or copied octants, and create overlapping processor neighborhoods on partition boundaries [8, 10, 14, 24]. The advantage of ghost octants is that once they are copied, they may be visited multiple times rather than having repeated communication requests for them. During mesh generation, only the layer of terminal octants and their *parents* on the partition boundary need to be copied to support the lateral neighbor-finding algorithms, Retrieve_ Terminal_ Face_ Neighbors and Retrieve_ Terminal_ Edge_ Neighbors. Although this approach consumes additional storage, it allows the processors to perform the mesh generation calculations asynchronously since each processor has the required set of local and ghost octants to support face and edge neighborhood queries for every local octant.

## 5. Application to Parallel Mesh Generation

The distributed octree and scalable algorithms to migrate octants, refine octants, and retrieve octree neighbors have been presented in Sections 2, 3 and 4. Here we present an overview of a parallel mesh generator [5 - 7], and the way in which it is supported by the distributed octree structure. Beginning from a geometric modeler's description of a domain to be meshed, the parallel mesh generator proceeds to (i) partition the geometric model based on estimates of meshing workload, (ii) construct a valid triangulation of the surface of the domain, and (iii) mesh the interior of the domain with tetrahedra. Parallel octree building occurs during steps (i) and (ii) and is described in Section 5.1. A combination of octant template and advancing front procedures then mesh the interior of the domain in step (iii).

One of the difficulties in starting from a geometric modeler is that the parallel mesh generator lacks direct information to predict the number of elements within a given processors's vol-

ume that are required to (i) obtain a valid finite element discretization, and (ii) adhere to specified sizes of mesh edges, faces, and regions. Thus, workload estimates in the initial stages of parallel mesh generation are based on less complete information in comparison to parallel finite element solvers, which have complete knowledge of the mesh [8, 9, 17]. It, therefore, becomes convenient to use the octree to estimate the number of elements in a given octant, and assign octants to processors based on those estimates.

### *5.1. Parallel Octree Building during Parallel Mesh Generation*

The mesh generator obtains meshing workload estimates by sampling points on the surface of the geometric model, and computes the sizes $s$ of mesh edges which satisfy the mesh control requirements at the point of interest [7]. The *level* of the tree required to obtain terminal octants of approximate length $s$ is then given by:

$$level = \log_2(S/s), \tag{14}$$

where $S$ is the dimension of the root octant. Approximate estimates of the meshing workload are, therefore, indirectly related to the cost of constructing subtrees at the specified levels. Where the tree depth is high, we expect a large number of small elements, and where the tree depth is low, we expect a small number of large elements. A "coarse octree" is refined until its meshing load can be balanced among the processors, and the octant members are subsequently partitioned using a divide and conquer strategy [7]. Octant migration procedures are called upon to distribute the octants in this tree to their assigned processors. Terminal octants which are migrated to a processor other than their parent's become *local roots* and are stored in their processor's *local root list* by the octant migration operator (section 3.1). Figure 11 shows an example of a distributed octree which was used to distribute a pre-triangulated boundary representation of a geometric model across four processors.

Once the "coarse octree" is partitioned, the surface mesh generator proceeds to refine the

*local roots* in parallel to the proper levels, Eq. (14), needed during the surface and volumetric meshing procedures. The octree is refined again in order to enforce the 2:1 criteria such that ($\Delta L \leq 1$) between a terminal octant and all of its *terminal face and edge neighbors*. Overlapping octree neighborhoods and the neighbor-finding algorithms described in section 4 support the procedure which enforces the 2:1 criteria.

Once an octree has been refined to the proper levels, the parallel mesh generator proceeds to construct a valid surface discretization of the geometric model [7]. Initially, model faces are split so they can be meshed in parallel. The resulting mesh faces are inserted into the octree in order to localize the mesh. Proper localization requires that each terminal octant store links to its bounding mesh data. The volume of the geometric modeling domain is then meshed in parallel by a combination of octree template meshing and advancing front procedures [6, 11, 16].

### *5.2. Octree Template Meshing*

Template meshing consists of meshing each interior, terminal octant which lies a sufficient distance within the geometric model. In order to construct template meshes in parallel, each processor traverses down the subtrees rooted at its *local root list.* A given interior terminal octant $O_i$ is meshed by (i) retrieving all of its terminal neighbors $O_N$ using the lateral neighbor-finding algorithm Retrieve_ All_ Neighbors (section 4), (ii) retrieving the appropriate template pattern octant $O_p$, based on the terminal neighbor set $O_N$, and (iii) meshing octant $O_i$. Octree neighborhood information is needed in order to generate a conforming tetrahedral mesh across octant boundaries. Overlapping octree neighborhoods are used in order to guarantee conformity on octree partition boundaries without the need to synchronize during template meshing.

Mesh entities on the partition boundary are augmented with interprocessor links which point to the location of the corresponding entities on neighboring processors [6, 19, 20] (see Figure 12). It is convenient to use the octree to link mesh entities on the partition boundary since (i) the template mesh data "lines up" with the octant topological entities and (ii) a mesh entity on a

partition boundary must be classified on an octant which is on a partition boundary. Octants on partition boundaries are copy-migrating once, along with their corresponding template mesh data (only the template data on the partition boundary) to the processors which contain duplicate instances or uses of their template mesh data in order to avoid repeated communications calls for shared off processor mesh data. A given mesh entitiy on a partition boundary, classified inside a given interior terminal octant $O_i$ on the partition boundary, is then linked to its off processor instances by retrieving the appropriate mesh entities inside the copied terminal octant neighbors of $O_i$. Figure 13 illustrates some example 3-D template meshes which were generated by the template meshing procedure.

### *5.3. Advancing Front Procedure to Complete the Volumetric Meshing*

Once templates have been applied to interior terminal octants, the region between the surface mesh and the template mesh is filled by advancing-front procedures [6, 11, 16]. A mesh face is removed from the front by connecting it to a neighboring mesh vertex. A list of candidate vertices to connect to a given mesh face is obtained from an appropriately sized terminal octree neighborhood [6] surrounding the mesh face. When a required octree neighborhood extends beyond the bounds of a processor's partition, the given face removal is postponed until the octree has been repartitioned such that the required neighborhood is all on one processor. When the required octree neighborhood for a given mesh face is all on processor, candidate mesh edges and faces resulting from a connection to a candidate vertex are inspected for possible intersections with existing mesh entities in the octree neighborhood. If an intersection is detected, the face removal procedure considers the next candidate vertex. If none of the candidate vertices form a valid element, then a new mesh vertex is created and the mesh face is connected to the new mesh vertex. The mesh front is continually updated as new elements are created and terminates when there are no remaining mesh faces in the front. The reader is referred to [6, 7] for specific details regarding the advancing front procedures and the way in which the mesh is repartitioned.

## 6. Performance Results

Performance measurements were obtained on various components of the parallel octree presented in Sections 3, 4 and 5, on a 32 processor IBM SP2 (power 266 MH thin node). In order to determine the scalability of an algorithm, it is useful to observe the growth rate as problem size (in this case $sum(N_{M_s})$, $s = 1, \ldots, N_p$) grows with the number of processors ($N_p$), such that problem size on a particular processor remains constant. Under these conditions, an algorithm is said to scale if its growth is $O(\log N_p)$ [4] or less.

### *6.1. Octant Migration*

For the specific scalability experiment in Figure 14 (a), a distributed octree is constructed by partitioning the octree at the highest possible level (eg., an 8-processor run is partitioned at the first tree level), and refining such that each processor contains approximately 600,000 octants. Each processor then exchanges a slice of 10,000 terminal octants with their remote neighbors in order to exercise each stage of the octant migration. The CPU times reported in Figure 14 (a), are the times that it takes to migrate $10000 \times N_p$ octants out of $600000 \times N_p$ octants. Thus, we are testing in the range of values where $N_p \ll N_{M_s}$. Under these conditions, we observe that the growth with $N_p$ appears to approach a constant value as $N_p \to \infty$ which indicates that the algorithm scales with $N_p$.

In order to observe the predicted growth with $max(N_{M_s})$, $s = 1, \ldots, N_p$, we hold the tree size fixed at 4,800,000 octants on 8 processors, and increase the number of migrating octants from 2,500 octants/processor to 17,000 octants/processor. Figure 14 (b) shows that doubling $max(N_{M_s})$ doubles the execution time.

### *6.2. Octant Refinement*

In order to observe the growth of the octant refinement algorithm and the processing of

face neighbor update messages, we construct a distributed octree by partitioning the tree at the highest possible level (as in the migration experiment), and refine each of the *local roots* in parallel one level at a time, until the total number of octants $N_{R_s}$ allocated on a processor $P_s$ is equivalent to a specified limit. Refining the octree in this manner produces a tree which is somewhat homogeneously refined across the domain, rather than heavily refined in a given part of the universe.

For the specific scalability experiment shown in Figure 15 (a), each processor allocated a total of 500,000 octants. The CPU times reported in Figure 15 (a) include the time it takes to allocate $500,000 \times N_p$ octants and to process the resulting face neighbor update messages. One of the factors influencing the growth with $N_p$, is the fact that none of the octree partitions are totally surrounded by 6 processors. This results in a steady increase in the time to refine the distributed octree due to an increasing number of face neighbor update messages being generated as $N_p$ is increased.

In order to observe the predicted growth with $max(N_{R_s})$, $s = 1, \ldots, N_p$, we hold $N_p = 8$, and increase the number of octants which are allocated on each of the 8 processors from 50,000 octants/processor to slightly under 1 million octants/processor. Based on the results in Figure 15 (b), the growth appears to be linear in $max(N_{R_s})$. The slight drop in performance past 750,000 octants/processor is likely to be the result of the octree consumption approaching the available memory resources (RAM).

*6.3. Neighbor-Finding*

We compare the growth rates of traversal-based neighbor-finding techniques which are used in [1, 2], with the lateral neighbor-finding techniques described in section 4 by partitioning the octree at the top levels and refining the *local roots* in parallel until each processor has approximately 30,000 octants. Overlapping neighborhoods are then exchanged between the processor set $P$ to support all neighborhood queries on the local octants of each processor. The subtrees are

then traversed in parallel, and neighbor queries in every direction are performed on each local terminal octant encountered in the traversal. Neighborhood queries of this kind are performed frequently during the parallel meshing calculations, therefore, this test is a good indication of the cost of neighborhood queries during mesh generation.

Figure 16 (a) contains the total CPU times required to perform face neighbor queries in each of the 6 directions $(+x, -x, +y, -y, +z, -z)$ and for each terminal octant in the distributed octree. The dashed, upper curve in Figure 16 (a) is the total CPU time which was needed by the traversal based technique that performs containment queries from the root down to find neighbors. The solid, upper curve in Figure 16 (a) is a scaled $N\log_8 N$ curve. Together these upper curves demonstrate that the traversal based technique grows as $\log_8 N$ per neighbor query which corresponds well to the predicted behavior. The lower curve in Figure 16 (a) is the total CPU time required by the lateral technique introduced in section 4. This result demonstrates that the lateral method grows as $O(1)$ per neighbor query as the theory predicts. Similar results were also obtained for the terminal *edge* and *vertex neighbor* retrieval algorithms. The results for the edge neighbor queries are shown in Figure 16 (b).

*6.4. Template Meshing*

Timing results for the meshing part of the template algorithm shown in Figure 17 (which does not include setting the interprocessor mesh links), demonstrate that a single node of the IBM SP2 is capable of generating template meshes, such as those shown in Figure 13, at a rate of approximately 3660 regions/second. The sharp increase in slope past $N = 100,000$ mesh regions is likely to be the result of the mesh consumption approaching the available memory resources (RAM).

To determine how well the template meshing algorithm scales, we construct a distributed octree, as in the scalability experiment for octree refinement, by partitioning the octree and refining until each processor allocates a specific number of octants. For the specific scalability experi-

ment shown in Figure 18 (a), approximately $6500 \times N_p$ terminal octants (out of a total of $7500 \times N_p$) are meshed, resulting in approximately 43,000 mesh regions/ processor. The CPU times reported in Figure 18 include both the time it takes to mesh the octants and to set interprocessor mesh links on the partition boundary. As seen, there is a slight growth for small $P$ which appears to approach a constant value for large $P$. The growth with $N_p$ is indicative of an increase in the number of mesh faces on the partition boundaries which results in longer times to set interprocessor mesh links after the template meshes are generated asynchronously in parallel. This growth in the size of the partition boundary with $N_p$ is verified by the results in Figure 18 (b) which is a plot of the number of surface mesh faces on the partition boundary divided by the total number of mesh regions. As shown, this surface to volume ratio initially increases with $N_p$, but then levels beyond 16 processors.

## 7. Conclusions

A distributed octree structure was developed to support efforts in parallel mesh generation [5 - 7]. The octree data structure augments a basic hierarchical octree structure to include interprocessor links to off processor octants, and also includes a *local root list* which gives each processor access to all its local octree data. The distributed octree also includes lateral links between octants of the same level which share common faces in the octree topology. These lateral links, known as *face neighbor links*, support neighbor-finding algorithms which operate in $O(1)$ time on a tree with controlled level differences.

Two basic algorithms are described to support the construction of a distributed octree with *face neighbor links*. An octant migration algorithm maintains the octree connectivity (*parents*, *children*, *face neighbor links*, and *local root lists*) when the octants are distributed or redistributed by an application. Results on the octant migration show that it scales well and has a growth rate proportional to $max(N_{M_s})$, $s = 1, \ldots, N_p$ mail, where $N_p$ is the number of processors in the set $P$, and $N_{M_s}$ is the number of octants which are migrating on processor $P_s$. An octant refinement

algorithm and its companion routine to process face neighbor update messages was written to asynchronously refine the *local roots* of a distributed octree in parallel. Results demonstrate that the growth rate of octree refinement is proportional to $max(N_{R_s})$, $s = 1, \ldots, N_p$, where $N_{R_s}$ is the number of octants allocated by the octant refinement algorithm on processor $P_s$.

Finally, we show how the distributed octree structure supports mesh generation in the distributed memory environment. In particular, we show how these scalable tools are used during parallel template meshing, and provide some example template meshes. Results for the template meshing and its companion routine to set interprocessor mesh links between duplicate mesh entities or *uses* on the partition boundary show that it scales well and is capable of generating elements at a rate of 3660 regions/second on a single node of an SP2 (power 266 MH thin node).

### Acknowledgements

The work of the first author was supported by the Office of Naval Research through grant number N00014-95-1-0892. Partial support for this work was provided by the DOE ASCI program through grant number B341495.

### References

1. J. Arvo, D. Kirk, A survey of ray tracing acceleration techniques, *in* "An Introduction to Ray Tracing," (A. Glassner, ed.), pp. 201-262, Academic Press, 1989.

2. P. L. Baehmann, S. L. Wittchen, M. S. Shephard, K. R. Grice, M. A. Yerry, Robust, geometrically based, automatic two-dimensional mesh generation, *International Journal for Numerical Methods in Engineering* **24**, (1987), 1043-1078.

3. M. W. Beall, M. S. Shephard, A general topology-based mesh data structure, *International Journal For Numerical Methods In Engineering* **40**, (1997), 1573-1596.

4. D. P. Bertsekas, J. N. Tsitsiklis, "Parallel and Distributed Computation," Prentice Hall, New Jersey, 1989.

5. H. L. de Cougny, M. S. Shephard, C. Ozturan, Parallel three-dimensional mesh generation, *Computing Systems in Engineering* **5**, (1994), 311-323.

6. H. L. de Cougny, M. S. Shephard, C. Ozturan, Parallel three-dimensional mesh generation on

distributed memory MIMD computers, *Engineering with Computers* **12**, 2 (1996), 94-106.

7. H. L. de Cougny, "Parallel Unstructured Distributed Three Dimensional Mesh Generation," Ph. D. thesis, School of Engineering, Rensselaer Polytechnic Institute, 1998.

8. K. D. Devine, J. E. Flaherty, R. M. Loy, S. R. Wheat, Parallel partitioning strategies for the adaptive solution of conservation laws, *in* "Proceedings of the IMA Workshop on Modeling, Mesh Generation, and Adaptive Numerical Methods for Partial Differential Equations," (I. Babuska, J.E. Flaherty, W.D. Henshaw, J.E. Hopcroft, J.E. Oliger, and T. Tezduyar, Eds.), pp. 215-242, Springer, Berlin, 1995.

9. K. D. Devine and J. E. Flaherty, Parallel adaptive *hp*-refinement techniques for conservation laws, *Appl. Numer. Math.* **20** (1996), 367-386.

10. K. D. Devine, "An Adaptive HP-Finite Element Method with Dynamic Load Balancing for the Solution of Hyperbolic Conservation Laws on Massively Parallel Computers," Ph. D. thesis, School of Science, Rensselaer Polytechnic Institute, 1994.

11 P. L. George, E. Seveno, The advancing-front mesh generation method revisted, *International Journal For Numerical Methods In Engineering* **37**, (1994), 3605-3619.

12. J. JaJa, "An Introduction to Parallel Algorithms", Addison-Wesley, New York, 1992.

13 A. Kela, Hierarchical octree approximations for boundary representation-based geometric models, *Computer Aided Design* **21**, (1989), 355-362.

14 C. Lee, C., M. Hamdi, Parallel image processing applications on a network of workstations, *Parallel Computing* **21**, (1995), 137-160.

15. R. Lohner, Some useful data structures for the generation of unstructured grids, *Communications in Applied Numerical Methods* **4**, (1988), 123-135.

16. R. Lohner, P. Parikh, Generation of three-dimensional unstructured grids by the advancing front method, *International Journal of Numerical Methods in Fluids* **8**, (1988), 1135-1149.

17. R. M. Loy, "Adaptive Local Refinement with Octree Load-Balancing for the Parallel Solution of Three-Dimensional Conservation Laws," Ph. D. thesis, School of Science, Rensselaer Polytechnic Institute, 1998.

18. MPI: A Message Passing Interface Standard, University of Tennessee, Knoxville, Tennessee, 1994.

19 C. Ozturan, H. L. de Cougny, M. S. Shephard, J. E. Flaherty, Parallel adaptive mesh refinement and redistribution on distributed memory machines, *Computer Methods in Applied Mechanics and Engineering* **119**, (1994), 123-137.

20. C. Ozturan, "Distributed Environment and Load Balancing for Adaptive Unstructured Meshes," Ph. D. thesis, School of Science, Rensselaer Polytechnic Institute, 1995.

21. H. Samet, Neighbor finding techniques for images represented by quadtrees, *Computer Graphics and Image Processing* **18**, (1982), 37-57.

22. H. Samet, "Applications of Spatial Data Structures, Computer Graphics, Image Processing, and GIS", Addison-Wesley, New York, 1989.

23 W. J. Schroeder, M. S. Shephard, A combined octree/delaunay method for fully automatric 3-d mesh generation, *International Journal for Numerical Methods in Engineering* **29**, (1990), 37-55.

24 C. N. Sekharan, V. Goel, R. Sridhar, Load balancing methods for ray tracing and binary tree computing using PVM, *Parallel Computing* **21**, (1995), 1963-1978.

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

26. M. L. Simone, "A Distributed Octree Structure and Algorithms for Parallel Mesh Generation," Ph. D. thesis, School of Engineering, Rensselaer Polytechnic Institute, 1998.

27 M. L. Simone, H. L. de Cougny, M. S. Shephard, Tools and techniques for parallel grid generation, *in*: "Fifth International Conference on Numerical Grid Generation in Computational Field Simulations," pp. 1165-1174, Mississippi, 1996.

28 M. A. Yerry, M. S. Shephard, A modified quadtree approach to finite element mesh generation, *IEEE Computer Graphics and Applications* **3**, (1983), 39-46 .

29 M. A. Yerry, M. S. Shephard, Automatic three-dimensional mesh generation by the modified octree technique, *International Journal For Numerical Methods In Engineering* **20**, (1984), 1965-1990.

**Author Biographies**

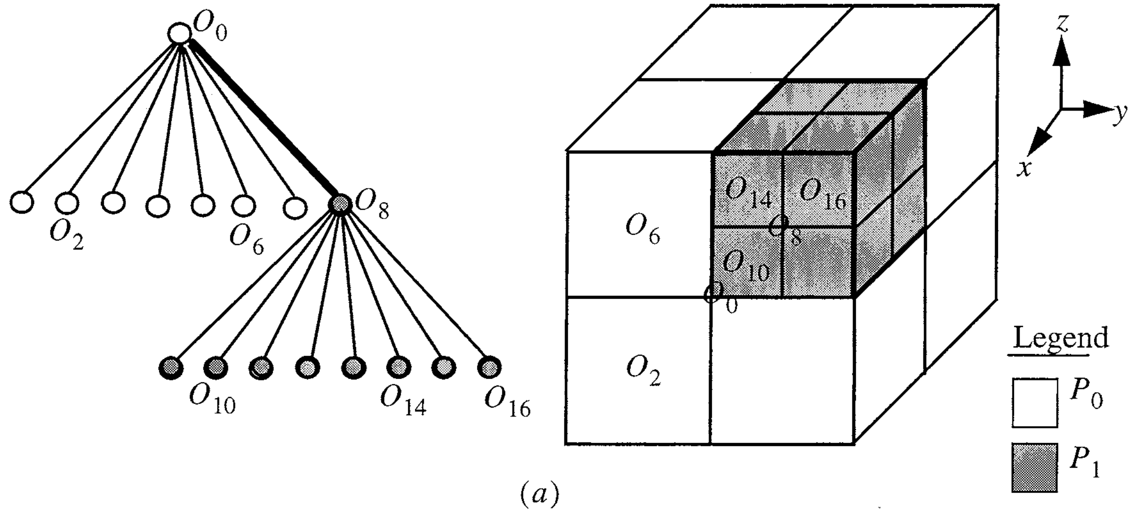
MICHELLE L. SIMONE is an associate research engineer with United Technologies Research Center. Dr. Simone received her B.S. in mechanical engineering from Cornell University (1990), M.S. (1993) and Ph.D. (1998) degrees in mechanical engineering from Rensselaer Polytechnic Institute. Her research interests include finite element modeling and software development in the area of numerical modeling.

RAYMOND M. LOY is a Postdoctoral Research Associate with the Scientific Computation Research Center of Rensselaer Polytechnic Institute. He received his B.S. in Computer Science from the Columbia University School of Engineering and Applied Science (1987); M.S.

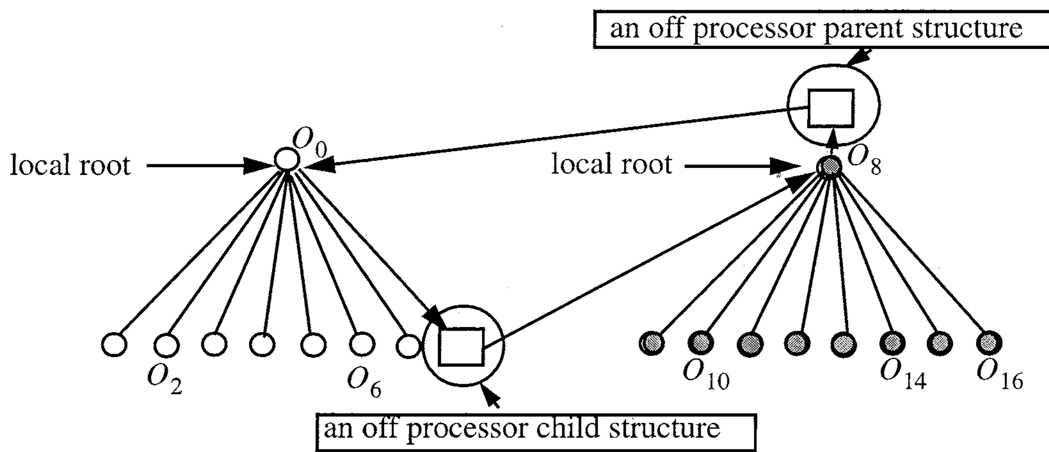
(1989) and Ph.D. (1998) degrees in Computer Science from Rensselaer Polytechnic Institute. His research interests include parallel load balancing and software support for parallel adaptive computation.

MARK. S. SHEPHARD is the Samuel A. and Elisabeth C. Johnson, Jr. Professor of Engineering at Rensselaer Polytechnic Institute. At Rensselaer he holds joint appointments in the departments of Civil Engineering; Mechanical Engineering, Aeronautical Engineering & Mechanics; and Computer Science. He is the director of RPI's Scientific Computation Research Center. Dr. Shephard has worked for a number of years on the development of automated adaptive computational technologies for solving partial differential equations

JOSEPH E. FLAHERTY is the Amos Eaton Professor of Computer Science at the Rensselaer Polytechnic Institute where he holds a joint appointment with the Departments of Computer Science and Mathematical Sciences. He received his Ph.D. (1969) degree from the Polytechnic Institute of Brooklyn. He conducts research in computational science with an emphasis on adaptive and parallel techniques. He is editor of Applied Numerical Mathematics, SIAM's Modeling and Simulation in Engineering, and the Problems and Algorithms Section of the SIAM Review. He is also on the Editorial Board of Computational Mechanics Advances and the SIAM Journal on Scientific Computing. He is a member of the American Ceramics Society, the Association for Computing Machinery, the IEEE Computer Society, the Executive Council of the Institute for Mathematics and Computers in Simulation, the Society for Industrial and Applied Mathematics, and the Executive Council of the U.S. Association for Computational Mechanics.




(*a*)




(*b*)

Figure 1. (a) A hierarchical octree structure and its associated octree domain (shading is used to indicate processor ownership of the octants). (b) An illustration of the actual data representation of the octree structure in (a). Local root lists are stored and maintained on each processor.

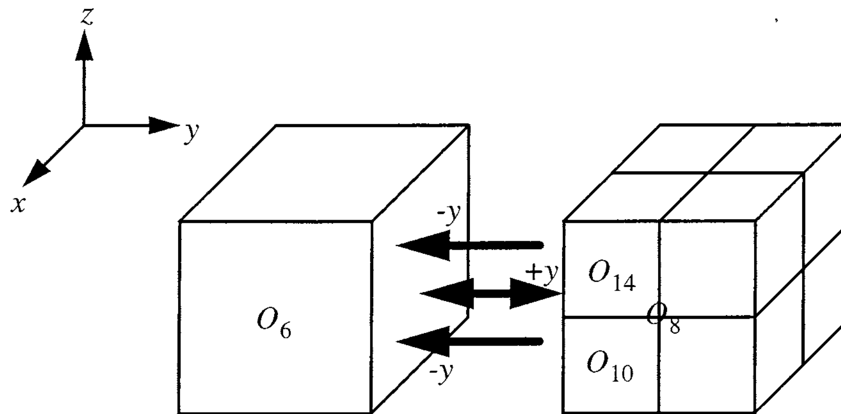



Figure 2. A portion of the octree of Figure 1 showing sample face neighbor links. Note that two-way links exist between equal face neighbors such as $O_6$ and $O_8$, while one-way links exist from octants such as $O_{10}$ and $O_{14}$ to their inherited face neighbor octant $O_6$.

**Procedure** Octant_Migration
**Input**: $O_M$: a set of octants to be migrated
$P_D$: a set of corresponding destination processors

**FOR ALL** members of processors set $P$ **do**

1. Allocate space for $O_{M_s}$ on $P_d$.
   1.1 **for each** $O_i \in O_{M_s}$ on $P_s$ **do**
   1.2 Send allocation request to $P_d$
   **end for**
   1.3 Compute number of incoming octants on $P_d$
   1.4 Process allocation requests on $P_d$; Send new addresses back to $P_s$
   1.5 Receive new addresses of migrating octants on $P_s$

2. Update the data attached to $O_{M_s}$.

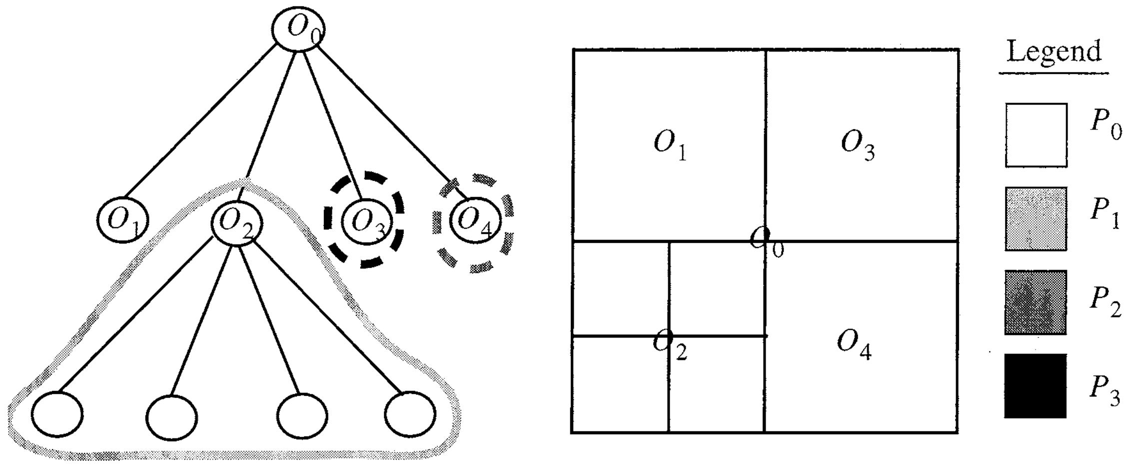
3. Update the relatives and neighbors of $O_{M_s}$ on the affected processor $P_a$
   3.1 **for each** $O_i \in O_{M_s}$ on $P_s$ **do**
   3.2 Update the $Parent$, *children*, and *face neighbors* connected to $O_i$
   **end for**

   3.3 **while** (messages_pending) on $P_a$ **do**
   3.4 Receive updates
   3.5 Set links to new addresses of migrating octants in affected relatives and neighbors
   3.6 Forward face neighbor update messages to affected children with inherited face neighbor links to a migrating octant
   **end while**

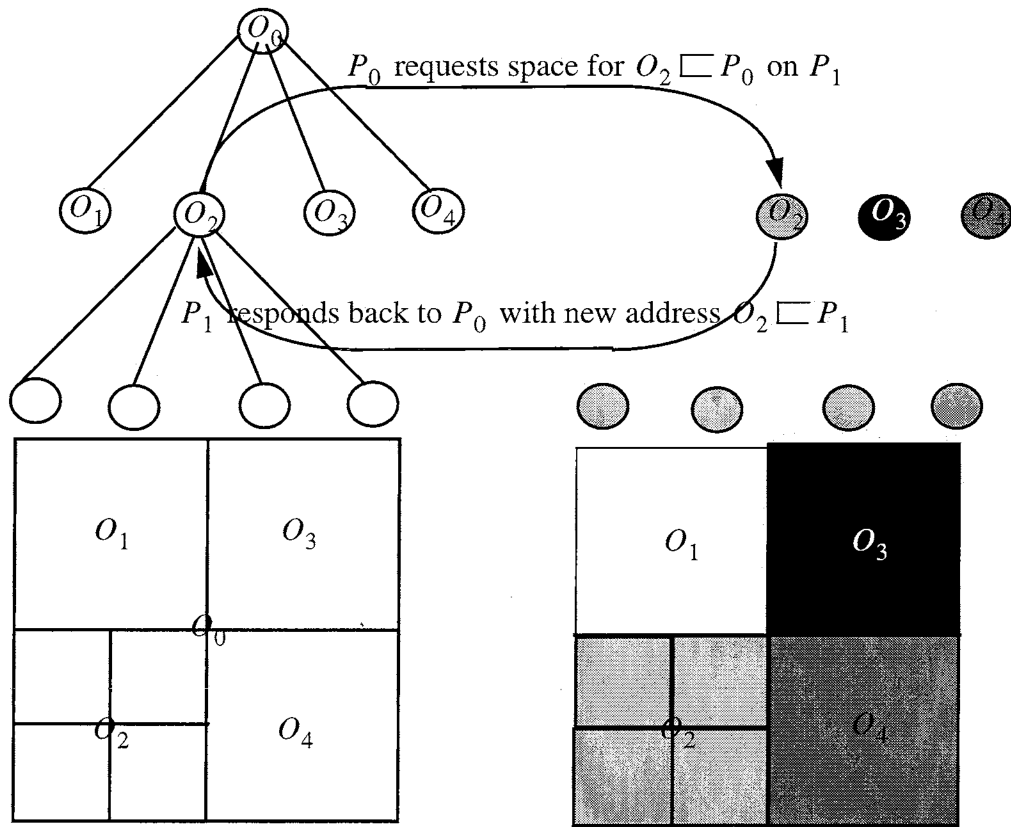
4. Final Migration
   4.1 Send the updated link data contained in $O_{M_s}$ to $P_D$.
   4.2 Free $O_{M_s}$ on $P_s$; update the local root list on $P_s$.
   4.3 Receive the migrating octants on $P_d$
   4.4 Set connectivity data for each octant; Update the local root list on $P_d$.

**END FOR**

Figure 3. Octant Migration Algorithm




(a)




(b)

Figure 4. (a) A quadtree on $P_0$ with quadrants targetted for migration to $P_1$, $P_2$, and $P_3$. Migrating quadrants are circled according to the shading of their destination processor. (b) An example of an allocation request which is processed by the set $\{P_0, P_1\}$ during the first stage of octant migration.

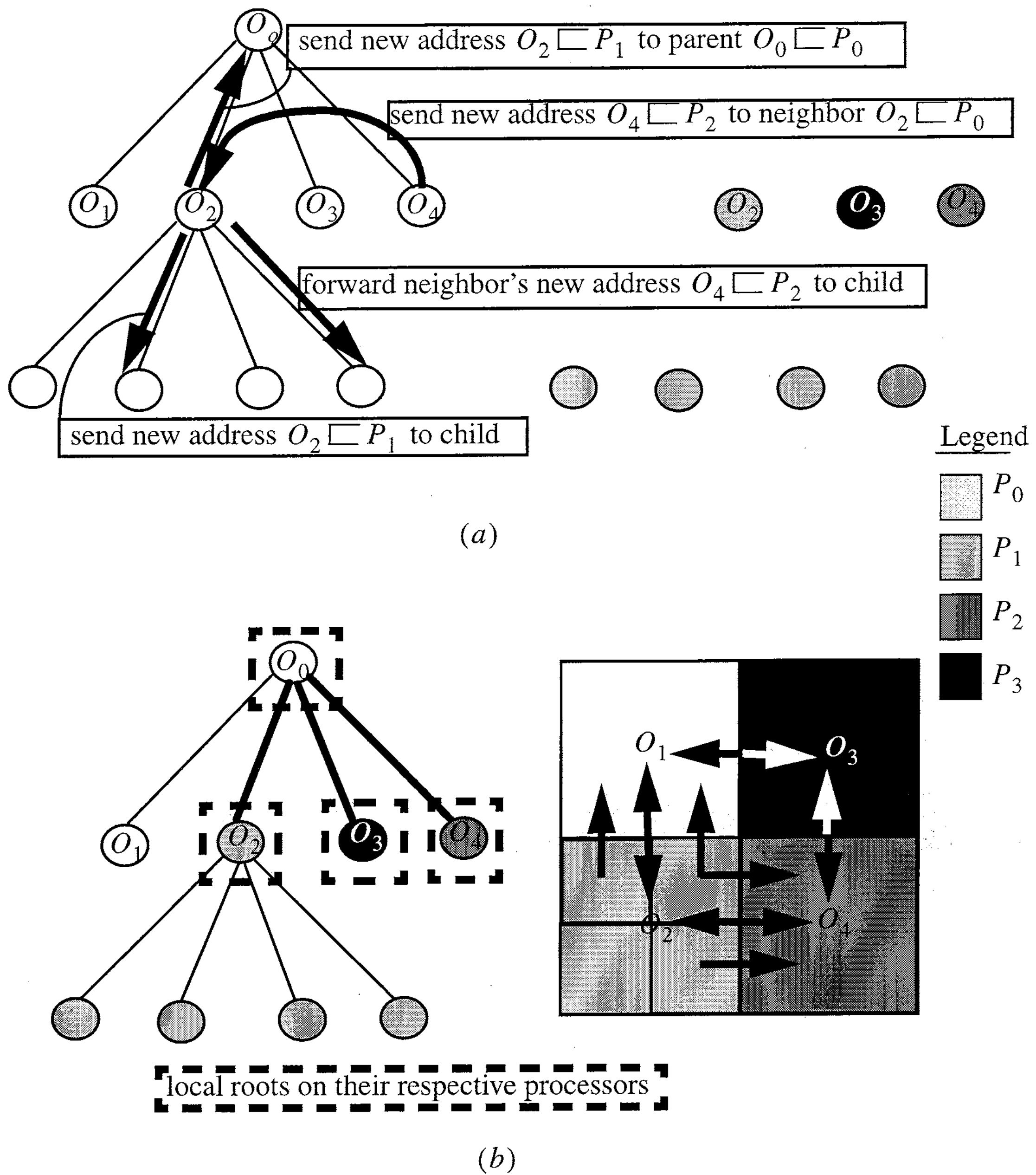



Figure 5. (a) Some link updates which are processed for $O_2 \sqsubset P_0$ during the third step of octant migration of Figure 3. (b) The quadtree after the final step in octant migration. All interprocessor parent-child and interprocessor neighbor links are shown in bold. Note that $O_{M_s}$ are freed on $P_s$, for $s = 1, \ldots, N_p$.

**Procedure** Octant_Refinement
**Input**: $O_i$: a terminal octant to be refined.
**Output**: $O_i$: a non-terminal octant .

1. Set parent-child connectivity between $O_i$ and its newly allocated children $O_c$
    **for each** child $O_c$ of $O_i$ **do**
        Allocate memory for a new *child* $O_c$.
        Set a *parent link* from $O_c$ *to* $O_i$.
        Set a *child link* from $O_i$ *to* $O_c$.
    **end for**

2. Set neighbor connectivity for each new child $O_c$ of $O_i$
    **for each** new child $O_c$ **do**
        **for each** direction $d$ (of octant face normals) **do**
2.1            Retrieve $O_c$*'s equal* **or** *inherited face neighbor* in direction $d$, $O_d$
2.2            Set a link from $O_c$ to $O_d$
2.3            **if** $O_d$ is an *equal face neighbor* **and not** a *sibling*
2.4                Reset $-d$ *face neighbor link* of $O_d$ to new *child* $O_c$
2.5                Visit all $descendents(O_d)$ with *inherited face neighbor links* to $Parent(O_c)$ and reset their *-d face neighbor link* to new *child* $O_c$
            **end if**
        **end for**
    **end for**

Figure 6. Octant Refinement Algorithm.

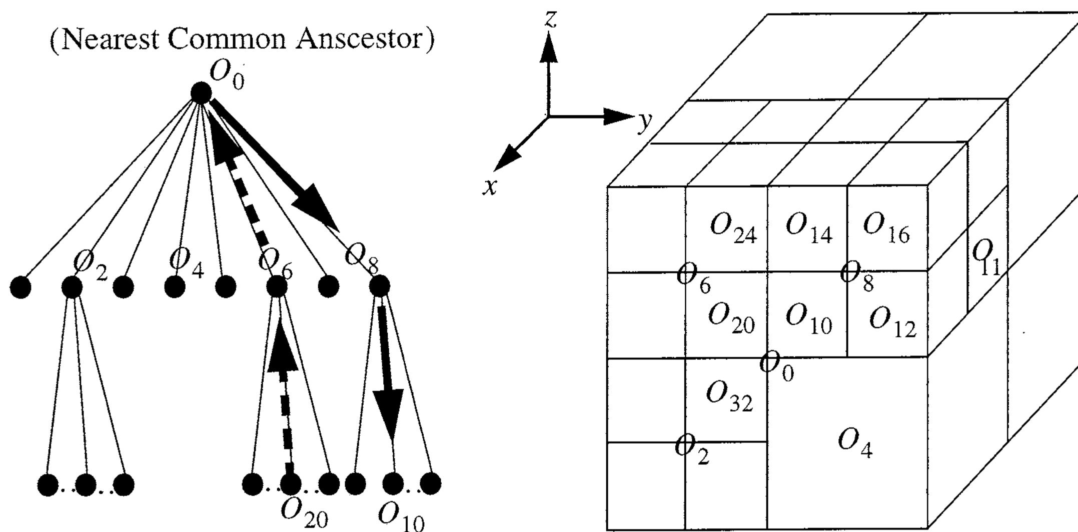



Figure 7. An illustration of the traversal required to find $O_{20}$'s terminal face neighbor in the $+y$ direction when only parent-child connectivity is stored.

**Procedure** Retrieve_Terminal_Face_Neighbors
**Input**: $O_i$: an octant
$d$: a specific face of octant $O_i$ $(+x, -x, +y, -y, +z, -z)$
**Output**: the terminal face neighbors of $O_i$ which intersect $\{O_i^2\}_d$.

$O_d \leftarrow$ Retrieve_Face_Neighbor($O_i$, $d$)
Traverse_to_Terminals($O_d$, $-d$)

(*a*)

**Procedure** Retrieve_Terminal_Edge_Neighbors
**Input**: $O_i$: an octant
$d = (d_1, d_2)$: a specific edge of octant $O_i$
**Output**: the terminal edge neighbors of $O_i$ which intersect $\{O_i^1\}_d$.

1. Get the terminal face neighbors of $O_i$ which intersect edge, $\{O_i^1\}_d$
   1.1 $O_{d_1} \leftarrow$ Retrieve_Face_Neighbor($O_i$, $d_1$)
   1.2 Traverse_to_Terminals($O_{d_1}$, $(-d_1, d_2)$)
   1.3 $O_{d_2} \leftarrow$ Retrieve_Face_Neighbor($O_i$, $d_2$)
   1.4 Traverse_to_Terminals($O_{d_2}$, $(d_1, -d_2)$)

2. Get the terminal neighbors which only intersect $O_i$ on its edge, $\{O_i^1\}_d$
   2.1 **if** $level(O_i) = level(O_{d_1})$ **or** Adjacent($O_i$, $d_2$, $Level(O_{d_1})$)
   2.2 $\quad O_{ge} \leftarrow$ Retrieve_Face_Neighbor($O_{d_1}$, $d_2$)
   2.3 $\quad$ **if** $level(O_i) > level(O_{ge})$
   2.4 $\quad\quad$ Determine $O_e$, the *descendent* of $O_{ge}$ which is equal in size to $O_i$ **or** the last terminal *descendent* of $O_{ge}$
   $\quad\quad O_{ge} \leftarrow O_e$
   2.5 $\quad$ Traverse_to_Terminals($O_{ge}$, $-d_1$, $-d_2$)
   **else**
   2.6 $\quad$ **Return** $O_{d_1}$

(*b*)

Figure 8. Algorithms to retrieve specific terminal (a) face and (b) edge neighbors of an octant $O_i$.

**Procedure** Retrieve_Terminal_Vertex_Neighbors
**Input**: $O_i$: an octant
$d = (d_1, d_2, d_3)$ :a specific vertex of octant $O_i$
**Output**: the terminal vertex neighbors of $O_i$ which intersect $\{O_i^0\}_d$.

1. Retrieve the terminal face neighbors of $O_i$ which intersect $\{O_i^0\}_d$
   1.1 $O_{d_1} \leftarrow$ Retrieve_Face_Neighbor$(O_i, d_1)$
   1.2 Traverse_to_Terminals$(O_{d_1}, (-d_1, d_2, d_3))$
   1.3 $O_{d_2} \leftarrow$ Retrieve_Face_Neighbor$(O_i, d_2)$
   1.4 Traverse_to_Terminals$(O_{d_2}, (d_1, -d_2, d_3))$
   1.5 $O_{d_3} \leftarrow$ Retrieve_Face_Neighbor$(O_i, d_3)$
   1.6 Traverse_to_Terminals$(O_{d_3}, (d_1, d_2, -d_3))$

2. Retrieve the terminal edge neighbors of $O_i$ which intersect $\{O_i^0\}_d$
   2.1 $O_{e_1} \leftarrow$ Retrieve_Equal_Edge_Neighbor$(O_i, O_{d_1}, d_2)$
   2.2 Traverse_to_Terminals$(O_{e_1}, (-d_1, -d_2, d_3))$
   2.3 $O_{e_2} \leftarrow$ Retrieve_Equal_Edge_Neighbor$(O_i, O_{d_1}, d_3)$
   2.4 Traverse_to_Terminal$(O_{e_2}, (-d_1, d_2, -d_3))$
   2.5 $O_{e_3} \leftarrow$ Retrieve_Equal_Edge_Neighbor$(O_i, O_{d_2}, d_3)$
   2.6 Traverse_to_Terminals$(O_{e_3}, (d_1, -d_2, -d_3))$

3. Retrieve the terminal neighbor which only intersects $O_i$ on its vertex, $\{O_i^0\}_d$
   3.1 **if** $level(O_i) = level(O_{e_1})$ **or** Adjacent$(O_i, d_3, level(O_{e_1}))$
   3.2 $\quad O_{gv} \leftarrow$ Retrieve_Face_Neighbor$(O_{e_1}, d_3)$
   3.3 $\quad$ **if** $level(O_i) > level(O_{gv})$
   3.4 $\quad\quad$ Determine $O_v$, the *descendent* of $O_{gv}$ which is equal in size to $O_i$ **or** the last terminal *descendent* of $O_{gv}$
   $\quad\quad O_{gv} \leftarrow O_v$
   3.5 $\quad$ Traverse_to_Terminals$(O_{gv}, (-d_1, -d_2, -d_3))$
   **else**
   3.6 $\quad$ **Return** $O_{e_1}$

Figure 9. An algorithm to retrieve specific terminal vertex neighbors of octant $O_i$

**Procedure** Retrieve_All_Neighbors
**Input**: $O_i$: an octant
**Output**: all the terminal neighbors of $O_i$.

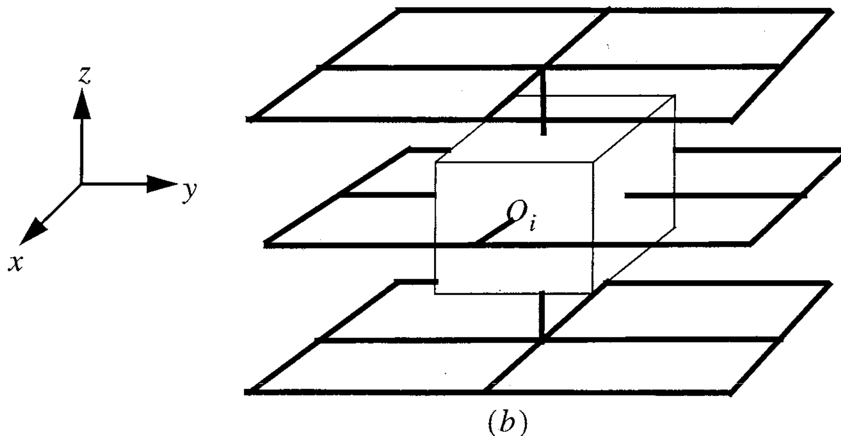
1. Define all the directions needed to get all the terminal neighbors of $O_i$

$$
\begin{aligned}
d = \{&+z, (+z, +y), (+z, +y, -x), (+z, -x), (+z, -x, -y), (+z, -y),\\
&\quad (+z, -y), (+z, -y, +x), (+z, +x), (+z, +x, +y)\\
&-z, (-z, +y), (-z, +y, -x), (-z, -x), (-z, -x, -y), (-z, -y),\\
&\quad (-z, -y), (-z, -y, +x), (-z, +x), (-z, +x, +y),\\
&+y,\\
&-x, (-x, -y), (-x, +y),\\
&+x, (+x, +y), (+x, -y),
\end{aligned}
$$

2. get all the terminal neighbors in each direction

**for each** direction $d$ **do**
  **if** $d$ is a face direction (has only 1 cartesian component)
    Retrieve_Terminal_Face_Neighbors($O_i$, $d$)
  **else if** $d$ is an edge direction (has 2 cartesian components)
    Retrieve_Terminal_Edge_Neighbors($O_i$, $d$)
  **else**
    Retrieve_Terminal_Vertex_Neighbors($O_i$, $d$)
  **end if**
**end for**

(*a*)




(*b*)

Figure 10. (a) An algorithm to retrieve all an octant's terminal neighbors and (b) the space filling curve mapped out by the algorithm when an octant's terminal neighbors are at the same level in the octree hierarchy.

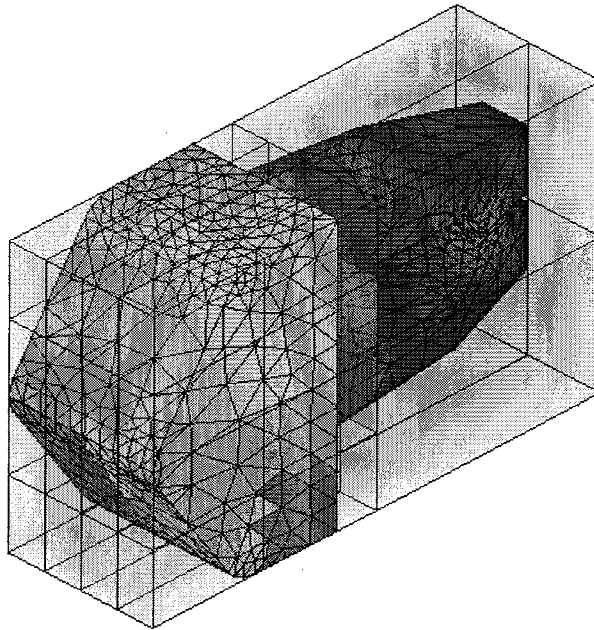


Figure 11. A pre-triangulated boundary representation of a geometric model which is partitioned across 4 processors by the octree. The shading indicates the classification of the mesh and octant data with respect to the processor set $P$.

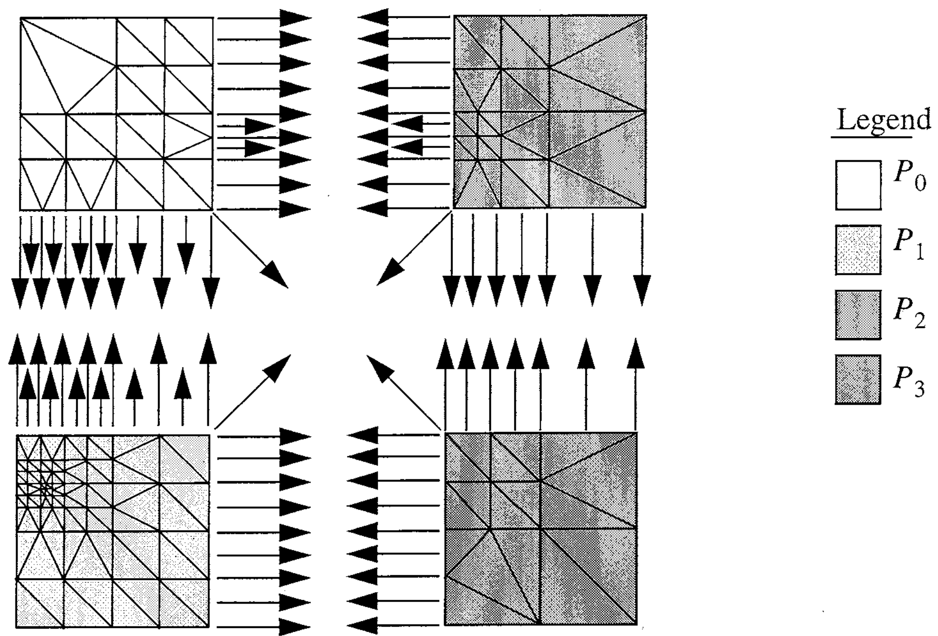



Figure 12. A portion of a distributed tree with its template mesh data. Interprocessor mesh links are indicated by arrows (2D setting).

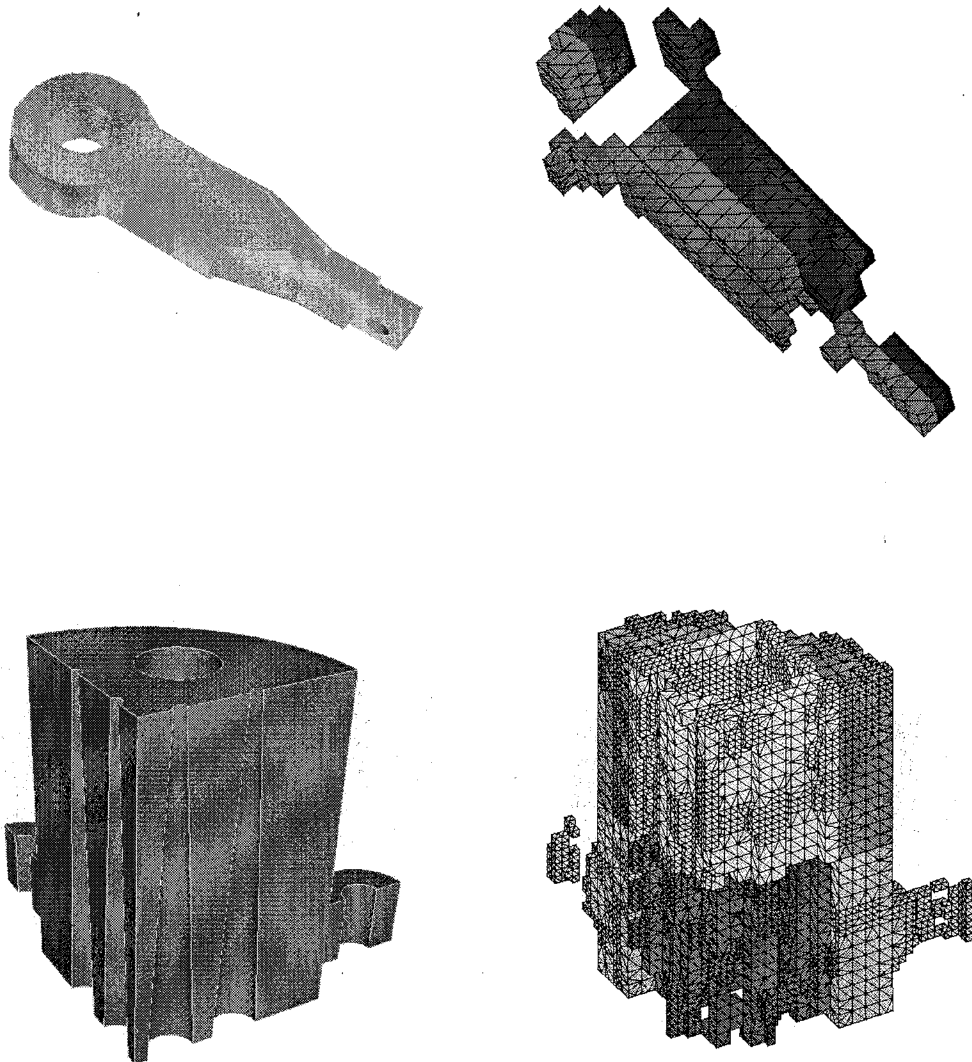
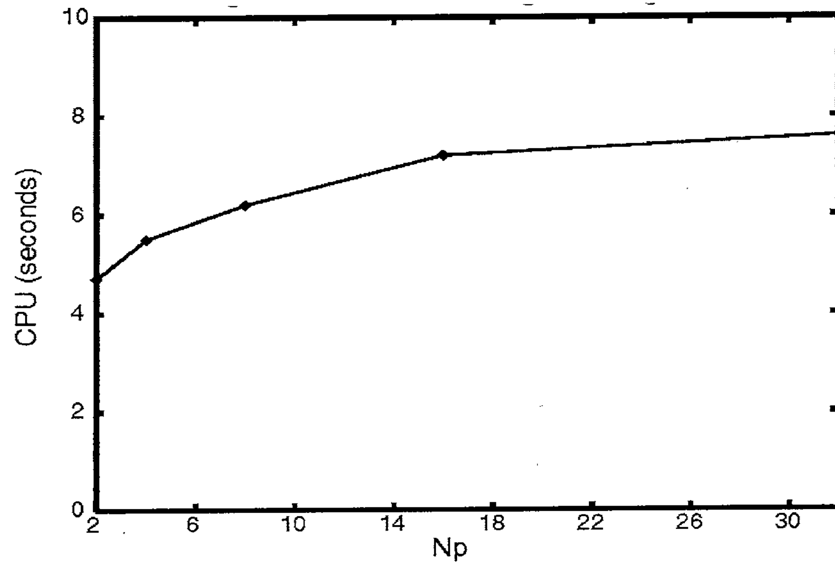
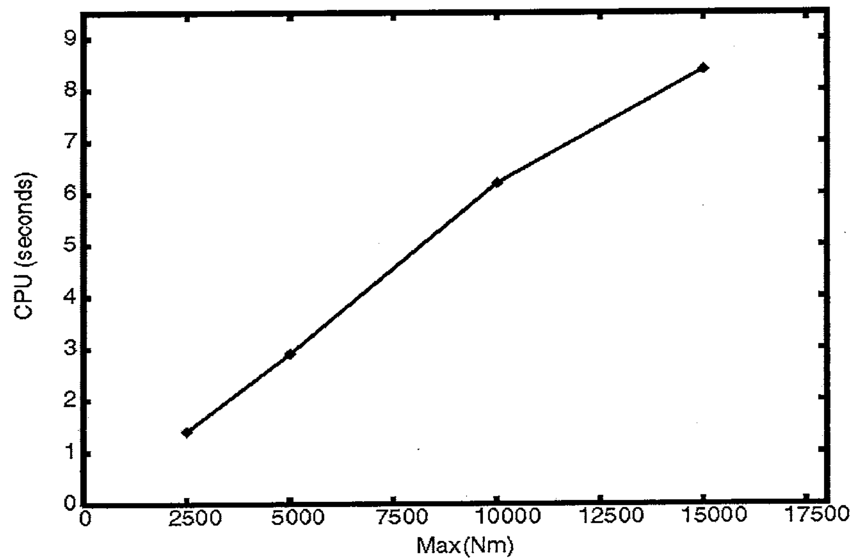


Figure 13. Template meshes of geometric models which are partitioned by two and four processors.

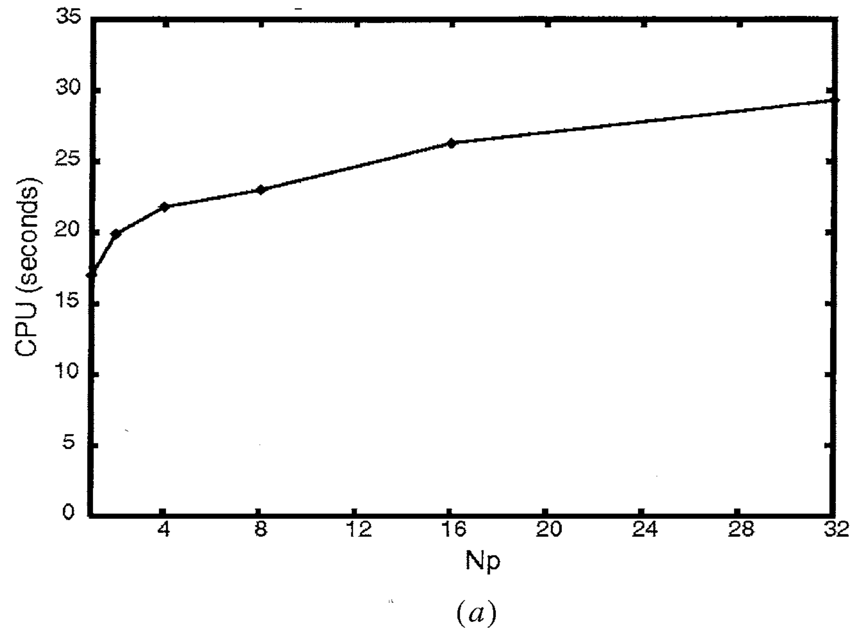



(*a*)

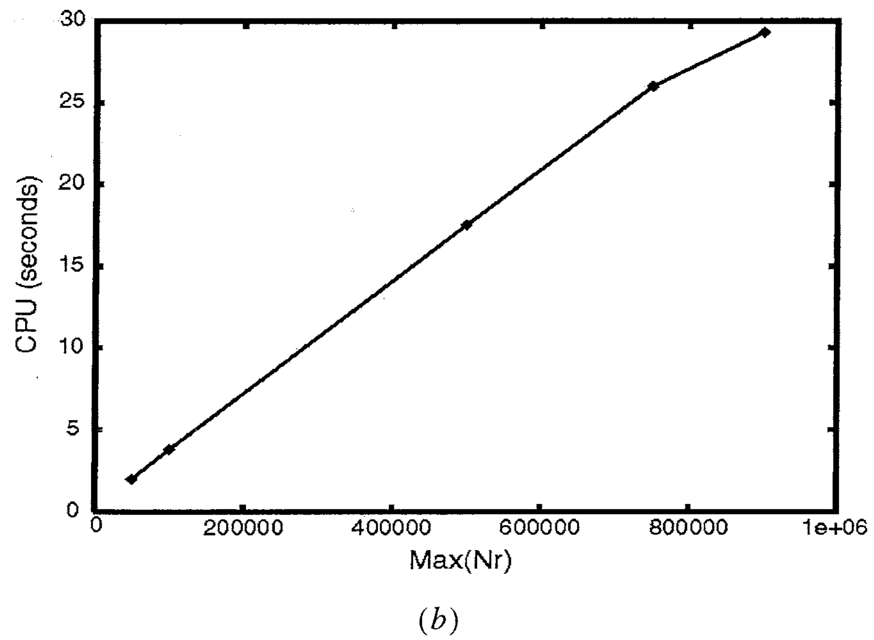



(*b*)

Figure 14. (a) Octant migration scalability experiment where $10,000 \times N_p$ octants, out of a total of $600,000 \times N_p$ octants, are migrated. (b) Results obtained by holding the total tree size fixed at 4,800,000 octants on 8 processors while migrating an increasing number of octants.

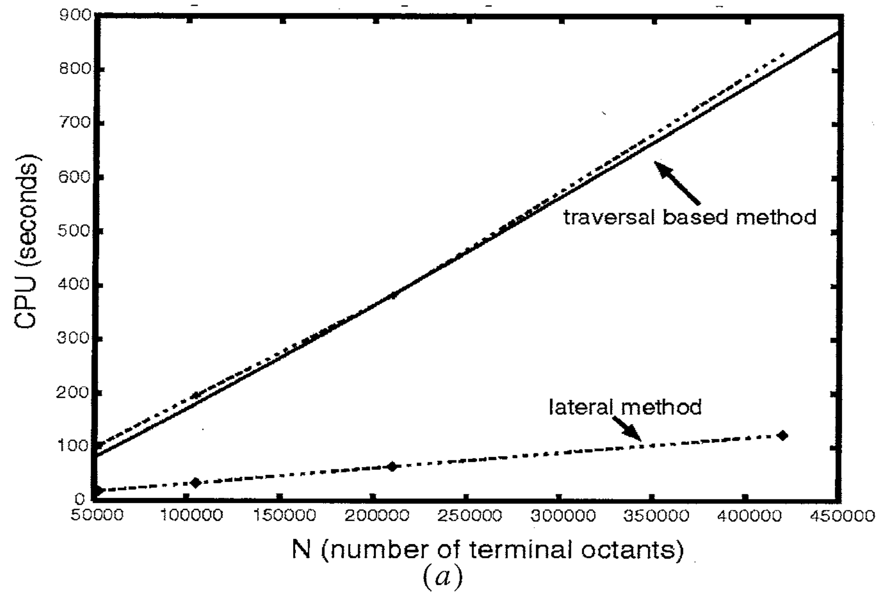



(*a*)

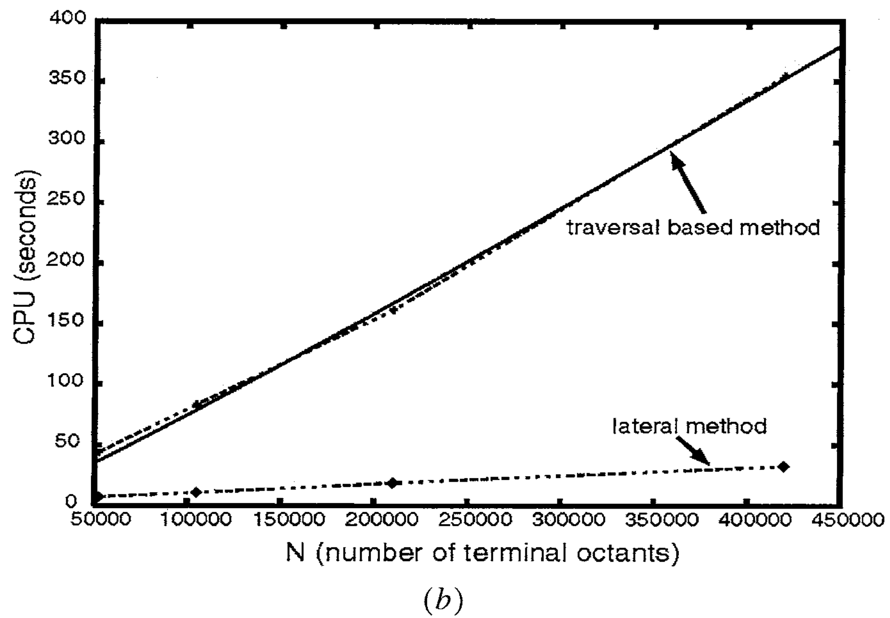



(*b*)

Figure 15. (a) Octant refinement scalability experiment obtained by allocating a total of 500,000 octants on each processor. (b) Timing of octree refinement algorithm on 8 processors.



($a$)



($b$)

Figure 16. (a) CPU results for face neighbor retrieval algorithms and (b) edge neighbor retrieval algorithms.

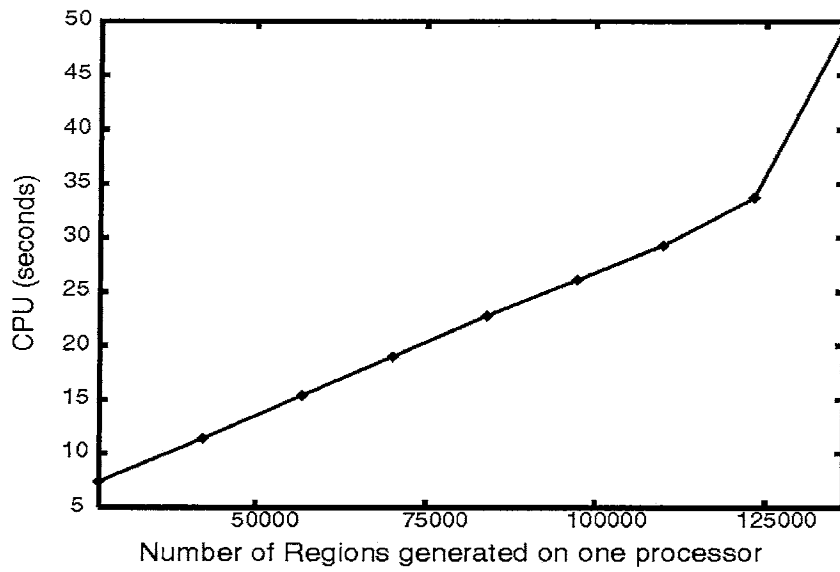
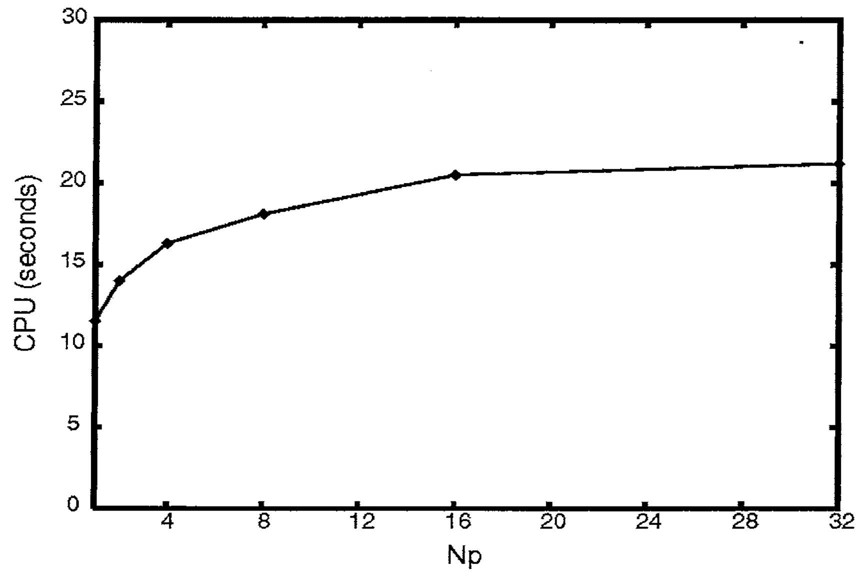
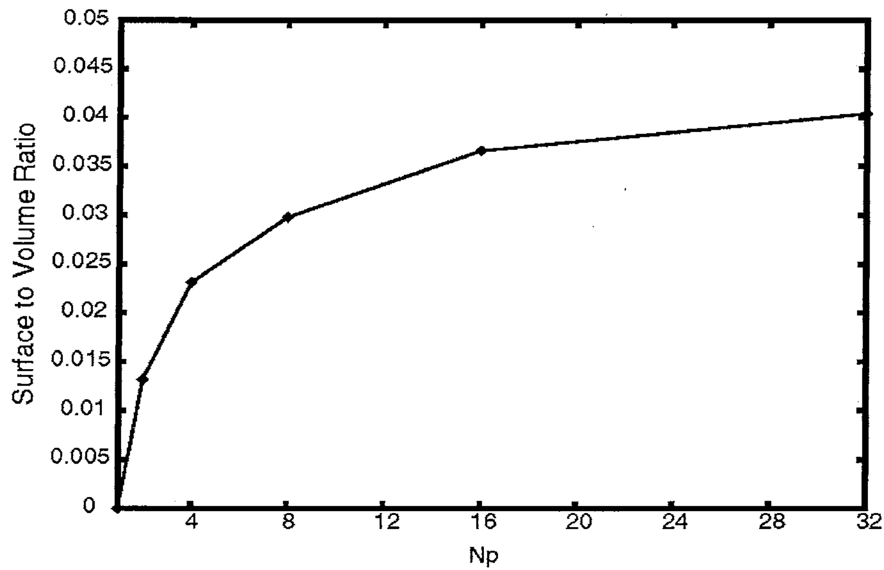



Figure 17. CPU results for meshing part of octree template algorithm.




(*a*)




(*b*)

Figure 18. (a) Scalability experiment of octree template algorithm obtained by meshing approximately $43,000 \times N_p$ regions and setting interprocessor mesh links on the partition boundary. The growth with $N_p$ is indicative of an increase in the number of mesh faces on the partition boundary as $N_p$ is increased (b).