24

Parallel Unstructured Grid Generation

Hugues L. de Cougny
Mark S. Shepherd

24.1 Introduction 24-1
24.2 Requirements for Parallel Mesh Generation................ 24-2
24.3 Classification of Parallel Mesh Generators 24-2
24.4 Meshing Interfaces Along with Subdomains 24-2
24.5 Premeshing Interfaces.. 24-5
Initial Coarse Mesh Partitioning • Tree Partitioning • Prepartitioning
24.6 Postmeshing Interfaces 24-10
24.7 Conclusion.. 24-17

24.1 Introduction

Scalable parallel computers have enabled researchers to apply finite element and finite volume analysis techniques to larger and larger problems. As problem sizes have grown into millions of grid points, the task of meshing models on a serial machine has become a bottleneck for two reasons: (1) it will take too much time to generate meshes, and (2) meshes will not fit in the memory of a single machine.

Parallel mesh generation is difficult, because it requires the ability to decompose the domain to be meshed into subdomains that can be handed out to processors. This is referred to as partitioning. Partitioning in the context of parallel mesh generation is hard because it has to be done with an input that is either a geometric model or a surface mesh. This means one is trying to partition a 3D domain having only the knowledge of its boundary, at least initially. In contrast, it is much easier to partition a 3D mesh, which is what finite element or finite volume parallel solvers typically do. Proper evaluation of the work load is also a challenge in parallel mesh generation. It is problematic to accurately predict the number of elements to be generated in a given subdomain, or how much computation per element will be required. This leads to difficulties in maintaining good load balance at all times.

There are two types of commercially viable parallel architectures: (1) distributed memory, and (2) shared memory [11]. Distributed memory machines are such that each node has its own local memory. They are often associated with message passing libraries, such as MPI [1]. With a message passing library, the programmer is explicitly responsible for communicating data across processors if needed. With a shared memory machine, there is a global address space that each node can read and/or write to. To gain full efficiency, and reduce communication (at the machine level) to a minimum, on today's shared memory computers, the programmer may have to arrange the data in a specific form depending on how the problem is partitioned. Also, high-level programming languages, such as FORTRAN 90 [12], may not be well-suited for parallel mesh generation because of the lack of a static structure to the problem. In the following, focus is given to the distributed memory parallel architecture. It is assumed parallelism is driven by a message passing library, and in particular, MPI [1].

0-8493-2687-7/99/$0.00+$.50
© 1999 by CRC Press LLC

The next two sections discuss the requirements that any parallel mesh generator should fulfill and how parallel mesh generators can be classified into three separate classes. The following three sections describe parallel mesh generation techniques presented in the literature using this classification. The last section will conclude this chapter with remark and comments.

24.2 Requirements for Parallel Mesh Generation

The ideal parallel mesh generator should be

1. Scalable with respect to time and memory
2. Efficient in a parallel sense
3. Stable

A process is considered "time" scalable if the running time increases slowly with the number of processors, assuming the ratio of problem size to number of processors stays constant. As an example, a process with a complexity of $O((n/n_p)\log(n_p))$, where n related to the problem size and n_p is the number of processors, is scalable since the $\log(n_p)$ term increases slowly with n_p. The same concept applies to "memory" scalability. The memory requirements on a single processor should increase slowly as the problem size increases with the number of processors at the same rate. Scalability is an absolute requirement. If the parallel procedure is not scalable, there will be a limit, sooner or later, on how big a problem can be. Parallel efficiency refers to how well the parallel procedure makes use of the computing resources that are available [11]. Idling processors should be avoided as much as possible. Parallel efficiency is usually related to how well the work load is balanced across the available processors (load balancing). Parallel efficiency should not be confused with "sequential" efficiency, which relates to "sequential" algorithms and has nothing to do with parallelism. In the following, efficiency will refer to parallel deficiency unless noted otherwise. Parallel efficiency is not an absolute requirement but is very desirable. Note that a parallel procedure can be scalable but inefficient, and vice-versa. Stability is with respect to the quality of the produced triangulations. If the quality degrades as the number of processors increases, the parallel mesh generator is not stable.

24.3 Classification of Parallel Mesh Generators

Parallel unstructured mesh generators presented to date all employ the concept of domain partitioning. Figure 24.1 shows a partitioned domain in 2D as well as the associated partition graph obtained by connecting neighboring subdomains with a graph edge. Typically, a processor will be given the task to mesh a subdomain. What differentiates the various approaches is how they treat the interfaces between subdomains. In this paper, three classes of parallel unstructured mesh generators are considered:

1. Those that mesh interfaces as they mesh the subdomains
2. Those that premesh the interfaces
3. Those that postmesh the interfaces

The first class of parallel mesh generators refers to those that neither premesh nor postmesh interfaces. Interfaces are meshed at the same time as subdomains. In the second class, objects are partitioned in such a way that subdomain meshing requires no communication. This is possible by meshing interfaces before the subdomains. In the third class subdomains are meshed, and interfaces are left out for later processing.

24.4 Meshing Interfaces Along with subdomains

The parallel implementations of the Bowyer–Watson algorithm [2, 21] (see Chapter 16) by Chrisochoides and Sukup [4] and Okusanya and Peraire [13] are examples of meshing interfaces at the same time as

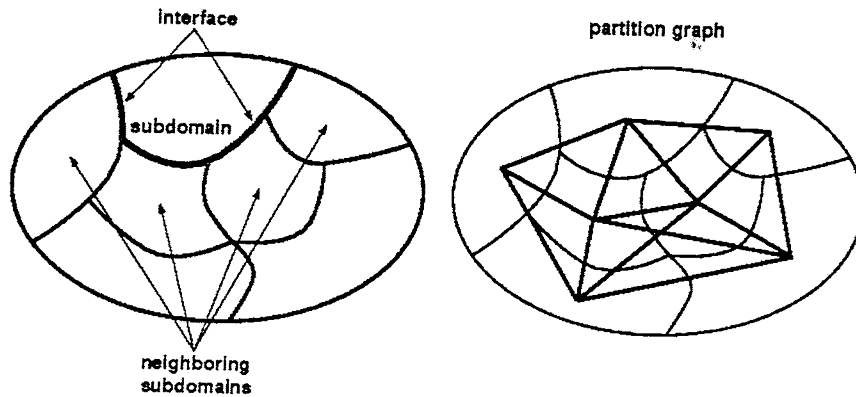

FIGURE 24.1 Example of partition in 2D.

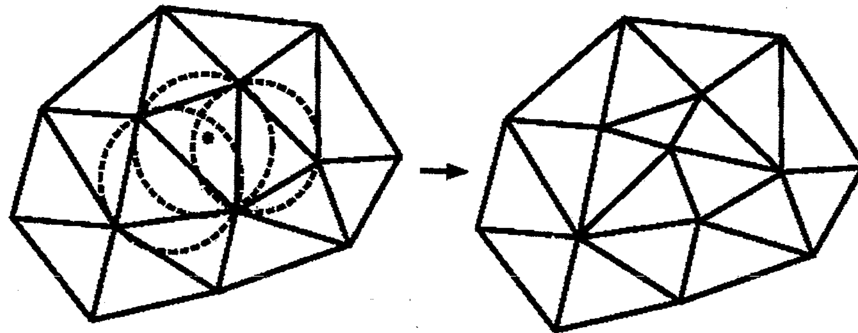

FIGURE 24.2 Delaunay insertion in 2D.

subdomains. The input is a distributed initial mesh that is boundary conforming. It should be noted that this initial mesh could potentially be obtained using the same algorithm, assuming a parallel boundary recovery procedure is available [9].

Assuming element sizes have been prescribed across the domain, any mesh edge in the triangulation that is too long is refined by inserting one or more vertices along the edge using the Bowyer–Watson algorithm [10]. In practice, imposed sizes are stored in a secondary structure such as a background grid or a tree (see Chapter 14 and 15). It should be noted that if the number of grid cells or octants is proportional to the size of the input, the grid or tree has to be distributed to ensure "memory" scalability.

Given a point to insert, the Bowyer–Watson algorithm proceeds as follows (in two dimensions):

1. Find one mesh face that contains the new point.
2. From that mesh face, find all mesh faces whose circumcircles contain the new point using mesh adjacency.
3. Delete the mesh faces (this creates a cavity).
4. Connect the boundary edges of the cavity to the new point.

A graphical description is given in Figure 24.2. If the mesh is distributed, the insertion of a new point on a given processor may not be possible if the cavity extends to neighboring processors due to the mesh being distributed.

The parallel Bowyer–Watson algorithm as described by Chrisochoides and Sukup [4] operates by looping over the following inner loop:

```
for each point to insert do
  get triangle that contains it
  perform task:
    expand cavity
    if cavity cannot be obtained then
      add uncompleted task to blocking-queue
      send a request to neighboring processor(s)
        for needed triangles
    else then
      delete cavity
      connect cavity's boundary to point
    endif
  poll for pending requests
  put received requests (if any) in ready-queue
  move any task from the blocking-queue
    which has been serviced (by a neighboring processor) to
    the ready-queue
  while ready-queue not empty do
    perform task
    if task can be serviced on processor then
      notify requesting processor that
        task has been serviced
    endif
  endwhile
endfor
```

The *blocking-queue* contains tasks that are suspended due to missing information residing on other processors. The *ready-queue* contains tasks that can be performed on a processor. Tasks can switch from the *ready-queue* to the *blocking-queue*, and vice-versa. The complete procedure is actually an outer loop that adds to the inner loop the processing of the *ready-queue* and a check for termination. The outer loop is needed since some processors may still have points to insert while others are done.

This procedure has been implemented using Active Messages [3] on the IBM-SP2. From Chang et al. [3], "Active Messages is a low-latency communication mechanism that minimizes overheads and allows communication and computation to be overlapped in multiprocessors." With Active Messages, a processor must poll for pending messages. If the poll is a hit, the message is received. Polling induces negligible overhead (at least on the IBM-SP2).

This procedure is scalable since a processor usually needs to communicate with its neighbors when inserting a point close to the partition boundary. This is usually true if the partitions are initially, and remain, "bulky." A "bulky" partition is such that the ratio of surface to volume is high.

For this procedure to work well, and therefore have a chance to be efficient, communication must overlap computation well. Beside this communication/computation overlapping issue, the efficiency of the above procedure depends upon how well the computation load is distributed. It is difficult to evaluate how much work is needed to "refine" a subdomain, or more exactly, how many vertices a processor will have to insert. It is assumed that the work required to insert a point is, on average, constant. Note that the number of vertices to be inserted on a subdomain is proportional to the number of elements that will be generated. A rough estimate of the number of elements that need to be generated on a given subdomain can be obtained after building a secondary structure such as a quadtree (in 2D) from imposed sizes that, for example, satisfies the maximum 2:1 level of difference rule. This tree construction is similar to the one described in [5]. The number of interior and boundary terminal quadrants (in 2D) provides a rough estimate of the number of elements that will be generated on the subdomain. Here load balancing is more difficult, since work on a given processor may be induced by a neighboring processor. This

typically happens when points are inserted near partition interfaces. It is assumed that, for a given processor, the work induced by neighboring processors should average out the work the processor itself has relayed to neighboring processors. This means that points to be inserted near interfaces should be evenly distributed among processors. Although very different from interface postmeshing, discussed later, this raises the same basic issue of how to partition the interface for proper load balance. Another issue related to efficiency is how much time is spent updating the various mesh data structures as neighboring processors answer to sent requests. The updating procedures must be very fast, typically as fast as the deletion and creation procedures used in the course of the Bowyer–Watson algorithm.

This parallel Bowyer–Watson algorithm is stable with respect to triangulation quality as the number of processors increases since (1) the Delaunay triangulation is unique for a random input, and (2) no interior artificial boundaries are introduced (see Section 24.5 for when this happens).

24.5 Premeshing Interfaces

This class has been further subdivided into three subclasses depending on how the partitioning into subdomains is performed:

1. Partitioning of an initial coarse mesh
2. Partitioning of a background tree
3. Direct partitioning (prepartitioning) of the input surface mesh

24.4.1 Initial Coarse Mesh Partitioning

A commonly used approach [7, 22] consists of the following:

1. Generate a coarse initial mesh.
2. Partition that coarse mesh into n_p subdomains.
3. Refine interface edges of coarse mesh to proper sizes.
4. Distribute the subdomains to the n_p processors.
5. Mesh subdomains.
6. Optimize locally the submeshes.
7. Optimize globally the assembled mesh.

Figure 24.3 gives a graphical description of the procedure.

Initial mesh generation, partitioning, and global optimization are performed on one processor (host), and are therefore not scalable. Subdomain mesh generation and local mesh optimization phases are performed in parallel. These steps are scalable.

The partitioning of the coarse mesh should be such that the subsequent parallel subdomain mesh generation phase is load-balanced. This is a difficult task. The best one can do is to define heuristics to estimate the number of elements the mesher will generate on a given subdomain. If partitioning is done well, then it is expected that the speed-up will be nearly perfect for the subdomain meshing generation.

It is important to keep in mind that the quality of the meshes generated should not degrade as the number of processors increases. This is a concern since this form of partitioning produces "artificial" boundaries. A constrained Delaunay mesher is usually likely to be minimally influenced by these artificial boundaries as long as they are not too close to "natural" boundaries. On the other hand, an advancing front method is likely to create triangulations that degrade is artificial boundaries multiply. This is due to the nature of the advancing front method which has, in general, a tendency to create poor elements as fronts collide [7]. To alleviate this problem, it is necessary in this case to optimize the mesh.

24.4.2 Tree Partitioning

Saxena and Perucchio [14] suggest a tree decomposition of the geometric model to drive the parallel meshing process. Here the input is a geometric model, not a boundary mesh. In 3D, the terminal octants

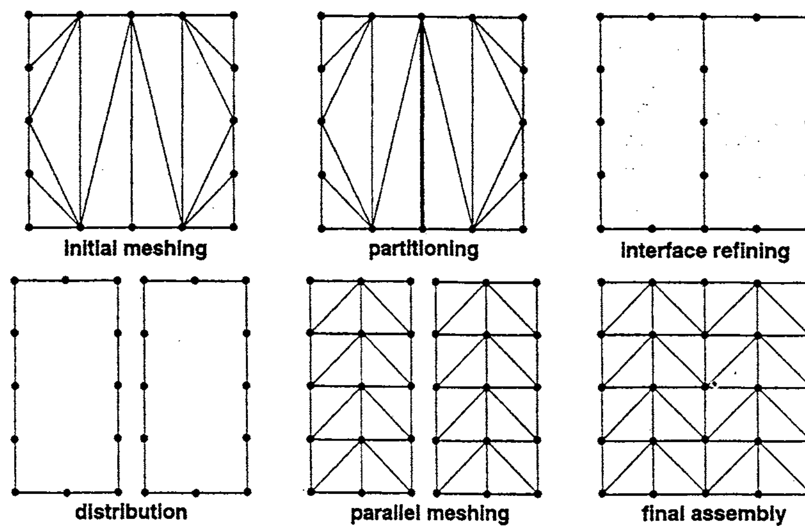

FIGURE 24.3 Meshing of subdomains coming from the partitioning of an initial coarse mesh.

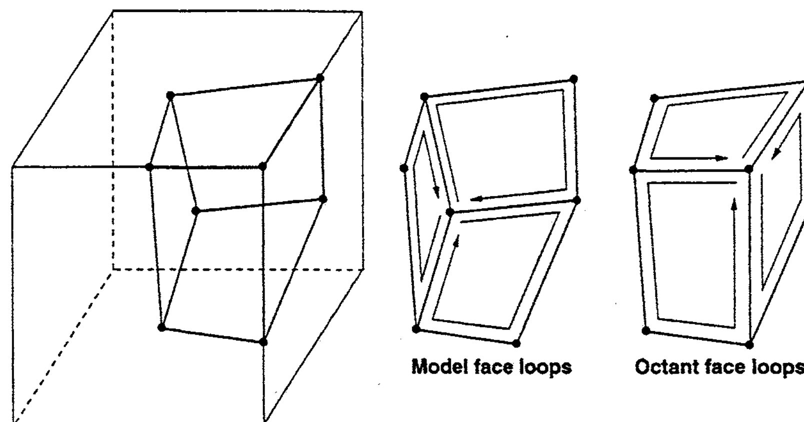

FIGURE 24.4 Example of model face and octant face loops.

are such that their sizes correspond to the sizes imposed by the meshing attributes. Interior terminal octants are meshed using meshing templates. Terminal octants that interact with the domain's boundary (boundary octants) are intersected with the model and then meshed using either a meshing template or an element extraction technique. The interaction between a boundary terminal octant and the model results in the creation of model face loops and interior (to the model) octant face loops [16]. Figure 24.4 shows model face loops and octant face loops for an octant that interacts with three model faces joining at a model vertex.

The set of interior and boundary octants is partitioned among the available processors. The process of intersecting boundary terminal octants with the model and meshing the terminal octants is performed in parallel and without communication. Since an octant face can be shared by several processors (two if the tree is uniform) and meshes on interfaces have to match, care must be taken when meshing octant face loops. The Delaunay triangulation is very attractive here since it is unique, assuming vertices are not in a degenerate situation (four vertices forming a rectangle). Because octant faces are rectangles, it is likely a loop on an octant face has degeneracies. By inserting loop vertices in a given order, the uniqueness of the Delaunay triangulation can again be guaranteed [15]. Note that the meshing of model face loops does not require any such consideration since model face loops cannot be on interfaces. Once octant

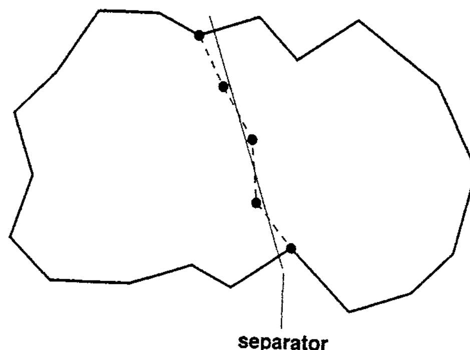

FIGURE 24.5 A separator and its associated triangulation in 2D.

face loops and model face loops have been meshed, interiors of octants are meshed with meshing templates or element extraction techniques.

Octree generation and partitioning are performed sequentially, and are therefore not scalable. It should be noted that a parallel scalable procedure to perform both at the same time is described later in the present chapter. The subdomain meshing procedure is performed in parallel and is scalable.

The performance of the parallel steps of the meshing procedure depends upon how well the partitioner can anticipate how much work will be spent meshing an octant. It is easy to figure this out for an interior octant. It is, however, difficult to estimate how much work will be spent on meshing a boundary octant since one does not know *a priori* how complex the interaction with the model will be.

Stability of the meshing procedure (with respect to triangulation quality) is not an issue here, since identical meshes are created irrespective of the number of processors.

24.4.3 Prepartitioning

Galtier and George [8] prepartition a surface mesh by triangulating appropriately placed separators. A separator cuts a domain into two parts. Given a surface mesh and a separator (say, a plane), the triangulation of the separator is such that

1. It separates, without modification, the initial surface mesh into two subsurface meshes.
2. Sizes of mesh entities on the separator are consistent with imposed sizes.

The separator is not triangulated in the usual sense. The geometry of the separator is used to guide the meshing of the domain, defined by the input surface mesh, in the vicinity of the separator. The triangulation associated with the separator is made of triangles. In other words, a separator and its associated triangulation have the same dimension. Figure 24.5 shows a line separator and its associated triangulation (dashed line segments) when the input is a 2D polygonal mesh (solid line segments). How separators are actually meshed is explained next after a short discussion of the properties of the "projective" Delaunay concept. (Delaunay mesh generation is covered in Chapter 16.)

The technique used to mesh the separator is based on a rather new concept, referred to as "projective" Delaunay. In classic Delaunay, given a set of vertices in 3D, the Voronoï domain at a vertex is defined as the locus of points that are closer to that vertex than to any other vertex in the set. Any two vertices whose Voronoï domains share a side are connected by an edge in the associated Delaunay triangulation. With projective Delaunay onto a surface, given a set of vertices in 3D space, the Voronoï domain at a vertex is defined as the locus of points on the surface that are closer to that vertex than to any other vertex in the set. This defines a Voronoï diagram on the surface. This Voronoï diagram on the surface defines a projective Delaunay triangulation in 3D space. The Voronoï diagram is constructed on the surface and the resulting projective Delaunay triangulation is built in 3D space by connecting vertices whose Voronoï domains on the surface are adjacent. The term "projective" is misleading in this context, since there is actually no projection involved here. Figure 24.6 shows a simple example of "projective"

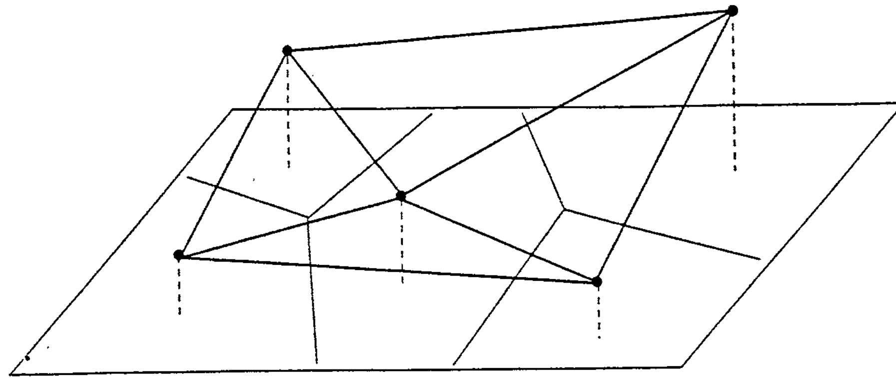

FIGURE 24.6 Example of projective Delaunay triangulation with respect to a plane.

Delaunay triangulation when the surface is a plane. Given a set of vertices in 3D space and a separator, only a subset of these vertices are involved in the projective Delaunay triangulation of the separator. This means the meshing of the separator is local to the separator.

Given an input surface mesh and a separator, the projective Delaunay triangulation of the separator is obtained as follows:

1. Define the poly-line boundary of what will be the triangulation associated with the separator by intersecting edges of the input surface mesh with the separator.
2. Build the "projective" Delaunay triangulation of the separator using only vertices from the input surface mesh.
3. Recognize the poly-line boundary.
4. Delete any mesh face that is outside the poly-line boundary.
5. Insert additional vertices on edges that are too long according to meshing attributes.

Figure 24.7 shows the meshing of a planar separator on a cube. The bottom left picture corresponds to step 2. The bottom right picture corresponds to step 5. Note that only one additional vertex has been inserted. This process is similar to the building of constrained Delaunay triangulations using insertion [9]. If the poly-boundary is not part of the projective Delaunay triangulation, there is no attempt in trying to recover the boundary. If boundary edges are missing, the meshing of the separator is aborted, and an alternate separator is considered. Mesh entities resulting from the meshing of two different separators cannot intersect each other because (1) the two projective Delaunay triangulations are part of the Delaunay triangulation of the set of vertices appearing on these two triangulations, and (2) the Delaunay triangulation is unique. It is assumed there are not Delaunay degenerate situations, that is, more than four vertices on a sphere. However, a mesh entity on a separator can possibly intersect another mesh entity on the surface mesh. If this happens, the meshing of the separate is again aborted. In the context of prepartitioning, if a separator cannot be meshed, a nearby separator can be considered in its place. This means that, even if a specific separator cannot be meshed, the prepartitioner may still succeed. Nevertheless, due to the possible failure of meshing a given separator, there is no real guarantee the prepartitioner will always succeed at placing the separators where they were meant to be. The cost of meshing a separator depends upon the number of generated mesh faces.

Two different techniques for prepartitioning are considered [8]:

1. Cuts along a single direction.
2. Recursive cuts.

In both methods, the separator surfaces are planes. It should be noted that separators do not have to be planes. A separator plane is always perpendicular to the cutting direction. It can be chosen so that it separates any domain into two subdomains with nearly equal number of surface mesh faces. Given the

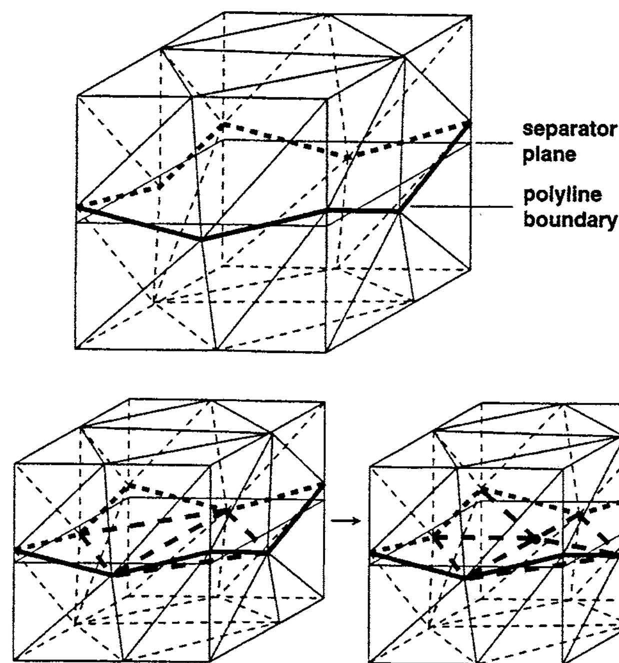

FIGURE 24.7 Meshing of a separator plane on a cube.

surface mesh of a model, the principal axis of minimum inertia that generates the lowest rotation momentum is a good choice for a cutting direction. In the first method, all cuts are made along that axis. In the second method, the first cut is along that axis. This defines two subdomains that can themselves be cut independently along their respective principal axes of inertia, thus, forming four subdomains. Recursive cutting continues until the desired number of subdomains has been obtained. *A priori*, the desired number of subdomains is equal to the number of processors.

Efficient parallelization of the first method requires separators to be sufficiently distant from each other so that there is no interference when meshing them. Each processor, except one, is responsible for the meshing of a separator. Note that this requirement may conflict with proper load balance during the volume mesh generation phase. The cost of meshing the separators in parallel depends on the maximum number (among all processors) of generated mesh faces. This does not scale since, as the number of surface mesh faces increases, the number of mesh faces to generate on a separator increases (at the same rate in the case of uniform sizing) irrespective of how many processors are used.

The second method can be easily parallelized with the *divide and conquer* paradigm [11]. At each recursive cut, one half of the problem goes to one half of the processors. The cost of meshing the first separator (corresponding to the first cut) depends upon the number of mesh faces generated. This again does not scale if only one processor is involved in meshing the separator. Scalability can be achieved only if the meshing of an individual separator can be performed in parallel. This was not discussed [8].

Once prepartitioning is complete, subdomain meshing is performed in parallel and without any communication using constrained Delaunay triangulation in 3D [9]. Note that prepartitioning implicitly distributes the input meshed needed by the volume mesher on each processor. Subdomain meshing is clearly scalable.

The quality of generated triangulations may degrade as the number of processors increases. This is due to the fact that artificial boundaries are created that can potentially be close to original boundaries. Because a Delaunay method is used here, the creation of artificial boundaries far enough away from original boundaries will not cause degradation. It is possible to check for closeness of mesh entities as the separators are being meshed and take appropriate decisions. Checking for closeness is, however, expensive and will lower the performance of prepartitioning in terms of speed.

24.6 Postmeshing Interfaces

This technique has been used first by Shostko and Löhner [18]. Given an input surface mesh, a background grid is built serially on processor 0 (host). The role of the background grid is twofold:

1. To keep track of desired element size information in space.
2. To control the parallel execution.

The task parallel paradigm drives the parallel mesh generator. Tasks are handed out by a dedicated host processor. Other processors are referred to as nodes. The host processor is responsible for:

1. Building a background grid.
2. Partitioning the background grid in at least n_p subdomains.
3. Handing out a background grid subdomain along with the front faces it contains to the next available node.

The individual n_p nodes do the following:

1. Mesh background grid subdomains given by the host.
2. Send back to the host the front faces that could not be processed.

On a given node, the advancing front method is used to mesh the subdomain defined by the background grid elements. To prevent overlapping of submeshes coming from different nodes, a mesh region will not be created if it crosses the subdomain's boundary. More precisely, it will not even be created if it is too close. Assuming the distance between interfaces is always large enough, stability of the parallel mesh generation procedure with respect to triangulation quality degradation will be maintained.

Subdomains should be such that the rate of success of the advancing front method is as high as possible. This rate of success can be defined as the ratio of front faces for which mesh regions could be created to the total number of mesh faces processed. If this ratio gets too low, nodes spend most of their effort determining that they cannot create mesh regions. Partitioning should define "bulky" partitions, that is, partitions with a low surface-to-volume ratio. Note that, when computing this ratio, only the surface shared by two processors should be considered. A "greedy" algorithm [6] that looks at element adjacency to build subdomains is used here. It should be stressed that this partitioning is performed on one processor. This is fine with respect to scalability as long as the size of the background grid is constant, that is, is not a function of the size of the input mesh. If the size of the background grid depends upon the input surface mesh, partitioning must be performed in parallel for that step to be scalable. Concerning "memory" scalability, if the size of the background grid (number of elements) is of the order of the size of the input surface mesh, the background grid should be distributed.

The host-nodes paradigm also poses a problem for scalability. This can be easily seen when the host is handing out subdomains to available nodes for meshing. Assume that the host initially holds n input mesh faces and that it is handing out n/n_p mesh faces to each processor in turn. The cost of this operation is $O(n)$, which is not scalable. The host-nodes paradigm poses a problem any time the host has to communicate with a nonconstant number of processors at the same time.

After the nodes have created the mesh regions within their respective subdomains, the space in between the meshed subdomains remains to be meshed. The "skeleton" of this empty space is made up of the interfaces between the subdomains. In 3D problems, there are three types of interfaces:

1. Faces.
2. Edges.
3. Vertices.

Figure 24.8 shows the interfaces after subdomain meshing on a simple 2D example with four processors. In this particular example, there are four edge interfaces and one vertex interface.

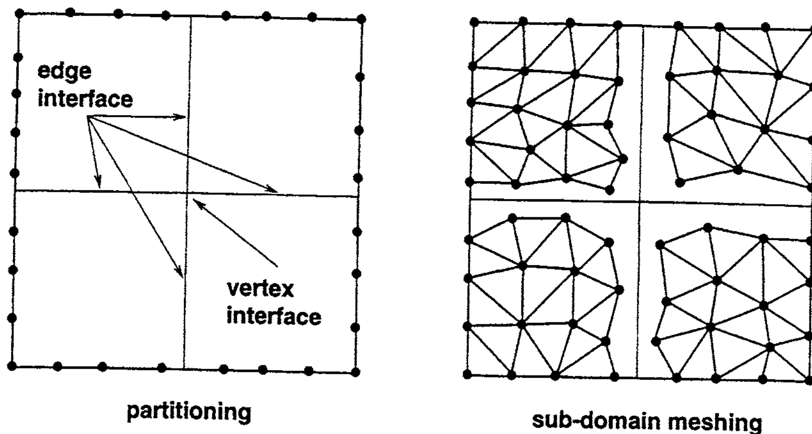

FIGURE 24.8 Interfaces in 2D.

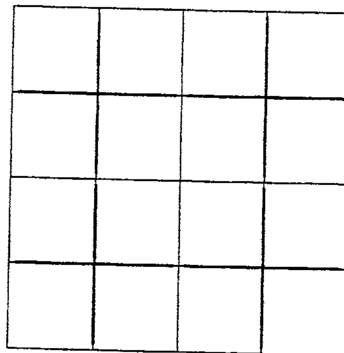

FIGURE 24.9 Interface meshing in 2D using "base" methodology.

The "base" methodology for interface meshing is as follows:

```
for each corner interface do
  if no adjacent subdomain is marked as being used then
    mark adjacent subdomains as being used
    hand out data to next available node (for meshing)
  endif
endfor
```

Implementation-wise, the node is given (by the host node) the background grid elements and the active faces of the subdomains adjacent to the corner interface being considered. This means the node will mesh all interfaces coming to that corner. The "base" methodology is repeated until there are no more interface vertices remaining. Figure 24.9 explains the "base" methodology on a 2D example. What is shown are 16 subdomains belonging to 16 processors. The thicker lines represent interfaces handed out, for meshing, to nodes. To be more precise, each + sign is given to a node for meshing.

The cost of one iteration of the "base" methodology is equal (in 3D) to the maximum number of face interfaces coming to a vertex interface times the maximum size of a face interface. The maximum number of iterations is equal to the maximum number of face interfaces coming to a vertex interface. As the

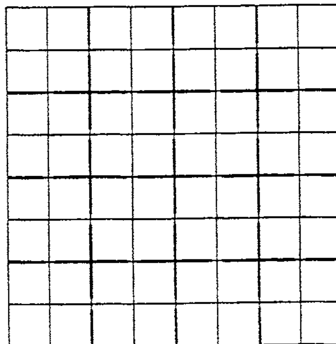

FIGURE 24.10 Subdomain refinement to improve interface meshing in "improved" methodology.

problem size increases proportionally with the number of processors, one can assume these quantities remain nearly constant. This means that the procedure used for meshing the interfaces is scalable. It is, however, not efficient with respect to parallelism. Considering Figure 24.9, only four processors out of the 16 available can work at meshing the interfaces at the same time. Better efficiency can be obtained by further subdividing the subdomains once subdomain meshing is complete. Figure 24.10 shows the effect of subdomain refinement on interface meshing. Note that the initial subdomains were the same as in Figure 24.9, meaning the vertex interfaces are at the same locations. Comparing with Figure 24.9 where only four processors could be work at the same time, here, all processors can work at the same time. This "improved" methodology leads to better load balance and, therefore, better parallel efficiency.

The parallel mesh generator presented by de Cougny, et al. [5] also uses an advancing front method to mesh the volume in between a surface mesh and template-meshed interior octants. Given a distributed surface mesh, the latest version of this procedure builds a distributed octree in parallel [20]. The octree is such that

1. The root octant fully encloses the input mesh.
2. The size of any terminal octant is comparable to the size of the input mesh entities it contains or will contain.
3. There is no more than one level of difference between octant edge neighbors.

The purpose of the tree is to

1. Enable data localization during volume meshing
2. Have a quickly defined spatial structure that can be partitioned
3. Use fast octant meshing techniques on interior terminal octants that are more than one element deep from the surface mesh

The input for tree building is a distributed array of points in 3D space associated with a tree level. It is referred to as the (*point, level*) array. This array is built by considering, for each mesh vertex on the input surface mesh, the average length of the connected mesh edges transformed into a tree level. Any length d can be transformed into a tree level by applying the formula, $level = \log_2(D/d)$ where D is the dimension of the root octant. Tree building is decomposed into two steps:

1. Local root building
2. Subtree subdivision

The process for local root building is as follows:

```
1. Initialize processor set to all processors
2. Create global root octant on each processor in processor set
```

```
3. Initialize control of global root octant to processor set
while all processor sets not of cardinality 1 do
   only consider terminal octants under processor set's control
   4. subdivide (once) any terminal octant
       that needs to be subdivided
   5. assign terminal octants to processors
       according to "load ratio rule"
   6. split processor set into subsets
endwhile
   7. Delete trees on all processors but 0
   8. Migrate terminal octants according to their owning processors
```

The concept of processor sets is an attractive feature of MPI [1]. It enables subdivision of a set of processors into subsets that can run in parallel independently of each other. A processor set can be subdivided into subsets, and each subset can be further subdivided as many times as desired.

In step 1, the processor set is initialized to the complete set of processors. Its cardinality is n_p. The goal of the procedure is to recursively split the processor sets until all processor sets contain a single processor. When a processor set has control of a terminal octant, it means that (1) the terminal octant exists on all processors in the set, and (2) only that set can make decisions regarding whether or not to refine it. In steps 2 and 3, the initial processor set is given control of the root octant. Critical to the effectiveness of the tree-building procedure is what happens within the *while* loop (steps 4, 5, and 6). Consider a processor set controlling a set of terminal octants. As a reminder, the set of terminal octants exist on all processors in the set at this time. The (*point, level*) array known to the processor set contains only entries relevant to the set of terminal octants the processor set is responsible for. If necessary, this array can be evenly distributed among the processors in the set for load balance using a simple data migration scheme. In step 4, the decision to refine, or not refine, at terminal octant is made after (1) having each processor in the set examined its (*point, level*) array keeping track of the maximum level, and (2) having communicated its maximum level with all processors in the set. If the global maximum level is more than the octant's level, the terminal octant is refined once at this point. The reason why a terminal octant is refined only once is because the tree must be as shallow as possible for subtree building, described next. The shallower the tree at that point, the more efficient the complete tree building procedure will be, since subtree building requires no communication. Once the terminal octants under the processor set's control have been processed for refinement (new ones are ignored), they are assigned to processors within the processor set according to a "load ratio rule" (step 5) and the current processor set is split into subsets (step 6).

The "load ratio rule" attempts to make sure that processor subsets will carry a load, measured by the number of points within the volumes of the terminal octants they will be in charge of, that is close to the load average times the number of processors in the subset. Considering Figure 24.11, if the number of processors in the set is three, then, octants 0 and 1 are assigned to processor 0, octants 2 and 3 are assigned to processor 1 and 2. The processor set is split into two subsets: one containing processor 0 and the other one containing processors 1 and 2. The (*point, level*) array is redistributed so that (1) each entry ends up on a processor in the subset that is in charge of the terminal octant that contains the point, and (2) processors in subsets hold the same number of entries. This guarantees locality of data and an even distribution of the (*point, level*) array. Each subset resulting from the split is now considered a processor set in the next iteration of the *while* loop. This process continues until all current processor sets contain a single processor. The rest of the procedure can then be run without using the concept of processor sets, since they have all been reduced to single processors. In Steps 7 and 8, the current tree is actually distributed by deleting it everywhere except on processor 0 and migrating terminal octants based upon which processors have control of them. This procedure builds a distributed "partial" tree where each terminal octant can be seen as a local root of a constructed subtree. Figure 24.12 shows a distributed "partial" tree in 2D. Each terminal octant exists only on a single processor. Details about the data structure

o_ind =	0	1
load =	5	18
o_ind =	2	3
load =	32	5

FIGURE 24.11 Example for the "load ratio rule."

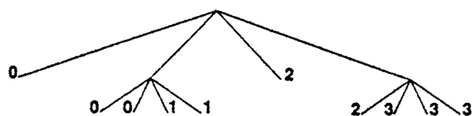

FIGURE 24.12 "Partial" distributed tree.

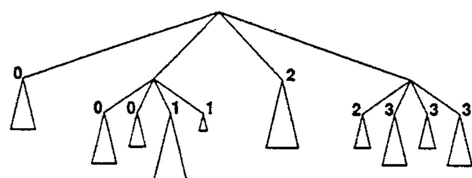

FIGURE 24.13 Complete distributed tree.

used for this distributed tree can be found in [20]. Subtree building, described below, is implicitly load balanced with respect to the (*point, level*) array, that is, each processor will have approximately the same number of points to insert into its subtree(s). This is due to the "load ratio rule" used in the above procedure for local root building.

The process of subtree subdivision is as follows:

```
Each processor is responsible for 1 or more local roots
for each (point, level) do
  get octant(s) the point is in
  if octant_level < level then
    refine octant recursively to level
  endif
enddo
```

Each processor builds subtrees rooted at the local roots. These subtrees only exist on the processors they have been built in. Here terminal octants can be recursively subdivided until the desired level is reached. This procedure requires no communication. Figure 24.13 shows the complete tree structure after subtree building in 2D.

The cost for local root building is $O((n/n_p)\log(n_p))$. The n/n_p term represents how many (*point, level*) each processor is holding. The $\log(n_p)$ factor reflects the number of iterations in the *while* loop. The cost for subtree subdivision is $O((n/n_p)\log(n/n_p))$. The n/n_p term indicates how many (*point, level*) each processor has to insert into its subtree(s). The $\log(n/n_p)$ term is for tree traversals of the subtree(s). The

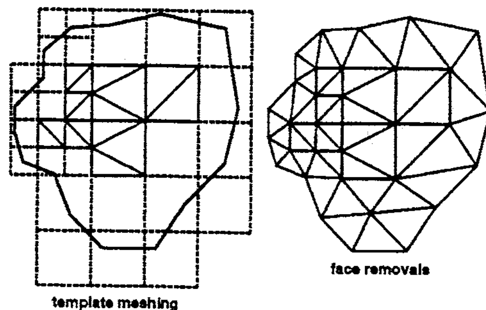

FIGURE 24.14 Template meshing and face removals.

total cost for tree building is dominated by the subtree subdivision cost. Tree building is therefore a scalable process.

Terminal octants are classified according to their interactions with the input surface mesh. If a terminal octant has mesh entities from the input mesh within its volume, it is classified boundary. Once all boundary terminal octants have been recognized, any unclassified terminal octant is classified either interior to a model region or outside. Interior terminal octants at least one element length away from the input surface mesh are meshed using meshing templates. Interior terminal octants are partitioned using a parallel recursive inertial bisection procedure, described below, to ensure load balance while meshing templates. The way templates have been designed is such that triangulations on octant faces shared by two processors are guaranteed to implicitly conform.

Once interior terminal octants have been meshed, the domain between the input surface mesh and the meshed octants is meshed in parallel by applying face removals. A face removal is the basic operation in the advancing front method which, given a front face, creates a mesh origin. Figure 24.14 graphically shows the two meshing steps of the procedure on a 2D example.

It is important to explain how face removals are performed in parallel in order to understand the full parallel face removal procedure. A front face is not removed if the tree neighborhood from which target vertices are drawn is not fully present on processor. This implicitly guarantees this parallel mesh generator is stable with respect to triangulation quality degradation. Face removals can be applied until there is no front face that can be removed. At that point, the tree is repartitioned. The process of applying face removals and repartitioning the tree continues until the front is empty. The parallel face removal procedure is as follows:

```
where there are boundary terminal octants then
  repartition the boundary terminal octants
  apply face removals
  reclassify boundary terminal octants which have no more
    front faces in their volumes as complete
endwhile
```

Tree repartitioning is performed by a parallel recursive inertial bisection procedure [17] based upon the *divide and conquer* paradigm [11]. The input to the parallel recursive inertial bisection procedure is a set of distributed boundary terminal octants. All processors participate in the first bisection along the axis of minimum momentum. Half the processors are given the task to further bisect the terminal octants before the median. The other half is responsible for the terminal octants after the median. The median is such that it separates the set of boundary terminal octants into two nearly equal parts. Bisection continues until the number of partitions is equal to the number of processors. Terminal octants are migrated according to their destinations. When a terminal octant is migrated, the front faces in its volume

are migrated to the same destination. If a front face is connected on one side to a mesh region, the mesh region is migrated. The cost of repartitioning n entities is $O((n/n_p)\log(n/n_p)\log(n_p))$. This is a scalable process. The $(n/n_p)\log(n/n_p)$ term comes from sorting the entities along the axis of minimum momentum. The $\log(n_p)$ factor represents how many times recursive bisection needs to be applied. Details about the implementation of this parallel repartitioning scheme can be found in [17]. This repartitioning method has been chosen because (1) it is relatively easy to parallelize, (2) it generates relatively good partitions [19], and (3) it is multipurpose in the sense that it can be used for other applications than parallel mesh generation. Note that other parallel repartitioners could be used. After a face removal step, boundary terminal octants that have been "filled up" with mesh regions are reclassified as complete and do not participate in the next tree repartitioning.

The number of available processors for meshing is reduced when the rate of success of the face removal step drops considerably. This rate of success is defined as the ratio of successfully removed faces to tried faces. To study the scalability of the face removal meshing loop, assume that, at each step, the number of processors is reduced by half. Without loss of generality, the initial number of processors is assumed to be a power of two. The proposed face removal meshing loop can only be scalable if the number of octants to repartition is reduced by half at each iteration. Reducing the problem by half at each iteration cannot be guaranteed in theory. Although test case results have shown promising speed-ups for up to 32 processors with removal rates greater than one half for most steps, scalability of the described face removal meshing loop is questionable in a theoretical sense.

Scalability can, however, be ensured by explicitly meshing the interface resulting from subdomain meshing. subdomain meshing is the combination of the first partitioning and face removal step. The following procedure to mesh interfaces is similar to the one described by Shostko and Löhner [18]. The main difference resides in the fact that here the host-nodes paradigm is not used. Decision making concerning the repartitioning of interfaces and the actual migration of data is performed in parallel. Interface meshing is hierarchical, that is, face, edge, and vertex interfaces are considered in turn. Also, here, "very" fine-grain parallelism coming from the tree is used to improve parallel efficiency. Since this procedure can be used by other parallel mesh generators, it is discussed without considering the use of template meshing for interior terminal octants. Template meshing on interior terminal octants reduces the sizes of face and edge interfaces, which makes the procedure, described next, more efficient.

A face interface can be meshed by migrating to one processor boundary terminal octants that are closer to that face interface than any other face interface. After the face removal step, each processor assigns its remaining boundary terminal octants, that is, those that have not been "filled up," to its bounding interfaces based on distance consideration. Within a subdomain, any remaining boundary terminal octant is assigned to the closest bounding interface. Figure 24.15 shows the assignment (to interfaces) of boundary terminal octants resulting from subdomain meshing. In this case, the subdomain of interest is bounded by three edge interfaces denoted as 0, 1, and 2. Assignment of boundary terminal octants to face interfaces is performed in parallel by all processors. Figure 24.15 only shows what happens on one processor. The idea is to have each face interface meshed by a single processor by migrating to that processor (unless already there) all boundary terminal octants associated with the face interface. In practice, to avoid unnecessary migration, a processor adjacent to the face interface will be chosen to mesh it. Since the initial partitioning is "bulky" and terminal octants are similar in sizes to the front mesh faces they contain, *a priori* all face interfaces can be meshed within the same step without interference except at edge and vertex interfaces, which is expected. The work needed to mesh a face interface can be accurately estimated by counting the boundary terminal octants that have been associated with it. This means that good load balance during face interface meshing is possible. The cost for face interface meshing is equal to the maximum number of face interfaces a subdomain has, times the maximum number of elements to be generated on a face interface. As remarked previously, this leads to a scalable procedure. Edge interface meshing uses the exact same methodology and is not described here. Vertex interfaces can efficiently be meshed independently of each other since they have become small and bounded subdomains. Note that the tree is not needed anymore when meshing the vertex interfaces.

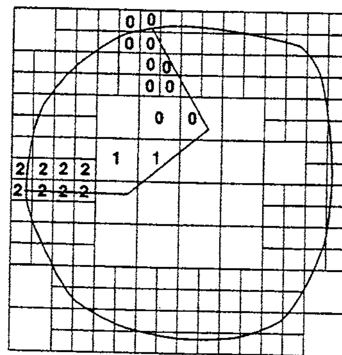

FIGURE 24.15 Assignment of boundary terminal octants to interfaces.

24.7 Conclusion

This chapter has reviewed parallel unstructured mesh generation procedures with respect to: (1) scalability, (2) parallel efficiency, and (3) stability relative to triangulation quality. Scalability appears to be a requirement since it guarantees that, as more processors are used, bigger problems can be solved in reasonable clock time. Parallel mesh generation is difficult because there is no real structure than can "perfectly" drive a parallel algorithm. The final structure appears only upon completion with the generated 3D mesh. Parallel mesh generation is also tedious because it usually involves several processes and data structures that all need to be "time" and "memory" scalable, respectively. The parallel mesh generation field is still very young, which means that the algorithms presented in this chapter are probably evolving very fast and completely new algorithms are being written. Because the development of efficient scalable parallel techniques takes much more time than their sequential counterparts, it may take a while before parallel mesh generation comes to a state of maturity.

References

1. Message passing interface forum MPI: a message-passing interface standard, *International Journal of Supercomputer Applications and High-Performance Computing.* 8(3/4), 1994. Special issue on MPI.
2. Bowyer, A., Computing Dirichlet tessellations, *The Computer Journal.* 1981, 24, 2, pp 162–166.
3. Chang, C.-C., Czajkowski, G., von Eicken, T., design and performance of active messages on the IBM SP-2, Technical report, Cornell University, 1996.
4. Chrisochoides, N. and Sukup, F., Task parallel implementation of the Bowyer–Watson algorithm, *5h Int. Conf. on Numerical Grid Generation in Computational Field Simulations.* Mississippi State University, 1996, pp 773–782.
5. de Cougny, H.L., Shephard, M.S., Özturan, C., Parallel three-dimensional mesh generation on distributed memory MIMD computers, *Engineering with Computers.* 1996, 12, pp 94–106.
6. Farhat, C. and Lesoinne, M., Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics. *Int. J. for Numerical Methods in Engineering.* 1993, 36, pp 745–764.
7. Gaither, A., Marcum, D., Reese, D., A paradigm for parallel unstructured grid generation, *5th Int. Conf. on Numerical Grid Generation in Computational Field Simulations.* Mississippi State University, 1996, pp 731–740.
8. Galtier, J. and George, P.-L., Prepartitioning as a way to mesh subdomains in parallel, *5th Int. Meshing Roundtable.* 1996, Pittsburgh, PA, pp 107–121.

9. George, P. L., Hecht, F., Saltel, E., Automatic mesh generator with specified boundary, *Computer Methods in Applied Mechanics and Engineering.* 92, pp 269–288.
10. George, P. L. and Hermeline, F., Delaunay's mesh of a convex polyhedron in dimension d. Application to arbitrary polyhedra, *Inte. J. for Numerical Methods in Engineering.* 1992, 33, pp 975–995.
11. Jájá, J., *An Introduction to Parallel Algorithms.* Addison-Wesley, 1992.
12. Metcalf, M. and Reid, J., *Fortran 90/95 Explained.* Oxford University Press, 1995.
13. Okusanya, T. and Peraire, J., Parallel unstructured mesh generation, *5th Int. Conf. on Numerical Grid Generation in Computational Field Simulations.* Mississippi State University, 1996, pp 719–729.
14. Saxena, M. and Perucchio, R., Parallel FEM algorithms based on recursive spatial decomposition. I. automatic mesh generation, *Computers & Structures.* 1992, 45(5–6), pp 817–831.
15. Schroeder, W.J. and Shephard, M.S., A combined octree/Delaunay method for fully automatic 3-D Mesh Generation, *Int. J. for Numerical Methods in Engineering.* 1990, 29, pp 37–55.
16. Shephard, M.S. and Georges, M.K., Automatic three-dimensional mesh generation by the finite octree technique, *Int. J. for Numerical Methods in Engineering.* 1991, 32(4), pp 709–749.
17. Shephard, M.S., Flaherty, J.E., de Cougny, H.L., Özturan, C., Bottasso, C.L., Beall, M.W., Parallel automated adaptive procedures for unstructured meshes, *Special Course on Parallel Computing in CFD.* AGARD, 1995, number R-807, pp 6.1–76.49.
18 Shostko, A. and Löhner, R., Three-dimensional parallel unstructured grid generation, *Int. J. for Numerical Methods in Engineering.* 1995, 38, pp 905–925.
19. Simon, H.D. and Teng, S.-H., How good is recursive bisection, technical report, NASA Ames Research Center, 1993.
20. Simone, M., de Cougny, H.L., Shephard, M., Tools and Techniques for parallel grid generation, *5th Int. Conf. on Numerical Grid Generation in Computational Field Simulations.* Mississippi State University, 1996, Vol. II, pp 1165–1174.
21. Watson, D., Computing the n-dimensional Delaunay tessellation with applications to Voronoï polytopes, *The Computer Journal.* 1981, 24(2), pp 167–172.
22. Wu, P. and Houstis, E.N., Parallel adaptive mesh generation and decomposition, *Engineering with Computers.* 1996, 12, pp 155–167.