AGARD-R-807

AGARD

ADVISORY GROUP FOR AEROSPACE RESEARCH & DEVELOPMENT

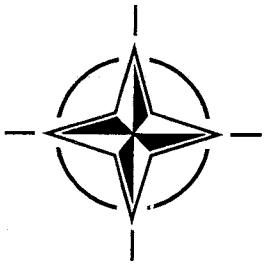
7 RUE ANCELLE, 92200 NEUILLY-SUR-SEINE, FRANCE

AGARD REPORT R-807

Special Course on Parallel Computing in CFD

(l'Aérodynamique numérique et le calcul en parallèle)

The material assembled in this report was prepared under the combined sponsorship of the AGARD Fluid Dynamics Panel, the Consultant and Exchange Program of AGARD, and the von Karman Institute (VKI) for Fluid Dynamics. It was presented in an AGARD-FDP-VKI Special Course at the VKI, Rhode-Saint-Genèse, Belgium, 15-19 May 1995 and 16-20 October 1995 at NASA Ames, United States.



NORTH ATLANTIC TREATY ORGANIZATION

Published October 1995

Parallel Automated Adaptive Procedures for Unstructured Meshes

M.S. Shephard, J.E. Flaherty, H.L. de Cougny, C. Ozturan, C.L. Bottasso and M.W. Beall
Scientific Computation Research Center
Rensselaer Polytechnic Institute
Troy, NY 12180–3590
USA

Summary

Consideration is given to the techniques required to support adaptive analysis of automatically generated unstructured meshes on distributed memory MIMD parallel computers. The key areas of new development are focused on the support of effective parallel computations when the structure of the numerical discretization, the mesh, is evolving, and in fact constructed, during the computation. All the procedures presented operate in parallel on already distributed mesh information. Starting from a mesh definition in terms of a topological hierarchy, techniques to support the distribution, redistribution and communication among the mesh entities over the processors is given, and algorithms to dynamically balance processor workload based on the migration of mesh entities are given. A procedure to automatically generate meshes in parallel, starting from CAD geometric models, is given. Parallel procedures to enrich the mesh through local mesh modifications are also given. Finally, the combination of these techniques to produce a parallel automated finite element analysis procedure for rotorcraft aerodynamics calculations is discussed and demonstrated.

Contents

for parallel mesh generation. Although the mesh enrichments dictated by an adaptive analysis can be satisfied through remeshing by the automatic mesh generator, the computational cost and need to project parameters between meshes indicates the desire to employ alternative mesh enrichment techniques when possible. Chapter 4 presents a set of local mesh modification procedures for the effective refinement and coarsening of meshes.

Given a set of parallel procedures for controlling mesh partitions, for the generation and enrichment of the mesh, the remaining ingredient of the automated adaptive analysis is the adaptive solver. Consistent with the other component procedures presented in these notes, it is assumed that the solver operates on an unstructured mesh which has been partitioned to the various processors of the parallel computer. Under this assumption, adaptive finite volume and finite element solvers are most appropriate. Chapter 5 presents the structure of such a solver. The specific solver discussed is a finite element based procedure which builds directly on the parallel mesh control tools of the earlier sections.

2. Parallel Control of Evolving Meshes

Central to the parallel automated adaptive analysis procedures considered here are tools to control the mesh and its distribution among the processors as the meshes are generated and analyzed. These tools must be able to maintain load balance as the mesh evolves during the computations in such a manner that the interprocessor communications are kept as small as possible. Its is also critical that these procedures operate in parallel and scale as the problem size grows so they do not become the bottleneck in the parallel computation process.

The tools required to support parallel automated adaptive analysis include:

1. data structures and operators to support the model representations employed
2. interprocessor communication control mechanisms
3. mechanisms to effectively move portions of the discrete models generated to various processors so load balance can be maintained
4. techniques to partition the mesh among the processors so the load is balanced and communications are minimumized
5. techniques to up-date the mesh partitions to regain load balance which was lost due to mesh modifications

The minimum data structures needed for an automated adaptive analysis are (i) a problem definition, in terms of a geometric model and analysis attributes, and (ii) a mesh, which the discrete representation used by the analysis procedures. The next section describes a general structure, based on boundary representations, for the problem definition and the mesh. This same form of structure is used to support the partition model used by the partition operators, mesh migration procedures and dynamic load balancing procedures. In additional to these data structures, several procedures described employ tree structures to support searching and spatial enumeration. The mesh partition procedures described in section 2.2 are designed to effectively collect groups of mesh entities for migration and, using the interprocessor communication operators, transfer the information and update all local data structures as needed.

A number of algorithms have been developed to partition a given mesh to a set of processors. The interested reader is referred to references [4, 20, 21, 56, 80] for more information. The current document focuses on procedures to update an existing set of mesh partitions after the mesh has been modified by a mesh adaptation procedure. Section 2.3 presents two classes of procedures for this purpose.

2.1. Mesh Data Structure to Support Geometry-Based Automated Adaptive Analysis

The classic unstructured mesh data structure of a set of node point coordinates and element connectivities is not sufficient for supporting automated adaptive analysis. Richer structures are required to support adaptive mesh enrichment procedures and to provide the links to the original domain definition needed by critical functions, including ensuring that the automatic mesh generator has produced a valid discretization of the given domain. A number of alternative mesh data structures have been proposed for various forms of mesh adaptation. Instead of describing and comparing these structures, a general data structure based on a hierarchy of topological entities is given.

The goal of an analysis process is to solve a set of partial differential equations over the geometric domain of interest, ${}_g\Omega$. Generalized numerical analysis procedures employ a discretized version of this domain in terms of a mesh. Since the mesh domain, $\overline{{}_m\Omega}$ may not be identical to the original geometric domain, $\overline{{}_g\Omega}$, and/or various procedures, such as automatic mesh generation, adaptive mesh refinement and element stiffness integration need to understand the relationship of the mesh to the geometric model, it is critical to employ a representational scheme which maintains the relationships between these two models. Although a number of schemes are possible for defining a geometric domain [58], the most advantageous for the current purposes are boundary-based schemes in which the geometric domain to be analyzed is represented as

$$\overline{{}_g\Omega}(g\{{}_gS\}, g\{{}_gT\}) \tag{1}$$

where $g\{{}_gS\}$ represents the information defining the shape of the entities which define the domain and $g\{{}_gT\}$ represents the topological types and adjacencies[1] of the

[1] In the context of a domain representation, adjacencies are the relationships among topological entities which bound each other. For example, the edges that bound a face, is a commonly used topological adjacency.

and any form of adaptive analysis on conforming unstructured meshes[3], all adjacencies are either stored, or can be quickly determined through a set of local traversals and sorts which are not a function of the mesh size. One set of relationships that can effectively meet these requirements is to maintain adjacencies between entities one order apart. Figure 1 graphically depicts this set of relationships as well as the classification with respect to the geometric domain representation.

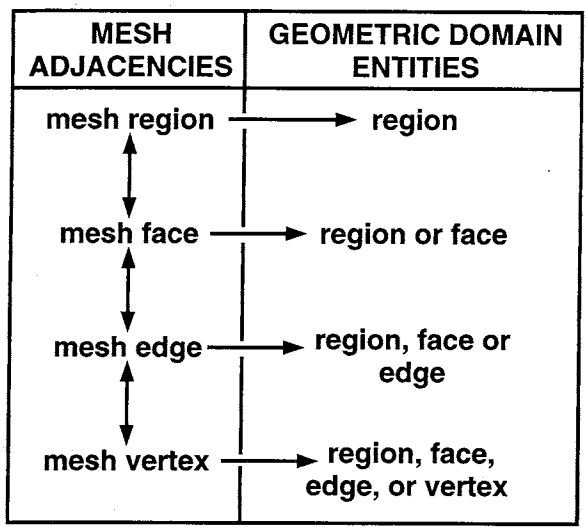


Figure 1. Mesh topological adjacencies and classification information

Since there are natural orderings for several of the adjacencies which prove useful to the operations performed, the forms of adjacencies employed are: an unordered list of n entities adjacent to entity ϑ signified by $\vartheta\{{}_vT^d\}^{(n)}$, a linear list of n entities adjacent to entity ϑ signified by $\vartheta\lfloor{}_vT^d\rfloor^{(n)}$, and a cyclic list of n entities adjacent to entity ϑ signified by $\vartheta[{}_vT^d]^{(n)}$. Specific entities also store directional knowledge of how that entity is used in the specific adjacency. In these cases the left superscript, $\pm$, on the entity, ${}^{\pm}_vT^d_i$, indicates a directional use of the topological entity ${}_vT^d_i$ as defined by its ordered definition in terms of lower order entities. A + indicates use in the same direction, while a $-$ indicates a use in the opposite direction.

The specific downward adjacencies stored are:
For mesh regions

$$ {}_mT^3_i\{{}^{\pm}_mT^2\}^{(n)} \tag{4} $$

which indicates the faces bounding the mesh region, where $n = 4$ for a tetrahedron, $n = 6$ for a hexahedron, etc.

[3] A conforming mesh is one where all mesh entities exactly match. For example, a situation where the mesh edge bounding one mesh face has two mesh edges from another mesh face lying exactly on top of it is not allowed. Although possible to extend the procedures presented here to support those situations, they will not be considered in the present document.

For mesh faces

$$ {}_mT^2_i[{}^{\pm}_mT^1]^{(n)} \tag{5} $$

which defines the loop of edges that bound the face, where $n = 3$ for a triangular face and , $n = 4$ for a quadrilateral face.
For mesh edges

$$ {}_mT^1_i\lfloor{}_mT^0\rfloor^{(2)} \tag{6} $$

which indicates the two vertices that bound the edge.
The specific upward adjacencies stored are:
For mesh vertices

$$ {}_mT^0_i\{{}_mT^1\}^{(n)} \tag{7} $$

which indicates the edges the vertex is on the boundary of.
For mesh edges

$$ {}_mT^1_i\{{}_mT^2\}^{(n)} \tag{8} $$

which indicates the faces the edge partly bounds.
For mesh faces

$$ {}_mT^2_i\lfloor{}_mT^3\rfloor^{(2)} \tag{9} $$

which indicates the zero, one, or two regions the face partly bounds.
An alternative set of adjacencies which can directly meet the needs of many applications is to maintain the same downward adjacencies and store only the single upward adjacency from the vertices to the highest order entities using them. In the case of a manifold mesh in 3–D this upward adjacency would be

$$ {}_mT^0_i\{T^3\} \tag{10} $$

which are the regions that the vertex bounds. In the case of general non-manifold models, it is the upward adjacencies form the vertices to any mesh entity it bounds which itself is not bounded by a higher order entity. In this case the adjacency relationship is a bit more complex being

$$ {}_mT^0_i\Big\{{}_mT^3, {}_mT^2, {}_mT^1 \;\Big|\; |{}_mT^2\lfloor{}_mT^3\rfloor| = 0, \; |{}_mT^1\{{}_mT^2\}| = 0\Big\} \tag{11} $$

This set includes the regions the vertex bounds, the faces the vertex bounds which do not bound any regions, and the edges the vertex bonds which do not bound any faces.

2.2. Partition Communication and Mesh Migration

Adaptive unstructured meshes on distributed memory computers require data structures which provide efficient queries for various entity and processor adjacency information as well as fast updates for changes in the mesh. The requirements for sequential implementations of hp-adaptive finite element methods can be satisfied by the SCOREC mesh database just given. For parallel applications, we first enumerate the major requirements of a distributed memory mesh environment. These requirements are met by the distributed mesh environment Parallel Mesh Database (PMDB) that is then described.

The `Tiling` system developed by Devine [18] is the first distributed environment to support hp-adaptive analysis and provides migration routines for regularly structured two dimensional meshes which can be hierarchically refined. Each tiling element stores pointers to neighboring four elements with partition boundary elements pointing to a ghost-element data which acts as a buffer during communication. The elements are assigned a unique id at the beginning and after refinements. The elements with unique ids are maintained in a balanced AVL tree [68] to allow efficient insertion and deletion during migration. The `Tiling` system supports only rectangular elements as the basic entity and the notion of shared entities like edges is implicit.

2.2.2 Distributed Mesh Model and Notation Used

The distributed mesh is viewed analogous to the modeling of non-manifold geometric objects. Figure 3 shows the hierarchical classification of the global mesh entities ${}_mT_i^d$, the processor model entities ${}_pT_i^{d'}$ and geometric model entities ${}_gT_i^{d''}$. Given the set of mesh entities $\{{}_mT\}$, a partitioning at the d_m dimension level divides the mesh into n_p parts, ${}_pT_{p_k}^{d_m}$, each of which is assigned to a processor with id $p_k = 0, \ldots, n_p - 1$. As a result of partitioning, some of the entities with dimension $d < d_m$ will be *shared* by more than one processor. The d_m-dimensional entity will be held by only one processor. Hence in general, partitioning with $d_m > 0$ defines a one-to-many relation from a mesh entity ${}_mT_i^d$ to its uses ${}_m^kT_i^d$ where $k \leq min(\Delta({}_mT_i^d), n_p)$. Here Δ defines the degree of an entity, i.e. given the dimension d of an entity, Δ is the number of $d+1$ dimensional entities which use it.

Since the procedures in a distributed memory environment operate on private local processor address space, we refer to each entity use ${}_m^kT_i^d$ in the global model as ${}_m^{(p_k,a_k)}T_i^d$ or in shorthand notation (p_k, a_k). The tuple (p_k, a_k) stands for the use of an entity by processor p_k at local address a_k. In the algorithm descriptions presented later this tuple is also called a *link* particularly if it is stored on a different processor than p_k.

For the implementation of owner computes paradigm, one of the processors holding a given entity ${}_mT_i^d$ is designated as the owner of that entity. In the distributed processor address space, we distinguish the owned entities as (p_o, a_o). Therefore, a partitioning in this case defines a one-to-one and onto mapping of global mesh entities onto the owned distributed mesh entities: Note that the inverse of this mapping exists and hence the pair (p_o, a_o) can serve as a global key of a distributed entity.

The uses of the shared entities are mapped onto the owner entity by a many-to-one relation :

$$\Phi : (p_k, a_k) \mapsto (p_o, a_o) \tag{12}$$

Figure 3 shows the relationship between the geometric model entities ${}_gT_i^{d''}$, the global mesh entities ${}_mT_i^d$ and the processor model. Given the uses (p_k, a_k) of an entity distributed over processors p_k, an agreement can be reached among these processors on whether they hold the identical entity by computing the ownership using the function Φ.

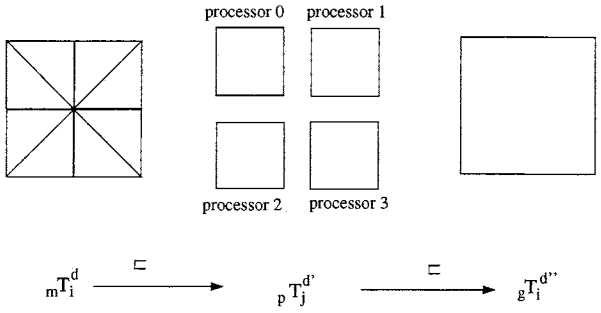


Figure 3. The relationship between the mesh model, processor model and the geometric model

2.2.3 Data Structures

PMDB data structures were designed to provide full variety of adjacency information. At the micro level of a partition boundary entity, one should be able to get all the uses or links of an entity on other processors. Each partition boundary entity stores all the uses on other processors as a linked list. This is shown in Figure 4. Note that one of the processors holding a shared entity is marked as an owner of that entity. The bold edges and vertices indicate the owners of the shared entities. This ownership information can be used in the implementation of the owner computes rule, for example, during link updates in mesh migration or scalar product computation in an iterative linear solver.

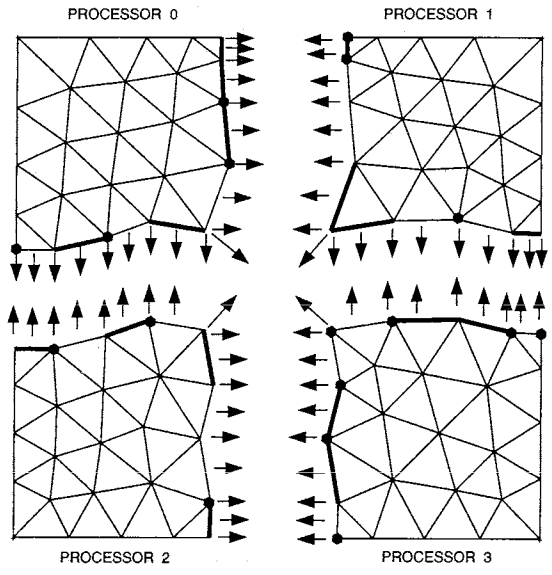


Figure 4. PMDB inter processor links and entity ownership

Since each processor stores the uses (p_k, a_k) on all the processors that hold a shared entity, the ownership can be computed as a function of these uses. An example of an ownership function Φ given in equation 12 is to choose the processor which has the tuple (p_k, a_k)

procedure mesh_migrate (P_s,$\{{}_mT^{d_i}_{s_i}\}$,P_r,$\{{}_mT^{d_i}_{r_i}\}$)
input: P_s: destination processors.
$\{{}_mT^{d_i}_{s_i}\}$: sets of regions to be migrated
output: P_r: source processors
$\{{}_mT^{d_i}_{r_i}\}$: sets of regions received
begin
/* 1. senders and receivers to owners */
1 Pack the mesh $\{{}_mT^{d_i}_{s_i}\}$ to be sent.
2 Find the migrated boundary.
3 Delete migrated internal entities
4 Pack the owners' uses corresponding to migrated boundary
5 Send packed submeshes and uses to P_{s_i}
6 Receive packed submeshes and uses from P_{r_i}
7 Unpack the submeshes to get $\{{}_mT^{d_i}_{r_i}\}$

/* 2. senders and receivers to owners */
8 Establish usage of both sent and received migrated boundary entities.
9 Pack local uses of migrated boundary and owners uses to be sent to owner processors P_o
10 Send packed local and owner uses to owner processors.
11 Receive packed uses from senders and receivers.

/* 3 owners to affected */
12 Owners update use lists by inserting/deleting received local uses into/from use lists pointed to by owner uses and generate new ownerships.
13 Pack updated uses list of entities to be sent to affected processors P_a.
14 Send updated use lists and ownership to owner processors.
15 Receive updated uses list and ownership from owner processors.
16 P_a update use lists and ownership.
17 Delete unused sent migrated boundary entities.
end

Figure 6. Mesh Migration Algorithm

Once the packed submesh has been received, the processors unpack it and insert it into the mesh $p_k\{{}_mT\}$ held by the processor p_k. It is also possible that when more than one submesh arrives from different processors, they all might share some common entities. Figure 7 shows an example of such a case. As shown processors 0 and 2 both migrate to processor 1. Among the migrated entities are those which are shared by both 0 and 2. In such a case, these commonly shared entities, once unpacked, should not be unpacked for the subsequent received submeshes which also contain them and comes from a different processor. This process is achieved by inserting the unpacked migrated boundary entities into a red-black tree [68] which has guaranteed logarithmic access for each inserted entity. A key is needed to represent the entity in the red-black tree. This key can be either a global key or the readily available (p_o, a_o) tuple which was discussed earlier. Currently, PMDB version 3.1 by default generates global numbers after mesh is refined. The global numbers can be used for debugging and also provides a readily available equation number for linear equation solvers which assemble the global matrix. A future version of PMDB will make the global number generation optional in order to save memory for applications which do not need it.

Senders and Receivers to Owners: These steps operate only on the sent and received migrated boundary entities. These entities are tested to see if they are used by ${}_pT$ on processor p. Determining the use on processor p of a $d-$dimensional entity requires determining if that entity is part of the boundary of a $d+1$ dimensional entity on processor p. The entity hierarchy data structures of SCOREC mesh database readily provide this d to $d+1$ dimensional entity adjacency relationship. If the entity is used, its use (p, a) is packed and identified by the (p_o, a_o) use to be sent to owner processor. If the entity is not used $(p, null)$ is packed. Once packed, this information is sent to the owner processors. The overall complexity of these steps is proportional to the size of the sent and received migrated boundaries.

Owners to Affected Processors: Owners receive updates targeting a particular entity (p_o, a_o) it owns. If a use (p, a) is received, it is inserted in the list of uses of the shared entity at address a_o. If $(p, null)$ is received, the use (p, a) is deleted from the list of uses at address a_o. Once all the updates are completed, the ownership of these entities are regenerated. The updated links are then packed and sent to the affected processors. The affected processors receive these uses and update the corresponding local shared entities' list of uses. At this point, the migrated boundary entities can be deleted and mesh migration completes.

Computing Number of Receives: The steps $5-6, 10-11$ and $14-15$ implement non-blocking sends and receives. Each processor needs to know how many messages are being sent to it by other processors so that it can post a corresponding number of receive statements. A simple way to compute the number of receives is by first having each processor initialize a vector r of length n_p and to set r_p to 1 if a message will be sent to processor p and 0 otherwise. A follow-up sum scan operation can then be executed by all the processors resulting in each location r_p containing the number of receives. This procedure has $O(n_p log n_p)$ run time complexity and requires a message of length n_p to be communicated during the combine operation. Whereas this scheme will be efficient for small n_p, it is nevertheless non-scalable. The DIME environment, for example, makes use of the crystal_router [24] which provides a scheme for this problem by utilizing $log(n_p)$ message exchanges across the dimensions of the hypercube multiprocessors.

Considering the fact that each processor p usually sends to a small number s_p of processors, a scalable strategy is desirable for large n_p. We can make this scheme scalable by making use of the radix sort routine [7]. Since the processor ids are in the range $0, \ldots, n_p - 1$,

2.2.5 Scalability of Mesh Migration and Extensions

In the mesh migration procedure presented above, the amount of communication involved is proportional to the volume of submeshes in the first stage of the algorithm and to the surface of submeshes during link updates in the second and third stages. As a result, if each processor migrates to a small number of processors, such as its neighbors, then we expect that the migration will scale as the number of processors is increased. Various tests have been performed to demonstrate scalability of migration. The data involving the maximum number of regions migrated by a processor, the total number of regions migrated by all processors, the time taken, and the throughput, that is, the number of regions sent by a single processor per second are plotted against the number of processors used.

Test 1: In the first test, we let each processor exchange a slice on its partition boundary with its neighbors. This test is a realistic representative of the migration patterns that occur in iterative dynamic load balancers since regions near partition boundaries are migrated in clusters to the neighborhood of a heavily loaded processor. Another application that performs this kind of migration is mesh coarsening [10]. Figure 8 shows the example mesh that was used before (a) and after migration (b). Figure 9(a) plots the maximum number of regions sent by a processor and (b) shows the wall time taken. From these plots, we see that execution time is proportional to the number of regions sent irrespective of the number of processors.

Figure 10 on the other hand plots the total number of regions sent by all processors. As the number of processors are increased the total number of regions at partition boundaries increases. Hence even though overall more regions have been moved, the time is proportional to the maximum sent by a single processor. This behavior demonstrates that when processors migrate to a small number of neighbors, the migration procedure scales well. Figure 10(b) plots the throughput attained.

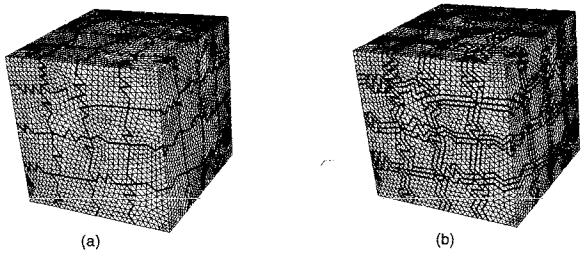


Figure 8. Neighborhood migration test; before boundary exchange (a), after boundary exchange (b)

Test 2: In the second test, we let each processor hold 2500 regions corresponding to a partition of the box mesh and migrate all its regions randomly targeted to s processors with $s = 1, \ldots, 2^i, \ldots, n_p - 1$. The plots of time taken for migration and the throughput per processor is shown in Figure 11. The plot in (a) shows that as the number of processors is increased, the time taken grows

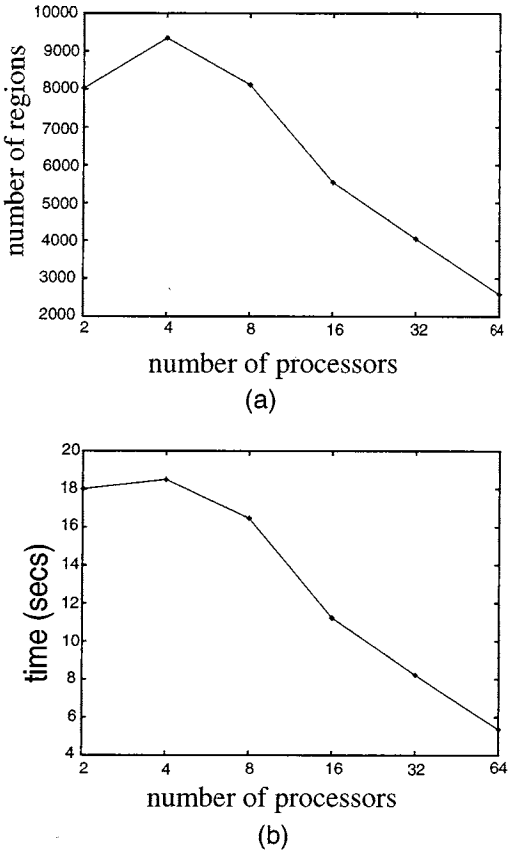


Figure 9. Neighborhood migration test for box ; maximum number of regions migrated by a processor (a), wall time (b)

slowly. In particular, if we look at the $s = 1$ case, we see a flat curve between 32 and 64 processors. The number of processors has been doubled, yet the execution time remains the same. As s is increased the execution time growth is larger as expected, since the number of total migrations is increased. In particular, if $s = n_p - 1$, we have all-to-all migration. Note that, there is a pronounced drop in the throughput as shown in Figure 11(b) between the cases $s = 1$ and 2. For example, with $n_p = 48$, the throughput is 519 regions for $s = 1$ and drops to 309 regions at $s = 2$. The major cause of this drop is not the mere increase in s, but rather the fact that when regions are assigned random destination, the union of the migrated boundary of the mesh entities being sent becomes proportional to the number of regions sent. In the case of $s = 1$, the migrated boundary is proportional to the surface of the mesh entities sent. As a result, since the cost of stages 2 and 3 of the mesh migration algorithm is dependent on the size of migrated boundary, these stages contribute greatly to the drop in cases $s > 1$. The sets of regions which are migrated in practice are clustered locally and hence the migrated boundary size is rarely proportional to the volume being sent. Therefore, higher throughput rates can be attained for larger s as is evident from Test 1 above.

This section discussed the data structures and the migration routines used in the PMDB library. PMDB library cur-

and conquer approach to pair processors and uses connectivity as well as coordinate information to decide which elements to migrate.

A disadvantage of the common implementations RB methods is they start with the entire mesh on a single processor and partition from there. Two problems with this approach in a parallel adaptive calculation are (i) the time required to gather the distributed mesh together on a single processor, and (ii) the fact that after the mesh has been adapted, it may have grown to the point that it can not fit on a single processor. These problems can be alleviated it the mesh remains distributed during the repartitioning process. The next subsection discusses a parallel implementation of *Inertial Recursive Bisection* that operates on a distributed mesh.

RB methods operate on the whole mesh and compute the direct destination for each element. Because of this, it is possible that RB methods may require complete remapping of the elements at the end. On the other hand, iterative local migration techniques propagate the excess load by local transfers to other processors. A disadvantage of iterative local migration techniques is that many iterations may be required to regain global balance and hence elements reach their final destination after many local transfers rather than directly. In particular, when elements are migrated, the full element data involving connectivity and local attached data are communicated. For parallel repartitioners based on coordinate bisection, only the centroids and region pointers need to be communicated during a parallel sorting phase. As a result this class of repartitioners may have better performance on machines in which the communication between any pair of processors is distance-independent.

Subsection 2.3.2 presents an iterative load balancing procedure based on the Leiss/Reddy heuristic of requesting load from the most heavily loaded neighbor. The performance of this procedure is compared with repartitioning by the parallel distributed inertia recursive bisection algorithm.

2.3.1 Geometry-Based Dynamic Balancing Procedures

Geometry-based dynamic balancing (or repartitioning) relies here on the Inertial Recursive Bisection (IRB) method [50] which is a variation of the more classic Orthogonal Recursive Bisection (ORB) [4]. ORB is a recursive process that bisects a set of entities by considering the median of the set of corresponding centroids with respect to a given coordinate axis. As ORB is recursively called, the choice of coordinate axis is circularly permuted (x,y,z,x, etc). Unlike ORB, IRB considers the inertial coordinate system (origin is at the center of gravity and the three axes are the principal axes of inertia) for the set of entities to be bisected. In three dimensions, the determination of the three principal axes of inertia is an eigenvalue problem of order 3. Once the inertial coordinate system is defined, the coordinates of the centroids are transformed and the cut is made at the median with respect to the first coordinate. This first coordinate is the "key" that the sorting algorithm described later in this section works on.

The main assumption for performing repartitioning in parallel is that the entities are distributed. It is also assumed that there is no reason for the number of entities stored on processor to be uniform across processors. The result of this repartitioning will be an equal number of entities per processor. It should be noted that, in this context, the goal of repartitioning is equivalent to the goal of dynamic load balancing [15, 55, 73, 54, 43, 85]. The key algorithm in IRB (and ORB) is the determination of the median for a given set of doubles (referred to as "keys") [68]. With respect to this paper, the "keys" are the first coordinates, in the inertial frame, of the entities to be bisected. The method used here is to sort the "keys" and then pick the entry at the middle of the sorted list. In this case, efficiently performing IRB in parallel can be reduced to efficiently sorting in parallel [34]. From the conclusions of the paper by Blelloch et al. [8] which compares different parallel sorting algorithms (Batcher's bitonic sort, radix sort, and sample sort), it appears that the sample sort algorithm is the fastest of the three for large data sets. Therefore, a parallel sample sort algorithm has been implemented in order to efficiently support IRB.

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

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

The goal of step 1 is to split the set of "keys" into p parts (buckets) as evenly as possible and as efficiently as possible. The $p-1$ splitters which are implicitly sorted (say with respect to increasing value) are labeled from 1 to $p-1$. All distributed "keys" below splitter 1 belong to bucket 0, all distributed "keys" between splitter i ($0 < i < p-1$) and splitter $i+1$ belong to bucket i, and all distributed "keys" above splitter $p-1$ belong to bucket $p-1$. Processor i ($0 \le i < p$) is responsible for the bucket labeled i. In step 2, assuming the $p-1$ splitters have been found and broadcasted to all processors, any distributed "key" can tell in which bucket it belongs and is rerouted to the processor that is responsible for that bucket. At this point, any processor has knowledge of all "keys" that belong to the bucket it has been assigned to. Step 3 can be performed using any efficient sequential sorting algorithm, like quicksort [68]. It is clear that the parallel efficiency of the sample sort algorithm depends on the sizes of the buckets. Parallel efficiency is maximal when the sizes of the buckets are near equal. A sampling method is used to obtain "good" splitters. Given the n input "keys", ps "keys" (s is an integer ≥ 1 called the over sampling ratio) are selected at random and sorted typically sequentially. The entries in the sorted list of ranks s, $2s$, ... , $(p-1)s$ are the $p-1$ splitters. The bound for bucket expansion (ratio of maximum bucket size to average) is

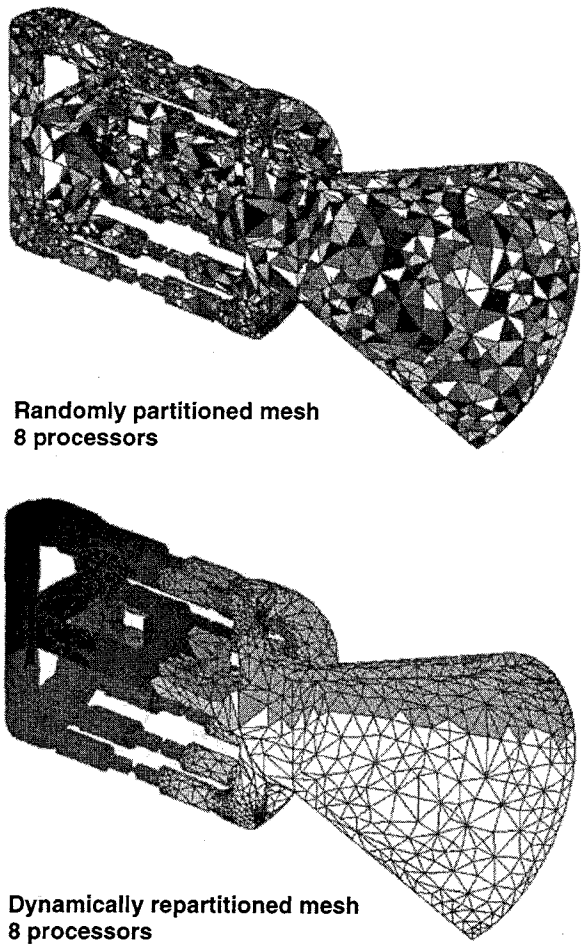


Figure 13. Dynamic repartitioning on a randomly distributed mesh

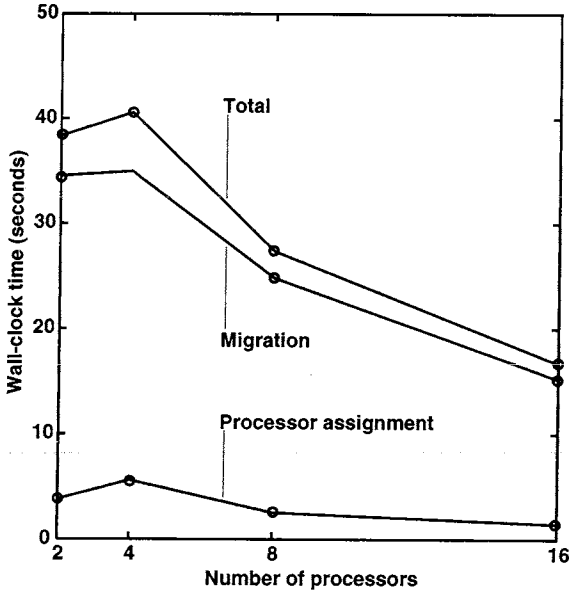


Figure 14. Timings for dynamic repartitioning

To incorporate more global information and to direct load transfers, we view the processor requests for load from heavily loaded processors as forming a forest of trees. Figure 15(a) shows an example of requests that can be formed. Given this hierarchical arrangement of processors as the nodes of trees, we balance the trees as shown in Figure 15(b) and iteratively repeat the process until the load distribution converges to optimal load balance within a user supplied tolerance. The full algorithm is given in Figure 16. The procedure details are given as follows.

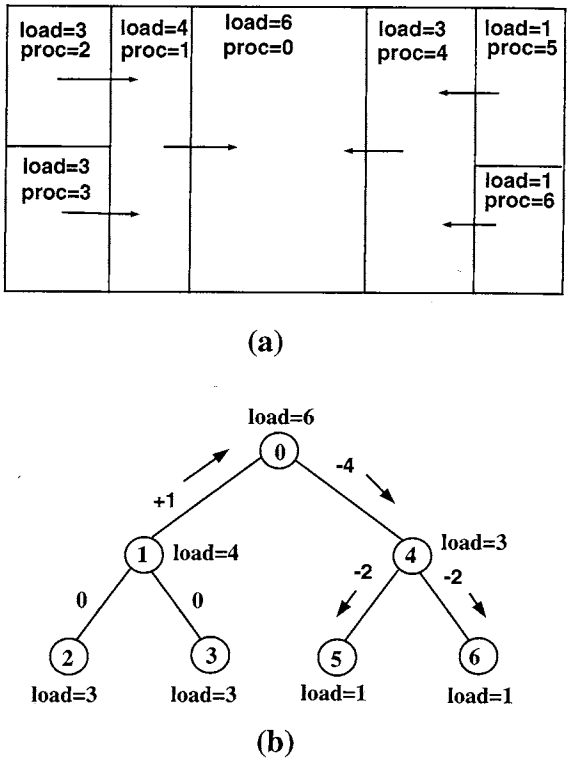


Figure 15. Load balancing example; load request (a) load migration on the tree (b)

procedure `tree_load_balance`(tol_{load}, max_{iter})
in tol_{load}imbalance load tolerance
in max_{iter} : maximum number of iterations
begin
1 iter = 0
2 **while** (max. load difference $> tol_{load}$) **and** (iter $< max_{iter}$) **do**
3 iter = iter + 1
4 Compute neighboring load differences.
5 Request load from neighbor processor having largest load difference (creates processor trees).
6 Linearize processor trees.
7 Compute amounts of load migration.
8 Select and migrate load.
9 **endwhile**
end

Figure 16. Tree based dynamic load balancing procedure

Steps of the procedure The steps of balancing the forest of trees are repeated until convergence is achieved. Assuming that load transfer occurs when there is a load difference of at least two units, Leiss/Reddy's algorithm

high frequency of load imbalances, the load balancer will have better performance. The repartitioner bypasses the effects of distance by directly sending load from heavily loaded to lightly loaded processors. On an architecture such as the IBM-SP2, in which communication cost is independent of the distance between the processors and hence the same between any pair of processors, the repartitioner will be advantageous since it directly sends the load to its final destination. The load balancer will be disadvantageous since it will incur expensive latency cost during many local transfers it performs.

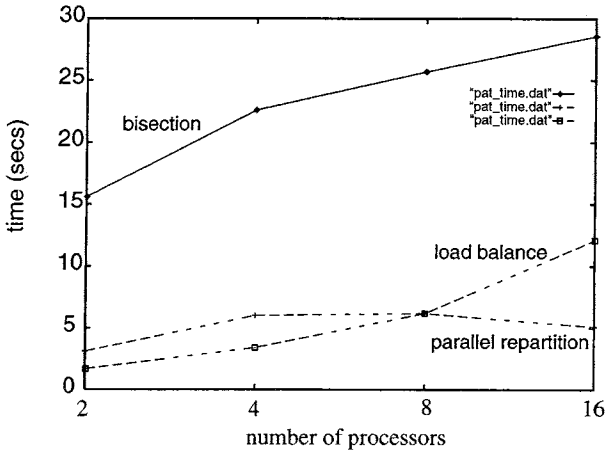


Figure 19. Time taken for load balance, parallel repartitioning and bisection

Finally, Figure 20 shows the quality of the partitions produced in terms of maximum and total percentage of faces cut. The load balancer's element selection criteria for migration dictates the quality of the partitions. The criteria currently used can be improved by incorporating coordinate information to selection decision.

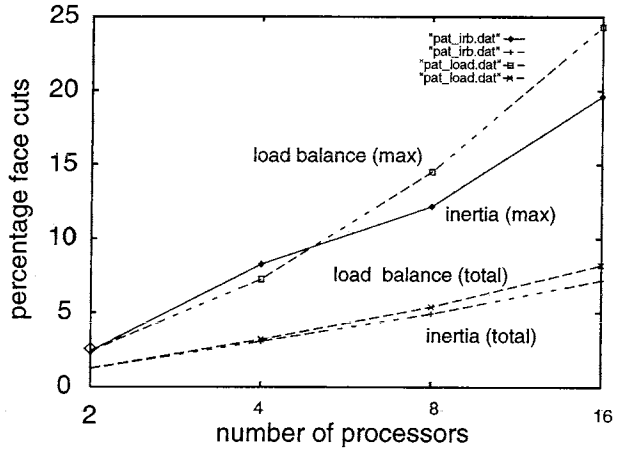


Figure 20. The total and maximum number of cut faces for each method

Test 2: In this test, various statistics are reported for the adaptively refined `onera-m6` wing mesh during an actual CFD analysis on 32 processors. At the beginning the mesh has 85567 tetrahedrons. Three stages of adaptive refinements are performed during which the number of tetrahedrons increase to 131000, 223501 and finally to 388837. Figure 21 shows the convergence history of the iterative load balancer. In all cases of load balancing after refinement, the imbalance reduces to less than 4% during the first 8 iterations and takes far more number of iterations to reduce this imbalance further to 0% imbalance. One need not run the `tree_balance` to full convergence. It can be stopped when a reasonable imbalance is achieved.

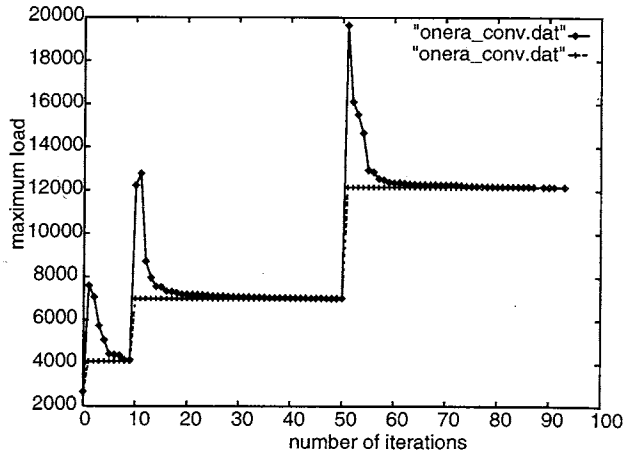


Figure 21. The convergence history for load balance during three stages of refinement

Table 2 shows the execution time comparisons between the `tree_load_balance` and the parallel moment of inertia partitioner. In all cases, moment of inertia outperforms `tree_load_balance` for the same reasons which was explained in Test 1.

refinement	1st		2nd		3rd	
percent imbalance	1.7	0	3.8	0	1.9	0
tree_balance (sec)	73	85	127	210	189	283
inertia partition (sec)		21		48		128

Table 2 Execution times (in seconds) for `tree_load_ balance` and inertia partitioner

Finally, Table 3 shows partition quality comparisons between the `tree_load_balance` and the moment of inertia partitioner. The percentage of maximum number and the total number of cut faces are given for both `tree_load_balance` and the inertia partitioner.

refinement	1st		2nd		3rd	
percent cuts	max	total	max	total	max	total
tree_balance	24	10	26	11	25	10
inertia partition	26	7	25	7	19	6

Table 3 Maximum and total percentage of cut faces for `tree_load_balance` and inertia partitioner

or more edges is enforced during this process to control smoothness of the mesh gradations. Once the octree is generated, the octants are classified as *interior*, *outside*, or *boundary*. Those classified as *outside* receive no further consideration. Some *interior* octants are reclassified *boundary* if they are too close to mesh entities classified on the boundary of the model region (*boundary-interior*). The purpose of this reclassification is to avoid the complexities caused when *interior* octant mesh entities (coming from the application of templates) are too close to the boundary and may lead to the creation of poorly shaped elements in that neighborhood. *Interior* octants are meshed using templates. Face removal procedures are then used to connect the boundary triangulation to the interior octants. Figure 23 graphically describes a face removal in a two-dimensional setting.

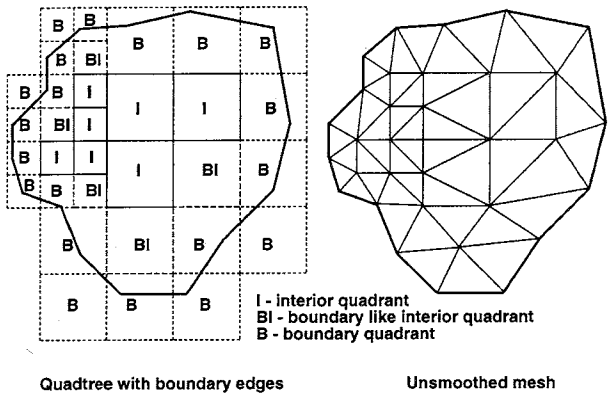


Figure 22. Graphical depiction of the basics of the presented mesh generator

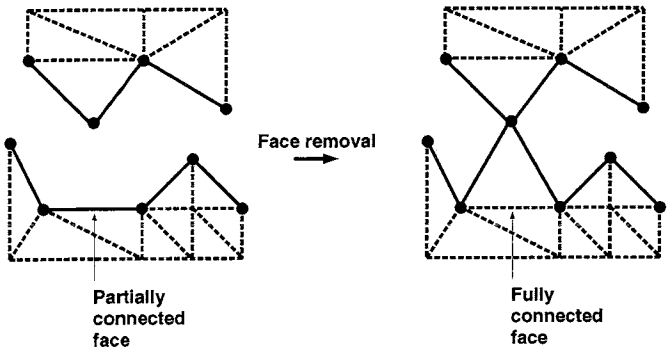


Figure 23. Face removal (2–D setting)

3.3. Sequential Region Meshing

As indicated above, the starting point for the region meshing process is a completely triangulated surface. The surface triangulation must satisfy the conditions of topological compatibility and geometric similarity [67] with respect to the model faces. The region meshing process consists of the three steps of (i) generation of the underlying octree, (ii) template meshing of interior octants, and (iii) face removal to connect the given surface triangulation to the *interior* octants.

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

3.3.1 Underlying Octree

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

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

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

$$octlev = \log_2 \left(\frac{rootlength}{size} \right) \qquad (16)$$

where $rootlength$ is the length of the root octant and $size$ is the size of the mesh entity (defined here as the average length of the bounding edges). It should be noted that this procedure does not theoretically ensure a match in size between every terminal octant and the partially connected mesh faces it knows about.

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

4. **for** each potential target vertex *vert* {
   a. Perform preliminary check on acceptability. If not acceptable, **continue**
   b. If the new element contains any mesh vertex belonging to the front, **continue**
   c. If the new element intersects any existing mesh entity, **continue**
   d. Evaluate how close the new element is to existing mesh entities (compute relative minimum distance *min_dist*)
   e. **if** (*min_dist* ≥ *dist_lim*) {
      - *target_vert* = *vert*
      - *max_min_dist*= *min_dist*
      - **break**

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

      }

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

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

   }

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

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

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

Figure 25 shows potential target vertices of type 1, 2, and 3 for the face to remove.

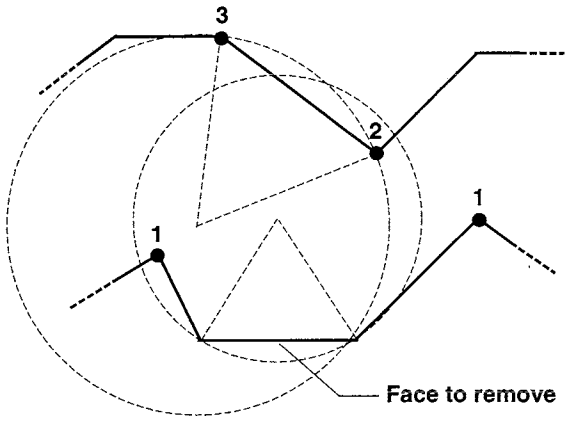


Figure 25. The three types of potential target vertices (2–d setting)

Given a potential target vertex, one has to make sure that any new mesh entity (resulting from the creation of the new mesh region) does not intersect an existing mesh entity. The creation of a new mesh region may result in the creation of a new mesh vertex, up to three new mesh edges, and up to three new mesh faces. New mesh edges are checked for intersection against nearby partially connected mesh faces. Given a virtual new mesh edge, the nearby partially connected mesh faces are obtained through the tree neighborhood of order 0 (of the new edge). If no intersection is detected, new mesh faces are checked for intersection against nearby front mesh edges. Given a virtual new mesh face, nearby front mesh edges

request, which certainly can degrade the overall performance if not done carefully. An easy solution is to make sure that all processors participate in the request (soft synchronization). On a sequential machine, performing tree traversals to obtain neighboring information, typically, getting all terminal octants that neighbor an octant entity (face, edge, or corner) can be avoided if octant face neighboring terminal octants are stored. The limited increase in data storage is well worth the constant time complexity for getting neighboring information. In a parallel setting, it is difficult to conceive such a scheme without having to communicate between processors.

3.4.2 Multiple Octant Migration

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

Any processor can send any number of terminal octants to another processor. When a terminal octant is migrated from one processor to another, the partially connected mesh faces not connected to any mesh region (these are the mesh faces remaining from the given surface triangulation) owned by the octant and/or the mesh regions that are bounded by at least one partially connected mesh face owned by the octant are migrated as well. An octant owns a mesh entity when it knows about it (has it within its volume) and has its centroid within its volume. Note that a partially connected mesh face not known by the octant may be migrated as part of a mesh region if that region is bounded by another partially connected mesh face whose owner is the octant. Also, if a mesh region is bounded by more than one partially connected mesh face known to the octant to be migrated (up to four), the ownership is arbitrarily dictated by the first partially connected mesh face to be processed (from the list of partially connected mesh faces known to the octant). Figure 28 shows a two-dimensional example of the mesh regions to be migrated within an octant. When the multiple octant migration completes, the processor is informed of the octants it has received. For each received octant, a list of associated mesh entities is also given, basically the partially connected mesh faces and/or mesh regions that were sent.

The primary complexity that arises when migrating octants and associated mesh information is the absence of a global labeling system for the mesh entities. Each processor employs a local labeling for the hierarchy of mesh entities that it is assigned. The interprocessor mesh adjacency links maintain the required knowledge of the adjacent mesh entities on neighboring processors. Although the mesh data for a partially connected face is on one

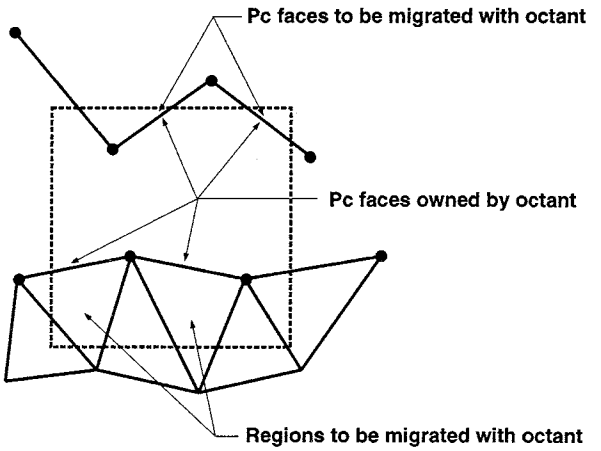


Figure 28. Octant migration

processor, the octants which refer to that face may be on multiple processors. Since the face removal procedure must perform geometric checks on all partially connected faces known to that octant, the time required to perform these operations would be greatly increased if the required information had to be fetched from neighboring processors. To eliminate this requirement, each partially connected face known to an octant will either be a pointer to face, when the face is actually on-processor, or a set of three coordinates when the face is stored off-processor. Although this approach avoids interprocessor communications, it complicates the process of updating references to partially connected mesh faces on and off-processor when octants are migrated. Concerning the update of processor assignment at the octant level, since the tree structure is currently stored on all processors, a broadcast is performed to all processors indicating the fact that octants have been relocated.

3.4.3 Dynamic Repartitioning

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

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

Repartitioning for stages 1 and 2 is done at the terminal octant level (1 with respect to terminal octants classified *interior* and 2 with respect to terminal octants classified *boundary*). Repartitioning for stage 3 is performed at the mesh region level. The strategy is identical for both cases, only the process of migrating differs. The methods used here are geometry-based dynamic balancing (repartitioning) procedures which are described in section 2.3.1.

3.5. Parallel Region Meshing

3.5.1 Underlying Octree

At this point in time, the octree is built sequentially on a single processor (processor 0). Since a sequential octree building can become a bottleneck when dealing with

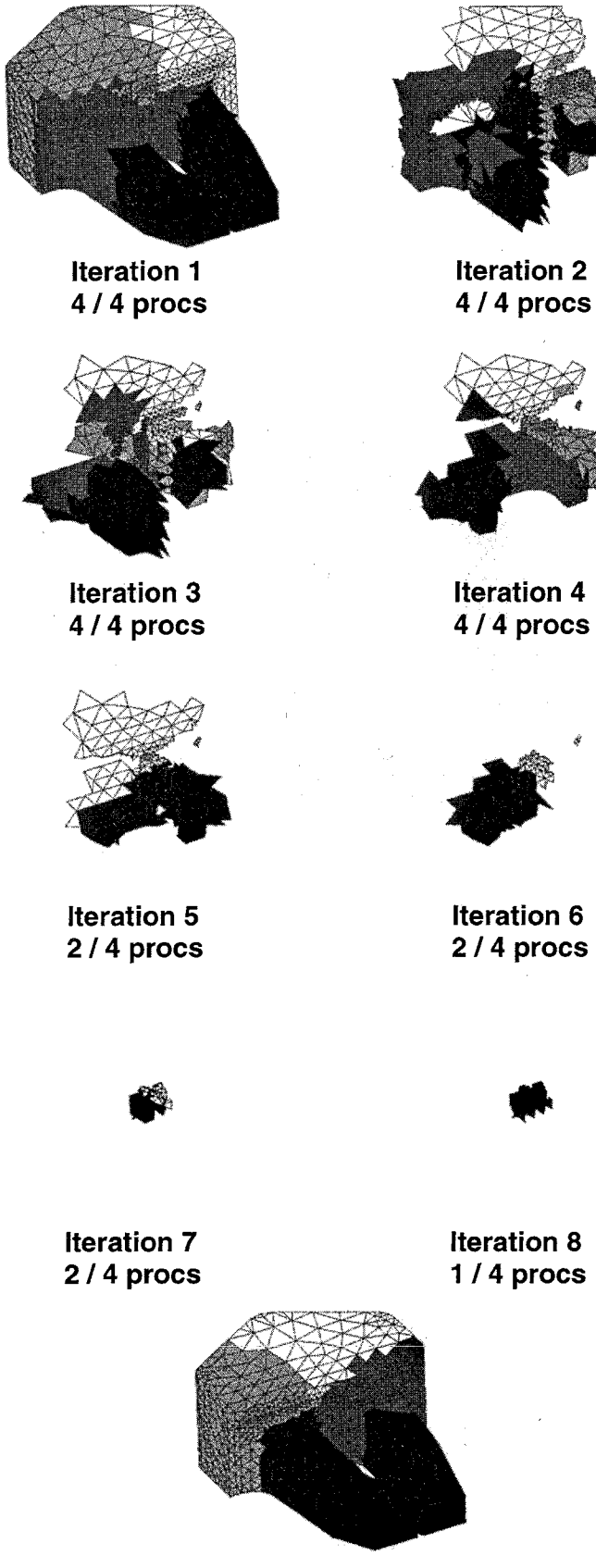


Figure 30. Successive face removal iterations and final repartitioned mesh for *chicklet*

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

Table 4 Face removal statistics for *connecting rod* (35,000 mesh regions created by face removals — 70,000 total)

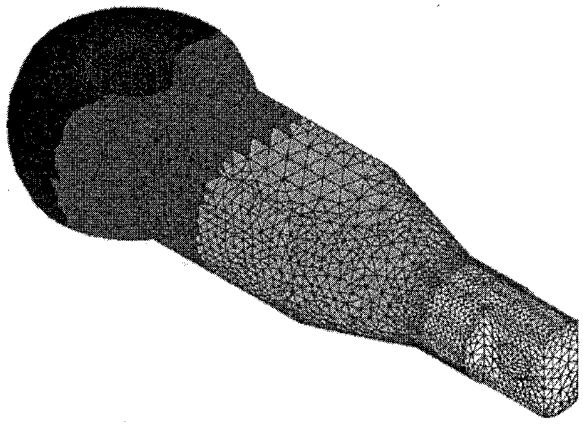


Figure 31. Final repartitioned mesh for *connecting rod* (4 processors)

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

Table 5 Face removal statistics for *blade* (60,000 mesh regions created by face removals — 90,000 total)

not include template meshing. Face removal speed-up indicates speed-up for step 2 of the parallel face removal procedure. Total speed-up indicates speed-up for all steps (1, 2, and 3). In that case, the first repartitioning (iteration 1) is not counted since it can be considered an initial partitioning step. Note that the time taken to perform the first repartitioning depends on the size of the problem and not the number of processors. The speed-up is by definition set to 1.0 for the run with the smallest number of processors. The results show good speed-ups as long as the size of the problem is adequate with the number of processors on hand.

Figure 35. Final repartitioned mesh for *mechanical part 2* (8 processors)

A few definitions related to triangulation quality relevant to local retriangulation tools are now given:
Triangulation quality: *If each mesh entity* ${}_mT_i^{d_i}$ *of a triangulation* ${}_m\Omega$ *is associated with a quality measure* q_i, *the quality of the triangulation is defined as* $Q = \min(q_i)$
Triangulation acceptability: *Given a quality threshold* q_l, *a triangulation* ${}_m\Omega$ *is acceptable with respect to triangulation quality if* $Q > q_l$.
Triangulation comparison: *A triangulation* ${}_m\Omega_i$ *of a set of points is considered better with respect to triangulation quality than another triangulation* ${}_m\Omega_j$ *of the same set of points if* $Q_i > Q_j$.

4.1.1 Edge Swapping

In two and three dimensions, a swapping step is performed after inserting a new node into the triangulation to transform a locally non-Delaunay triangulation into a Delaunay one. Aside from the refinement issue, it is a method to incrementally build a Delaunay triangulation of a set of points.

Swapping relies on the general result given by Lawson which states that a set of $n+2$ points in R^n may be triangulated in at most two ways [42]. In two dimensions, there are two ways to triangulate a strictly convex quadrilateral. Edge swapping consists of switching diagonals for the quadrilateral resulting from the union of the two connected triangles (if convex). In three dimensions, there are two ways to triangulate a strictly convex triangular hexahedron containing five and only five points (the five apices of the triangular hexahedron). Joe provided a set of workable swappable configurations for the three-dimensional case [35]. If a mesh face ${}_mT_1^2$ is not locally optimal (does not satisfy the Delaunay criterion) and corresponds to one of the two situations on the left side of figure 36, it is swapped. If $[{}_mT_4^0, {}_mT_5^0] \cap {}_mT_1^2 \neq \emptyset$ ($[{}_mT_4^0, {}_mT_5^0]$ being the line segment spanning from ${}_mT_4^0$ to ${}_mT_5^0$), the triangular hexahedron initially containing two tetrahedra is retriangulated with three. If $[{}_mT_4^0, {}_mT_5^0] \cap (S_1 \cup S_2 \cup S_3) \neq \emptyset$ (where the S_i's are plane sectors appearing shaded in figure 36) and $\exists\ {}_mT_1^3 \mid \{{}_mT_2^0, {}_mT_3^0, {}_mT_4^0, {}_mT_5^0\} \in \partial({}_mT_1^3)$, the triangular hexahedron initially containing three tetrahedra is retriangulated with two.

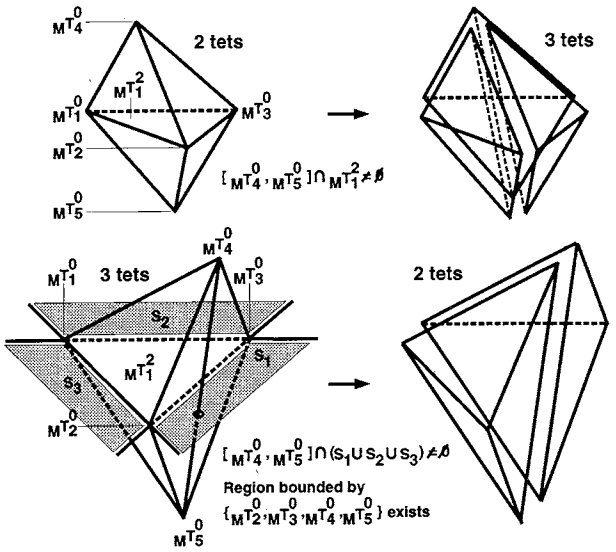


Figure 36. 2–to-3 and 3–to-2 swaps in three dimensions

These swaps, commonly referred to as 2–to-3 and 3–to-2, are suited for Delaunay triangulations and by extension for regular triangulations [19]. Referring to Fig. 36, if $[{}_mT_4^0, {}_mT_5^0] \cap (S_1 \cup S_2 \cup S_3) \neq \emptyset$ and $\forall {}_mT_i^3\ \{{}_mT_2^0, {}_mT_3^0, {}_mT_4^0, {}_mT_5^0\} \notin \partial({}_mT_i^3)$ or $[{}_mT_4^0, {}_mT_5^0] \cap ({}_mT_1^2 \cup S_1 \cup S_2 \cup S_3) = \emptyset$, there is no possible swap. When dealing with Delaunay triangulations (or regular triangulations), theoretical results indicate that non swappable faces (in Joe's sense) are not critical. However, when dealing with any other criterion, non swappable faces (in Joe's sense) may be critical. The other non swappable configuration from figure 36 which corresponds to $[{}_mT_4^0, {}_mT_5^0] \cap ({}_mT_1^2 \cup S_1 \cup S_2 \cup S_3) = \emptyset$ consists of four tetrahedra bounded by $\{{}_mT_1^0, {}_mT_2^0, {}_mT_4^0, {}_mT_5^0\}$, $\{{}_mT_2^0, {}_mT_3^0, {}_mT_4^0, {}_mT_5^0\}$, $\{{}_mT_1^0, {}_mT_2^0, {}_mT_3^0, {}_mT_4^0\}$, and $\{{}_mT_1^0, {}_mT_2^0, {}_mT_3^0, {}_mT_5^0\}$, respectively [35]. It is clear that there is no other way of triangulating this convex hull. The ideas presented by Brière de l'Isle and George [17] about edge removal enable the extension of the classic 3–to-2 swap [35, 19].

4.1.2 Edge Removal

Brière de l'Isle and George [17] have proposed an edge removal technique as part of an algorithm to optimize the quality of a given mesh. It can also be used as part of a scheme to recover the faceted boundary of a model [28]. A mesh edge ${}_mT_1^1 \sqsubset {}_gT_1^3$ which is bounded by vertices ${}_mT_1^0$ and ${}_mT_2^0$ can be eliminated by retriangulating the polyhedron of all connected tetrahedrons. The polyhedron is retriangulated by: (i) triangulating the polygon

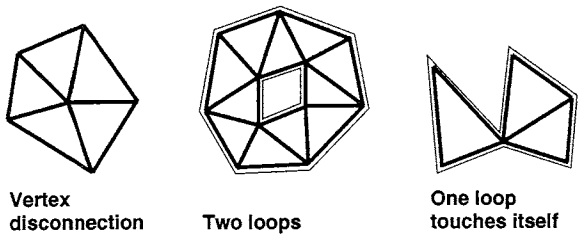


Figure 38. Cases when the topological state of the set of mesh faces $\{{}_mT_i^2\}$ prevents multi-face removal

$\{{}_mT_i^2\}$ (as well as any mesh face in the set) is said to be topologically removable.

Since the goal of the presented optimization algorithm is to get rid of undesirable mesh regions, the input to the multi-face removal procedure is a mesh region ${}_mT_1^3$ and a bounding mesh face ${}_mT_1^2$ from which the simply connected set of mesh faces is constructed. The description of the algorithm follows:

1. Get vertex ${}_mT_1^0$ opposite ${}_mT_1^2$ in ${}_mT_1^3$ (Fig 39.a)
2. Get region ${}_mT_2^3$ on other side of ${}_mT_1^2$
3. Get the vertex ${}_mT_2^0$ opposite ${}_mT_1^2$ in ${}_mT_2^3$ (Fig 39.b)
4. Gather all pairs of face-connected mesh regions such that one mesh region connects to ${}_mT_1^0$ and the other connects to ${}_mT_2^0$. Keep track of the mesh faces in-between pairs of mesh regions ($\{{}_mT_i^2\}$). The set of gathered mesh regions defines $pol(\{{}_mT_i^2\})$ (Fig 39.c)
5. If retriangulation would create invalid elements, do not perform removal
6. Compute quality of initial triangulation Q_{org}
7. Compute quality Q_{new} of triangulation that would result from connecting all boundary faces of $pol(\{{}_mT_i^2\})$ to ${}_mT_1^0$
8. If $Q_{new} < Q_{org}$, do not perform removal
9. Delete the mesh regions in $pol(\{{}_mT_i^2\})$ to form a polyhedral cavity
10. Connect all faces of polyhedral cavity to ${}_mT_1^0$ (Fig 39.d)

The initial triangulation of the polyhedron (Fig 39.c) is such that there are:

1. *m* mesh regions (note that *m* is an even number),
2. *3m/2–2* interior (with respect to the polyhedron) mesh faces, and
3. *m/2–1* interior mesh edges each connected to *4* mesh regions.

The resulting triangulation (Fig 39.d) is such that there are:

1. *m/2+2* mesh regions,
2. *m/2+2* interior mesh faces, and
3. *1* interior mesh edge connected to *m/2+2* mesh regions.

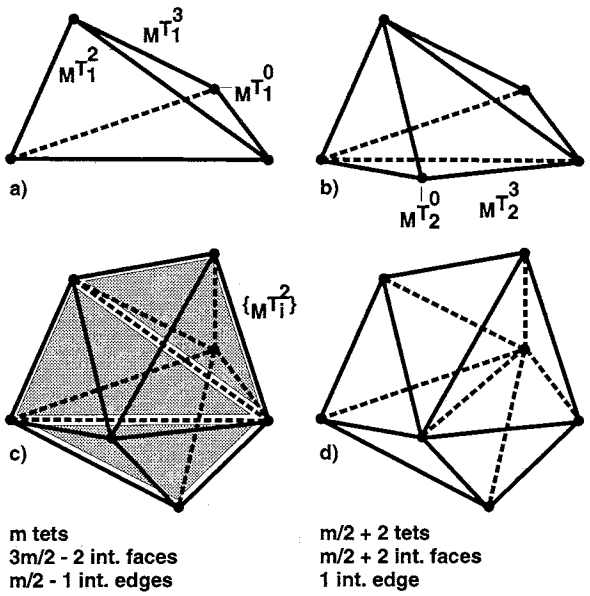


Figure 39. Multi-face removal in three dimensions

4.1.4 Triangulation Optimization Using Local Retriangulation Tools

The goal of the optimization algorithm is to improve the quality of Geometric triangulations with respect to a given criterion (e.g., element shape). The optimization procedure described here makes use of the local retriangulation tools described above, namely edge removal and multi-face removal. Other local retriangulation tools which change the number of mesh vertices like mesh entity splitting, edge collapsing, and even local remeshing are not incorporated into this specific optimization procedure. Also, smoothing techniques (vertex repositioning) [23, 14] are not addressed. In this discussion, triangulation optimization can be used over the whole triangulation or locally over a sub-triangulation resulting from adaptive enrichments such as refinement and derefinement.

The optimization procedure is region based, that is, it looks for mesh regions that are not acceptable (quality below q_l) and attempts to remove them from the triangulation with local retriangulation tools. Given a non acceptable mesh region ${}_mT_1^3$, one can potentially remove that mesh region from the triangulation by considering edge removal with respect to any of its four bounding edges or multi-face removal with respect to any of its four bounding faces. The optimization algorithm is described as follows:

1. Initialize queue *Qu* of non acceptable mesh regions (quality below q_l)
2. If (*Qu* empty) or (there is no edge removal or multi-face removal that can successfully be applied to any mesh region in *Qu*), end
3. Pop a region from *Qu*
4. Consider which edge removal (with respect to any bounding edge) or multi-face removal (with respect to any bounding face) gives the best quality improvement of the corresponding polyhedron

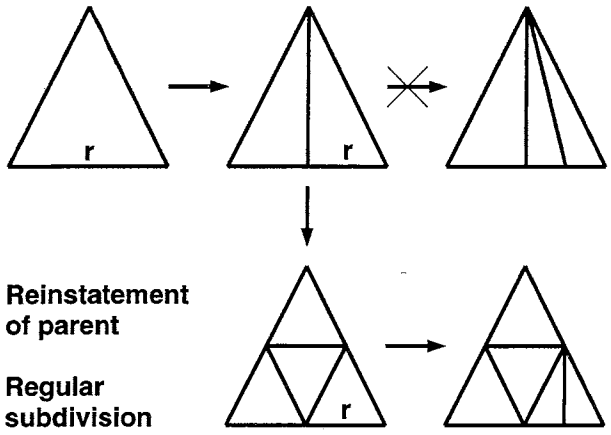


Figure 42. Reinstatement of parent element followed by regular subdivision in two dimensions

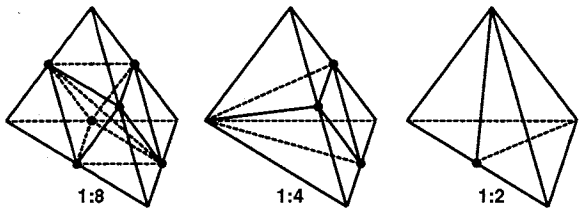


Figure 43. Classic element subdivision patterns in three dimensions

at the centroid of the element and the element is split accordingly. If a marking pattern does not correspond to a predefined configuration, it is upgraded to the closest one. The process terminates in a finite number of steps. It has been shown that the regular 1:8 subdivision scheme is stable as long as the proper (shortest) inner diagonal is chosen [29, 5]. Algorithms based on subdivision patterns are stable if irregular child elements (not resulting from regular subdivision) are never further subdivided, in other words, parents of those are reinstated and subdivided with the 1:8 subdivision scheme prior to any further subdivision [57, 48, 9]. Note that ${}_m\Omega_i$ is always nested into ${}_m\Omega_1$ but ${}_m\Omega_{i+1}$ may not be nested into ${}_m\Omega_i$ due to the possible reinstatement of parents ($i \geq 2$). Any refinement scheme based on subdivision patterns which does not have all possible subdivision patterns and/or reinstates some parent elements prior to further subdivision will in general over-refine, that is, produce more refinement than requested by the adaptive procedure. Also, using subdivision patterns which add a centroidal vertex when not actually needed will over-refine as well.

4.2.2 Generalized Bisection

In two dimensions, an element is refined by bisecting its longest edge (two–triangle algorithm) [59]. Elements with non-conforming edges are subdivided following the patterns of Figure 44. The process terminates in a finite number of steps. Following the results of Rosenberg and Stenger [61] and Stynes [82] about longest edge bisection, the scheme is stable, furthermore, interior angles are always greater than one half of the lowest angle in the initial triangulation ${}_m\Omega_1$ [59].

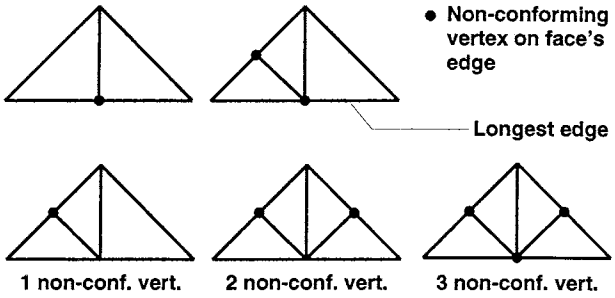


Figure 44. Non-conforming elements and their triangulations in two dimensions

This method of subdivision along the longest edge has been extended to three dimensions [60]. Elements to be refined are bisected along their longest edges. Non-conforming elements are subdivided along their longest edges in a recursive fashion. Unlike the two-dimensional case, an element that needs refinement or is non-confoming must be bisected at its longest edge.

This scheme guarantees nesting. In two dimensions, following the longest edge bisection results of Rosenberg and Stenger [61] and Stynes [82], the scheme is stable, furthermore, interior angles are always greater than one half of the lowest angle in the initial triangulation ${}_m\Omega_1$ [59]. In three dimensions, to this point in time, no one has yet presented a proof of the stability of the scheme probably because (i) the longest edge in a mesh region is not necessarily opposite the largest dihedral angle and (ii) the sum of all dihedral angles of a mesh region is not constant. However, the scheme seems to be "experimentally" stable. Because the non-conformity can propagate, this scheme will in general over-refine.

Joe [45] has proven that the infinite bisection of a tetrahedron is stable using generalized bisection on a mapped special tetrahedron. Note that this result does not prove that generalized bisection in the real space is stable. Liu and Joe [44] have presented a stable refinement algorithm that makes use of this result. In ${}_m\Omega_1$, for each element, a bisected edge is uniquely chosen (this does not mean that all elements will be subdivided). Elements that need to be subdivided are bisected along their bisected edges. When an element is subdivided into two elements, the bisected edges for the two new elements are imposed according to rules given in [44]. Once all elements that need refinement have been subdivided, there may be some non-conforming elements in the triangulation. The process of subdividing elements continues until there are no more non-conforming elements in the mesh. At this point, the scheme guarantees nesting, is stable, and will in general over-refine. After all levels of refinement have been applied, local transformations [35] are applied to further improve the quality of the final mesh. It should be noted that if local transformations are applied after each refinement iteration, a priori control of stability is lost. From experimental results given in [44], this scheme over-refines less than the scheme by Rivara and Levin [60] especially as the number of refinement levels becomes high. As a

Muthukrishnan et al. [53] sort mesh regions that are to be refined with respect to increasing length of their longest edges. The first region to be refined is the one at the end of the list. Before splitting the longest edge, the regions connected to the edge are examined. If a connected region has a longest edge different from the edge to be split, it is put in the list of regions to be refined at the appropriate rank. After the split, the list is updated. This refinement scheme is actually identical to the scheme described by Rivara and Levin [60] and therefore has the same properties. It is followed by a node repositioning procedure (nesting is lost).

Lo [46] sorts (in an approximate way) the mesh edges marked for refinement with respect to increasing length. The mesh edge at the end of the list is split and the list is updated. This scheme is different from the one by Rivara and Levin [60] and Muthukrishnan et al. [53] since only edges marked for refinement will be split. At this point, the scheme guarantees nesting, is not stable, and does not artificially refine. It is followed by a triangulation optimization procedure which makes use of node repositioning and local transformations (nesting is lost). These local transformations are:

1. 2–to-3,
2. 3–to-2, and
3. 4–to-4 which is an edge removal when there are four mesh regions connected to an edge.

4.2.6 Refinement Using Full Set of Subdivision Patterns

Refinement is performed by marking appropriate mesh edges for refinement and applying subdivision patterns to each mesh region. Each mesh region has from zero to six marked edges. Subdivision patterns for each possible configuration of marked edges have been developed in order to annihilate any over-refinement. There are ten possible patterns which are as follows (Fig. 48):

1. 1–edge: this is the classic 1:2 subdivision pattern (one template)
2. 2–edge (this is also the Green II in [9]):
   a. One face has two marked edges (two templates)
   b. All faces have one marked edge (one template)
3. 3–edge:
   a. One face has three marked edges: this is the classic 1:4 subdivision pattern (one template)
   b. Two faces have two marked edges (four templates)
   c. Three faces have two marked edges (eight templates)
4. 4–edge:
   a. One face has three marked edges (four templates)
   b. All faces have two marked edges (sixteen templates)
5. 5–edge (four templates)
6. 6–edge: this is the classic 1:8 subdivision pattern (one template)

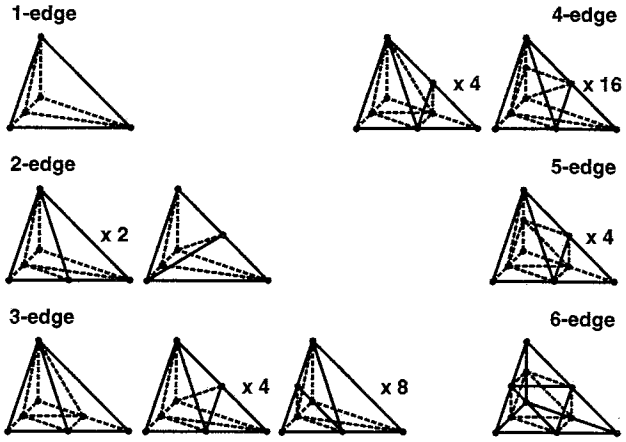


Figure 48. Subdivision patterns in three dimensions

When only the 1:2, 1:4, and 1:8 subdivision patterns are used, there is no possible triangulation incompatibility at the face level, in other words, the subdivision patterns on both sides of a face with either one or three marked edges will always match (at the face level). Inclusion of all the refinement types requires explicit consideration of triangulation compatibility at the face level. If a face with two and only two marked edges has been triangulated due to the subdivision of one region using that face, the template used to subdivide the other region must match the face triangulation. Since there are a priori two ways to triangulate a face with two marked edges (Fig. 49), any pattern which has N faces with two and only two marked edges needs 2^N templates.

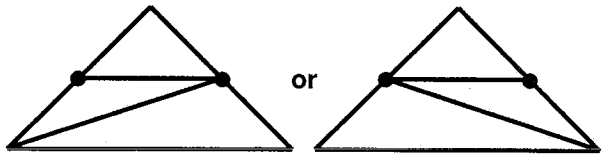


Figure 49. The two ways to triangulate a mesh face with two marked edges

As is, this refinement scheme is not stable since it is possible, and likely, that an angle (solid) will be bisected more than once when multiple refinements are applied in the same areas. However, it can be made stable at the price of some over-refinement. Assuming the quality of the initial triangulation ${}_m\Omega_1$ is Q_1, stability requires that for any subsequent triangulation ${}_m\Omega_i$ $(i > 1)$ its quality Q_i is such that $Q_i \geq q_l$ with $q_l = \alpha Q_1$ where α is some constant. Given a mesh region with at least one marked edge but fewer than six, the template corresponding to the number of marked edges is applied and the optimization procedure (with q_l as the threshold) is applied locally to the subdivided mesh region. If the optimization procedure is successful, nothing else has to be done for that mesh

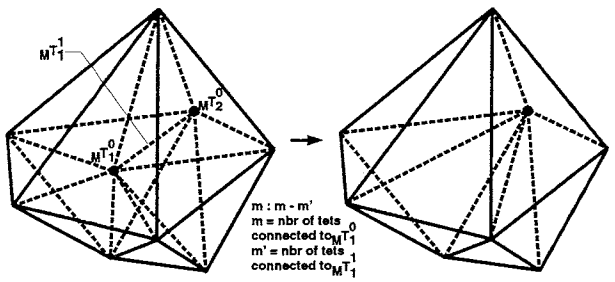


Figure 52. Edge collapsing in three dimensions

if the mesh edge can be collapsed (FALSE otherwise). Figure 54 illustrates graphically some of the cases where edge collapsing is not possible which are pointed out in the pseudo-code.

Before physically collapsing the edge, the geometry of the mesh regions to be created can be predicted exactly (this check refers to step 2 of the algorithm). The volumes of the new mesh regions can be computed by considering all mesh regions which are connected to ${}_mT_1^0$ but not connected to ${}_mT_1^1$ and virtually moving ${}_mT_1^0$ to ${}_mT_2^0$. Since the computation of the volume of a mesh region always consider the bounding vertices in a certain order, the (virtual) movement of one of its bounding vertices is valid only if the new volume is positive. Therefore, one can always tell beforehand if the to-be created mesh regions are invalid. The quality of the to-be created mesh regions can be predicted as well. If the quality of the to be created elements is not good enough with respect to some predetermined threshold, the derefinement of the edge need not be performed. This is important in order to guarantee the stability of the refinement/derefinement scheme. Also, assuming both end vertices are candidates to be the target vertex, the target vertex that would create the "better" triangulation of the two is chosen.

4.4. Complete Mesh Adaptation Procedure

The actual implementation of the mesh adaptation scheme uses the following steps:

1. Derefinement using edge collapsing as described above
2. Global optimization with $q_l = Q_1$
3. Refinement using full set of subdivision patterns without consideration for stability
4. Refinement vertex snapping (to the model boundary)
5. Global optimization with $q_l = Q_1$

So far, problems due to the non-stability of the implemented refinement scheme have not appeared. If they happen, the refinement can be made stable as described above at the price of some over-refinement.

4.5. Parallelization of Mesh Adaptation

Today's CFD computations are costly both in CPU time and memory. For big enough problems, the flow solver cannot be run on a classic scalar workstation for which performance and memory are limited. For large-scale analysis of fluid flows, it is necessary to use a parallel flow solver. Since the mesh adaptation is an integral part

```
Get bounding vertices (mT1^0 ⊏ gT1^d1^0, mT2^0 ⊏ gT2^d2^0)
of edge mT1^1 ⊏ gT1^d1^1
if d1^0 = d2^0
    if gT1^d1^0 = gT2^d2^0
        if d1^0 = 3 return TRUE (ok to collapse)
    else return FALSE (cannot collapse) (Fig.
    54.a)
else
    if d1^0 = 3 or d2^0 = 3 return TRUE (target
    vertex is the one classified on lower
    order model entity)
    At this point, the two mesh vertices are
    classified on model boundary
    if d1^1 = 3 return FALSE
    Switch (if necessary) mT1^0 and mT2^0 so that
    d1^0 > d2^0 (from now on, target vertex will
    be mT2^0 if collapsing is possible)
    if gT1^d1^1 ≠ gT1^d1^0 return FALSE (Fig. 54.b)
At this point, the two vertices are classified
on model boundary and the edge is classified
on the model entity of higher order
for each pair of mesh edges (mT2^1 ⊏ gT2^d2^1, mT3^1 ⊏
gT3^d3^1) that connect to mT1^0 and mT2^0 respec-
tively and connect to each other
    if d2^1 = 3 or d3^1 = 3 continue
    if d2^1 = d3^1
        if gT2^d2^1 ≠ gT3^d3^1 return FALSE
        else if d2^1 = 1 return FALSE
    At this point, the two edges are classi-
    fied on same model face or one is clas-
    sified on model face and the other is
    classified on the model face's boundary
    Switch (if necessary) mT2^1 and mT3^1 so
    that d2^1 > d3^1
    Find face mT1^2 ⊏ gT1^d1^2 bounded by
    {mT1^1, mT2^1, mT3^1}
    if mT1^2 does not exist, return FALSE
    if gT1^d1^2 ≠ gT2^d2^1 return FALSE (Fig. 54.c)
for each pair of mesh faces (mT1^2 ⊏ gT1^d1^2, mT2^2 ⊏
gT2^d2^2) that connect to mT1^0 and mT2^0 respec-
tively and connect to each other by a mesh
edge
    if d1^2 = 2 and d2^2 = 2 return FALSE (Fig.
    54.d)
    if (mT1^2, mT2^2) do not bound a mesh region,
    return FALSE
return TRUE
```

Figure 53. Pseudo-code for checking topological validity for edge collapsing

of the flow solver, it must be running in parallel as well in order not to become a bottleneck.

4.5.1 Derefinement

If a mesh edge ${}_mT_1^1$ is marked for derefinement, it is attempted to be collapsed. If the polyhedron $pol({}_mT_1^0)$ is on processor p_i, the edge collapsing is performed on p_i. If $pol({}_mT_1^0)$ is not fully on p_i, the missing mesh regions are requested from the appropriate processors. When all processors are done traversing their lists of mesh edges, the processors that have received requests send (migrate)

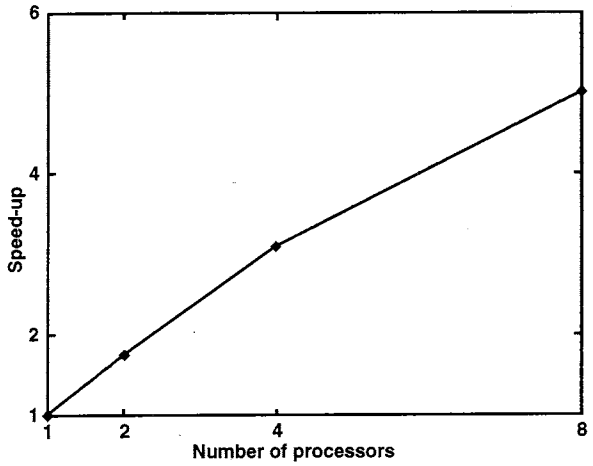


Figure 57. Speed-ups for the optimization procedure (85,000 elements)

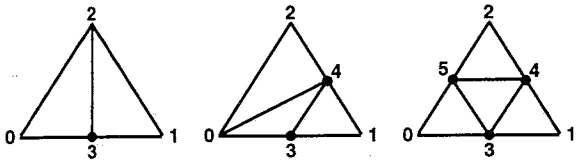


Figure 58. Subdivision patterns at the mesh face level

Once all mesh faces on the partition boundary are subdivided, links for all new mesh entities are updated. Then, each processor can apply the three-dimensional templates on any mesh region with at least one marked edge (as described above) without any communication.

Once all appropriate mesh regions have been subdivided, the refinement vertices which are classified on the model boundary need to be snapped to the corresponding model entity. Since snapping makes use of the local retriangulation tools, the technique to parallelize that process is similar to the one used to parallelize the derefinement and optimization steps. All processors iterate on a two step process: (i) (sequential) vertex snapping along with requests for missing mesh regions, and (ii) sending of requests and migration of requested mesh regions until all refinement vertices have been attempted to be snapped. At the end of the refinement step, the processors may not be well balanced for two reasons: (i) refinement is selective, and (ii) mesh regions have been migrated (due to snapping). Therefore, a load balancing step is applied before going further. Figure 59 shows speed-ups for the refinement procedure on 36,000 elements when 20% of the mesh edges are refined (resulting triangulation has 88,000 elements).

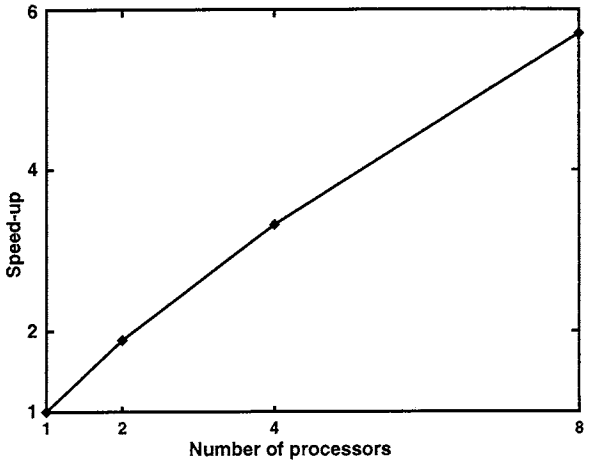


Figure 59. Speed-ups for parallel refinement

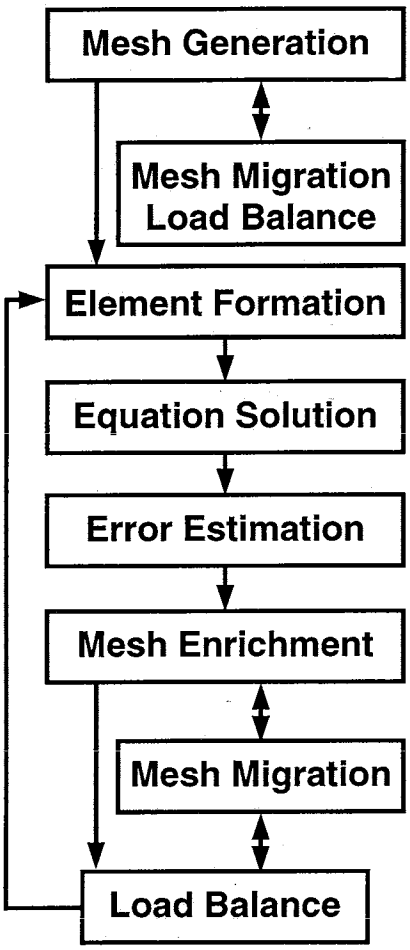


Figure 60. Components of a parallel adaptive analysis procedure

5. Parallel Adaptive Analysis Procedures

5.1. Structure of a Parallel Adaptive Analysis Procedure

Although the most computationally intensive operations in an adaptive analysis are of the same type as those of a fixed mesh analysis, an adaptive analysis must use more general structures which effectively account for the evolution of the discretization. The structure of a parallel adaptive analysis procedure follows directly from the procedures used for the parallel control of evolving meshes presented in the previous sections. Figure 60 presents an overall flow chart of a parallel automated adaptive analysis procedure.

Krylov space solvers. However, they can complicate their parallelization, leading to increased data communications or the need for a global ordering. However, depending upon the underlying problem, local preconditioners may prove adequate to assure convergence in a reasonable number of iterations, or the preconditioner may be calculated one time, stored, and used repeatedly.

5.2. Finite Element Code for Rotorcraft Aerodynamics

This section presents an parallel adaptive procedure for the automated aerodynamic analysis of helicopter rotors based on the procedures discussed in this paper. Adaptive analyses on unstructured discretizations represent an effective and accurate method to address the complex physical phenomena that characterize rotorcraft systems. The problem of the accurate numerical simulation of these phenomena has recently stimulated a vigorous research effort in the scientific community, certainly prompted by the fact that rotor–body interactions, transonic effects, wake effects and blade stall, all have a major impact on the performance, stability and noise characteristics of helicopter rotors.

One of the most important characteristics and distinguishing features of the software presented here is that all the different phases of the analysis, namely the mesh partitioning, the finite element solution, the error indication, the mesh adaptation and the subsequent load balancing, are realized without leaving the parallel environment. In contrast with other procedures that perform only part of the analysis in parallel, as for example just the finite element solution phase, our approach has the advantage of making better use of the power of a distributed memory architecture, leading to an integrated software environment, reducing the i/o and avoiding the bottlenecks that are always present when one tries to solve certain phases of the analysis in serial, especially when very large problems are addressed.

This integrated approach to the parallel adaptive solution of PDE's has lead us to select the message passing paradigm as our method of choice for the parallel programming. This is in contrast with the trend shown by some recent publications [36, 39, 52], where parallel finite element methodologies on fixed meshes have been developed based on data parallel techniques. In fact, we believe that the software development is more easily accomplished in a message passing programming model when one has to deal with adaptive strategies and mesh modification techniques. With the idea of developing a uniform software environment, we have used portable message passing protocols in each stage of the analysis. The implementation has been carried out using the message passing library standard MPI [1] and it has been tested on IBM SP–1 and SP–2 systems.

The procedure developed employs a stabilized finite element formulation which is valid for forward flight and for hovering rotor problems, as well as for general unsteady and steady compressible flow problems. The linear algebra is solved by means of a scalable implementation of the standard and matrix–free GMRES algorithms. Simple techniques are used for estimating regions of high error with the purpose of driving the adaptive procedures.

Techniques to effectively handle the far–field and symmetry boundary conditions for a hovering rotor are considered. Results are presented to demonstrate the ability of the parallel adaptive procedures to solve rotorcraft aerodynamics problems.

Consideration is also given to measures of efficiency and scalability of the parallel adaptive procedures that have been developed. The importance of these measures are demonstrated.

5.2.1 Finite Element Formulation

The initial/boundary value problem can be expressed by means of the Euler equations in quasi–linear form as

$$\mathbf{U}_{,t} + \mathbf{A}_i \cdot \mathbf{U}_{,i} = \mathbf{E}, \qquad (i = 1, \ldots, n_{sd}) \qquad (17)$$

plus well posed initial and boundary conditions. In equation (17), n_{sd} is the number of space dimensions, while $\mathbf{U} = \rho\,(1, u_1, u_2, u_3, e)$ are the conservative variables, $\mathbf{A}_i \cdot \mathbf{U}_{,i} = \mathbf{F}_{i,i}$ where $\mathbf{F}_i = \rho\, u_i (1, u_1, u_2, u_3, e) + p\,(0, \delta_{1i}, \delta_{2i}, \delta_{3i}, u_i)$ is the Euler flux, and $\mathbf{E} = \rho\,(0, b_1, b_2, b_3, b_i u_i + r)$ is the source vector. In the previous expressions, ρ is the density, $\mathbf{u} = (u_1, u_2, u_3)$ is the velocity vector, e is the total energy, p is the pressure, δ_{ij} is the Kronecker delta, $\mathbf{b} = (b_1, b_2, b_3)$ is the body force vector per unit mass and r is the heat supply per unit mass.

The Time–Discontinuous Galerkin Least–Squares finite element method is used in this effort [70, 71]. The TDG/LS is developed starting from the symmetric form of the Euler equations expressed in terms of the entropy variables $\mathbf{V}$ and it is based upon the simultaneous discretization of the space–time computational domain. A least–squares operator and a discontinuity capturing term are added to the formulation for improving stability without sacrificing accuracy. The TDG/LS finite element method takes the form

$$\begin{aligned}
&\int_{Q_n} \Big(-\mathbf{W}^h_{,t} \cdot \mathbf{U}(\mathbf{V}^h) - \mathbf{W}^h_{,i} \cdot \mathbf{F}_i(\mathbf{V}^h) + \mathbf{W}^h \cdot \mathbf{E}(\mathbf{V}^h)\Big)\, \mathrm{d}Q \\
&+ \int_{\mathcal{D}(t_{n+1})} \mathbf{W}^{h^-} \cdot \mathbf{U}(\mathbf{V}^{h^-})\, \mathrm{d}\mathcal{D} - \int_{\mathcal{D}(t_n)} \mathbf{W}^{h^+} \cdot \mathbf{U}(\mathbf{V}^{h^-})\, \mathrm{d}\mathcal{D} \\
&+ \int_{P_n} \mathbf{W}^h\, \mathbf{F}_i(\mathbf{V}^h) \cdot \mathrm{d}\mathbf{P} \\
&+ \sum_{e=1}^{(n_{el})_n} \int_{Q_n^e} \Big(\mathcal{L}\mathbf{W}^h\Big) \cdot \tau\Big(\mathcal{L}\mathbf{V}^h\Big)\, \mathrm{d}Q \\
&+ \sum_{e=1}^{(n_{el})_n} \int_{Q_n^e} \nu^h \hat{\nabla}_\xi \mathbf{W}^{\mathbf{h}} \cdot \mathrm{diag}\,[\tilde{\mathbf{A}}_0] \hat{\nabla}_\xi \mathbf{V}^{\mathbf{h}}\, \mathrm{d}\mathbf{Q} = 0. \qquad (18)
\end{aligned}$$

Integration is performed over the space–time slab Q_n, the evolving spatial domain $\mathcal{D}(t)$ of boundary $\Gamma(t)$ and the surface P_n described by $\Gamma(t)$ as it traverses the time interval $I_n =]t_n, t_{n+1}[$. $\mathbf{W}^h$ and $\mathbf{V}^h$ are suitable spaces

5.2.2 Boundary Conditions for Hovering Rotors

The imposition of the correct far–field boundary conditions is a critical issue in the analysis of hovering rotors, when one wants to give an accurate representation of the hovering conditions within a finite computational domain. For determining the inflow/outflow far–field conditions we have adopted the methodology suggested by Srinivasan *et al.* [81], where the 1–D helicopter momentum theory is used for determining the outflow velocity due to the rotor wake system. The inflow velocities at the remaining portion of the far–field are determined considering the rotor as a point sink of mass, for achieving conservation of mass and momentum within the computational domain.

Another important condition that must be considered for the efficient simulation of hovering rotors is the periodicity of the flow field. This allows consideration of a reduced computational domain given by the angle of periodicity $\psi = 2\,\pi/n_b$, n_b being the number of rotor blades.

The introduction of the periodicity conditions in the rotating wing flow solver has been implemented treating them as linear 2–point constraints applied via transformation as part of the assembly process. This approach has the double advantage of being easily parallelizable and of avoiding the introduction of Lagrange multipliers. On the other hand, it requires the mesh discretizations on the two symmetric faces of the computational domain to match on a vertex by vertex basis. Since this is not directly obtainable with the currently used unstructured mesh generator, a mesh matching technique has been developed for appropriately modifying an existing discretization.

In order to simplify the discussion, define one of the symmetric model faces as "master" and the other as "slave". The face discretization of the slave model face is deleted from the mesh, together with all the mesh entities connected to it. The mesh discretization of the master model face is then rotated of the symmetry angle ψ about the axis of rotation and copied onto the slave model face, yielding the required matching face discretizations. The matching procedure is then completed filling the gap between the new discretized slave face and the rest of the mesh using a face removal technique followed by smoothing and mesh optimization.

The imposition of the constraints can be formalized in the following manner. Consider the partition of the unknowns $\mathbf{V}$ in internal ($\mathbf{V}_i$), master ($\mathbf{V}_m$) and slave ($\mathbf{V}_s$), as

$$\mathbf{V} = (\mathbf{V}_i, \mathbf{V}_m, \mathbf{V}_s).$$

The slave unknowns $\mathbf{V}_s$ can be expressed symbolically as functions of the master unknown $\mathbf{V}_m$ as

$$\mathbf{V}_s = \mathbf{G} \cdot \mathbf{V}_m$$

or, for the j–th master–slave pair of nodes as

$$\mathbf{V}_s^j = \mathbf{G}^j \cdot \mathbf{V}_m^j,$$

where

$$\mathbf{G}^j = \begin{bmatrix} \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{R} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} \end{bmatrix}, \tag{23}$$

$\mathbf{R}$ being the rotation tensor associated with the rotation of the symmetry angle ψ about the axis of rotation.

The minimal set of unknowns $\bar{\mathbf{V}} = (\mathbf{V}_i, \mathbf{V}_m)$ is related to the redundant set $\mathbf{V}$ by

$$\mathbf{V} = \mathbf{\Gamma} \cdot \bar{\mathbf{V}} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \\ \mathbf{0} & \mathbf{G} \end{bmatrix} \cdot \bar{\mathbf{V}}. \tag{24}$$

The unconstrained linearized discrete equations of motion read

$$\mathbf{J} \cdot \Delta\mathbf{V} = \mathbf{r},$$

where $\mathbf{J}$ is the tangent matrix and $\mathbf{r}$ is the residual vector. Applying the transformation $\mathbf{\Gamma}$ to the unconstrained system yields the constrained reduced system

$$\mathbf{\Gamma}^T\mathbf{J}\mathbf{\Gamma} \cdot \Delta\bar{\mathbf{V}} = \mathbf{\Gamma}^T \cdot \mathbf{r}. \tag{25}$$

Refer to [74] for implementation details of this technique.

5.2.3 Subsonic and Transonic Hovering Rotors

Caradonna and Tung [12] have experimentally investigated a model helicopter rotor in several subsonic and transonic hovering conditions. These experimental tests have been extensively used for validating CFD codes for rotating wing analysis. The experimental setup was composed of a two–bladed rotor mounted on a tall column containing the drive shaft. The blades had rectangular planform, square tips and no twist or taper, made use of NACA0012 airfoil sections and had an aspect ratio equal to six.

Figure 61 shows the experimental and numerical values of the pressure coefficients at different span locations for three subsonic test cases investigated by Caradonna and Tung, namely $\theta_c = 0^o$ and $M_t = 0.520$, $\theta_c = 5^o$ and $M_t = 0.434$, $\theta_c = 8^o$ and $M_t = 0.439$. The agreement with the experimental data is good at all locations, including the section close to the tip. Only two pressure distributions are presented for each case for space limitations, however similar correlation with the experimental data was observed at all the available locations. Relatively crude meshes have been employed for all the three test cases, with the coarsest mesh of only 101,000 tetrahedra being used for the $\theta_c = 0^o$ case, and the finest of 152,867 tetrahedra for the $\theta_c = 8^o$ test problem.

The analysis was performed on 32 processing nodes of an IBM SP–2. Reduced integration was used for the interior elements for lowering the computational cost, while full integration was used at the boundary elements for better resolution of the airloads, especially at the trailing edge of the blade. The GMRES algorithm with block–diagonal preconditioning was employed, yielding an average number of GMRES iterations to convergence of about 10. The analysis was advanced in time using one single Newton

sity and Mach number, which was employed for driving the parallel adaptation of the mesh. For the new vertices created by the adaptation process, the solution was projected from the coarser mesh using simple edge interpolation. The solution obtained in this way was used for restarting the analysis, which was advanced for 60 time steps with a CFL number of 15. Similarly, a second adaptation was performed, yielding the final mesh for which another 40 time steps were performed at a CFL of 20, until convergence in the energy norm of the residual. The average number of GMRES cycles per time step throughout the analysis was 8.

Figure 64 shows the mesh at the upper face of the blade tip, before and after refinement. The different grey levels indicate the different subdomains, i.e. elements assigned to the same processing node are denoted by the same level of grey. Note the change in the shape of the partitions from the initial to the final mesh, change generated by the mesh migration procedure for re–balancing the load after the refinement procedure has modified the discretization. Note also how the mesh nicely follows the shock.

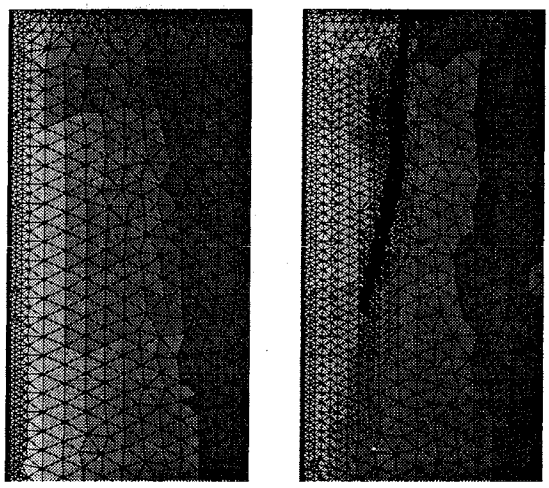


Figure 64. Meshes with partitions on the upper surface of the blade tip, $\theta_c = 8^o$, $M_t = 0.877$. At left: initial coarse grid with IRB partitions. At right: final adapted grid with partitions obtained by migration.

5.3. Effectiveness of Parallel Adaptive Analysis Procedures

The evaluation of the efficiency and performance of a parallel adaptive analysis is a task complicated by the numerous aspects that must be considered. In the following we will try to address at least some of them with the help of a classical problem in CFD, namely that of the Onera M6 wing in transonic flight, that we have used in the early stages of development of our code for validation purposes. This wing has been studied experimentally by Schmitt and Charpin [65] and it has been employed by numerous researchers for validating both structured and unstructured flow solvers. The wing is characterized by an aspect ratio of 3.8, a leading edge sweep angle of 30^o, and a taper ratio of 0.56. The airfoil section is an Onera D symmetric section with 10% maximum thickness–to–cord ratio.

We consider a steady flow problem characterized by an angle of attack $\alpha = 3.06^o$ and a value of $M = 0.8395$ for the freestream Mach number. In such conditions, the flow pattern around the wing is characterized by a complicated double–lambda shock on the upper surface of the wing with two triple points.

We first address the scalability of the parallel solver on a fixed mesh, i.e. we analyze the speed–ups attained by the code using one single mesh and varying the number of processing nodes. This is a classical measure of efficiency, and it is important to show that the implemented procedure performs well with respect to it before measuring other properties that are more pertinent to an adaptive analysis.

The simulation was performed using a mesh consisting of 128,172 tetrahedra, using the matrix–free GMRES algorithm with reduced integration of the interior elements and full integration of the boundary elements. A local time stepping strategy was employed with one single Newton iteration per time step, using a CFL condition of 5 in the first 20 time steps and a CFL equal to 10 for another 80 time steps, attaining a drop in the residual of three orders of magnitude. The mesh was partitioned using a parallel implementation of the IRB algorithm. The time for partitioning, even if small when compared with the time needed for achieving convergence in the finite element analysis, is not considered in the following. The analysis was run on 4, 8, 16, 32, 64, 128 processors of an IBM SP–2 and the results are presented in Figure 65 in terms of the inverse of the wall clock time versus the number of processing nodes. The highly linear behavior of the parallel algorithm shows the excellent characteristics of scalability of the code.

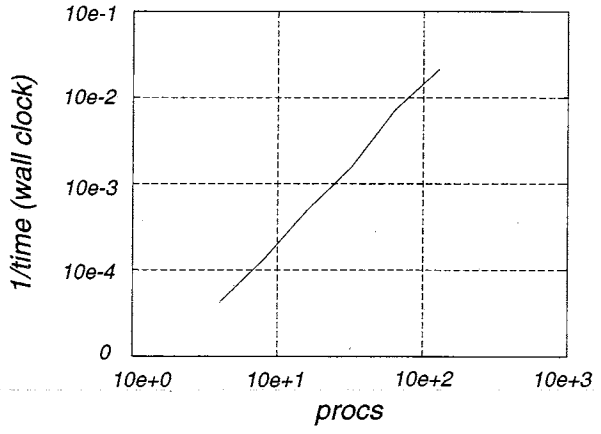


Figure 65. Parallel efficiency evaluated at fixed mesh for the Onera M6 wing in transonic flight. 128,172 tetrahedra, IRB partitions.

The same problem was then adaptively solved in order to more accurately resolve the complicated features of the flow. An initial coarse mesh of 85,567 tetrahedra was partitioned with the IRB algorithm on 32 processing nodes

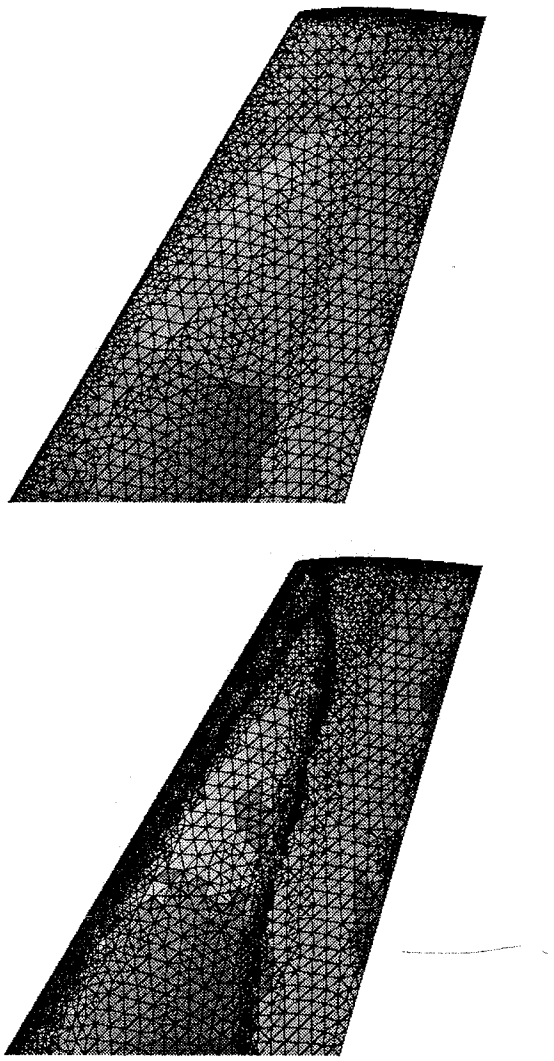


Figure 67. Onera M6 wing in transonic flight, $\alpha = 3.06^o$, $M = 0.8395$. Initial and final meshes. Grey levels indicate processor assignment.

gorithm that can then operate on balanced or nearly balanced meshes. This "incremental" rebalancing capability represents a nice advantage of the iterative load balancing scheme over other algorithms. The parallel repartitioning algorithm was instead activated just once at the end of each adaptive step.

The meshes obtained during the two previously mentioned parallel adaptive simulations of the Onera M6 wing were analyzed for gathering data on the overall performance of the analysis. Figure 68 reports plots of the boundary faces and neighbor statistics. The quantities plotted are defined as:

(i) Surface–to–volume measures:

$$S_{\max} = \max_i (\text{Boundary Faces}_i/\text{Faces}_i),$$

$$S_{\text{glob}} = \text{Boundary Faces/Faces}.$$

(ii) Neighbor measures:

$$N_{\max} = \max_i (\text{Neighbors}_i/(\text{Procs} - 1)),$$

$$N_{\text{avrg}} = (\sum_i \text{Neighbors}_i/(\text{Procs} - 1))/\text{Procs}.$$

All these quantities are reported in Figure 68 versus the number of tetrahedra in the mesh at a certain adaptive level normalized by the number of tetrahedra in the initial mesh. The solid line represents the values of the parameters obtained for the parallel adaptive analysis where the iterative mesh migration procedures were employed. The dashed line corresponds to the parallel adaptive analysis where the refined meshes were repartitioned after each adaptive step using the parallel IRB algorithm.

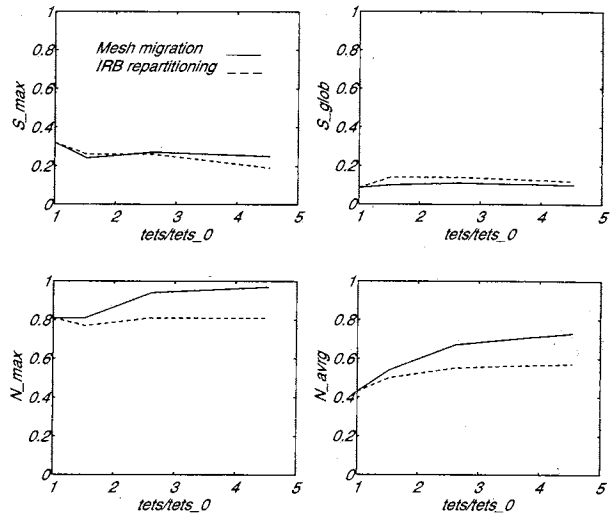


Figure 68. Boundary faces and neighbor statistics for the parallel–adaptive analysis of the Onera M6 wing in transonic flight using the mesh migration and IRB rebalancing schemes.

From the analysis of the first two plots at the top of Figure 68, it is clear that the migration procedures implemented in this work control very effectively the surface–to–volume ratios, which in fact remain constant and fairly similar to the ones obtained with the IRB partitioning for the whole simulation. On the other hand, the second two plots of the same figure show that the number of neighbors of each subdomain tends to increase with the number of adaptive steps performed. A more detailed analysis shows that in general each subdomain is connected by a significant amount of mesh entities (vertices, faces, edges) only with a reduced number of neighbors, while it shares a very limited number of mesh entities with the other neighbors. We are currently investigating ways of removing such small contact area interconnections, in order to achieve a better control on the number of neighbors.

The different partition statistics provided by the two rebalancing algorithms and shown in the previous figure

[13] S. D. Connell and D. G. Holmes. 3-dimensional unstructured adaptive multigrid scheme for the euler equations. *AIAA J.*, 32:1626–1632, 1994.

[14] H. L. de Cougny. Automatic generation of geometric triangulations based on octree/Delaunay techniques. Master's thesis, Civil and Environmental Engineering, Scientific Computation Research Center, Rensselaer Polytechnic Institute,Troy, NY 12180-3590, May 1992. SCOREC Report # 6-1992.

[15] H. L. de Cougny, K. D. Devine, J. E. Flaherty, R. M. Loy, C. Ozturan, and M. S. Shephard. Load balancing for the parallel solution of partial differential equations. *Applied Numerical Mathematics*, 16:157–182, 1994.

[16] H. L. de Cougny, M. S. Shephard, and C. Ozturan. Parallel three-dimensional mesh generation on distributed memory mimd computers. *Engineering with Computers*, 1995. submitted.

[17] B. E. de l'Isle and P. L. George. Optimization of tetrahedral meshes. INRIA, Domaine de Voluceau, Rocquencourt BP 105 Le Chesnay France, 1993.

[18] K. M. Devine. *An adaptive HP-finite element method with dynamic load balancing for the solution of hyperbolic conservation laws on massively parallel computers*. PhD thesis, Computer Science Dept, Rensselaer Polytechnic Institute, Troy, New York, 1994.

[19] H. Edelsbrunner and N. R. Shah. Incremental topological flipping works for regular triangulations. In *8th Annual Comp. Geometry*, pages 6–43, 1992.

[20] C. Farhat. A simple and efficient automatic FEM domain decomposer. *Computers and Structures*, 28:579–602, 1988.

[21] C. Farhat and M. Lesoinne. Automatic partitioning of unstructured meshes for the parallel solution of problems in computational mechanics. *Int. J. Numer. Meth. Engng.*, 36:745–764, 1993.

[22] M. Fiedler. Algebraic connectivity of graphs. *Czechoslovak Math. J.*, 23:298–305, 1973.

[23] D. A. Field. Laplacian smoothing and Delaunay triangulations. *Comm. Appl. Num. Meth.*, 4:709–712, 1987.

[24] G. C. Fox and W. Furmanski. Communication algorithms for regular convolutions and matrix problems on the hypercube. In M. T. Heath, editor, *Conference on Hypercube Multiprocessors*, pages 223–238, Philadelphia, 1986. SIAM.

[25] W. H. Frey and D. A. Field. Mesh relaxation: A new technique for improving triangulations. *Int. J. Numer. Meth. Engng.*, 31:1121–1133, 1991.

[26] P. L. George. *Automatic Mesh Generation*. John Wiley and Sons, Ltd, Chichester, 1991.

[27] P. L. George. Generation de maillages par une methode de type voronoi partie 2: Le cas tridimensionnel. Technical Report INRIA 1664, INRIA, Domaine de Voluceau, Rocquencourt BP 105 Le Chesnay France, 1992.

[28] P. L. George, F. Hecht, and E. Saltel. Fully automatic mesh generator for 3d domains of any shape. *Impact of Comp. in Sc. and Engng.*, 2:187–218, 1990.

[29] M. E. Go Ong. *Hierarchical Basis Preconditioners for Second Order Elliptic Problems in Three Dimensions*. PhD thesis, Univ. of California, Los Angeles CA, 1989.

[30] N. Golias and T. Tsiboukis. An approach to refining three-dimensional tetrahedral meshes based on delaunay transformations. *Int. J. Numer. Meth. Engng.*, 37:793–812, 1994.

[31] W. Gropp. Simplified linear equation solvers users manual. Technical Report ANL-98/8-REV 1, Mathematics and Computer Science Division, Argonne National Laboratory, 1993.

[32] E. L. Gursoz, Y. Choi, and F. B. Prinz. Vertex-based representation of non-manifold boundaries. In M. J. Wozny, J. U. Turner, and K. Priess, editors, *Geometric Modeling Product Engineering*, pages 107–130. North Holland, 1990.

[33] S. W. Hammond. *Mapping Unstructured Grid Computations to Massively Parallel Computers*. PhD thesis, Computer Science Dept., Rensselaer Polytechnic Institue, Troy, 1991.

[34] J. JaJa. *An introduction to Parallel Algorithms*. Addison Wesley, Reading Mass., 1992.

[35] B. Joe. Three-dimensional triangulations from local transformations. *SIAM J. Sci. Stat. Comp.*, 10(4):718–741, 1989.

[36] Z. Johan. *Data Parallel Finite Element Techniques for Large-Scale Computational Fluid Dynamics*. PhD thesis, Stanford University, July 1992.

[37] Z. Johan, T. J. R. Hughes, K. K. Mathur, and S. L. Johnsson. A data parallel finite element method for computational fluid dynamics on the connection machine system. *Comp. Meth. Appl. Mech. Engng.*, 99:113–134, 1992.

[38] Y. Kallinderis and P. Vijayan. Adaptive refinement-coarsening scheme for three-dimensional unstructured meshes. *AIAA J.*, 31(8):1440–1447, August 1993.

[39] J. G. Kennedy, M. Behr, V. Kalro, and T. E. Tezduyar. Implementation of implicit finite element methods for incompressible flows on the cm-5. In *Army High-Performance Computing Research Center*, number 94-017, University of Minnesota, 1994.

[40] B. W. Kernigham and D. M. Ritchie. *The C programming Language*. Prentice Hall, Inc., 1990.

[41] C. P. Kruskal, L. Rudolph, and M. Snir. Efficient parallel algorithms for graph problems. *Algorithmica*, 5:43–64, 1990.

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

[73] M. S. Shephard, C. L. Bottasso, H. L. de Cougny, and C. Ozturan. Parallel adaptive finite element analysis of fluid flows on distributed memory computers. In *Recent Developments in Finite Element Analysis*, pages 205–214. Int. Center for Num. Meth. in Engng., Barcelona, Spain, 1994.

[74] M. S. Shephard, S. Dey, and M. K. Georges. Automatic meshing of curved three-dimensional domains: Curving finite elements and curvature-based mesh control. In *Proceedings of the IMA Summer Program Modeling Mesh Generation and Adaptive Numerical Methods for Partial Differential Equations, July 6-23, 1993*. Springer Verlag, 1994.

[75] M. S. Shephard and P. M. Finnigan. Toward automatic model generation. In A. K. Noor and J. T. Oden, editors, *State-of-the-Art Surveys on Computational Mechanics*, pages 335–366. ASME, 1989.

[76] M. S. Shephard and M. K. Georges. Automatic three-dimensional mesh generation by the Finite Octree technique. *Int. J. Numer. Meth. Engng.*, 32(4):709–749, 1991.

[77] M. S. Shephard and M. K. Georges. Reliability of automatic 3-D mesh generation. *Comp. Meth. Appl. Mech. Engng.*, 101:443–462, 1992.

[78] M. S. Shephard and N. P. Weatherill, editors. *Int. J. Numer. Meth. Engng.*, volume 32. Wiley-Interscience, Chichester, England, 1991.

[79] A. Shostko and R. Löhner. Three-dimensional parallel unstructured grid generation. *Int. J. Numer. Meth. Engng.*, 38:905–925, 1995.

[80] H. D. Simon. Partitioning of unstructured meshes for parallel processing. *Comput. Sys. Engng.*, 2:135–148, 1991.

[81] G. Srinivasan, V. Raghavan, and E. Duque. Flowfield analysis of modern helicopter rotors in hover by Navier-Stokes method. In *International Technical Specialist Meeting on Rotorcraft Acoustics and Rotor Fluid Dynamics*, Philadelphia, PA, 1991.

[82] M. Stynes. On faster convergence of the bisection method for all triangles. *Math. of Computation*, 35(152):1195–1201, October 1980.

[83] B. K. Szymanski and A. Minczuk. A representation of a distribution power network graph. *Archiwum Elektrotechniki*, 27(2):367–380, 1978.

[84] T. E. Tezduyar, M. Behr, S. Mittal, and J. Liou. A new strategy for finite element computations involving moving boundaries and interfaces - the deforming-spatial-domain/space time procedure: I. the concept and preliminary tests. *Comp. Meth. Appl. Mech. Engng.*, 94:339–351, 1992.

[85] A. Vidwans, Y. Kallinderis, and Venkatakrishnan. Parallel dynamic load-balancing algorithm for three-dimensional adaptive unstructured grids. *AIAA Journal*, 32(3):497–505, March 1994.

[86] R. F. Warming, R. M. Beam, and B. J. Hyett. Diagonalization and simultaneous symmetrization of the gas-dynamic matrices. *Math. of Comp*, 29:1037–1045, 1975.

[87] D. F. Watson. Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes. *The Computer J.*, 24(2), 1981.

[88] N. P. Weatherill and O. Hassan. Efficient three-dimensional delaunay triangulation with automatic point creation and imposed boundary constraints. *Int. J. Numer. Meth. Engng.*, 37:2005–2039, 1994.

[89] K. J. Weiler. The radial-edge structure: A topological representation for non-manifold geometric boundary representations. In M. J. Wozny, H. W. McLaughlin, and J. L. Encarnacao, editors, *Geometric Modeling for CAD Applications*, pages 3–36. North Holland, 1988.

[90] R. Williams. *DIME: Distributed Irregular Mesh Environment*. Supercomputing Facility, California Institute of Technology, 1990.

[91] R. D. Williams. Performance of dynamic load balancing algorithms for unstructured grid calculations. Technical Report C3P913, Pasadena, 1990.

[92] R. D. Williams. Voxel databases: A paradigm for parallelism with spatial structure. *Concurrency*, 4:619–636, 1992.

[93] R. D. Williams. Dime++: A parallel language for indirect addressing. Technical Report CCSF-34, Caltech Concurrent Supercomputing Facilities, Pasadena, June 1993.

[94] M. A. Yerry and M. S. Shephard. Automatic three-dimensional mesh generation by the modified-octree technique. *Int. J. Numer. Meth. Engng.*, 20:1965–1990, 1984.

[95] H. Zima and B. M. Chapman. Compiling for distributed memory systems. Technical Report ACPC/TR 92-17, Austrian Center for Parallel Computation, University of Vienna, 1992.