3D PARALLEL UNSTRUCTURED MESH GENERATION*

T. Okusanya† and J. Peraire‡

Fluid Dynamics Research Laboratory
Department of Aeronautics and Astronautics
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

ABSTRACT

A strategy for the parallel generation of 3D tetrahedral unstructured meshes is proposed based on the extension of the 2D algorithms presented by Okusanya *et al* [1]. These are based on a sequence of point insertion and load balancing cycles. Points are inserted concurrently within each subdomain and dynamic load-balancing is incorporated to ensure an even work distribution during the generation process. The inter-processor boundaries are allowed to move to accomodate element migration between subdomains.

INTRODUCTION

Rapid advances in unstructured mesh methods for computational fluid dynamics (CFD) have been made over recent years and, for the computation of inviscid flows, have achieved a considerable level of maturity. As a result, a number of systems incorporating automatic mesh generators and flow solvers have been built which are currently being used in a semi-production mode by industry and research establishments [2, 3, 4]. The main advantage of the unstructured mesh approach is that, for complex geometries, it allows to significantly reduce the time period associated with the CFD analysis cycle. This feature is especially valuable during the early stages of design of aircraft when a large number of options needs to be analyzed.

The reasons for distributed mesh generation are twofold. The first one is to reduce the computational times required for the generation of large grids. The second one, and more important, is to avoid the very large single-processor memory requirements in sequential generators. At present, the size of the largest computations is determined by the ability to generate grids on serial machines. For instance, the generation of a tetrahedral unstructured grid containing 16 million cells (which is well within the capabilities of existing parallel solvers [5]) requires a serial machine with 1Gb of core memory using the FELISA system [3]. Once generated, this grid needs to be partitioned, again using a serial machine, and sent to the parallel processors to carry out the flow solution. Hence, a strategy capable of generating a grid in parallel such that when the grid generation process is finished, the grid is already partitioned and in place for the flow solution, is desirable.

This paper addresses the development of a parallel 3D unstructured mesh generator designed to operate on parallel MIMD machines with distributed memory. Other attempts at parallel mesh generation include Weatherhill [6], Löhner *et al* [7] and Saxena *et al* [8] which in general follow the rule of domain partitioning with independent meshing of the subdomains. Depending on the implementation, the discretization of the inter-subdomain regions can take place either before or after the subdomain meshing. Efficient parallel algorithms require a balance of work between the processors while maintaining inter-processor communication to a minimum. A key to determining and distributing the work load is the knowledge of the type of analysis being performed. For mesh generation, the only knowledge available at the beginning of the process is the geometric model, the complexity of which has little or no relationship to the work required to generate the mesh.

A more general approach has been proposed by Shephard *et al* [9, 10]. Shephard *et al* discuss the development of an octree-based parallel three-dimensional mesh generator [9] which is later coupled to an adaptive refinement system and a parallel mesh database into the distributed mesh environment [10].

A set of algorithms for parallel mesh generation in 2D were

---

*Submitted to the Joint ASME/ASCE/SES Summer Meeting, Special Symposium on Trends in Unstructured Mesh Generation, January 1997

†Graduate Research Assistant

‡Associate Professor

proposed by Okusanya *et al* [1] which form the basis for the 3D parallel mesh generator presented here. Point insertion within each subdomain is performed concurrently. Mesh migration required for the implementation of the constrained Delaunay and load balancing algorithms is accomplished by allowing the interprocessor boundaries to move during the generation process. The point insertion process is interspersed with dynamic load balancing to maintain an even work load.

MESH GENERATION ALGORITHMS

The mesh generation procedures are based on a constrained Delaunay triangulation with respect to a discretization of the domain boundary. The point insertion algorithm due to Watson [11] based on the circumcircle criterion is considered and modified to ensure grid consistency in the constrained case.

The interior point creation strategies of Rebay [12], Chew [13] and Weatherhill [14] are implemented. In Rebay's algorithm, the new points are placed on the Voronoï segment whereas in Chew's and Weatherhill's algorithms, the new points are placed on the element circumcenters and centroids respectively. The interior points are inserted to meet an element size distribution which is specified by means of a *background mesh* and a *source distribution* [3]. The selected interior point creation stategies require an initial triangulation of the computational domain boundary. This initial triangulation is currently performed a priori by discretizing, using the background mesh, a CAD representation of the domain boundary using Ferguson splines and surfaces.

PARALLEL IMPLEMENTATION

The data structures and procedures as described in [1] were extended and implemented in 3D . The parallel mesh generation process is based on a cycle of point insertion and load balancing operations. Points are inserted within each subdomain until a prescribed number of elements have been generated. The mesh is then balanced to ensure an even distribution of the work load. Implementation follows a message driven master/slave paradigm and the number of slave processors coincides with the number of subdomains.

1. Data Structure

The mesh is partitioned across processors such that each element is uniquely owned by one processor. The usual 4 vertices and neighbors representation of each tetrahedral element is employed. Inter-processor information is dealt with by having faces, edges and points common to two adjacent subdomains shared (duplicated). The mesh generation procedures operate on this model of distributed grid representation. The concept of face, edge and point sharing is embodied in a data structure termed a *front face* which is defined for each shared inter-processor face and duplicated on each sharing subdomain. This is depicted in the 3-D context in Fig. 1. For a given inter-processor interface such as shown in Fig. 1, the associated front on each sharing subdomain contains information regarding the remote processor *Unique Identifier*, identifier of the remote processor front face, and the identifiers of the local and remote front vertices. These vertex identifiers have a one-to-one correspondence on the local and remote processors.

2. Mesh Migration

The load balancing and element request procedures to be described later require the transfer of elements between subdomains. During mesh generation, the process of element migration from one processor to another is carried out in three stages. The first stage involves verifying that the elements to be migrated can be transferred to the destination processor. This requires a *front lock* on all neighbouring processors (excluding the sender and receiver) with fronts affected by the migration. This is necessary to avoid situations, such as the simultaneous transfer of adjacent elements in different subdomains, in which the proper updating of the shared information cannot be guaranteed.

The second stage involves the transfer of mesh entities between the sender and receiver. This is depicted in the 2D context in Fig. 2 and involves packing the information regarding the boundary, topology and geometry of the migration element set. Note that the set of elements to be transferred need not be contiguous. In order to determine the boundary and topology of the elements to be transferred, a Breadth First Search (BFS) algorithm is employed. The first step in this stage is the packing of the boundary faces and assignment of proper values for the remote topological entity identifiers. The final step is the packing of the interior entities.

The third stage involves the update of the front faces on all the processors affected by the transfer excluding the sender and the receiver. The mesh migration process is completed when the receiver processor issues appropriate messages to unlock the locked fronts.

The performance of the migration algorithm is illustrated in Fig. 3 which shows the migration throughput achieved for a typical example on the IBM SP2. The figure includes comparisons between the 2D [1] and 3D versions and shows the CPU time required to migrate elements between two processors as a function of the number of elements. From this figure, the average throughput in 2D is seen to be approximately 5000 elements per second and in 3D is approximately 8000 elements per second. This is attributed to the improvements in the 3D internal formulation such as the use of an Alternating Digital Tree [15] for searches.

3. Load Balancing

The dynamically evolving nature of the unstructured mesh during the mesh generation process creates a load imbalance among the processors. Dynamic load balancing is accomplished by means of *partitioning* and *dynamic redistribution* of the elements. The algorithm is based on a modified Coordinate Bisection in which the partitioning of the domain into subdomain blocks is done using planes perpendicular to the cartesian axes.

Processor assignment of the elements is based on the element centroid coordinates. The number of blocks ($\mathbf{n_X} \times \mathbf{n_Y} \times \mathbf{n_Z}$) is based on the number of processors available and selected in such a way that $\mathbf{n_X}$, $\mathbf{n_Y}$ and $\mathbf{n_Z}$ are as close as possible. The partition separators are obtained by iteratively solving a linear system of equations derived from a spring analogy. Each block is represented as three linear springs in the x, y and z directions such that in each direction, the "spring constant" is proportional to the ratio between the number of elements in the block and block length along that direction. This linear system is depicted in the 2D context in Fig. 4. After the partition locations are obtained, the final destination of each element is determined and a migration schedule is set up to redistribute the elements. This algorithm is in some respects similar to the "Strip" partitioning algorithm developed by Vidwans *et al* [16].

For meshes which contain regions of densely clustered elements, the iterative procedure may lead to an oscillatory behaviour which requires an unreasonable number of iterations to converge. This situation is easily remedied by introducing band limiters which limit the minimum and maximum displacements for a partition within each iteration.

The performance of the load balancing algorithm is assessed by generating a mesh on one processor, random assignment and migration of elements to subdomains followed by load balancing. It should be noted that random distribution represents the worst-case scenario for the load balancing algorithm. Figure 5 shows the CPU time required to balance 144600 randomly distributed elements generated around a Dassault Falcon (Fig. 7) as a function of the number of processors. The total CPU times depicted have been decomposed into processor assignment times and element migration times. For comparison, the 2D load balancing results for 50000 elements (Fig. 6) is included. We find that in general, the migrations times are fairly constant for a given total number of elements. The slight increase in the total time is due to the increasing overhead involved in processor assignment for the elements. As in the 2D case, the relatively low time required for the two processor case in 3D may be explained by noting that the migration algorithm simplifies considerably when only two processors are involved. Figure 8 shows a 4 processor example of load balancing during the process of mesh generation about a Dassault Falcon.

4. **Element Request**

Point insertion algorithms [11, 17] in general require the global element set which will be affected by the insertion of a newly created point. This is required by the point insertion algorithms to guarantee that the mesh is locally Delaunay. Hence in order to retrieve required off-processor elements, new procedures must be derived. The initiator of the request may be either granted or denied remote elements by any affected processor based on the existence of an overlap between any remote topological entity in use and the requested elements.

The general procedure is initiated given an initial set of local elements $\{\Omega_i{}^r\}$ affected by a newly created point. Request messages are propagated to the subdomains which lie across any processor front elements attached to $\{\Omega_i{}^r\}$. From the point of view of a processor in the request processor set, a search is performed to obtain the required elements $\{\Omega^t\}$. In order to maintain mesh consistency, a front lock request must be made on all front elements attached to $\{\Omega^t\}$. If the elements requested are not in use and locks are granted, the request initiator is granted the element set $\{\Omega^t\}$. If this is not the case, the initiator is denied that set of elements and must consider the insertion of the point at a later stage.

5. **Point Insertion Buffering**

During the process of element request, idle time which is being wasted waiting for a response from remote processors can be put to better use by buffering the element request into an internal queue and attempting to insert another point. When a prescribed buffer limit is reached, mesh generation is halted until some or all buffered requests are answered. The usefulness of this is that each processor may now do other work if available. It is found that in general, buffering of few elements yields the best results.

PARALLEL MESH GENERATION RESULTS

The above described algorithms are incorporated into a parallel mesh generation code and the performance obtained for a typical case (Fig. 7) using the centroid node placement algorithm [14] is depicted in Figs. 9, 10, 11 and 12. Figures 9 and 11 show the mesh generation timings based on average and maximum processor CPU times respectively as a function of the number of processors for 1.8 million elements without a final fine-resolution load balance. Figures 10 and 12 show the mesh generation timings based on average and maximum CPU times respectively for the same case with a final fine-resolution load balance. The total CPU times have been decomposed into meshing, load balance, element request and polling times. The latter makes reference to the time the processor is handling requests or data from other processors. The figures show linear scaling with the number of processors for the actual time spent in mesh generation. When compared to the data in [1] for the 2D case, the curves show the same trend. Scalability of the total time is slightly worse in 3D than in 2D. This is due to the fact that the time required for element request in 3D is a much larger fraction of the total time than in 2D. In general, one finds that in 3D, more elements are potentially affected by the insertion of a point. This translates directly into a greater element migration volume between subdomains and thus increased element request time.

CONCLUSIONS

An approach to parallel unstructured 3D mesh generation has been developed for operation on distributed memory MIMD computers based on the extension of the 2D work by Okusanya *et al* . The developed algorithms include parallelization of the point insertion schemes and the inclusion of element migration and dynamic load balancing to ensure even work load distribution. The algorithms have been implemented on an IBM SP2 system using MPI but are easily portable to other platforms supporting standard communication libraries. Partial scalability of the mesh generation algorithm has been illustrated based on timings for different processor sizes. However the main thrust of this paper is the reduction in the memory bottleneck associated with serial generators. The scalability achieved can be maintained for a larger number of processors provided that the granularity i.e number of elements per subdomain is sufficiently high.

ACKNOWLEDGEMENTS

This material is based upon work supported under a National Science Foundation Graduate Research Fellowship. Computing time on the IBM SP2 was provided by the NASA Langley Research Center under grant NAG1-1587.

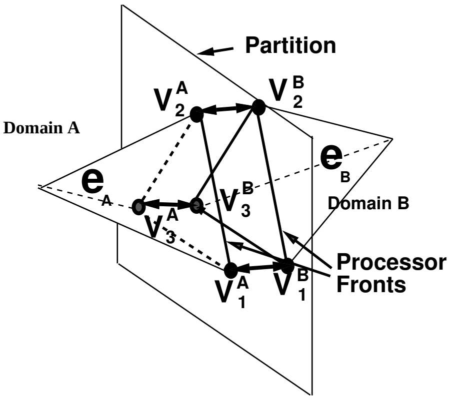


Figure 1: Interprocessor Front

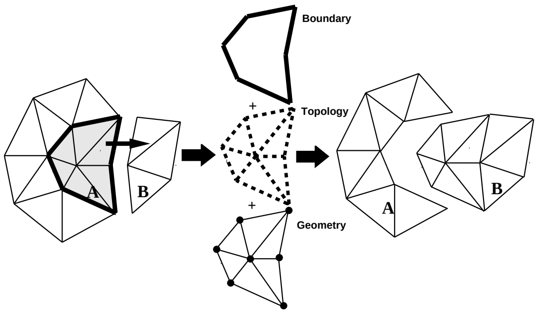


Figure 2: Element Migration Philosophy

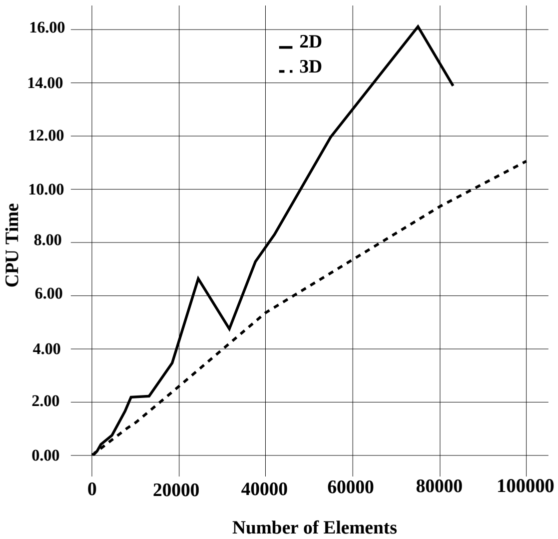


Figure 3: Mesh Migration Throughput

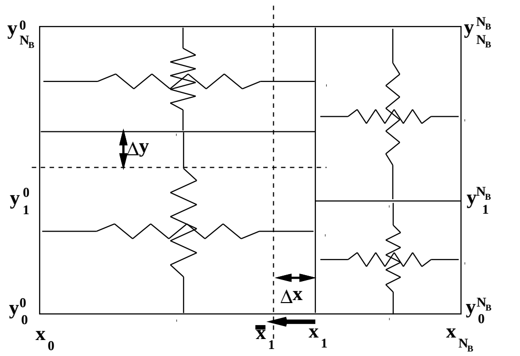
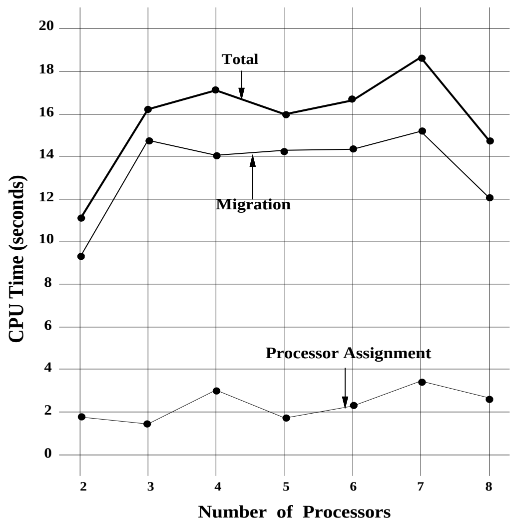


Figure 4: Linear 2D Load Balance System



3D: 146600 Elements

Figure 5: Load Balancing

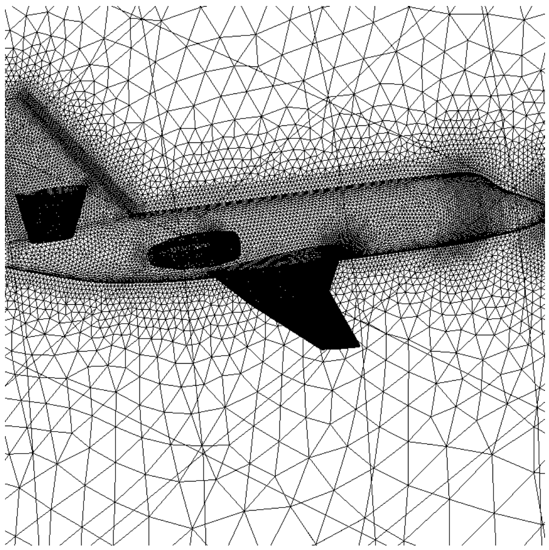
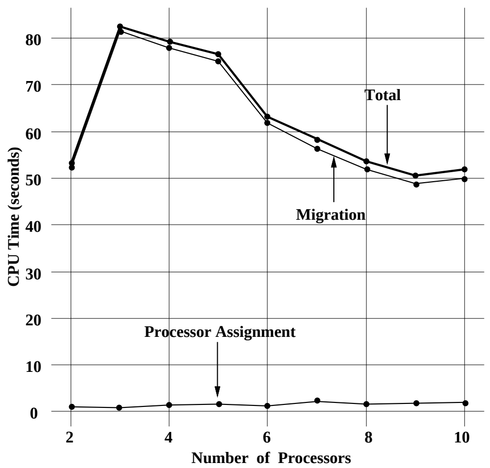


Figure 7: Dassault Falcon



2D: 50000 Elements

Figure 6: Load Balancing

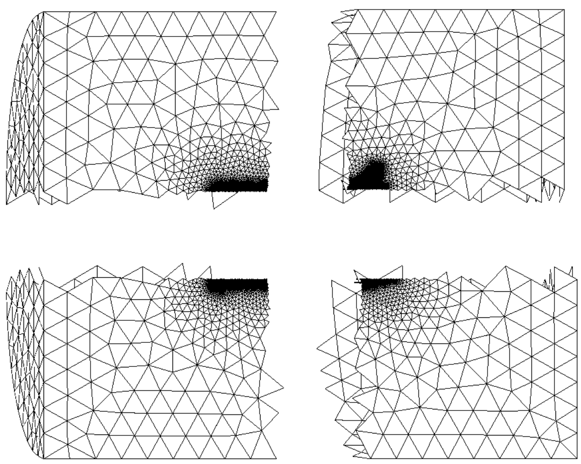
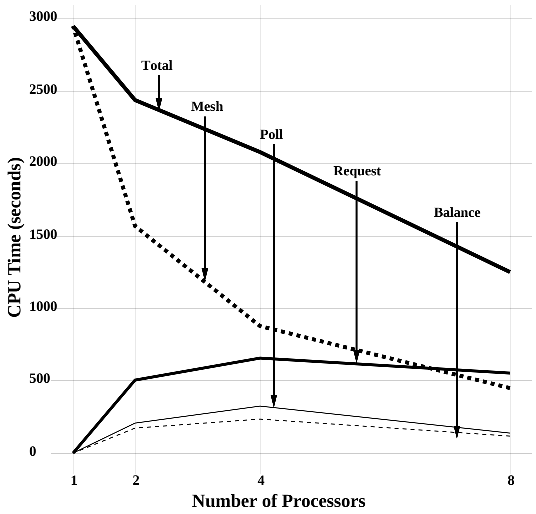
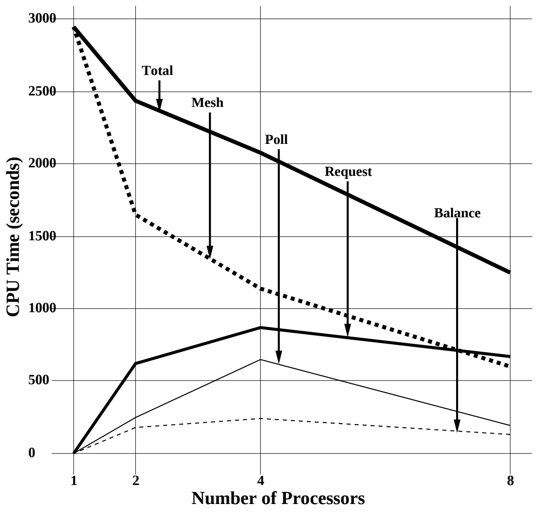


Figure 8: 3D Parallel Mesh Generation -
4 processor example



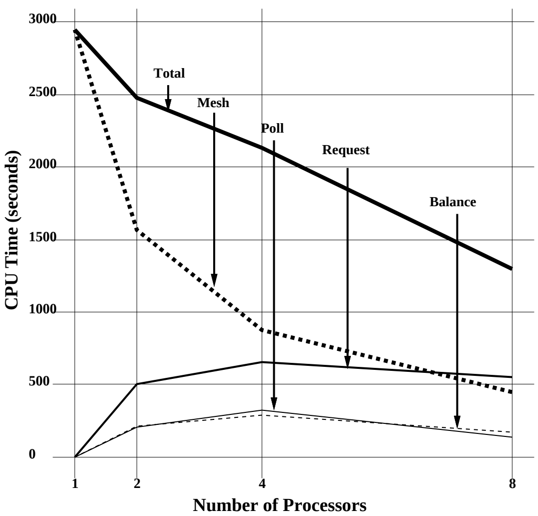
Avg. times w/o Final Balance

Figure 9: Mesh Generation



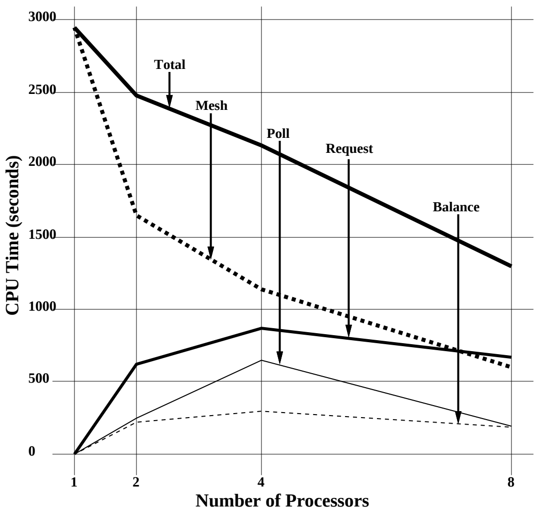
Max. times w/o Final Balance

Figure 11: Mesh Generation



Avg. times w/ Final Balance

Figure 10: Mesh Generation



Max. times w/ Final Balance

Figure 12: Mesh Generation

References

[1] T.Okusanya and J.Peraire. Parallel Unstructured Mesh Generation. *5th Int. Conf. on Numerical Grid Generation*, 1996.

[2] A. Jameson. Computational Algorithms for Aerodynamic Analysis and Design. *Applied Numerical Mathematics*, pages 383–422, 1992.

[3] J.Peraire, J.Peiro, and K.Morgan. The FELISA system. An unstructured mesh based system for aerodynamic analysis. Technical report, Computational Aerospace Science Laboratory/M.I.T, 1983.

[4] N.T.Frink, P.Parikh, and S.Pirzadeh. A Fast Upwind Solver for the Euler Equations on 3D Unstructured Meshes. *AIAA*, 1991.

[5] K.L.Bibb, J.Peraire, and C.J.Riley. Hypersonic Flow Computations On Unstructured Meshes. *AIAA*, 1997.

[6] N.P.Weatherhill, N.A.Verhoeven, and K.Morgan. Dynamic load balancing in a 2D parallel Delaunay mesh generator. *Parallel Computational Fluid Dynamics*, pages 641–648, 1995.

[7] R.Löhner, J.Camberos, and M.Merriam. Parallel Unstructured Grid Generation. *Comp. Meth. Appl. Mech. Engg.*, pages 343–357, 1992.

[8] M.Saxena and R.Perruchio. Parallel FEM Algorithms Based on Recursive Spatial Decompositions. *Computers and Structures*, pages 817–831, 1992.

[9] H.L. de Cougny, M.S. Shephard, and C. Ozturan. Parallel Three-Dimensional Mesh Generation on Distributed Memory MIMD Computers. Technical Report SCOREC Report #7, Rensselaer Polytechnic Institute, 1995.

[10] M.S.Shephard, J.E.Flaherty, H.L. de Cougny, C.Ozturan, C.L.Bottasso, and M.W.Beal. Parallel Automated Adaptive Procedures for Unstructured Meshes. *Parallel Computing in Computational Fluid Dynamics, AGARD-FDP-VKI*, 1995.

[11] D.F.Watson. Computing the n-dimensional Delaunay Tessellation with Application to Voronoi Polytopes. *The Computer Journal*, pages 167–171, 1981.

[12] S.Rebay. Efficient Unstructured Mesh Generation by Means of Delaunay Triangulation and Bowyer-Watson Algorithm. *Journal of Computational Physics*, pages 125–138, 1993.

[13] P.Chew. Mesh generation, curved surfaces and guaranteed quality triangles. Technical report, IMA Workshop on Modeling, Mesh Generation and Adaptive Num. Meth. for Part. Diff. Eqs, Univ. Minnesota, Minneapolis, 1993.

[14] N.Weatherhill and O.Hassan. Efficient three-dimensional grid generation using the Delaunay triangulation. *1st European Comp. Fluid Dynamics Conf., Brussels*, 1992.

[15] J.Bonet and J.Peraire. An Alternate Digital Tree for Geometric Searching and Interaction Problems. *Intl. Journal Num. Methods in Eng.*, 1995.

[16] A. Vidwans, Y.Kallinderis, and V. Venkatakrishnan. Parallel Dynamic Load-Balancing Algorithm for Three-Dimensional Adaptive Unstructured Grids. *AIAA*, pages 497–505, 1994.

[17] P.J.Green and R.Sibson. Computing the Dirichlet Tessellation in the Plane. *The Computer Journal*, pages 168–173, 1977.